TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky a mezioborových inženýrských studií

BAKALÁŘSKÁ PRÁCE

Vyhodnocení podobnosti textových souborů

Evaluation of similarity of text files

Liberec 2006 Radek Bartman

TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky a mezioborových inženýrských studií

Studijní program: B2612 – Elektrotechnika a informatika

Studijní obor: 2612R011 – Elektronické informační a řídící systémy

Vyhodnocení podobnosti textových souborů

Evaluation of similarity of text files

Bakalářská práce

Autor: **Radek Bartman**

Vedoucí bakalářské práce: Ing. Tomáš Pluhař

Konzultant: Ing. Tomáš Martinec

V Liberci 17. 5. 2006

Prohlášení

Byl jsem seznámen s tím, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 o právu autorském, zejména § 60 (školní dílo).

Beru na vědomí, že TUL má právo na uzavření licenční smlouvy o užití mé bakalářské práce a prohlašuji, že s o u h l a s í m s případným užitím mé bakalářské práce (prodej, zapůjčení apod.).

Jsem si vědom, že užít své bakalářské práce či poskytnout licenci k jejímu využití mohu jen se souhlasem TUL, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených univerzitou na vytvoření díla (až do jejich skutečné výše).

Bakalářskou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím bakalářské práce a konzultantem.

Datum:

Podpis:

PODĚKOVÁNÍ

Děkuji Ing. Tomáši Pluhaři za odborné vedení bakalářské práce a poskytnuté konzultace a Ing. Tomáši Martincovi za poskytnuté konzultace.

Děkuji také diskutujícím na serveru www.builder.cz, jelikož mi poskytli cenné rady a informace.

V neposlední řadě děkuji svým rodičům a celé rodině za psychickou i finanční podporu při studiu.

Anotace

Bakalářská práce se zabývá vyhodnocením podobnosti textových souborů, jejich podobnost je posuzována dle množství navrhnutých kritérií. Jednotlivá kritéria jsou zpracována pomocí filtrů do knihoven funkcí, které umožňují modulární řazení filtrů za sebou. K těmto filtrům je vytvořeno unifikované rozhraní, které umožňuje spuštění a konfigurování filtru pomocí stejných programovacích prostředků. Bakalářská práce také obsahuje ukázkový program, který používá navrhnuté filtry na vzorek souborů a vyhodnocuje jejich podobnost. Výsledky jsou pro každý soubor ukládány do databáze, aby se mohli v budoucnu porovnávat s novými soubory.

Annotation

This bachelor labor is based on similarities of text file evaluation; the similarity is assessed due to number of suggested criteria. A particular criterion is elaborated according to filters into function libraries, which enable modular sorting subsequently. Along with filters unified interface is created. This separation provides activation and configuration of filter using resembling programming resources. Bachelor work also includes sample program, which applies drafted filters on sample files and analyses their similitude. The results are deposited for each file into the database to be compared in the future with new files.

Obsah

Úvod

Cílem bakalářské práce je vytvoření programu, který umožní vyhodnocení podobnosti textových souborů. Program by měl mít intuitivní ovládání a měl by umožnit snadné zařazení jednotlivých filtrů do stromové struktury za sebou. Tento program by měl být konfigurovatelný jak co se týče konfigurovatelnosti jednotlivých filtrů, tak co se týče konfigurace samotné stromové struktury filtrů. Samotný program se bude skládat ze dvou částí – v první se bude nastavovat konfigurace hodnotících kritérií, ve druhé se budou posuzovat výsledky použitých filtrů aplikovaných na daný soubor s množstvím již dříve vyhodnocených souborů.

Celý program musí být co nejvíce otevřený, aby se mohli kdykoliv v budoucnu napsat nové funkce (filtry), které do něj budou moci být začleněny, aniž by se musel hlavní program znovu kompilovat.

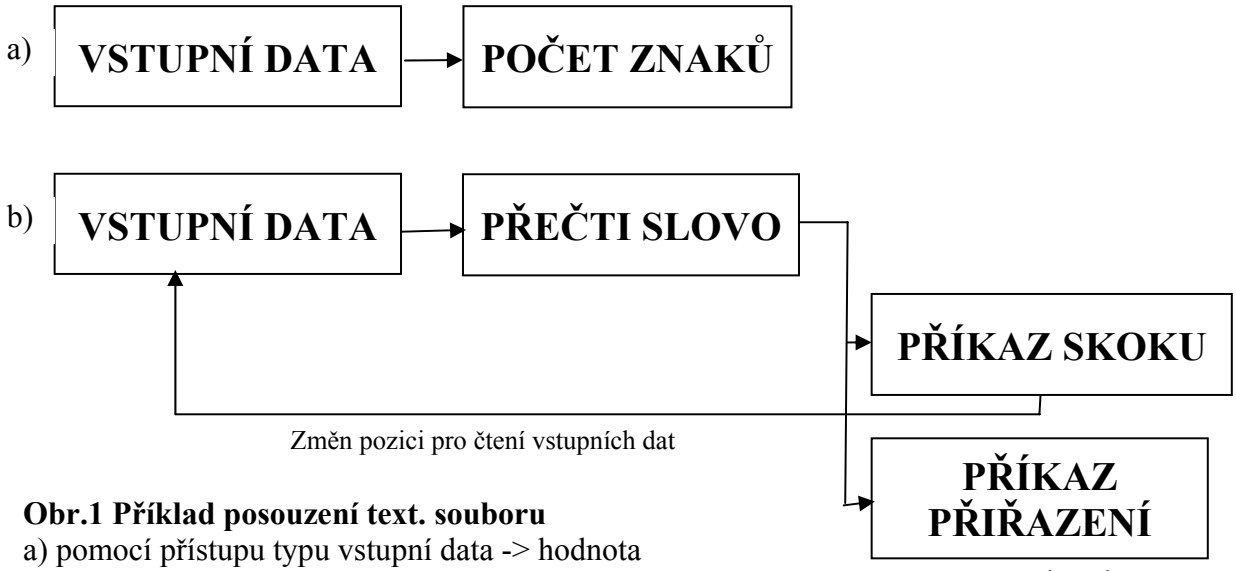
Jednotlivé filtry by měli být v zásadě dvou typů; buď pouze vstupní (pouze analyzují vstupní data), nebo vstupně-výstupní (provedou zadanou úpravu vstupních dat a ty zpřístupní k dalšímu zpracování pro vstupní filtr). Výsledkem vstupního filtru by mělo být celé číslo, které přesně vystihuje zadané kriterium. Výsledek vstupně-výstupního filtru nemusí být definován, protože se dá nahradit kombinací vstupně-výstupního filtru a jednoho(nebo více) vstupních filtrů. Filtry by měli být složeny ze dvou částí – grafické konfigurace filtru(nezávislé na použitém algoritmu) a samotné výkonné části. Filtr by měl být schopen zapamatovat si svou konfiguraci.

Jako programovací jazyk bude použit jazyk C++, jež je plně objektově orientován a nabízí i podporu kontejnerů, proudů, a snadnou dynamickou alokaci objektů, což bude jistě k užitku. Jako vývojové prostředí bude zvolen C++ Builder ve verzi 6.0, jež oproti konkurenci nabízí integrovanou podporu souborů INI, dynamických knihoven, databází a velké množství grafických komponent.

1 Přístupy posuzování textových souborů

Obor, kterým se zabývá tato bakalářská práce není příliš rozšířen a není mi dosud známa žádná dostupná literatura, která by se porovnáváním textových souborů zabývala. Budou proto použity termíny, které nejsou nikde zavedeny nebo mi nejsou známy.

Textové soubory se dají posuzovat pomocí několika přístupů, jedná se buď o přístup typu *vstupní data -> hodnota daného kritéria* (viz obr 1a), nebo o přístup *strukturální*, kdy se analyzují vstupní data postupně a podle toho jakého druhu jsou se inkrementují hodnoty jednotlivých kriterií, případně se mění pozice ve vstupním souboru (viz Obr. 1b)



Obr.1 Příklad posouzení text. souboru
a) pomocí přístupu typu vstupní data -> hodnota daného kriteria
b) pomocí strukturálního přístupu

Z principu je jasné že přístup *vstupní data -> hodnota daného kritéria* bude aplikačně jednodušší a také relativně implementačně nezávislý – u spousty kritérií nás nezajímají jednotlivé řídící příkazy. Prostor pro vznik chyby tu je pouze velmi malý – a i když chyba vznikne, tak nám s největší pravděpodobností ovlivní pouze hodnotu

malého množství filtrů. Výsledkem filtru používající tento přístup bude celočíselná hodnota, charakterizující počet výskytů určitého jevu.

Strukturální přístup bude složitější, musí totiž uvažovat naprosto všechny možné možnosti vstupních dat (např. odkazy), vyhodnocovací algoritmus musí vědět ke každému slovu i písmenku jeho přesný význam; zato nám však nejspíše poskytne nejvíce kompletní informace i s údaji o umístění výskytů. Tento přístup je velice náchylný na chyby, protože pokud špatně identifikujeme jen jediný řetězec, může to negativně ovlivnit hodnoty všech kriterií. Např. i tak dokonalý prostředek CodeInsight od firmy Borland, který je použit pro doplnění psaného textu dle logické souvislosti obsahuje chyby. Pokud nastane tato chyba, je celý tento nástroj vyřazen z činnosti. Výsledkem filtru pracujících pomocí tohoto přístupu je obecně struktura, která může obsahovat i nečíselné hodnoty – řetězce, nebo další struktury.

Je možný také *částečný strukturální přístup*, kdy se prohledává nejprve hlavní blok či smyčka souboru, tento nalezený blok se vyhodnotí přístupem *vstupní data -> hodnota*. Pokud by tento blok obsahoval další podblok, tak se pro jeho vyhodnocení opět použije ta samá funkce; tento přístup je tudíž rekurzivní. Výsledkem tohoto typu filtru je opět struktura, která obsahuje celočíselné hodnoty a může také obsahovat sama sebe. Tento přístup je aplikačně relativně jednoduchý a poskytuje nám velké množství informací, problém však nastává s interpretací získaných dat. Tato data by se dala snadno zobrazit ve stromové struktuře, avšak vyhodnocení podobnosti takovýchto dat je problematičtější.

Nejprve jsem se chtěl pokusit zpracovat dané téma pomocí *strukturálního přístupu*, avšak posléze jsem zjistil, že jednotlivá kriteria by nemohla být implementačně nezávislá (lišila by se např. pro programovací jazyky DELPHI a C++, nebo pro textové soubory *.txt), a museli by se napsat pro každý programovací jazyk zvlášť, takže by nebyla obecná. *Částečný strukturální přístup* zase naráží na interpretaci výsledků, a opět by jednotlivá kriteria nebyla implementačně nezávislá.

Pro tuto implementační náročnost jsem zadanou úlohu zpracoval pomocí přímé metody, kdy zjišťuji hodnotu každého parametru zvlášť pomocí různých filtrů. Snažil jsem se aby jednotlivé filtry byly co nejvíce konfigurovatelné, aby mohly být použity téměř pro jakákoli vstupní data.

2 Použité programovací techniky

Ve vlastním programu i v jeho dílčích knihovnách používám specifické programovací techniky, jako je práce s proudy, operace s množinami, volaní funkcí WIN API pro nahrávání a uvolňování dynamických knihoven, práce s inicializačními soubory INI, či příkazy jazyka SQL. V této kapitole si shrneme jejich základní vlastnosti.

2.1 Proudy (streamy)

Protože hlavním úkolem této bakalářské práce je práce se soubory, musíme si blíže vysvětlit vstupní a výstupní techniky programovacího jazyka C++ pro práci se s vstupem/výstupem. Pro vstupy a výstupy(dat) má jazyk C++ velice silný nástroj - proudy (anglicky stream). Proud je zvláštní třída, která má přetížené operátory „<<" a „>>" pro vstup a výstup z/do proudu.

