**Technická univerzita v Liberci**

Fakulta Mechatroniky a mezioborových inženýrských studií

# Bakalářská práce

**Liberec 2009** **Krček Miroslav**

# Technická univerzita v Liberci

Fakulta Mechatroniky a mezioborových inženýrských studií

Studijní program: **B 2612 – Elektrotechnika a informatika**

Studijní obor: **Informatika a logistika**


# Flexibilní vstupní rozhraní pro program Flow123d

# Flexible input interface for program Flow123d

**Bakalářská práce**


Autor: **Miroslav Krček**

Vedoucí bakalářské práce: **Mgr. Jan Březina, Ph.D.**

Ústav nových technologií a aplikované informatiky


V Liberci 30.4.2009

Zde bude list, který sem dostal jako zadání

# Prohlášení

Byl(a) jsem seznámen(a) 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 BP a prohlašuji, že **s o u h l a s í m** s případným užitím mé bakalářské práce (prodej, zapůjčení apod.).

Jsem si vědom(a) toho, 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(a) samostatně s použitím uvedené literatury a na základě konzultací s vedoucím diplomové práce.

Datum:

Podpis:

# Poděkování

Na tomto místě bych chtěl poděkovat Mgr. Janu Březinovy, Ph.D., který byl nejen zadavatelem této práce, ale také cenný rádce a průvodce, bez jehož ochoty a vstřícnosti by se jen těžko tato práce realizovala.

## Abstrakt

Cílem této bakalářské práce byl návrh flexibilního formátu vstupních dat pro program Flow123d a implementace funkcí pro čtení tohoto formátu v programovém jazyce C. Formát vstupních dat měl být snadno rozšiřitelný, pozičně nezávislý a tím umožňující kompatibilitu i s budoucími verzemi programu Flow123d. Navržený formát obsahuje spolu s daty i popis jejich významu pomocí klíčových slov. Čtecí funkce pak umožňují čtení hodnoty pro zadané klíčové slovo. Tento přístup významně oslabuje vazbu mezi vstupními daty a způsobem jejich čtení. Přiřazení klíčových slov je zadáno jednou pro celý soubor dat, takže zbytečně nezvětšuje velikost vstupních souborů. Díky navrženému rozhraní se ušetří mnoho času, stráveného pracným přepracováním nekompatibilních datových souborů.

## Abstract

The purpose of this bachelor work is a design of a flexible input data format for the program Flow 123d and an implementation of functions for reading of this format in the language C. The input data format should be extensible, positional independent, and thereby easily compatible even with future versions of the program Flow 123d. The proposed format contains the data together with a description of their meaning using a concept of key words. Then, the reading functions allow access to the values in terms of key words. This approach significantly relaxes the coupling between the input data and the way of their reading. The key words are assigned to the whole bunch of the input values so it does not magnify the size of input files. Thanks to the proposed interface we safe a lot of time spend in recasting incompatible data files.

# Obsah

# 1 Úvod

Cílem této bakalářské práce je poznat a naučit se programovací jazyk C, naučit se pracovat s **ukazateli** (3.1), umět využívat všech možností **řetězců** (3.2) a **struktur** (3.3), které tento programovací jazyk nabízí a s jejich pomocí naimplementovat rozhraní pro čtení formátových tabulek do programu **Flow 123d** (1.1). Důvodem vzniku této implementace je problém se **vstupními soubory** (2.2), kterými jsou tabulky čísel, kde význam čísel se může změnit verzí programu, tímto se vstupní soubor stane "nečitelný".

Právě proto bylo navrženo toto rozhraní, které využívá služeb následujících funkcí. Ke každé tabulce se připojí její formát a čtení bude provádět pomocí tohoto formátu. Formát vždy obsahuje určitý počet klíčových slov, popřípadě definici jejich typu, a podle pořadí umístění těchto klíčových slov se pomocí funkce **Parse_table_format** (5.1.3) vytvoří spojový seznam v definované struktuře. Spojový seznam bude tak dlouhý, kolik je počet klíčových slov ve formátu. Poté funkce **read_line** (5.1.1) v tabulce nalezne hodnotu, která se vyskytuje na totožném pořadí na řádku, jako příslušné klíčové slovo a vytvoří se ukazatel na nový spojový seznam, který bude naplněný klíčovými slovy, popřípadě jejich typy a hodnotami k nim náležejícími. Podle požadavků programátora, které upřesní v jedné z posledních funkcí, které rozhraní může nabídnout, dané podle specifických typů hodnot nebo v podobě stringu. Jsou jimi funkce **TableGetString** (5.3)**, TabkegetInt** a **TabkegetReal**. Toto umí vytvořená implementace. Dokáže nezávisle najít a rozpoznat námi požadovanou hodnotu a dokáže ji vrátit v potřebné formě a pořadí. Jak při záměně pořadí hodnot tak i při změně počtu hodnot. Otevírá se tím možnost číst starší data a účelně je využít. Účelů, ke kterým se daná implementace dá využít je mnoho, je to vlastně prostředek pro úpravu nově vzniklého formátu pro elektronickou výměnu dat a tudíž všude, kde se rozhodnou tento formát použít se tato implementace dá použít.