Program v jazyce C++ se dívá na vstup a výstup jako na proud bajtů. Při vstupu načte jednotlivé bajty ze vstupního proudu a my s nimy v hlavním programu pracujeme jako s proměnnými. Pokud je soubor textový, program se na každý byte dívá jako na znak (pro tuto operaci, kdy se jednotlivým bytům přiřazuje jejich znakový ekvivalent existuje normou definovaná ASCII tabulka, viz např. [4]) Do vstupního proudu mohou bajty přicházet z klávesnice, ale také z pevného disku, z nějakého místa v paměti, z jiného programu nebo například ze sítě. Stejně tak mohou bajty z výstupního souboru téci na obrazovku, na tiskárnu, pevný disk, či na jakékoliv výstupní medium pro které máme definovanou výstupní třídu. Příjemným důsledkem tohoto přístupu je, že můžeme zacházet se vstupem z klávesnice s naprosto stejnou technikou, jako kdyby vstup pocházel z pevného disku nebo z nějakého vzdáleného počítače. Podobně můžeme za pomocí proudů zpracovat výstup nezávisle na místě určení bajtů. Samotný vstup a výstup se tedy sestává ze dvou částí:

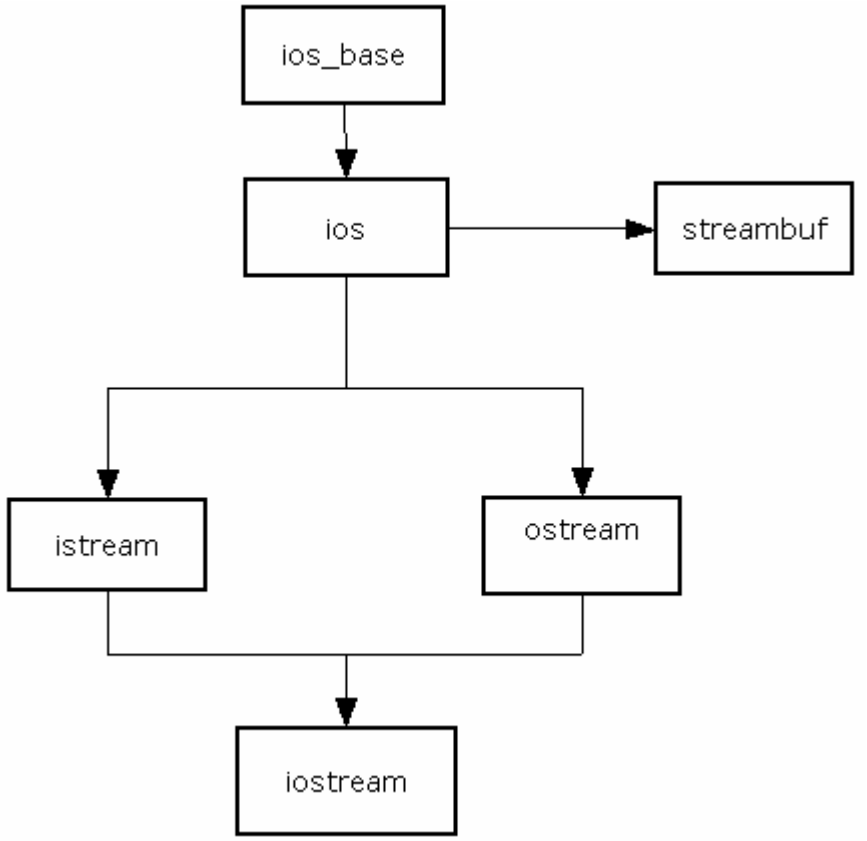
- Spojení proudu se vstupem do programu
- Spojení proudu se souborem, pamětí, klávesnicí, ..

Každý proud tedy potřebuje 2 spojení – na každém konci jedno. Pro vstupní proud tak máme na jedné straně proud bajtů, kterým proudí bajty z proudu do programu,a na

straně druhé proudí bajty do proudu. Obdobně správa u výstupního proudu vyžaduje spojit program s proudem jako zdroj, a spojení nějakého výstupu jako cíl. Je to stejné jako potrubí, kterým místo vody protékají informace(bajty).

Obvykle je práce s výstupním médiem mnohem rychlejší pokud použijeme vyrovnávací paměť – tzv. buffer. Jedná se vlastně o blok paměti, který je dočasný a v samotném přenosu bajtů tvoří jakéhosi prostředníka. Pokud například čteme ze souboru jednotlivé znaky, je pro počítač mnohem výhodnější přečíst celý blok dat než přímé čtení znaku po znaku. Kdyby měl počítač pokaždé když chceme zpracovat jeden znak přistupovat k pevnému disku, tak by výsledná doba zpracování celého souboru byla mnohem delší, než když si program načte blok o velikosti např. 512 bytů do operační paměti, a pak přistupuje při zpracování jednotlivých znaků místo na disk do tohoto místa v paměti. Operační paměť je totiž obvykle mnohem rychlejší. Při vstupu z klávesnice toto sice není potřeba, ale vyrovnávací paměť nám umožní vrátit se o znak zpět, pokud omylem stiskneme špatnou klávesu. Program pak celou znakovou sekvenci obvykle vyhodnocuje až po stisknutí klávesy ENTER.

Protože pro práci s proudy a vyrovnávací pamětí neexistují na hardwarové úrovni žádné jednoduché funkce, definuje si jazyk C++ několik tříd navržených pro implementaci proudů a vyrovnávacích pamětí. Tyto třídy dostaneme do programu vložením hlavičkového souboru *iostream*. Kmen předků třídy ***iostream*** je na následujícím obrázku.



Obr.2 Předkové třídy iostream

Třída ***streambuf*** poskytuje vyrovnávací paměť pro náš proud; uchovává si informace o její velikosti, aktuální pozici a má metody pro její zaplnění, vyprázdnění a správu.
Třída ***ios_base*** obsahuje obecné vlastnosti proudu, jako je režim otevření (binární/textový), či zda je vůbec otevřen.
Třída ***ios*** je založena na třídě ***ios_base*** a obsahuje ukazatel na objekt třídy ***streambuf***.
Třída ***ostream*** je odvozená od třídy ***ios*** a obsahuje metody pro výstup do proudu.
Třída ***istream*** je také odvozená od třídy ***ios*** a obsahuje metody pro vstup z proudu.
Společným potomkem tříd ***istream*** a ***ostream*** je třída ***iostream***, ta dědí metody jak pro vstup, tak pro výstup.

Pro výstup do proudu se používá přetížený operátor „<<", takže například příkaz *cout << "Dobrý den";* nám do objektu *cout* , což je standardní proud pro výstup na obrazovku, vloží řetězec „Dobrý den". Třída ***iostream*** obsahuje přetíženou verzi tohoto operátoru pro většinu základních typů, takže např. příkaz *cout << 15;* nám vypíše na obrazovku číslo 15 bez toho, že bychom museli provádět nějakou konverzi mi v programu - tuto činnost zajistí správná verze přetíženého operátoru „<<". Tento

operátor má jednu velice zajímavou vlastnost, a to že nám vrací referenci na objekt třídy *ostream* – toho se dá využít na jednoduché řetězení příkazů – např. příkaz
cout << "Hodnota čísla a je "<< cisloa; nám zobrazí na obrazovku hlášku „Hodnota čísla a je -7" (tedy pokud proměnná *cisloa* = -7).

Pro vstup z proudu se používá přetížená verze operátoru „>>", takže příkazy
int a; double b;
cin >> a; cin>>b;
nám z objektu *cin*, což je standardní proud pro vstup z klávesnice, uloží do celočíselné proměnné *a* číslo zadané na klávesnici, a do proměnné *b* se uloží zadané reálné číslo.

Nejpodstatnější funkce pro práci s vstupem co jsou použity v této práci (tj. většina znakově orientovaných) jsou:

istream& get (char& c);

- uloží do proměnné c znak ze vstupního proudu a posune ukazatel ve *streambuf* (znak čte neformátovaně!)

istream& unget ( );

- vrátí naposledy přečtený znak do vstupního proudu, takto vrácený znak se při dalším čtení chová jako by se s ním před tím nic nedělo

int Peek();

- tento příkaz přečte byte ze vstupního proudu aniž by ho přitom z toho proudu vyňal, podle dokumentace by neměl ovlivňovat hodnoty stavových bitů, ale zjistil jsem že tyto bity nastavuje stejně jako funkce *get*! Je proto důležité hlavně na konci souboru používat metodu *clear();* viz dále

stream& ignore (streamsize n = 1, int delim = EOF);

- tato funkce vyjme ze vstupního proudu maximálně n znaků; pokud nedosáhne konce proudu nebo nenarazí na znak delim(může být např. znak konce řádky)

pos_type tellg();

- vrátí pozici ve vstupním proudu

istream_type& seekg(pos_type& position);

- nastaví absolutní pozici v proudu na pozici *position*

istream_type& seekg(off_type& offset, ios_base::seekdir dir);

- nastaví pozici souboru relativně o celočíselný počet míst - parametr *offset* typu *streambuf*
  - od začatku (parametr *dir = ios_base::beg*)
  - konce (parametr *dir = ios_base::end*)
  - od aktuální polohy (parametr *dir = ios_base::cur*)

void ios::clear (iostate state = goodbit);

- tento příkaz používáme, pokud chceme změnit stav proudu (ignorujeme přitom jeho skutečný stav), parametrem *iostate* určujeme stavový bit který chceme vymazat – možné hodnoty jsou **badbit** (kritická chyba ve vyrovnávací paměti), **eofbit** (dosažen konec souboru), **failbit** (selhání přístupu k proudu), **goodbit** (všechno v pořádku)

Nejpodstatnější funkce pro práci s výstupem jsou:

ostream& put (char ch);

- vloží do výstupního proudu znak *ch*

ostream& seekp (streampos pos);

- vykonává stejnou funkci jako obdobná metoda *seekg* pro vstupní proudy

ostream& seekp (streamoff off, ios_base::seekdir dir);

- vykonává stejnou funkci jako obdobná metoda *seekg* pro vstupní proudy

streampos tellp ( );

- vykonává stejnou funkci jako obdobná metoda *tellg* pro vstupní proudy

void ios::clear (iostate state = goodbit);

- tato metoda je již popsána u metod pro vstup (pochází ze společného předka tříd ***ostream*** a ***istream*** – třídy ***ios***)

Pro vstup a výstup do souboru se používají třídy ***ifstream*** a ***ofstream*** definované v hlavičkovém souboru *fstream*. Tyto třídy mají konstruktor ve tvaru

*explicit ifstream (const char * filename, openmode mode = in);*

- otevře soubor určený řetězcem *filename* módem určeným proměnnou *mode*

*explicit ofstream (const char * filename, openmode mode = out | trunc);*

- otevře soubor určený řetězcem *filename* módem určeným proměnnou mode

proměnná *mode* může nabývat hodnot určených libovolnou kombinací těchto bitů (definovaných ve třídě ***ios_base***):

app	(append)	Při každém výstupu přesune ukazatel na konec souboru
ate	(at end)	Při otevření přesune ukazatel na konec souboru
binary	(binary)	Použije binární mód místo textového
in	(in)	Povolí vstupní operace s proudem
out	(out)	Povolí výstupní operace s proudem
trunc	(truncate)	Při otevření zkrátí soubor na nulovou velikost

Pro bližší informace o proudech, jako jsou stavy proudů, manipulátory, formátování incore a další viz [3], [4], [8].

2.2 Množiny a multimnožiny

Pro práci s proměnným počtem parametrů se dá s výhodou použít kontejneru typu množina (set); pro operace kdy zjišťujeme četnost určitého slova či slov opět s výhodou použijeme kontejneru multimnožina (multiset). Jejich základní vlastnosti si shrneme v této kapitole.

Základní rozdíl mezi množinou a multimnožinou je ten, že v množině může být každý prvek uložen jen jednou, kdežto v multimnožině může být jeden prvek uložen vícekrát. Oba dva druhy kontejnerů ukládaná data třídí – řetězce podle abecedy, čísla podle hodnoty; toho se dá využít například při počítání četnosti slov nebo při jednoduchém třídění hodnot.

Množina i multimnožina jsou součástí tzv. **Standard Template Library (STL)**, což je standardizovaný soubor šablon, které musí každý překladač C++ obsahovat. I pouhý výčet by vydal na několik stran, proto zde uvedu pouze základní objekty a jejich metody, které mi byly prostředkem k realizaci vlastního programu.

Konstruktory používáme ve tvaru:
set <identifikátor typu> x;

- deklaruje *x* jako množinu typu *<identifikátor typu>* např. *set <char> x;*

multiset <identifikátor typu> x;

- deklaruje x jako multimnožinu typu *<identifikátor typu>* např. *multiset <char> x;*

Základní používané metody pro oba typy kontejnerů jsou:
void clear();

- vymaže všechny prvky množiny/multimnožiny

void insert(const value_type& x);

- vloží x do množiny/multimnožiny

bool empty();

- vrací TRUE pokud je množina / multimnožina prázdná (v opačném případě vrací false)

void erase(const value_type& x);

- vyjme prvek *x* z množiny / multimnožiny

size_type count(const value_type & x);

- vrátí počet výskytů prvku *x* v kontejneru; pro množinu tedy vrátí buď 0 (prvek se nevyskytuje), nebo 1 (prvek se vyskytuje)

Kontejnery by byly slabý nástroj, pokud by neexistovala možnost získávat jednotlivá data v něm uložená – pro tuto činnost nám jazyk C++ poskytuje tzv. iterátory. Iterátor si můžeme představit jako takový generalizovaný ukazatel určený pro kontejnery; u ukazatele např. operátor ++ způsobí přechod na další prvek v poli, iterátor udělá to samé, ale následující prvek nemusí být v paměti uložen bezprostředně za prvkem předchozím. Výhoda iterátorů spočívá v tom, že pomocí nich mohu pracovat s libovolným kontejnerem, aniž bych věděl o jaký druh kontejneru se jedná. Tuto vlastnost hojně využívají algoritmy knihovny STL.
Pro práci s kontejnery existují ještě 2 základní metody které jsme si neuvedli – jsou to metody begin() a end():

iterator begin();

- vrací iterátor který ukazuje na první prvek v kontejneru

iterator end()

- vrací iterátor který ukazuje bezprostředně ZA poslední prvek v kontejneru

Bližší informace o množinách, multimožinách, iterátorech a dalších šablonách knihovny STL viz např. [4].

2.3 Funkce WIN API pro nahrávání a uvolňování dynamicky linkovaných knihoven

Jelikož výsledná aplikace má být co nejvíce otevřená novým filtrům, vzniká zde požadavek aby jednotlivé filtry mohli být přidávány do programu až za běhu programu. Pomocí staticky linkovaných knihoven se toho nedá docílit; ty jsou vhodné pouze pro programy které mají svou činnost jasně a přesně definovanou, a tam, kde není požadavek na snadnou rozšiřitelnost aplikace. Proto jsem se rozhodl použít knihovny DLL, které mají spoustu předností:

1) DLL vytvořená v jednom vývojovém prostředí může být použita v jiném vývojovém prostředí
2) DLL je nezávislá na programovacím jazyce. Tak je možno vytvořit DLL v jazyce C++ a použít jí v programu který je vytvořen např. v jazyce BASIC
3) Protože se DLL knihovna linkuje dynamicky (až při spuštění aplikace), je snadná revize knihovny bez nutnosti rekompilace aplikace (nová verze musí zpětně kompatibilní)

Vývojové prostředí C++ Builderu a jeho knihovny obsahují funkce jak pro statické, tak pro dynamické knihovny; avšak pro dynamické knihovny vyžaduje aby byla při kompilaci přítomna knihovna importů (soubor s příponou LIB), která obsahuje seznam hlaviček všech funkcí a tříd, které jsou v DLL označeny jako exportní. Já jsem použil třetí možnou cestu, a to že volám funkce WIN API pro nahrávání a uvolňování knihovny sám explicitně.

Vybrané funkce WIN API pro práci s DLL:

LoadLibrary();

Funkce *LoadLibrary()* mapuje specifikovaný proveditelný modul(knihovnu) do adresového prostoru aplikace a zvyšuje počítadlo odkazů dané DLL. Tato funkce se obvykle volá pro získání handle daného modulu. Hlavička je:

HINSTANCE LoadLibrary(LPCSTR lpLibFileName);

lpLibFileName je řetězec názvu knihovny (např. „definice.dll“)

Návratová hodnota: Při úspěchu vrací handle modulu, jinak vrací NULL.

FreeLibrary();

Funkce *FreeLibrary* sníží počítadlo odkazů dříve nahrané DLL. Jeli počítadlo odkazů rovno 0, bude DLL vyjmuta z adresového prostoru volajícího procesu a její handle nebude již dále platný. Hlavička je:

BOOL FreeLibrary(HMODULE hLibModule);

-*hLibModule* je handle uvolňované DLL
Návratová hodnota: Při úspěchu vrací true, při neúspěchu vrací false.

DLLEntryPoint();

Funkce *DLLEntryPoint()* je volitelná funkce knihovny a zajišťuje vstup do DLL. Je volána v při každém volání *LoadLibrary()* nebo *FreeLibrary()*, a její parametr *fwdReason* obsahuje důvod proč byla zavolána(blíže viz [5]). Hlavička je:

BOOL WINAPI DLLEntryPoint(HINSTACE hinstDLL, DWORD fwdReason, LPVOID lpvReserved)

-*hinstDLL* je handle DLL
-*fwdReason* obsahuje důvod proč byla funkce zavolána (blíže viz [5])
-*lpvReserved* rezervováno
Návratová hodnota: Pokud inicializace neproběhla úspěšně, vrací true.

GetProcAddress();

Funkce *GetProcAddress()* vrátí adresu specifikované exportní funkce obsažené v DLL. Hlavička je

FARPROC GetProcAddress(HMODULE module, LPCSTR lpProcName);

-*hModule* Handle DLL(získáno voláním *LoadLibrary*)

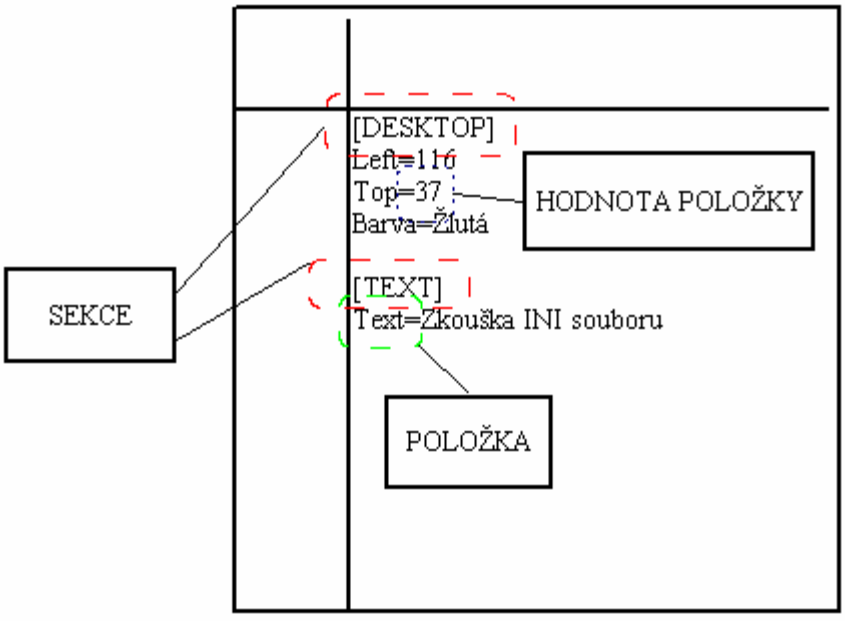
-*lpProcName* řetězec jména funkce, nebo ordinální číslo funkce. Tento řetězec NENÍ stejný jako jméno funkce v naší aplikaci. C++ názvy funkcí komolí; pokud je chceme zjistit, můžeme např. použít utilitu IMPDEF fy. Borland, nebo program Dependecy Walker(viz [19]).

Funkce *GetProcAddress()* vrací adresu funkce ve formě ukazatele typu FARPROC, při chybě vrací NULL.

2.4 Soubory INI

Přípona (INI) těchto souborů jasně vystihuje jejich funkci. Tyto soubory jsou často používané pro inicializaci a konfiguraci aplikace. Může v nich být např. velikost okna, barva písma, či řetězec titulku aplikace. Aplikace si tyto informace načte při svém spuštění; při ukládání je zase může uložit. Velkou výhodou těchto souborů je jejich jednoduchá organizace a hlavně že jsou textového formátu, takže se dají editovat v běžném textovém editoru.

Soubor INI obsahuje sekce (sections) (poznáme je podle toho, že jejich názvy jsou uvedeny mezi „[“ a „]”), a položky (keys), jež jsou členy dané sekce (poznáme je podle toho, že jejich názvy jsou následovány „=“). Položky mají svou hodnotu (value) (poznáme je podle toho, že jejich názvům předchází =).



Obr.2 Struktura INI souborů

Třída TIniFile:

Třída *TIniFile* je definovaná v knihovně *inifiles.hpp* a obsahuje následující metody a položky:

AnsiString ***FileName*** – název INI souboru, na který se odvoláváme instancí TIniFile

__fastcall TIniFile(const AnsiString FileName);
– konstruktor; *FileName* je jméno *IniSouboru*

long _fastcall ***ReadInteger****(AnsiString Section, AnsiString Ident , long Default);*
- načte ze sekce *Section* hodnotu položky *Ident*, pokud se položka nevyskytuje, vezme se hodnota *Default*

ReadString(), ReadBool() – to samé pro string a bool

void ***WriteInteger****(AnsiString Section, AnsiString Ident , long Value);*
- zapíše do sekce *SECTION* položku *IDENT* s hodnotou *VALUE*

WriteString(), WriteBool() – to samé pro string a bool

void fastcall ***DeleteKey****(AnsiString Section, AnsiString Ident);*
- vymaže ze sekce *Section* položku *Ident*

void ***EraseSection****(AnsiString Section);*
- vymaže sekci *Section*

2.5 Jazyk SQL

Pro komunikaci s databázemi se v dnešní době používá především dotazovací jazyk SQL, což je standardní počítačový jazyk pro přístup a manipulaci s databází. Tento jazyk je navržen tak, že jeho příkazy odpovídají přirozenému jazyku, konkrétně angličtině. (jeho původní název byl SEQUEL - Structured English Query Language).

Naučit se takový jazyk je díky tomu celkem jednoduché, a základy se dají zvládnout během krátké doby. Z tohoto důvodu se jazyk těší velké oblibě u uživatelů i u výrobců (např. Microsoft, Firebird, Informix, Interbase, MySQL, Oracle).

Příkazy jazyka SQL se dají rozdělit do 6 skupin:

1) Příkazy pro získání dat
2) Příkazy pro manipulaci s daty
3) Příkazy pro přenos dat
4) Příkazy pro definici dat
5) Příkazy pro řízení přístupu k datům
6) Další příkazy (komentáře)

V této práci byly použity pouze příkazy z prvních dvou skupin, proto budou popsány pouze tyto (a k tomu jen některé z nich), další jsou popsány např. v [12] a [13].

Ad 1) Příkazy pro získání dat

Jedná se především o příkaz *SELECT*, který nám slouží pro získání dat z tabulky.

Jeho plná syntaxe je ***SELECT*** *co* ***FROM*** *tabulka.* Tento příkaz nám zvolí položky *co* z tabulky *tabulka*. Parametr *co* nemusí být pouze jeden sloupec databáze, ale může se jednat o více sloupců, nebo dokonce o celou tabulku(v takovém případě za parametr *co* dosadíme „*“). Za parametrem *tabulka* může být použit ještě upřesňující kvalifikátor, nebo jich může být i více. Jedná se zejména o kvalifikátory

WHERE – česky kde, upřesňuje hodnoty jednotlivých sloupců, např. ***WHERE*** *Jmeno='Natália'* nám specifikuje že chceme vybrat záznam jehož položka *Jmeno* má hodnotu *Natália*.

ORDER BY – se používá pro třídění vrácených položek, například ***ORDER BY*** *Jmeno* nám položky vrátí jako setříděné podle sloupce *Jmeno* (abecedně). Za ním ještě může být použito klíčové slovo ***ASC***(vzestupně) nebo ***DESC***(sestupně) pro určení směru třídění.

GROUP BY – se používá pokud se větší počet záznamů má vypisovat na menší počet řádků, u tzv.agregačních funkcí (typicky suma, průměr), a my potřebujeme upřesnit z kterého sloupce se má ta suma, průměr počítat (např. ***SELECT*** *Company,*

***SUM**(Amount)* ***FROM** Sales* ***GROUP BY** Company* nám vrátí v sloupci ***SUM**(Amount*) sumu hodnot specifickou pro každou *Company*).

HAVING – se opět používá společně a agregačními funkcemi. Používá se, pokud chceme např. zvolit všechny položky jež mají dohromady sumu (průměr, …) hodnot větší/menší než specifikovanou (***HAVING SUM**(Plat) > 95400*).

Tyto kvalifikátory se můžou také řetězit za sebe pomocí standardních logických operací (**AND**, **OR**), např. ***WHERE** Suma=5* ***AND** Jmeno='Tomáš'*.