Důvodem výběru této bakalářské práce byl zájem o seznámení s programovacím jazykem C a jeho komponent, který za tříletou dobu studia nebyl náplní žádného z akreditovaných předmětů.

## 1.1 Flow 123d

Flow 123d je program pro modelování proudění kapaliny puklinovým prostředím založený na principu diskrétních stochastických puklinových sítí. Program poskytuje výsledky v podobě textových souborů čitelných programem GMSH, který je následně používán pro grafické zobrazení. Probíhají v něm výpočty proudového pole zaměřující se na podzemní skalní masivy a ty jsou charakterizovány výskytem horninových poruch puklin. Proudění podzemní vody a transport kontaminantů obsažených v podzemní vodě takovýmito skalními masivy se děje téměř výhradně v této síti puklin. Z tohoto důvodu nelze tyto procesy popsat standardními modely, které předpokládají porézní médium. Problematickým místem většiny modelů tohoto typu je diskretizace modelové puklinové sítě složené ze 2D útvarů typu disků či polygonů do formy sítě. Automatické generování těchto sítí je značně problematické a vede na sítě s velmi nízkou regularitou. Ruční opravy již vytvořené sítě či dokonce její generování je potom prakticky neproveditelné, díky velkému počtu puklin v oblasti. To ve většině případů vede na značné omezení na velikost oblasti, počet puklin a jejich hustotu v oblasti. V těchto modelech se prostředí skalního masivu charakterizuje sítí puklin aproximovaných plošnými eliptickými disky, jejichž frekvence, velikost, přiřaditelné rozevření a orientace jsou statisticky charakterizovány z geologických měření“([1] str.35).

## 1.2 charakteristika programu GMSH

Systém GMSH je obecný pre- a post- procesor pro matematické modely založené na metodách diskretizujících prostorové oblasti do podoby sítí. GMSH je freewarová aplikace šířena pod licencí GNU-GPL, což zamená, že může být zdarma používána pro komerční i nekomerční aplikace, pouze při zachování copyrightu původních autorů. Program je možno též upravovat a dále vyvíjet, při zachování původní licence a copyrightu. Autory programu jsou Christophe Geuzaine a Jean-Francois Remacle, domovská stránka programu je http://www.geuz.org/gmsh/ [3].

Důvody, proč byl GMSH vybrán jako vizualizační prostředek pro puklinové modely jsou následující:

- Umožňuje zobrazovat sítě tvořené plošnými útvary umístěnými v prostoru.
- Umožňuje interaktivní zobrazení (posun/rotace/zoom pohledu).
- Umožňuje zobrazení rozložení skalárních a vektorových veličin na takovýchto sítích.
- Formát vstupních dat je jednoduchý a dobře dokumentovaný.

# 2 Formát dat pro Flow123d

## 2.1 Jiné obecné datové formáty

Vstupní soubor není vlastně nic jiného než prostředek pro elektronickou výměnu dat (EDI - zkratka anglického originálu Electronic Data Interchange) je to výměna strukturovaných zpráv mezi počítači, respektive mezi počítačovými aplikacemi. Data jsou strukturována podle předem dohodnutých standardů a ve formě zpráv následně elektronicky automaticky přenášeny bez přispění člověka. Běžně se jako EDI rozumí specifické metody výměny zpráv, jež byly dohodnuty na úrovni národních nebo mezinárodních standardizačních společenství pro přenosy dat.

Na začátku projektu se rozhodovalo mezi třemi standardizovanými formáty. První z nich byl odlehčený formát pro výměnu dat **JSON**. Je jednoduše čitelný i zapisovatelný člověkem a snadno analyzovatelný i generovatelný strojově. Je založen na podmnožině Programovacího jazyka JavaScript. JSON je textový, na jazyce zcela nezávislý formát, využívající však konvence dobře známé programátorům jazyků rodiny C (C, C++, C#, Java, JavaScript, Perl, Python a dalších). Díky tomu je JSON pro výměnu dat opravdu ideálním jazykem. Avšak bohužel díky své jednoduchosti nebyl schopný využít veškeré potřebné parametry. Dalším velmi známým značkovacím jazykem je jazyk **XML**(eXtensible Markup Language, česky rozšiřitelný značkovací jazyk). Umožňuje snadné vytváření konkrétních značkovacích jazyků pro různé účely a široké spektrum různých typů dat. Jazyk je určen především pro výměnu dat mezi aplikacemi a pro publikování dokumentů. Jazyk umožňuje popsat strukturu dokumentu z hlediska věcného obsahu jednotlivých částí, nezabývá se sám o sobě vzhledem dokumentu nebo jeho částí. Prezentace dokumentu (vzhled) se potom definuje pomocí kaskádových stylů. Další možností je pomocí různých stylů provést transformaci do jiného typu dokumentu, nebo do jiné struktury XML. Avšak tento jazyk vnucuje uživateli mnoho zbytečných informací, což bylo důvodem k zamítnutí i této varianty. Nakonec se uvažovalo o jazyku **YAML**(nástroj pro ladění kódu webových stránek), který však zase neprošel z důvodu jeho složitosti a proto nezbývalo nic jiného než vytvořit si vlastní formát pro výměnu dat.

## 2.2 Specifikace formatu dat pro Flow123d

Tento soubor se skládá z tabulek, které obsahují data. Dále pak obsahuje množství komentářů, které jsou uvozovány znakem “#”. Tyto komentáře jsou v obsahu umístěné ve všech představitelných částí, a je jej potřeba odstranit pomocí naší implementace, protože pro naše účely jsou zbytečné, a spíše naopak by do výsledku mohli vznést chaos. Řádek v souboru je potřeba spojit s následujícím do jednoho vstupního řádku v případě, že daný řádek končí symbolem “\\”. Jako příklad pro objasnění situace můžeme použít tuto ukázkovou tabulku.

```
$sekce_XY [<format>] # zacatek sekce s volitelnym zadanim formatu dat
# pocet radku
2
# tady jsou data
1 2 \\ # prvn__ _r_adek dat
3 4 # st_ale je_st_e prvn__ _r_adek
"a b" # jedna hodnota
$Endsekce_XY # konec sekce
```

Zde hodnoty 1 2 uzavírá právě daný znak “\\”, tím pádem se daný řádek spojí s následujícím a vstupní řádek bude mít hodnoty 1 2 3 4. V obou případech, jak u znaku komentáře, tak u zpětných lomítek se tyto znaky do výsledku nezapíší.

## 2.3 tabulky

Ukázková tabulka nahoře začíná znakem “$“, po kterém následuje název tabulky tak jak je vygeneruje GMSH, dále následuje mezera za kterou je formát (viz kap. 2.1.1). Hned na dalším řádku by měla následovat hodnota “size”, která vyjadřuje počet řádků tabulky. Tato hodnota je spíše informativního typu, implementace ji sice vyseparuje a dá možnost uživateli ji použít, ale ve strukturách implementace je ošetřeno přepočítání skutečného počtu řádků tabulky. Dále následují hodnoty, na každém řádku leží přesně tolik hodnot kolik je ve formátu vyseparovaných klíčů. A konec tabulky je ošetřen opět znakem “$“, po kterém následuje název tabulky shodný s názvem na začátku.

### 2.3.1 Formát

Formát obsahuje potřebné informace o podobě, pozici a chování hodnot, které se nacházejí dále v tabulce. Formát je posloupnost klíčů (identifikátorů) volitelně následovaných specifikací typu a to vše je na jednom řádku. Typ může být Int, Real, String. Uvádění skalárních typů Int, Real, String je nepovinné, umožňují pouze lepší kontrolu kompatibility vstupních dat a programu.

Syntaxe formátu je následující:
<name1>[:<type1>] <name2>[:type2] ...

Tato implementace je dále připravena na rozšíření o specifický typ dat **Enum** (<val1>[=<nick1>] <val2>[=<nick2>]…), který umožňuje zadávat pouze celočíselné hodnoty <valN> případně jejich řetězcové ekvivalenty <nickN>. O typ **Array** (<key>[<type>]) znamená, že od dané pozice následuje proměnný počet hodnot se skalárním typem <type>, přičemž počet hodnot na konkrétním řádku je dán aktuální hodnotou klíče <key>. Tento klíč musí ve formátu(tedy i na řádcích dat) předcházet použití typu Array a musí být typu Int nebo bez uvedeného typu. A o poslední typ **Var** (<enum_key>; <enum_val1> <format1>; <enum_val2> <format2>; ....) znamená, že dále bude čteno podle toho formátu <formatN>, pro který se <enum_valN> shoduje s aktuální hodnotou klíče enum_key, který musí být typu Enum. Hodnoty <enum_valN> se musí shodovat (včetně pořadí) s hodnotami klíče <enum_key>.