Ad 2) Příkazy pro manipulaci s daty

Zde jsou nejdůležitější příkazy ***INSERT***, ***UPDATE*** a ***DELETE***.

***INSERT INTO** tabulka [sloupec1, sloupec2, ...]* ***VALUES** (hodnota1,hodnota2, ...)* nám vloží do tabulky *tabulka* specifikovaný záznam; specifikace sloupců nemusí být uvedeny (pak se data vkládají postupně od prvního sloupce). Např. příkaz ***INSERT INTO** telseznam (jmeno, cislo)* ***VALUES** ('Rudolf Veselý', 777535934)* nám vloží do tabulky *telseznam* záznam o Rudolfu Veselém jehož telefonní číslo je 777535934. Pokud jsou *jmeno* a *cislo* první sloupce tabulky, můžeme závorku *(jmeno, cislo)* vynechat.

UPDATE** table* ***SET** column=value (column1=value1, ...)* nám změní hodnoty jednotlivých sloupců *column*(může jich být klidně i více) v tabulce *table*. Za tímto příkazem bývá obvykle kvalifikátor ***WHERE.

***DELETE FROM** table* ***WHERE** column=value (column1=value1, ...)* maže všechny záznamy z tabulky *table*, pro jejichž sloupce je splněna podmínka *column=value*.

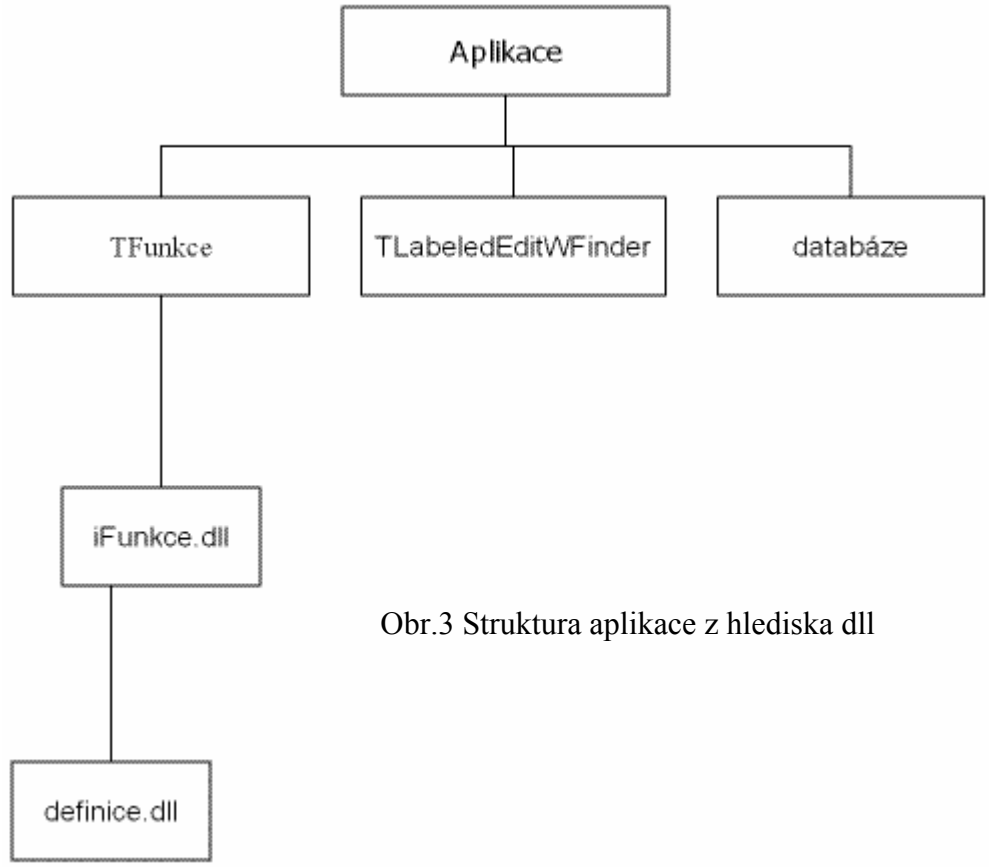
Ve všech použitých příkladech byl použit relační operátor "=", ale je samozřejmě možné používat i operátory ">" či "<".

Další použité techniky:

1) Funkce WIN API pro vyhledávání souborů (viz [5], [14])
2) Komponenty VCL (viz [5])

3 Organizace programu z hlediska DLL

Vlastní program je stavěn na několika úrovních; na nejvyšší úrovni je samozřejmě vlastní aplikace a její formulář, který obsahuje instanci třídy *TFunkce* pro návrh vyhodnocovacího schématu + spuštění vyhodnocení, instanci třídy *TlabeledEditwFinder* pro zaměřování výsledků ze schématu, a také obsahuje funkce pro práci s databází přes příkazy jazyka SQL.



Obr.3 Struktura aplikace z hlediska dll

Jak je vidět z obrázku 3. třída TFunkce není vytvořena jako jeden velký celek, ale obsahuje dynamicky linkovanou knihovnu DLL (zde „iFunkce.dll“, Funkce zde může být např. „Počet Znaků“, „Počet Slov“, podle toho jaké filtry použijeme), která obsahuje vlastní algoritmus pro vyhodnocování. V této DLL je vlastně uložen daný vyhodnocovací filtr. Dále také *TFunkce* obsahuje komponenty pro grafické zobrazení, Popup Menu pro nastavovaní parametrů a odkazy na další *TFunkce*(viz kapitola 6.1).

Knihovna „iFunkce.dll“ se ve skutečnosti nazývá např. „iPocetSlov.dll“. „iPocetZnaku.dll“) a skrývá v sobě 4 funkce s přesně definovanou hlavičkou pro nastavení vstupního souboru, přečtení výstupního souboru, konfiguraci daného filtru (vizuální dialog), a vlastní výkonnou část, kde se provádí definovaná funkce. Jiné nároky nejsou na tuto funkci kladeny, a ona si může dané funkce provádět jakýmikoliv prostředky; např. by mohla klidně vstupní soubor určený k vyhodnocení stáhnout z internetu.

Knihovna „iFunkce.dll“je v této stromové hierarchii nad knihovnou „definice.dll“; tato knihovna (definice.dll) je jádrem celé práce, a obsahuje všechny výkonné rutiny. Dosud všechny knihovny a instance tříd pracovali se jmény souborů a soubory samotnými; tato knihovna je však natolik obecná, že používá jako své parametry proudy. To je její velká výhoda, protože proud nemusí být pouze souborový (i když to je jeho nejčastější podoba), takže můžeme jednotlivé funkce používat s jakýmkoliv potomkem tříd istream a ostream, i s uživatelsky definovaným.

Toto členění programu nám poskytuje několik výhod:

1) Pokud nefunguje správně jedna část programu, je usnadněna lokalizace problému, protože je relativně snadné zjistit která část kódu způsobuje problém.
2) Oddělení algoritmů pro vlastní výkon daného filtru („definice.dll“) a jeho grafickou interpretaci a zadávání parametrů („iFunkce.dll“) nám umožňuje úpravu algoritmu pouhou opravou DLL která má v sobě uloženou vlastní výkonnou část – to nám může ušetřit spoustu času, pokud zjistíme že daná funkce nepracuje správně. Pokud tuto chybovou funkci používá více filtrů, nemusíme ji opravovat několikrát, ale pouze jednou v dané DLL.
3) Snadná rozšiřitelnost hlavního programu o nové funkce a filtry. Díky tomu že vlastní aplikace přistupuje k filtrům pomocí rozhraní třídy *TFunkce*, může být do programu přidán jakýkoliv filtr (tedy dynamická knihovna), který splňuje určitá pravidla, jako jsou přesně definované (co se týče názvu a parametrů) funkce a názvu dané knihovny.

4 Organizace dynamické knihovny definice.dll

Dynamická knihovna definice.dll je jádrem celého programu, jsou v ní uloženy výkonné části všech filtrů, tudíž její organizace je klíčová. Tato knihovna je pro přehlednost rozdělena do dvou částí, tak jak je zvykem v jazyce C++; souboru s deklaracemi všech funkcí (hlavičkový soubor *definice.h*) a souboru kde jsou jednotlivé funkce definovány (soubor *definice.cpp*). Přeložená knihovna má příponu **DLL** a kromě ní se při překladu vytvoří také knihovna importů s příponou **LIB**. Pokud chceme použít funkce z této knihovny, stačí připojit soubor *definice.lib* do projektu, vložit hlavičkový soubor *definice.h*, a o zavedení knihovny *definice.dll* se už překladač postará sám automaticky.

Hlavičkový soubor *definice.h* má v sobě vloženy kromě knihoven *iostream* a *vcl* nezbytných pro výkonnou část knihovny také hlavičkový soubor *deflong.h*, což je soubor, ve kterém je uvedena definice datového typu *LONG*, jež může být definován jako libovolná celočíselná hodnota. Tato definice je zde z toho důvodu, že kdybychom někdy v budoucnu chtěli použít některou s funkcí spolu s velkými soubory, nemusel by nám klasický *int* postačovat, v tom případě stačí změnit pouze tento hlavičkový soubor, a nemusíme se o nic jiného starat. Dále je zde také uvedena definice svazu *TDelka*, jež používají funkce pro určení velikosti souboru. *TDelka* je definována jako

```
union TDelka {
 unsigned __int64 Celkova;
 struct{
       DWORD Nizsi,Vyssi;
      };
 };
```

a můžeme k němu tedy přistupovat buď jako k celkové 64bitové hodnotě přes proměnnou *Celkova*, nebo pro krátké soubory můžeme využít pouze proměnnou *Nizsi* (*Vyssi* samozřejmě můžeme používat také).

Soubor *definice.cpp* obsahuje kromě definic jednotlivých funkcí také vložené knihovny *set.h* a *windows.h*, jejichž funkce používají některé z naprogramovaných funkcí.

Jednotlivé funkce, které obsahuje tato knihovna, by se dali rozdělit do 4 kategorií:

1) Funkce vyhledávající soubory dle zadané masky
2) Funkce vstupně-výstupní, formátující vstup (spíše formátování odstraňují)
3) Funkce jež kvantifikují určité kriterium
4) Funkce pro základy strukturovaného přístupu (většinou jde o funkce vyhledávací)

Několik z následujících funkcí používá nespecifikovaný počet parametrů, využívá k tomu operátoru výpustka (...). Tyto parametry musí být ukončeny nějakým přesně definovaným symbolem, většinou se používá nulový ukazatel NULL. Samozřejmě že by se mohli místo takto složitého předávání parametrů předávat množiny nebo multimnožiny, jenže ty nemusejí být definovány v každém programovacím jazyce, a tyto funkce by se pak v těchto jazycích nedali použít. Proto jsem zvolil raději tento způsob předávání parametrů.

4.1 Funkce vyhledávající soubory dle zadané masky

Jedná se o funkce které mají za úkol vrátit seznam souborů vyhovujících zadané masce a počátečního adresáře.

*void FindFilesInDir (const char *ret_cesta, const char *ret_maska,*
*TStrings *seznam, TDelka max_delka);*

Parametry této funkce jsou:

**ret_cesta* – tento řetězec obsahuje cestu k adresáři, ve kterém se má hledat

**ret_maska* – tento řetězec určuje masku vyhledávaných souborů

**seznam* – je seznam řetězců do kterého se uloží všechny názvy souborů, jež splňují daná kriteria

max_delka – je maximální přípustná délka hledaných souborů (v bytech)

Tato funkce vyhledává všechny soubory v adresáři určeným *ret_cesta*, které splňují masku *ret_maska* a jsou menší než *max_delka*. Nejprve se zavolá funkce WIN API

FindFirstFile(), která nám vyhledá první soubor který vyhovuje zadané masce, a pak se volá funkce *FindNextFile()* až do doby, dokud je co hledat. U každého souboru se porovnává jeho velikost, a je-li soubor menší než *max_delka*, tak se přidá do seznamu *seznam* (metodou *Add*)

*void FindFilesThroughDir(const char *ret_cesta, const char *ret_maska,*
*TStrings *seznam, TDelka max_delka);*

Parametry této funkce jsou stejné jako u funkce předešlé, na rozdíl od předchozí však prohledává zadaný adresář včetně podadresářů. Tato funkce prohledává pomocí funkcí *FindFirstFile()*, *FindNextFile()* zadaný adresář, a pokud narazí na podadresář, tak volá funkci *FindFilesThroughDir()* - tedy sama sebe. Samozřejmě také před každým vyhledáváním adresářů volá také funkci *FindFilesInDir()*, aby se načetli soubory z aktuálního adresáře.