## 2.4 Příklad vstupního souboru

Nyní si zde namodeluji ukázkový případ vstupního souboru s názvem test_table, se všemi ošetřenými událostmi, které u vstupního souboru mohou nastat. A za pomoci prvků komentáře, které se běžně ve vstupních souborech vyskytují, stručně objasním problematiku.

```
Zde se vyskytují poznámky \\
Které vygeneruje GMSH\\
Implementace tuto část textu díky lomítkům čte stále jako první řádek
#Vyskytuje se zde i mnoho takto uvozovaných komentářů
#Ve své podstatě však tato část souboru pro nás nehraje žádnou roli.
#Sice se tato část textu přečte a patřičně upraví, ale kromě načtení čísla řádku, z důvodu
#dohledatelnosti umístění případného problému, se jinak nic
```

```
#jiného neprovede a finální podoba řádku se zahodí a odblokuje


$Tabulka1 el:int retezec cisla:int body vrcholy
#Toto je začátek tabulky1, za kterou se vyskytuje formát
#Pokud v rozhraní zadáme, že chceme hledat hodnoty v tabulce1, tak
#právě toto je řádek, který naše implementace hledá.
#Uloží si obsah formátu do struktury, a začne přeskakovat tyto komentáře, dokud nenarazí na
#hodnotu size, která se vyskytuje na dalším řádku.
6
1 prvni 911 123 161718 #zde již začíná tabulka hodnot
2 druhy 912 456 131415 #načítají se řádky tabulky
3 treti 913 789 101112 # a číslo řádku se porovnává s hodnotou size
4 ctvrty 914 101112 789
#tento komentář jakožto prázdný řádek se automaticky přeskočí a nezapočítá jako řádek tabulky
5 paty 915 131415 456
6 sesty 916 161718 123 #řádek číslo 6
$End # konec tabulky1
$Tabulka2 pismena cislo  #Toto je začátek tabulky2, za kterou se vyskytuje formát

3

a 1
b 2
c 3
$End # konec tabulky2
# konec souboru
```

Tento zdroj hodnot, i když se to na první pohled nemusí tak zdát je naprosto reálný zdroj dat, který později otestujeme a vyzkoušíme, zda-li implementace funguje, tak jak má.

# 3 Popis interface

Řekněme, že programátor bude potřebovat ze vstupního souboru koordinovat některé hodnoty do jiného pořadí, než jak ve vstupním souboru zrovna jsou. Na příkladu, u kterého využijeme ukázkový vstupní soubor test_table(kap 2.2), si ukážeme, jak toho pomocí implementace docílí. Zaprvé musí zavolat knihovny s funkcemi implementace, v našem případě na řádku 1 "input_interface.h". Poté musí vytvořit řetězce FileName a TableName do kterých přiřadí příslušný název vstupního soboru a název tabulky, ze které chce čerpat data (řádek 5,6). Pro zjištění počtu řádků tabulky musí nadefinovat proměnnou datového typu int, v našem případě s názvem size. A v případě, že chceme data číst pomocí for cyklu měla by se nadefinovat i pomocná proměná, v našem případě řádek 7. Dále se nadefinuje struktura typu TableCtx (8) a proměnné do kterých se budou zapisovat hodnoty nalezené v tabulce (9).

Zavolá se TableNew (viz kap 5.1) pro otevření soboru(11), nalezení/nenalezení tabulky, naplní se ve struktuře položka formátu. Funkce použije námi zadané jméno souboru a název tabulky a vrátí nám naalokovanou strukturu s načteným formátem a ve struktuře vytvoří spojový seznam z hodnot formátu. Vrátí nám hodnotu size, která nám informativně slouží pro zjištění velikosti tabulky. Řádky 12,15,17 a 19 nám zde slouží pouze pro přehlednější vypsání dosažených výsledků.

TableReadLine (viz kap 5.2) do spojového seznamu pomocí klíčů vyseparovaných z formátu uloží hodnoty z aktuálního řádku tabulky a pomocí for cyklu se postupně funkce opakuje až do přečtení všech řádků tabulky (13,14)

Funkce TableGetString (viz kap 5.3), které se jako první parametr zadá název struktury, kterou používáme, jako druhý název hodnoty kterou hledáme (musí být v uvozovkách jako řetězec) a třetí parametr představuje pointer, který bude vracet požadované hodnoty, které programátor/uživatel požadoval (16,18). Nakonec funkce TableDestroy na řádku 21 odebere všem strukturám přidělenou paměť.

```
1 #include "input_interface.h"
2
3
4 int main(void){
5    char FileName[MAX_LINE]="test_table";
6    char TableName[MAX_LINE]="Tabulka2";
7    int size,k;
8    TableCtx *struktura;
9    char *hodnota1,*hodnota2;
10
11    TableNew(TableName,FileName,&struktura,&size);
12    printf("table size: %d\n",size);
13    for (k=0;k<size;k++){
14        TableReadLine(struktura);
15    printf("line: %d\n",k);
16        TableGetString(struktura,"cisla",&hodnota1);
17    printf("hondota 1: %s\n",hodnota1);
18        TableGetString(struktura,"pismena",&hodnota2);
19    printf("hondota 2: %s\n",hodnota2);
20    }
21    TableDestroy(struktura);
22    return 0;
23 }
```

Po spuštění programu je výsledek výstupu následující:

```
table size 3
line: a 1 buff: a key: pismena
line: a 1 buff: 1 key: cisla
line: 0
hodnota 1: 1
hodnota 2: a
line: b 2 buff: b key: pismena
line: b 2 buff: 2 key: cisla
line: 1
hodnota 1: 2
hodnota 2: b
line: c 3 buff: c key: pismena
line: c 3 buff: 3 key: cisla
line: 2
hodnota 1: 3
hodnota 2: c
```