*unsigned __int64 VratDelkuSouboru(const char *ret_nazev);*

Tato funkce má jako parametr řetězec, ten určuje název souboru pro který chceme zjistit jeho délku. Pro zjištění délky se nejprve soubor musí otevřít, a pak se volá funkce WIN API *GetFileSize()*.

4.2 Funkce vstupně-výstupní, formátující vstup

Následující 2 funkce si všímají faktu, že prakticky všechny moderní jazyky používají komentáře vložené přímo do zdrojového kódu. Tyto komentáře samozřejmě nemají na činnost programu žádný vliv, jsou určeny pouze pro lepší orientaci ve zdrojovém kódu. Jelikož tyto komentáře nemají na činnost programu vliv, pre-compiler je před vlastní kompilací programu vymaže. Přesně tuto činnost vykonávají také tyto dvě funkce:

bool _export DeCommentVR(ostream& str_vystupni, istream& str_vstupni,
*istream::pos_type pos_konec, const char *ret_levy, const char *ret_pravy);*

Tato funkce odstraňuje víceřádkové komentáře. To jsou komentáře, které jsou uvozeny nějakou definovanou sekvencí, a nějakou jinou sekvencí jsou ukončeny (i když tato sekvence může být teoreticky také stejná). Takto se může komentovat třeba i půlka zdrojového kódu a pak by nebylo žádoucí vyhodnocovat komentovaný obsah.

Parametry této funkce jsou:

&str_vystupni – což je reference na výstupní proud, do kterého se bude zapisovat text zbavený komentářů (proud musí být otevřen!)

&str_vstupni – což je reference na vstupní proud, ze kterého se bude číst zdrojový text (proud musí být otevřen!)

pos_konec – což je „ukazatel" na maximální pozici ve vstupním proudu; pokud se proud dostane až za pozici *pos_konec*, ukončí se výstup do výstupního proudu (vlastně skončí celá operace *DeCommentVR*)

**ret_levy – co*ž je řetězec, určující levou, otevírací sekvenci komentáře

**ret_pravy* – což je řetězec, určující pravou, zavírací sekvenci komentáře

Tato funkce pracuje tak, že nejprve cyklicky prohledává každý znak souboru (pomocí konstrukce *while (str_vstupni.get(znak)) { }*) a porovnává, zdali je nalezený znak prvním znakem otevírací sekvence. Pokud zjistí že ano, vrátí naposled přečtený znak do vstupního proudu, a pokusí se načíst řetězec znaků který je dlouhý stejně jako *ret_levy*. Pokud se načtený řetězec shoduje, znaky které následují se nezapisují do výstupního proudu. Pokud se načtený řetězec neshoduje, funkce nastaví pozici vstupního proudu za původně načtený znak, ten se uloží do výstupního proudu a pokračuje se další otočkou. Stejným způsobem se hledá konec komentáře. Až se nalezne konec komentáře, čtené znaky se opět začnou posílat do výstupního proudu. Funkce na začátku každého cyklu kontroluje zdali nebylo dosaženo pozice *pos_konec*, a pokud zjistí že ano, ukončí se.

*bool DeComment1R(ostream& str_vystupni, istream& str_vstupni, istream::pos_type pos_konec, const char *ret_jednoradkovy);*

Tato funkce odstraňuje jednořádkové komentáře. Jednořádkový komentář je komentář, který začíná po definované sekvenci znaků a končí znakem konce řádku.

Parametry této funkce jsou:

&str_vystupni – což je reference na výstupní proud, do kterého se bude zapisovat text zbavený komentářů (proud musí být otevřen!)

&str_vstupni – což je reference na vstupní proud, ze kterého se bude číst zdrojový text (proud musí být otevřen!)

pos_konec – což je „ukazatel“ na maximální pozici ve vstupním proudu; pokud se proud dostane až za pozici *pos_konec*, ukončí se výstup do výstupního proudu (vlastně skončí celá operace *DeCommentVR*)

**ret_jednoradkovy* – což je řetězec, jež určuje otevírací sekvenci pro jednořádkový komentář

Funkce pracuje tak, že cyklicky načítá jednotlivé znaky ze vstupního proudu (a přitom testuje jestli se nedostala za pozici *pos_konec*, v tom případě ukončí svou činnost) a pokud narazí na znak, kterým začíná znaková sekvence, tak vrátí do vstupního proudu naposledy přečtený znak a zkusí načíst celý řetězec. Pokud se řetězec shoduje, všechny následující znaky ze vstupního proudu jsou ignorovány až do konce řádku (pomocí metody *Ignore()* třídy ***istream***). Pokud se řetězec neshoduje, vrátí pozici za původně přečtený znak, uloží tento znak do výstupního proudu, a pokračuje v další smyčce.

Další dvě funkce jsou určeny pro odstranění nadbytečných znaků ze zdrojového souboru. Jedná se většinou o znaky mezera, tabulátor a nový řádek.

bool TotalDeleteWhiteZnak(ostream& str_vystupni, istream& str_vstupni, stream::pos_type pos_konec, char prvni, ...);

Tato funkce vymaže všechny bílé znaky. Parametry této funkce jsou:

&str_vystupni – což je reference na výstupní proud, do kterého se bude zapisovat text zbavený komentářů (proud musí být otevřen!)

&str_vstupni – což je reference na vstupní proud, ze kterého se bude číst zdrojový text (proud musí být otevřen!)

pos_konec – což je „ukazatel“ na maximální pozici ve vstupním proudu; pokud se proud dostane až za pozici *pos_konec*, ukončí se výstup do výstupního proudu

*char prvni – j*e první ze seznamu bílých znaků, které se mají ze vstupního proudu odstranit (znaky které se nezapisují do výstupního proudu). Za touto proměnnou

následuje libovolný počet dalších znakových proměnných (bílých znaků), jež se nebudou zapisovat do výstupního proudu. Posledním parametrem této funkce musí být *NULL*.

Funkce si nejprve pomocí standardních maker jazyka c *va_start()* a *va_end()* načte všechny bílé znaky uvedené jako parametr do definované množiny *mn_char_wz* a poté cyklicky prohlíží vstupní proud. Přitom u každého načteného znaku vyhodnocuje zdali se nachází v množině *mn_char_wz*. Pokud ano, tak se nezapíše do výstupního proudu; pokud ne tak se do výstupního proudu zapíše. Funkce samozřejmě také hlídá zdali se nenachází za pozicí *pos_konec*. Výstupem této funkce je vlastně jeden dlouhý neformátovaný řetězec.

bool DeleteWhiteZnak(ostream& str_vystupni, istream& str_vstupni, istream::pos_type pos_konec, char prvni, ...);

Tato funkce má stejné parametry jako funkce předchozí, její funkce se však mírně liší. Jejím úkolem není vymazat všechny bílé znaky, ale pouze ty nadbytečné – například pokud se za sebou opakují 2 či více mezer, funkce nechá jen jednu. Opakování těchto znaků není omezeno na jeden druh bílého znaku (např. 3x mezera), ale maže sekvenci jakýchkoliv bílých znaků (např. ze sekvence nový řádek, tabulátor, mezera uloží pouze první znak – nový řádek). Funkce pracuje tak, že si pamatuje zdali byl minulý znak v množině bílých znaků, a pokud ano, tak se do výstupního proudu nepošle následující znak (v opačném případě samozřejmě ano)

4.3 Funkce jež kvantifikují určité kriterium

Všechny tyto funkce mají jako parametr vstupní proud, který musí být otevřen. Všechny tyto funkce proud nechávají v původním stavu; tj. po ukončení v něm nastaví pozici na místo, jakou měl před zpracováním.

LONG KolikJeMalych(istream& str_vstupni,istream::pos_type pos_konec);

Jedná se o funkci, která spočítá počet všech malých písmen. Tato funkce cyklicky prochází vstupní proud znak po znaku, a pokud je načtený znak malé písmeno, zvýší si své interní počítadlo. Zjištění jestli je znak malý probíhá přes ASCII tabulku – Malá písmena jsou v této tabulce od dekadické hodnoty 97(znak „a“) po hodnotu 122(znak „z“). Parametry této funkce jsou:

&str_vstupni – reference na vstupní proud

pos_konec – „ukazatel“ na maximální pozici ve vstupním proudu; pokud se proud dostane až za pozici *pos_konec*, ukončí se činnost této funkce

LONG KolikJeVelkych(istream& str_vstupni,istream::pos_type pos_konec);

Jedná se o obdobnou funkci, která však místo malých písmen počítá velká písmena. To jsou písmena, jež v ASCII tabulce leží mezi dekadickou hodnotou 65(znak „A“) a hodnotou 90(znak „Z“).

LONG KolikJeCislic(istream& str_vstupni,istream::pos_type pos_konec);

Jedná se o funkci, ketrá počítá počet číslic. Číslice jsou v ASCII tabulce umístěny mezi dekadickou hodnotou 48(znak „0“) a hodnotou 57(znak „9“).

LONG KolikJeZnaku(istream& str_vstupni,istream::pos_type pos_konec);

Tato funkce počítá všechny znaky jež nejsou alfanumerické a nejsou řídící. To jsou znaky, jež v ASCII tabulce leží nad (včetně) dekadickou pozicí 32(znak mezera) a nejsou to číslice ani písmena. (leží na pozicích 32-47, 58-64, 91-96, 123-255).

LONG KolikJeRadku(istream& str_vstupni,istream::pos_type pos_konec);

Tato funkce počítá počet všech řádků. Řádky identifikuje podle toho, že každý řádek je ukončen znakem, jež má v ASCII tabulce dekadickou hodnotu 10, jedná se o tzv. „**LINE FEED**“ – česky posun řádků.

LONG KolikJeVsechZnaku(istream& str_vstupni,istream::pos_type pos_konec);

Tato funkce spočítá počet všech znaků v proudu. Cyklicky načítá znaky ze vstupního proudu až do pozice *pos_konec* a zvyšuje hodnotu proměnné *pocitadlo*, kterou poté vrací jako výsledek. Tato funkce nevrací skutečnou velikost proudu! Je

to dáno tím, že bere sekvenci ‘\n\r‘ (znak nového řádku+návrat na začátek řádky)jako jeden znak, i když ve skutečnosti je reprezentován znaky dvěmi. Pokud přičteme k hodnotě, jež nám vrátí tato funkce, počet řádků, tak bude v případě že *pos_konec* je nastaven na konec proudu výsledek stejný jako vrací funkce *VratDelkuSouboru()* (viz kapitola 4.1).

LONG PocetSlov(istream& str_vstupni, istream::pos_type pos_konec,char ch_prvni, ...);

Funkce *PocetSlov()* dělá přesně to, k čemu ji její název předurčuje. Kromě standardních parametrů *&str_vstupni*, *pos_konec* má také další, minimálně dva parametry znakové (*ch_první, ...*). V těchto parametrech jsou uvedeny všechny oddělovací znaky, které mohou být před (a po) každém slovu. Tyto znaky se nejprve načtou do pomocného pole, a pak se s použitím cyklu *while(str_vstupni.get(c)) {}* načítají postupně znaky ze vstupního proudu. Na začátku cyklu se kromě ověření pozice proudu (zdali už nedosáhla *pos_konec*) pokaždé předpokládá, že přečtený znak není oddělovač. Zdali je se zjišťuje hned v zápětí, kdy se pomocí cyklu *for* porovnává přečtený znak se znaky uloženými v poli oddělovačů. Pokud zjistí, že se čtený znak v tomto poli vyskytuje, nastaví příslušný příznak. Na konci celé smyčky se porovnává typ znaku v tomto cyklu (řetězcový či oddělovací) s typem znaku v minulé smyčce, a pokud jsou různé tak pro přechod *“znak -> oddělovač“* zvýší počítadlo počtu slov *pocitadlo*. Na konci funkce tuto proměnnou vrací jako výsledek.

LONG uoVyskytKlicSlova(istream& str_vstupni,istream::pos_type pos_konec,const char ret_slovo);*
LONG uoVyskytKlicSlov(istream& str_vstupni,istream::pos_type pos_konec,const char ret_slovo, ...);*

Tyto dvě funkce mají předponu *uo*, jež znamená “*un-optimized*“. Měli bychom je použít, pokud z nějakého důvodu nechceme, aby náš program pracoval s množinami. Tyto funkce se v hlavním programu který je používá neosvědčili, protože byly příliš pomalé (místo aby pracovali s množinami a výskyt klíčového slova určovali pomocí nich, pracují se znaky a neustálé přestavování pozice

v proudu je značně časově náročné), a tak jsem je přepsal do následující funkce, která již pracuje s množinami a má také parametry jimiž si můžeme definovat, jaké znaky budou ve funkci oddělovače slov.

LONG VyskytKlicSlov(istream& str_vstupni,istream::pos_type pos_konec,char prvni, ...);

Tato funkce počítá výskyt zadaného klíčového slova. Kromě dříve vysvětlených parametrů *&str_vstupni* a *pos_konec* má ještě několik parametrů, jež určují použité oddělovače mezi slovy a také vlastní klíčová slova. Seznam těchto parametrů začíná znakovou proměnnou *prvni*, což je první z možných oddělovačů. Následující oddělovače jsou odděleny standardně čárkou, a za posledním oddělovačem musí být nulový ukazatel *(NULL)*. Tímto ukazatelem parametry nekončí, ale následují řetězce (oddělené čárkou), jež určují hledaná klíčová slova. Poslední řetězec musí mít hodnotu *AnsiString(NULL).c_str()* (což je řetězec "0"). Funkce pracuje tak, že každé slovo které je ukončeno oddělovačem je porovnáváno se zadanými klíčovými slovy (které jsou uloženy v množině *mn_slova*) a pokud se nachází v této množině klíčových slov, přidá se do multimnožiny *mn_vysledky*. Na konci funkce se po ukončení cyklu jednoduše vrací velikost množiny *mn_vysledky*. (šlo by samozřejmě místo této multimnožiny použít jednoduchou proměnnou *pocitadlo*, avšak takto je funkce připravena na další rozšíření, např. by mohla vracet počty výskytů jednotlivých klíčových slov)

LONG PocetRetezcu(istream& str_vstupni, istream::pos_type pos_konec, const char c_retezcovy, const char c_znakovy);

Tato funkce je určena pro počítání řetězců. Za řetězce považuje sled znaků, který je z obou stran obklopen určitým znak (parametr *c_retezcovy*). Protože *c_retezcovy* může být také hodnota jednoznakové proměnné (která je z obou stran uvozena znakem *c_znakovy*), musí funkce znát i znak *c_znakovy*. Činnost funkce spočívá opět v sekvenčním čtení vstupního proudu; a pokud se narazí na znak, kterým začíná řetězec (*c_retezcovy*), funkce má informaci o tom, že se jedná řetězec a na jeho konci zvýší počítadlo *pocitadlo*, jež vrací jako výsledek.

LONG DelkaVsechRetezcu(istream& str_vstupni, istream::pos_type pos_konec, const char c_retezcovy, const char c_znakovy);

Tato funkce je jakousi variací funkce minulé, rozdíl je v tom, že funkce počítá počet všech znaků které řetězce obsahují. Počítadlo *pocitadlo* se inkrementuje vždy dokud je čtený znak uvnitř řetězce.

bool Cetnost(istream& str_vstupni, istream::pos_type pos_konec, int i_pole[], int n, ...);

Tato funkce slouží pro vyčíslení četnosti slov. Tato slova nevrací, vrací jen seřazený počet výskytů jednotlivých slov. Udělá si přehled o všech slovech, co se v proudu vyskytují, ta si seřadí podle četnosti výskytu, a poté tuto četnost výskytů vrací přes pole *i_pole[]*. Parametry této funkce jsou *&str_vstupni*(vstupní proud), *pos_konec*(maximální pozice v proudu),

int i_pole[] – je pole do kterého ukládá výslednou četnost (ukládá se vzestupně, tj. na pozici *i_pole[0]* je počet výskytů nejčetnějšího slova)

n – počet prvků pole.

Další parametry za parametrem n jsou jednotlivé oddělovače slov (standardně mezera, tabulátor, …), ukončené nulovým ukazatel (NULL).

Funkce si nejprve uloží všechny oddělovače do množiny *mn_char_oddelovace*, a poté pomocí konstrukce *while (FindWord(str_vstupni, pos_konec, retezec, oddel[0], oddel[1], ...) {}* (funkce *FindWord()* - viz kapitola 4.4) vyhledává další slova. Tato slova si ukládá do multimnožiny *multimn_slova*. Po skončení tohoto cyklu si zjišťuje počet výskytů jednotlivých slov (pro prohlížení této multimnožiny využívá iterátorů a cyklu *for*) a tyto jednotlivé počty výskytů každého slova ukládá do multimnožiny *mn_int_vyskytu*.V této multimnožině už je vlastně náš požadovaný výsledek, teď už jen stačí ho převést do pole a ještě k tomu opačně, protože v multimnožině jsou jednotlivé výskyty uspořádány od největšího počtu výskytů po nejmenší.

LONG PocetFunkcnichBloku(istream& str_vstupni, istream::pos_type pos_konec, int uroven, ...);

Tato funkce zjišťuje počet funkčních bloků na jednotlivých úrovních. Z tohoto důvodu se dá použít jen ve strukturovaných jazycích jako je např. C++ a Pascal. V těchto jazycích je zdrojový program členěn do jakési stromové struktury. V základní struktuře je definována většina funkcí a samotný prováděný program (v C++ funkce *main()*)

Každá funkce je z hlediska funkce *PocetFunkcnichBloku()* považována za funkční blok. Přesněji - tato funkce vyhledává výskyt řetězců, jež uvozují a ukončují blok příkazů. Jednotlivé funkční bloky můžou obsahovat další bloky příkazů, čehož se často využívá při podmínkách a cyklech. Kromě již popsaných parametrů *&str_vstupni*, *pos_konec* má tato funkce ještě parametr *uroven*, jež udává na které úrovni chceme funkční bloky počítat (0 – znamená základní úroveň). Další parametry jsou :

1) řetězce které začínají blok (např. "*begin*", "*{*"), oddělené čárkami, ukončeny nulovým ukazatelem NULL,

2) řetězce které ukončují blok (např. "*end*", "*}*") , oddělené čárkami, ukončeny nulovým ukazatelem NULL,

3) jednotlivé oddělovače, které jsou uloženy jako řetězce, oddělené čárkami, ukončeny nulovým ukazatelem NULL.

Funkce postupně "pročítá" vstupní proud, a pokud narazí na řetězec, jímž začíná blok, zvýší si své počítadlo *int_levy_otvirac.* Pokud narazí na řetězec, jímž je blok ukončen, zmenší počítadlo *int_levy_otvirac,* a pokud je poté jeho hodnota rovna zadanému parametru *n*, zvýší hodnotu celočíselné proměnné *vysledek*. Tuto proměnnou poté vrací jako výsledek celé funkce.

Tato funkce je psána na podobném základu jako funkce *FindNextExpression()* (viz. kapitola 4.4].

4.4 Funkce pro základy strukturovaného přístupu

Tyto funkce v případě úspěchu vrací *true*, v případě neúspěchu vrací *false*. Na rozdíl od dosud popsaných funkcí v případě úspěchu mění pozici v proudu; jsou to totiž funkce které mají za úkol vyhledávat. Protože popis jejich funkce by byl nad rámec

tohoto dokumentu (některé z nich aplikují docela složité algoritmy), uvedeme si pouze jejich činnost, pro přesnou funkci viz. zdrojové kódy - *příloha A*.

bool _export FindChar(istream& str_vstupni, istream::pos_type pos_konec, const char c_znak);

Tato funkce vyhledá v proudu první výskyt znaku *c_znak* a nastaví pozici v proudu před něj tak, že další volání funkce *str_vstupni(znak)* obdrží právě hledaný znak

*bool _export FindString(istream& str_vstupni, istream::pos_type pos_konec, const char *retezec);*

Tato funkce vyhledává první výskyt specifického řetězce *retezec* v proudu *str_vstupni*, a v případě že ho najde nastaví pozici v proudu těsně před něj jako u to děla funkce *FindChar()*

bool _export BackFindChar(istream& str_vstupni, istream::pos_type pos_konec, const char c_znak);

Tato funkce činí totéž co první funkce z této kapitoly, avšak neprohledává proud směrem dopředu, ale směrem vzad. Ukazatel v proudu nastaví v případě úspěchu opět před vyhledaný znak

*bool _export BackFindString(istream& str_vstupni, istream::pos_type pos_konec, const char *retezec);*

Tato funkce dělá činnost obdobnou funkci *FindString()*, ale opět místo směrem vpřed vyhledává řetězec ve vstupním proudu směrem vzad. To ovšem neznamená že by hledaný řetězec interpretovala také pozpátku; pokud chceme vyhledat např. řetězec „and“ vyhledá nám skutečně řetězec „and“ a nikoliv „dna“.

*bool _export FindWord(istream& str_vstupni,istream::pos_type pos_konec,char *retezec, char prvni, ...);*

Tato funkce nám ve vstupním proudu vyhledá další slovo. Slovo identifikuje podle oddělovačů z obou stran. Ty jsou dány seznamem parametrů začínajících

parametrem *prvni*, následují další znaky-oddělovače oddělené od sebe čárkou a jejich seznam musí být ukončen nulovým ukazatelem (*NULL*). Tuto funkci používá funkce *CetnostSlov()*.

*bool _export FindNextExpression(istream& str_vstupni, istream::pos_type *pos_zacatek,istream::pos_type *pos_konec, bool *slozeny, ...);*

Tuto funkci jsem chtěl využít v době, kdy jsem se snažil o částečný strukturální přístup. Tato funkce nám ve zdrojovém kódu najde další výraz. Výraz může být buď složený (blok); pak nám funkce v parametru *slozeny* vrací *true*, nebo se může jednat o jednoduchý příkaz (pak *slozeny = false*). Další parametry jsou:

1) řetězce které začínají blok (např. "*begin*", "*{*"), oddělené čárkami, ukončeny nulovým ukazatelem NULL

2) řetězce které ukončují blok (např. "*end*", "*}*") , oddělené čárkami, ukončeny nulovým ukazatelem NULL

3) řetězce které ukončují příkaz (např."*;*") , oddělené čárkami, ukončeny nulovým ukazatelem NULL

Této funkce se dá použít k naprogramování porovnávacího programu, který by kvantifikoval různá kriteria z kapitoly 5.3 v rámci jednotlivých bloků příkazů, a pak by porovnával jednotlivé bloky různých souborů mezi sebou.

5 Organizace dynamických knihoven Funkce.dll

Každá dynamická knihovna jež graficky reprezentuje nějakou funkci, musí mít určité standardní rozhraní, aby mohla být začleněna do hlavního programu skrze rozhraní třídy TFunkce.Určil jsem si, že aby filtr mohl být úspěšně začleněn, musí splňovat následující podmínky:

1) Pokud se má jednat o vstupní funkci, její název MUSÍ začínat znakem “i“ nebo “I“
2) Pokud se má jednat o výstupní funkci, její název MUSÍ začínat znakem “d“ nebo “D“

3) Každá funkce, nehledě na to jestli má být vstupní nebo výstupní musí mít definovány tyto 4 funkce, jejichž prostřednictvím komunikuje s okolím:
   void Konfiguruj(TComponent Owner, AnsiString NazevIni, int intindex);*
   int Proved(AnsiString NazevIni, int index);
   AnsiString CtiVystupniSoubor(AnsiString NazevIni, int index);
   void SetVstupniSoubor(AnsiString NazevIni, int index, AnsiString NazevSouboru);

Každý filtr si musí být schopen zapamatovat své nastavení; proto si každý filtr ukládá údaje o svém nastavení do INI souboru do své vlastní sekce, která se jmenuje stejně jako filtr, avšak bez přípony ".DLL"; za tímto názvem je ještě celočíselná hodnota (*index*), která určuje index filtru. Tato hodnota se předává jako další parametr těchto funkcí. Index je potřeba z toho důvodu, aby mohl být jeden filtr aplikován vícekrát na stejný zdrojový soubor, pokaždé samozřejmě na jiné úrovni. Jako parametr každé z těchto 4 funkcí je tedy vždy jméno INI souboru do kterého se má uložit nastavení – *NazevIni,* a index filtru – *index*.

void Konfiguruj(TComponent Owner, AnsiString NazevIni, int intindex);*
- Je funkce, která zobrazí grafický formulář, v kterém se nastavují všechny parametry požadovaného kriteria. Tento formulář se samozřejmě pro každou funkci liší. Jako parametr má ukazatel *Owner*, což je ukazatel na vlastníka formuláře; protože se tento zobrazuje jako modální, tak s vlastníkem nemůže být prováděna žádná operace dokud se tento formulář nezavře. Typicky se tedy za **Owner* dosazuje ukazatel na *Form*. Při svém zavření rozlišuje, zda-li byla zavřena pomocí tlačítka "OK" nebo "Zrušit". V případě že bylo stisknuto tlačítko OK, tak se změny projeví v INI souboru; v opačném případě se změny zapomenou.

int Proved(AnsiString NazevIni, int index);
- Je funkce, jež provádí požadovanou činnost daného filtru, pracuje tedy se souborem. Tato funkce vrací hodnotu prováděné funkce jako svůj výsledek a také tuto hodnotu uloží do souboru INI pod položkou "Výsledek"

AnsiString CtiVystupniSoubor(AnsiString NazevIni, int index);

- Tato funkce nám vrací řetězec, jež určuje výstupní proud (pro vstupně-výstupní filtry). Pokud není filtr výstupní, vrací hodnotu *AnsiString(NULL)*. Používá se u třídy *TFunkce* při přidání nového potomka do schématu.

void SetVstupniSoubor(AnsiString NazevIni, int index, AnsiString NazevSouboru);
- Tato funkce se používá pro nastavení vstupního souboru do filtru. Toto nastavení se samozřejmě provádí také ve funkci *Konfiguruj()*, zde je to však bez grafického dialogu. Funkce se spouští před provedením každého výpočtu (celkové analýzy vstupního souboru) spíše pro kontrolu, abychom se ujistili že je vyhodnocovací schéma správě sestaveno.