Z ukázkového vstupního souboru test_table se klíčové slova “písmena“ a “cisla“ vrací v opačném pořadí, než jak jsou zadaná ve vstupní tabulce. Mezi výslednými hodnotami jsou ještě zobrazeny šedivě zvýrazněné řádky, které zde slouží pouze pro

demonstraci chování a obsahu spojového seznamu. Line představuje řetězec aktuálně načteného řádku, buff aktuální hodnotu, která se přiřadí do spojové seznamu pod hodnotu value k příslušnému klíči, který zde zobrazuje hodnota key.

## 3.1 Čtení dvou tabulek najednou

Implementace si dokáže poradit i se dvěma tabulkami najednou, přečte se první tabulka. Poté se struktura pomocí funkce TableDestroy odalokuje, a bez jakýchkoliv problémů se ihned mohou načíst hodnoty z druhé tabulky. Nebo místo odalokování použiju dva ukazatele na strukturu TableCtx.

```
#include "input_interface.h"

int main(void){
    char FileName[MAX_LINE]=" test_table ";
    char TableName[MAX_LINE]="Tabulka1";
      char TableName2[MAX_LINE]="Tabulka2";
    int size,k;
    TableCtx *struktura;
      char *hodnota1,*hodnota2,*hodnota3;



    TableNew(TableName,FileName,&struktura,&size);
    printf("table size: %d\n",size);
    for (k=0;k<size;k++){
        TableReadLine(struktura);
      printf("line: %d\n",k);
        TableGetString(struktura,"cisla",&hodnota1);
      printf("hondota 1: %s\n",hodnota1);
        TableGetString(struktura,"retezec",&hodnota2);
      printf("hondota 2: %s\n",hodnota2);
      TableGetString(struktura,"el",&hodnota3);
      printf("hondota 3: %s\n",hodnota3);
    }
      TableDestroy(struktura);

    TableNew(TableName2,FileName,&struktura,&size);
    printf("table size: %d\n",size);
    for (k=0;k<size;k++){
        TableReadLine(struktura);
      printf("line: %d\n",k);
        TableGetString(struktura,"cisla",&hodnota1);
      printf("hondota 1: %s\n",hodnota1);
        TableGetString(struktura,"pismena",&hodnota2);
      printf("hondota 2: %s\n",hodnota2);

    }
    TableDestroy(struktura);
    return 0;
}
```

výsledek výstupu je následující:

```
table size 6
line: 1 prvni 911 123 161718 buff: 1 key: el
line: 1 prvni 911 123 161718 buff: prvni key: retezec
line: 1 prvni 911 123 161718 buff: 911 key: cisla
line: 1 prvni 911 123 161718 buff: 123  key: body
line: 1 prvni 911 123 161718 buff: 161718 key: vrcholy
line: 0
hodnota 1: 911
hodnota 2: prvni
hodnota3: 1
line: 2 druhy 912 456 131415 buff: 2 key: el
line: 2 druhy 912 456 131415 buff: druhy key: retezec
line: 2 druhy 912 456 131415 buff: 912 key: cisla
line: 2 druhy 912 456 131415 buff: 456 key: body
line: 2 druhy 912 456 131415 buff: 131415 key: vrcholy
line: 1
hodnota 1: 912
hodnota 2: druhy
hodnota3: 2
line: 3 treti 913 789 101112 buff: 3 key: el
line: 3 treti 913 789 101112 buff: treti key: retezec
line: 3 treti 913 789 101112 buff: 913 key: cisla
line: 3 treti 913 789 101112 buff: 789 key: body
line: 3 treti 913 789 101112 buff: 101112 key: vrcholy
line: 2
hodnota 1: 913
hodnota 2: treti
hodnota3: 3
line: 4 ctvrty 914 101112 789 buff: 4 key: el
line: 4 ctvrty 914 101112 789 buff: ctvrty key: retezec
line: 4 ctvrty 914 101112 789 buff: 914 key: cisla
line: 4 ctvrty 914 101112 789 buff: 101112 key: body
line: 4 ctvrty 914 101112 789 buff: 789 key: vrcholy
line: 3
hodnota 1: 914
hodnota 2: ctvrty
hodnota3: 4
line: 5 paty 915 131415 456 buff: 5 key: el
line: 5 paty 915 131415 456 buff: paty key: retezec
line: 5 paty 915 131415 456 buff: 915 key: cisla
line: 5 paty 915 131415 456 buff: 131415  key: body
line: 5 paty 915 131415 456 buff: 456 key: vrcholy
line: 4
hodnota 1: 915
hodnota 2: paty
hodnota3: 5
line: 6 sesty 916 161718 123 buff: 6 key: el
line: 6 sesty 916 161718 123 buff: sesty key: retezec
line: 6 sesty 916 161718 123 buff: 916 key: cisla
```

```
line: 6 sesty 916 161718 123 buff: 161718 key: body
line: 6 sesty 916 161718 123 buff: 123 key: vrcholy
line: 5
hodnota 1: 916
hodnota 2: sesty
hodnota3: 6
table size 3
line: a 1 buff: a key: pismena
line: a 1 buff: 1 key: cisla
line: 0
hodnota 1: 1
hodnota 2: a
line: b 2 buff: b key: pismena
line: b 2 buff: 2 key: cisla
line: 1
hodnota 1: 2
hodnota 2: b
line: c 3 buff: c key: pismena
line: c 3 buff: 3 key: cisla
line: 2
hodnota 1: 3
hodnota 2: c
```

Podle této reálné demonstrace je vidět, že program pracuje tak jak má. V první polovině kompletně rozebere první tabulku řádek po řádku, správně uloží hodnoty ke klíčovým slovům, nalezne hledané hodnoty, a přiřadí je do proměnných, které jsme si definovali. A postup u druhé tabulky se neliší od první demonstrace, kde jsme tabulku načítali samostatně.

# 4 Implementace v jazyce C

Tato implementace byla vynucena možností použití v kódu Flow123d. Pro její zvládnutí bylo potřeba nastudovat základy jazyka C a jeho další spektrum působnosti. Jako pomocí při studiu mi byla doporučena učebnice Jazyka C od Pavla Herouta (IV. Přepracované vydání)[5]. S níž jsem byl nanejvýše spokojený, pojetí této publikace bylo pro mě více než ideální, srozumitelnost předvedená na názorných příkladech si dokázala poradit i s nepříjemnými úskalími, které tento jazyk skrývá a které jsem při řešení potřeboval. Vyzdvihnu zde komponenty, které byli při práci klíčové. Jimiž jsou pointery(ukazatele, viz 3.1), řetězce(viz 3.2) a struktury(3.3).

Neméně důležitá je formální úprava programu z důvodu lepší srozumitelnosti kódu a bezproblémové orientace v případě hledání chyb. Základními prvky této problematiky jsou. Každou funkci uvozovat formátovým komentářem, který bude popisovat, co daná funkce umí, co je její vstup a co její výstup. Často používat komentáře v průběhu funkce, popisovat děje, významy a účely cyklů ve funkci, prostě jednoduše řečeno komentářů není nikdy dost. Dále je potřeba Volit jména proměnných tak, aby co nejlépe vystihovali jejich význam, i za cenu dlouhého názvu. A úplně posledním avšak možná úplně nejdůležitějším prvkem je přidělování paměti(3.2.4), což je nejčastější důvod zhroucení programu, pokud není přidělena správně.

## 4.1 Pointery

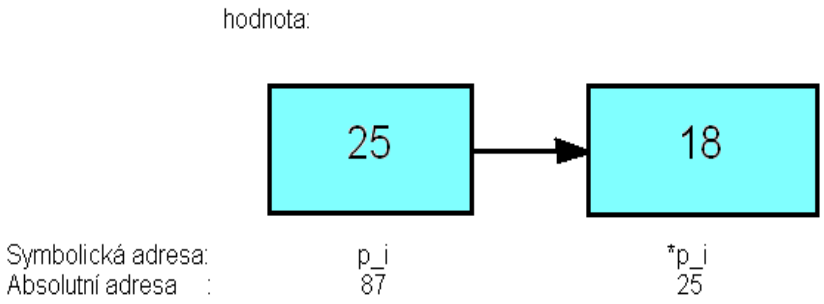
Pointery, jinak řečeno ukazatele podle mého názoru představují nejtěžší a zároveň klíčovou část tohoto jazyka. Pointer je proměnná jako každá jiná, pouze hodnota v této proměnné uložená má odlišný význam než u klasických proměnných.

Pointer představuje adresu v paměti a na této adrese se teprve skrývá příslušná hodnota proměnné.

Pointer je vždy svázán s nějakým datovým typem. Správně by se místo termínu „pointer“ mělo vždy uvádět „pointer na typ … “ . Hlavní funkce ukazatelů, kterých jsem při psaní práce používal byla možnost přenášet hodnoty proměnných z funkce do funkce bez toho abych musel používat globální proměnné, což by sice bylo daleko jednoduší ale z hlediska optimalizace nesrovnatelné. Ukazatel nemusí ukazovat pouze na proměnnou, řekněme typu čísla (int, real,…) nebo řetězec, ale i na datový typ typu struktury, nebo se dokonce dá odkazovat na funkci samotnou. A aby toho nebylo málo, tak ještě dokáže ukazovat i sám na sebe. Takzvaný pointer na pointer.