6 Organizace tříd TFunkce, TLabeledEditwFinder

V této kapitole je vysvětlen popis vlastních vizuálních tříd, jež mají za úkol graficky reprezentovat výsledky (třída *TLabelEditwFinder*), nebo jež jsou určeny pro návrh vývojového schématu, podle kterého se vyhodnocují dané zdrojové soubory(třída *TFunkce*). Začněme nejprve druhou z tříd, protože první jmenovaná je na ní závislá:

6.1 Třída TFunkce

Jedná se o třídu, která tvoří grafické rozhraní mezi jednotlivými filtry a vlastním programem. Tato třída umožňuje návrh schematické struktury, pomocí které se posléze vyhodnocuje daný zdrojový soubor.
Konstruktor má tvar:
*TFunkce(TObject *Sender, AnsiString CestaKDLL, AnsiString IniSoubor,int n, int x, int y, bool vyst, bool ShowConfig);*
Jednotlivé parametry mají tento význam:
**Sender* – je vlastník a rodič všech vizuálních komponent *TFunkce*
CestaKDLL – je název DLL daného filtru (včetně cesty)
IniSoubor – je jméno INI souboru, s kterým se má použít daný filtr
x, *y* – od těchto souřadnic se vykreslí vizuální prvky *TFunkce* (vůči rodiči, relativně)
vyst – určuje, zdali je filtr vstupně-výstupní (*true*) nebo jen vstupní (*false*)

ShowConfig – pokud je *true*, vyvolá se bezprostředně po inicializaci grafická konfigurace daného filtru

Třída má dvojí grafickou reprezentaci – podle toho, zda se jedná o vstupní nebo vstupně - výstupní filtr. Všechny grafické prvky jsou vytvořeny dynamicky. Nejprve si popíšeme vstupní filtr, protože vše co o něm platí, platí také u filtru jež je zároveň výstupní.

Vstupní filtr je složen z grafického panelu (komponenta TPanel), na němž jsou umístěny 2 komponenty Tlabel (Jedna pro výpis názvu filtru a druhá pro určení indexu filtru). Tento panel musí mít barvu "*clGradientActiveCaption*", protože pomocí této barvy je posléze "zaměřován" daný filtr (viz kapitola 6.2). Každá komponenta má vlastnost *tag(*32 bit hodnota*)*, která je k dispozici pro programátora, já jsem ji využil pro uložení adresy dané *TFunkce*. Tato adresa je pak opět využita pro zaměřování (viz kapitola 6.2). Panel má definováno *PopupMenu*, které slouží pro změny nastavení, mazání či spuštění daného filtru. Jednotlivé položky *PopupMenu* jsou:

"*Změnit DLL*" – tato volba vyvolá dialog pro změnu použitého filtru

"*Změnit Index*" – změní index filtru

"*Smaž tento blok*" - slouží pro vymazaní filtru (a daného objektu TFunkce)

"*Spusť filtr*" – spustí filtr (zavolá funkci Proved() daného filtru); tato funkce je zde pouze pro ladící účely, protože nespouští předcházející filtry (může se stát, že soubor který je uveden jako vstupní pro daný filtr není definován)

Všechny položky tohoto *PopupMenu* jsou přístupně pouze pokud je povolen editační mód (viz kapitola 7.2); při události *OnPopup* se totiž zjišťuje pomocí metody *ZjistiEditacniMod()* hlavního *Form*u jestli je či není povolen editační mód, a v závislosti na výsledku nastaví vlastnost *Enabled* jednotlivých položek menu.

Každý *Label* má definovanou funkci *OnDblClick*, která se vyvolá po dvojitém poklepu myší; tato událost je naprogramována pro vyvolání grafické konfigurace filtru – funkce *Konfiguruj()*;

Výstupní filtr musí mít oproti vstupnímu filtru prostor pro uložení dalších filtrů, jež pracují se souborem, který je výsledkem "rodičovského" filtru, proto obsahuje ještě další vizuální komponenty – jedná se zéjména o komponentu *TPanel*(objekt

PanelUloziste), která slouží pro vkládání a zoobrazování potomků (objektů typu *TFunkce*), a o z hlediska funkce programu nevýznamnou komponentu *TShape* (objekt *ShapeSipka*), která jen vizuálně spojuje oba panely, aby bylo vidět že patří stejné *TFunkci*.
PanelUloziste má opět definováno *PopupMenu*, které má tentokrát jenom jednu položku:
"*Přidej další funkci*" – slouží pro přidání další funkce (a filtru); tato nová funkce bude pracovat se souborem, který produkuje daný výstupní filtr
Tato položka je opět přístupná pouze pokud je zapnut editační mód.

Definované filtry vykonávají činnost jež je jasně dána jejich názvem; zpravidla pouze volají funkci s obdobným názvem z knihovny *definice.dll*, která má obdobný název + přizpůsobují jejich parametry. Dostupné filtry jsou:

"**Basic.dll**" – musí být uveden v každém schématu jako první, jedná se o speciální vstupně-výstupní filtr, který pouze kopíruje vstupní soubor do výstupního
"**deComment1R.dll**" – volaná funkce : *deComment1R()*
"**deCommentVR.dll**" – volaná funkce : *deCommentVR()*
"**DeleteWhiteZnak.dll**" – volaná funkce : *DeleteWhiteZnak()*
"**DeleteWZTotal.dll**" – volaná funkce : *TotalDeleteWhiteZnak()*
"**iCetnost.dll**" – volaná funkce : *Cetnost()*
"**iDelkaVsechRetezcu.dll**" – volaná funkce : *DelkaVsechRetezcu()*
"**iKolikJeCislic.dll**" – volaná funkce : *KolikJeCislic()*
"**iKolikJeMalych.dll**" – volaná funkce : *KolikJeMalych()*
"**iKolikJeRadku.dll**" – volaná funkce : *KolikJeRadku()*
"**iKolikJeVelkych.dll**" – volaná funkce : *KolikJeVelkych()*
"**iKolikJeVsechZnaku.dll**"–volaná funkce : *KolikJeVsechZnaku()*
"**iKolikJeZnaku.dll**" – volaná funkce : *KolikJeZnaku()*
"**iPocetFunkcnichBloku.dll**" – volaná funkce : *PocetFunkcnichBloku()*
"**iPocetKlicovychSlov.dll**" – volaná funkce : *VyskytKlicSlov()*
"**iPocetRetezcu.dll**" – volaná funkce :*PocetRetezcu()*
"**iPocetVsechSlov.dll**" – volaná funkce : *PocetVsechSlov()*

6.2 Třída TLabeledEditwFinder

V této třídě jsou dvě grafické komponenty – první *LabeledEdit* je určena pro zobrazení výsledku filtru, druhá, *Edit* je určena pro nastavení maximální povolené tolerance, pro kterou se ještě soubor považuje za podobný vůči souborům, které jsou uloženy v databázi (viz dále), nebo je použit pro nastavení váhy daného filtru (viz dále).

Komponenta *LabeledEdit* se skládá ze dvou relativně nezávislých částí – první je klasický *TEdit*, v tomto *TEdit*u zobrazujeme výsledky filtrů. Tyto výsledky se čtou z INI souboru (ten je pro každý *TLabeledEditwFinder* definován jako soukromá položka). Druhá část je klasický *TLabel*, který slouží jako popis onoho *Editu*. Pro tento *Label* je definováno další *PopupMenu* s jedinou položkou "*Zaměřit zdrojovou Funkci*". Po kliknutí na tuto volbu se zobrazí malý červený kruh (objekt typu *TShape*), který má povoleno posouvání po návrhovém formuláři, a je určen pro spárování filtru s objektem *TLabeledEditwFinder*. Spárování se provede tak, že toto kolečko umístíme nad panel daného filtru a upustíme ho zde. Poté se nám změní vlastnost *Caption Labelu* na jméno daného filtru a indexu.

Další komponenta *Edit* je určena pro nastavování relativní tolerance/váhy filtru. Je možno ji také ovládat pomocí komponenty *UpDown* která je k ní přiřazena.

Tolerance se nastavuje v rozmezí od -999 % po + 999%. Tolerance je brána relativně vůči výsledku (zobrazenému komponentou *LabeledEdit*), tj. pokud nastavíme toleranci např. 20%, v databázi se za podobné (z hlediska jediného filtru) berou hodnoty, které jsou v rozmezí *výsledek ± 0,2*výsledek* (hodnota filtru pro daný vstupní soubor).

Váha se nastavuje pomocí stejné komponenty jako tolerance, pro jeho nastavování platí tedy stejná pravidla, avšak zde nastavená čísla jsou brána v absolutní hodnotě. Pokud je nastaven procentuální mód vyhodnocování, číslo uvedené v této komponentě se reprezentuje jako tolerance, pokud je nastaven váhový mód, reprezentuje se toto číslo jako váha. Čím větší váhu nastavíme, tím je větší váha výsledku daného filtru. V tomto módu se za podobné považují soubory, pro které jsou jednotlivé výsledky všech filtrů u sebe v prostoru nejblíže (resp. blíže než maximální specifikovaná hodnota). Soubor výsledků se totiž bere jako *r*-rozměrný Eukleidovský prostor (*r* je počet výsledků), v kterém se vzdálenost dvou bodů spočítá pomocí vzorce:

$$\rho(X,Y) = \sqrt{(y_1 - x_1)^2 + (y_2 - x_2)^2 + \cdots + (y_r - x_r)^2}$$

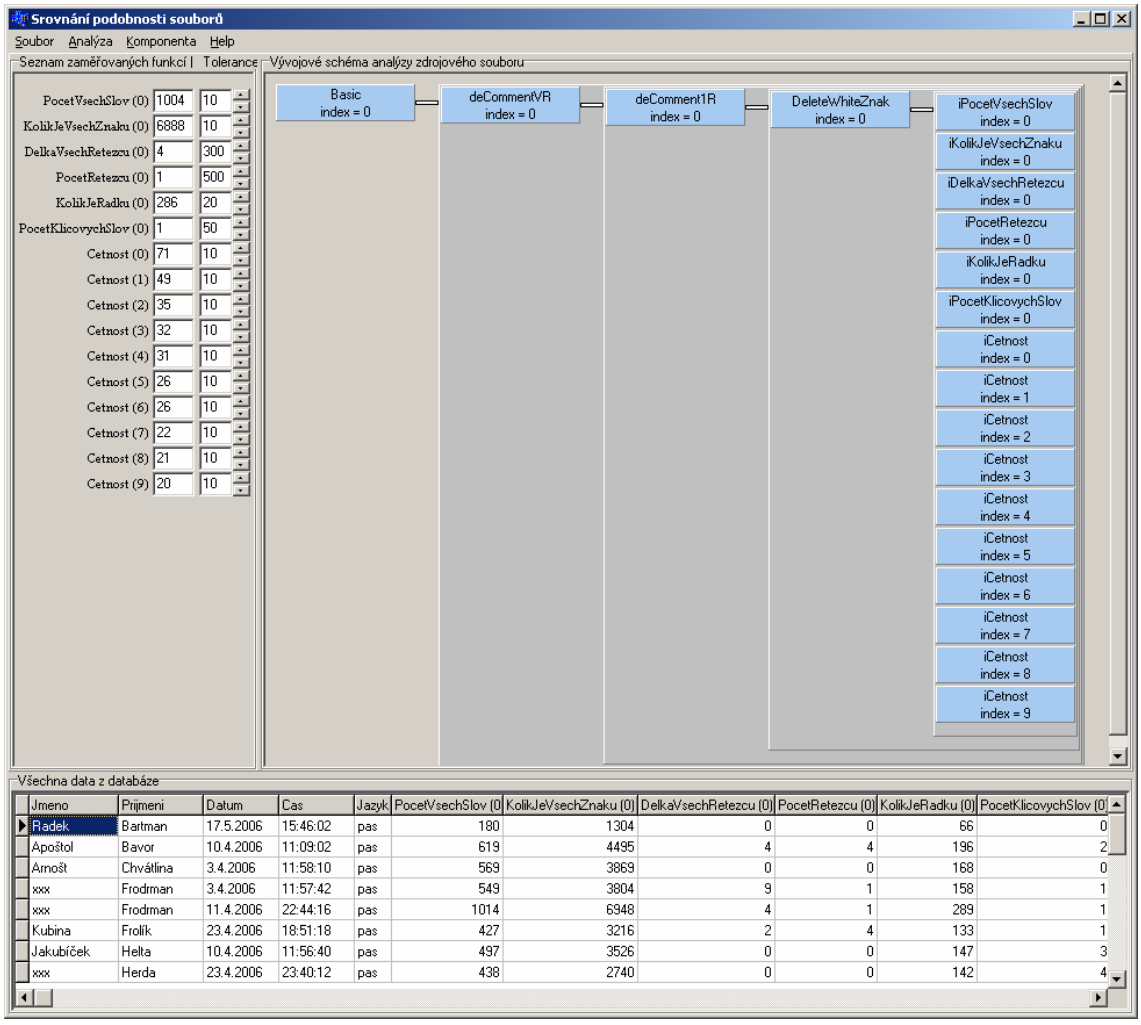
Nastavení jednotlivých vah má ten efekt, že před umocněním výrazu ($y_m - x_m$) se tento výraz ještě vynásobí zadanou vahou.

7 Program – vzhled, ovládání, použité komponenty

Program je vyvíjen jako klasická WIN32 aplikace; pro jeho ovládání je tedy určena myš a klávesnice. Vzhled programu včetně jednotlivých menu nevybočuje ze zažitých zvyklostí z operačního systému WINDOWS, takže jeho ovládání by po prvotním zorientování se nemělo činit potíže. Zejména část určená pro porovnávání souborů je z hlediska ovládání velice přívětivá.

7.1 Pracovní plocha formuláře

Pracovní plocha formuláře je rozdělena do tří částí.



Vpravo nahoře je umístěn *GroupBox* pro vlastní návrh vyhodnocovacího schématu, v tomto *GroupBoxu* je zobrazen zpravidla jeden základní objekt třídy *TFunkce*, který je výstupní. Tento základní objekt používá knihovnu (filtr) "Basic.dll". Do výstupního panelu tohoto základního objektu můžeme poté umístit (v případě že je povolen editační mód) libovolný počet dalších objektů třídy *TFunkce*.

Vlevo nahoře je umístěn další *GroupBox*, tentokrát pro zobrazení výsledků jednotlivých filtrů. Do tohoto *GroupBoxu* můžeme přidávat jednotlivé objekty *TLabeleditwFinder* pomocí položky menu *Komponenta/Přidat zobrazovač výsledků*.

Ve spodní části je *GroupBox*, který obsahuje pouze jednu komponentu – *DBGrid*, která slouží pro prohlížení záznamů z databáze. Po spuštění programu jsou v ní vidět všechny záznamy, po zapsání daného výsledku do databáze se v případě nalezení souborů které jsou podobné zobrazí jen ony.

7.2 Typické ovládání programu

Položkou *Nový* z nabídky *Soubor* si vytvoříme nový konfigurační soubor; zobrazí se jeden filtr – *Basic* a jeho konfigurace. Poté pomocí voleb popupmenu přidáváme a konfigurujeme jednotlivé filtry. Po tom co máme hotové vyhodnocovací schéma umístíme do formuláře prvky pro zobrazování výsledků (objekty TLabeledEditwFinder). Po kliknutí na *Komponenta/Přidat* zobrazovač se nám vždy přidá jeden tento objekt. Po tom co je všechny zacílíme vypneme editační mód (položka *Soubor/Vypnout editační mód*), nastavíme požadovaný vyhodnocovací mód (položka *Analýza/Mód vyhodnocení/*požadovaná volba; pokud zvolíme váhový mód, musíme ještě definovat max. vzdálenost – *Analýza/Definuj maximální vzdálenos v prostoru*) a uložíme konfiguraci (*Soubor/Uložení konfigurace*). Nyní můžeme analyzovat jednotlivé soubory, stačí zvolit volbu *Soubor/Změna vstupního souboru*. Po stisknutí na *Analýza/Spuštění analýzy* se nám vyplní jednotlivé Edity aktuálními výsledky. Vedle Editů jsou boxy s tolerancemi/váhami, které ještě musíme upravit. Pro uložení výsledků do databáze stiskneme *Analýza/Zapsání výsledků do databáze* (vyhodnocení a zároveň zápis do databáze lze také uskutečnit zkráceně pomocí volby *Analýza/Spustění & zapsání*). Po vyplnění identifikačních údajů (jméno, příjmení) nám program automaticky vyhledá podobné soubory podle nastavení tolerancí/vah jednotlivých filtrů.

Pokud nějaké nalezne, vypíše o tom hlášku na obrazovku a v komponentě DBGrid se zobrazí jen podobné záznamy (v opačném případě zobrazí všechny záznamy).

8 Ověření funkčnosti

Ověření funkčnosti jsem prováděl na vzorku cca 25 reálných souborů – zdrojových kódů jazyka Delphi, které řešili stejnou úlohu. Některé byly vyřešené, jiné byly pouze rozpracované. Ověření bylo provedeno pomocí procentuálního módu. Soubory jsem nejprve přidal do databáze (s nastavenou nulovou tolerancí), poté jsem zjistil jaké položky se liší nejvíce a nejméně a podle toho jsem upravil nastavení jednotlivých filtrů. Poté jsem všechny soubory z databáze odstranil, a zkoušel jsem je přidávat postupně do databáze. Mezi soubory nebyl žádný vyloženě stejný soubor, (počet znaků, řádků, …) a tak jsem si jeden takovýto vytvořil. Zkopíroval jsem jeden zdrojový soubor a změnil jsem v něm názvy proměnných, odstranil jsem většinu formátování, a přidal jsem k němu jednu jednoduchou funkci. Výsledky této analýzy byly takové, že mnou upravený zdrojový soubor program rozpoznal jako podobný souboru originálnímu, avšak jako podobný označil i dva soubory, jež podobné nebyly. Tyto dva soubory měli stejný počet funkcí, přibližně stejnou délku, ale implementace byla rozdílná. Pokud by bylo definováno více podmínek (např. klíč. slov), program by mohl mít i lepší úspěšnost.

Zajímavé výsledky by nám mohl dát váhový mód, ale nepodařilo se mi u něj v dostupném čase nastavit váhy jednotlivých filtrů tak, aby byly výsledky tohoto algoritmu použitelné.

Závěr

Cílem této práce bylo navrhnout různá kriteria pro hodnocení textových souborů a jejich aplikace na reálný zdroj dat. Po vyzkoušení na vzorku 25 reálných souborů můžeme konstatovat, že navržená kriteria jsou velice dobře použitelná jako zdroj dat, který je třeba pečlivě vyhodnotit. Ukázková aplikace se o to pokouší, a její úspěšnost lze hodnotit jako velmi dobrou; pokud je soubor vyložená kopie, program to zjistí se 100 % úspěšností, pokud se v souboru zamění některé řetězce (např. názvy proměnných), vyhodnocovací algoritmus tento soubor opět vyhodnotí s velkou pravděpodobností jako podobný. Pokud by byly citlivěji nastavená jednotlivá kriteria, je pravděpodobné, že by se úspěšnost správného vyhodnocení podobnosti ještě zvýšila.

Pokud by se pro každý možný problém použila jiná konfigurace vyhodnocovacího schématu, která by danému problému byla „ušita na míru“, úspěšnost vyhodnocení podobnosti souborů by se pohybovala nad hranicí 90 %. Pro potvrzení této domněnky by bylo ale třeba použít větší vzorek souborů, nicméně některá kriteria dávají velice přesvědčivé výsledky i v případě obecného zadání vyhodnocovacího schématu. Jedná se zejména o kriteria “Četnost slov”, “Počet klíčových slov” a “Počet funkčních bloků”.

Programová část je psána otevřeným způsobem, takže je snadné rozšíření možností aplikace o nové filtry a kriteria. Pokud by jsme chtěli pro zlepšení přesnosti navrhnout filtry orientované jen na jeden programovací jazyk, je to s minimálními úpravami programu možné. Mohli bychom dokonce napsat pro specifický programovací jazyk obdobu *parseru*, který by díky strukturovanému přístupu a zaměření jen na jeden programovací jazyk mohl poskytovat přesnější výsledky než stávající aplikace.

Při psaní zdrojového kódu k této aplikaci nejdéle trvalo samotné ladění. Samotné algoritmy sice nejsou složité, prostor pro vznik chyb zde však díky komplexnosti a provázanosti jednotlivých programových částí na sebe byl velký. Podstatnou částí programu je vizuální prostředí, které je intuitivní a přehledné. Pokud by nebylo předmětem této práce (resp. by bylo již hotové), vzniknul by větší prostor jít více do hloubky problému a aplikovat některé algoritmy strukturovaného přístupu; s největší pravděbodobností by však navrhnuté filtry nebyly tak obecné jako jsou nyní.

Seznam použité literatury

[1] Herout P.: Učebnice jazyka C, Kopp nakladatelství, České Budějovice, 2001

[2] Herout P.: Učebnice jazyka C 2. díl, Kopp nakladatelství, České Budějovice, 2000

[3] Virius M.: Od C k C++, Kopp nakladatelství, České Budějovice, 2002

[4] Stephen P.: Mistrovství v C++, Computer Press, Brno, 2004

[5] Matoušek D.: C++ Builder vývojové prostředí 1. díl, BEN nakladatelství, Praha 2002

[6] Matoušek D.: C++ Builder vývojové prostředí 2. díl, BEN nakladatelství, Praha 2003

[7] Matoušek D.: C++ Builder vývojové prostředí 3. díl, BEN nakladatelství, Praha 2003

[8] C++ reference: http://www.cplusplus.com/ref/indexr.html

[9] C / C++ reference: http://www.cppreference.com/index.html

[10] Dostál R.: Asociativní pole v C++, http://www.builder.cz/art/cpp/asocspp.html

[11] Dostál R.: Množina v C++, http://www.builder.cz/art/cpp/mnozinacpp.html

[12] SQL Tutorial http://www.w3schools.com/sql/default.asp

[13] Wikipedia SQL, http://en.wikipedia.org/wiki/SQL

[14] FindNextFile http://www.cs.rpi.edu/courses/fall01/os/FindNextFile.html

[15] Dependency Walker http://dependencywalker.com/

[16] Eukleidovský prostor http://www.ft.utb.cz/czech/um/ostravsk/skripta/KSkap1.htm

Seznam příloh

Příloha A - CD se všemi zdrojovými kódy a bakalářskou prací v elektronické podobě

Použité technické prostředky

Hardware:

Základní deska ASUS A7N8X-E DELUXE

Procesor AMD XP 1900+

1 GB RAM

Grafická karta MSI NX-6800

Pevný disk 80 Gb SEAGATE Barracuda 7200.7

Software:

Microsoft Windows XP Professional verze 2002, Build 2600 - Service Pack 2

Microsoft Office Professional Edition 2003

Borland C++ Builder Enterprise Suite version 6.0 (Build 10.157)

Pro správný chod aplikace je nutné nainstalovat knihovnu bdeinst.dll příkazem ***regsvr32.exe bdeinst.dll***