Význam obrázku (obr 3.2.1) je:

- proměnná p_i je pointer (leží na symbolické adrese p_i)
- Hodnota p_i je 25 (hodnota uložená na symbolické adrese p_i je 25)
- Číslo 25 se nevyužije přímo k výpočtu, ale představuje absolutní adresu v paměti
- Na absolutní adrese 25 v paměti je hodnota 18 (což je číslo, které se již k výpočtu použít dá).
- p_i ukazuje na hodnotu 18, ale sám má hodnotu 25.




Obrázek 3.2.1pointer

Ukazatele definujeme podobně jako jiné proměnné. Rozdílem je, že před identifikátorem musíme uvést ještě znak '*', kterým dáme překladači vědět, že definujeme pointer. Není potřeba navzájem od sebe oddělovat definice obyčejné proměnné a pointeru, jejichž typy (typ a bázový typ) jsou stejné. V případě, že jsou proměnné statické, budou mít nulovou hodnotu. Jsou-li automatické, bude jejich hodnotou nějaká náhodná adresa. I přesto ale lze s pamětí, kam tyto pointery ukazují, pracovat, tedy číst a zapisovat do ní. V takovém případě můžeme lehce přepsat paměť používanou jinými programy nebo systémem samotným. To má většinou za následek zhroucení programu nebo celého systému. Proto musíme ukazatel inicializovat nějakou smysluplnou hodnotou ještě před jeho prvním použitím.

Referenční operátor & Abychom mohli do ukazatelové proměnné přiřadit hodnotu (nějakou adresu), musíme ji nejprve někde získat. K tomu nám může posloužit referenční operátor '&'. Jeho zapsáním před identifikátor proměnné získáme adresu paměti, kde je tato proměnná uložena. Tuto adresu pak můžeme přiřadit nějaké ukazatelové proměnné.

Jak už bylo zmíněno, pomocí pointeru můžeme přistupovat k datovému objektu, na který ukazuje. K tomu nám poslouží operátor dereference '*'. Použitím tohoto operátoru na ukazatel získáme objekt, na který ukazatel odkazuje, přičemž typ takto získaného objektu bude shodný s bázovým typem dereferovaného ukazatele. Navíc je tento objekt l-hodnotou, což znamená, že může stát na levé straně přiřazovacího příkazu.

## 4.2 Řetězce

Řetězec (datový typ string) je speciální typ jednorozměrného pole. Je vždy složen z prvků typu char a v podstatě se chová jako každé jiné jednorozměrné pole.

- Řetězec může mít libovolnou délku, omezenou pouze velikostí paměti. Z této celkové přidělené paměti je ale „aktivní“ (právě využitá) jen její část od začátku až do prvního znaku ´\0´. Veškeré další informace uložené až za ´\0´ jsou při standardním zpracování řetězců nedostupné, protože práce s řetězcem končí vždy dosažením prvního znaku ´\0´.
- Při definování řetězce, tedy alokování místa pro něj, musíme alokovat o jeden bajt více, právě pro tuto ´\0´.
- Pokud zapomeneme na konec řetězce dát znak ´\0´ nebo ten znak omylem přepíšeme, považuje se za řetězec celá následující oblast paměti tak dlouho, dokud se někde dále v paměti tento znak neobjeví. To samozřejmě vede většinou k chybné funkci programu, zvlášť pokud do této paměti zapisujeme.

Protože jsou práce s řetězci velmi časté, poskytuje jazyk C díky své standardní knihovně množství funkcí, bez jejich pomoci bych si vypracování této práce dokázal jen těžko představit. Chceme li tyto funkce využívat, je nutné připojit do našeho programu standardní hlavičkový soubor **string.h. příkazem : #include <string.h>** . Protože řetězce jsou vlastně jen obyčejná pole, není možné je přiřazovat, sčítat je, ani provádět s

nimi jiné podobné operace. Naštěstí jazyk C nabízí pro tyto účely již předdefinované funkce, jejichž služeb můžeme využít. Z nichž podle mého názoru nejpoužívanější je funkce pro kopírování řetězce. Pro kopírování řetězců slouží funkce s obecným předpisem **char *strcpy(char cil[], char zdroj[])**. Pro kopírování jednoho řetězce do druhého je v C připravena funkce strcpy, která má dva argumenty, řetězce cil a zdroj. Bez ohledu na obsah a velikost řetězce cil je do něho postupně, znak po znaku, kopírován obsah řetězce zdroj, a to až do doby kdy se narazí na znak EOS. Tento znak je posledním zkopírovaným znakem. Při používání funkce strcpy je nutné zajistit, aby pole cíl mělo vždy dostatečnou velikost na to, aby se do něj řetězec zdroj vešel. To ale platí i pro většinu ostatních funkcí pracujících s řetězci. Návratovou hodnotou funkce strcpy je ukazatel na řetězec cil, ale protože o souvislosti polí s ukazateli si povíme až někdy jindy, můžeme prozatím tuto informaci pominout.

## 4.3 Datový typ struktura

Nezřídka se setkáváme se skutečnostmi, k jejichž popisu potřebujeme více souvisejících údajů. Navíc různého typu. Užitečnou možností je konstrukce, která takovou konstrukci dovolí a pro její snadné další použití i pojmenuje.

Definice struktury, její korektní syntaktický předpis je následující:

```
struct [<struct type name>] {
   [<type> <variable-name[, variable-name, ...]>] ;
   [<type> <variable-name[, variable-name, ...]>] ;
   ...
 } [<structure variables>] ;
```

Konstrukci uvádí klíčové slovo struct. Následuje nepovinné pojmenování struct type name, které jako v případě výčtového typu obvykle nepoužíváme. Následuje blok definic položek struktury. Po něm opět můžeme definovat proměnné nově definovaného typu. Položky jsou odděleny středníkem. Jsou popsány identifikátorem typu type, následovaným jedním, nebo více identifikátory prvků struktury variable-name. Ty jsou navzájem odděleny čárkami.

Pro přístup k prvkům struktury používáme selektor struktury (záznamu) . (je jím tečka). Tu umístíme mezi identifikátory proměnné typu struktura a identifikátor

položky, s níž chceme pracovat. V případě, kdy máme ukazatel na strukturu, použijeme místo hvězdičky a nezbytných závorek raději operátor ->.

Podívejme se na příklad. Definujeme v něm nové typy complex a vyrobek. S použitím druhého z nich definujeme další typ zbozi. Typ zbozi představuje pole mající POLOZEK_ZBOZI prvků, každý z nich je typu vyrobek. Typ vyrobek je struktura, sdružující položky ev_cislo typu int, nazev typu znakové pole délky ZNAKU_NAZEV+1. Teprve takové definice nových typů, někdy se jim říká uživatelské, používáme při deklaraci proměnných.

```
typedef
  struct {float re, im;} complex;
typedef
  struct {
          int ev_cislo;
          char nazev[ZNAKU_NAZEV + 1];
          int  na_sklade;
          float cena;
         } vyrobek;
typedef vyrobek zbozi[POLOZEK_ZBOZI];
```

Syntaxe struct sice nabízí snadnější definice proměnných použitých v programu, otázkou zní, jak čitelné by pak bylo například deklarování argumentu nějaké funkce jako ukazatel na typ zboží. Jinak řečeno, Konstrukci struktury pomocí typedef oceníme spíše u rozsáhlejších zdrojových textů. U jednoúčelových krátkých programů se obvykle na eleganci příliš nehledí.

Dále se podívejme na přiřazení hodnoty strukturované proměnné při její definici.

```
vyrobek *ppolozky,
    a  = {8765, "nazev zbozi na sklade", 100, 123.99};
```

Konstrukce značně připomíná obdobnou inicializaci pole. Zde jsou navíc jednotlivé prvky různých typů.

## 4.4 Alokace paměti

Práce s tzv. staticky alokovanou pamětí, tedy s paměťovými objekty, o jejíž alokaci se stará náš přeložený program sám není problém. Pokud jsme například v programu nadefinovali proměnnou, program pro ni automaticky alokoval paměť a po skončení její životnosti tuto paměť zase uvolnil. Problém ale nastává v okamžiku, kdy v programu potřebujeme pracovat s daty, pro která nemůže být paměť alokována staticky,

například proto, že předem není známá její potřebná velikost. V takových případech se používá tzv. dynamická alokace paměti, tedy taková, kdy sám programátor může v programu rozhodnout, kolik paměti alokovat. Ještě před samotnými příkazy pro alokaci paměti se ale podíváme na operátor **sizeof**, který právě při dynamické alokaci často využijeme. Tento operátor slouží k zjišťování velikosti svého operandu. Existují dva typy operandů, na které můžeme operátor sizeof aplikovat. Je to buď identifikátor některého datového typu: **sizeof( identifikátor_datového_typu )** nebo libovolný výraz: sizeof výraz. V případě, že operandem bude identifikátor některého datového typu (musí být uveden v závorkách), je výsledkem vyhodnocení velikost, jakou by v paměti zabíral objekt (např. proměnná) tohoto typu. Množina použitelných typů, na které lze `sizeof` aplikovat, není nijak omezena a je možné použít tento operátor i na uživatelsky definované typy.

# 5 IMPLEMENTACE

V implementaci se používají dvě struktury. Struktura LineItem a TableCtx. Které se v interface pomocí funkcí TableNew, TableReadLine, TableGetString plní příslušnými hodnotami. TableCtx obsahuje položky file, kde se uloží ukazatel na aktuálně otevřený soubor. Do name se uloží název tabulky kterou aktuálně hledáme, size je velikost, počet řádků tabulky. Line_no je aktuální řádka v souboru a numer_line_table je aktuální řádka v tabulce, samozřejmě se začíná načítat, až ve chvíli kdy se začínají číst hodnoty z tabulky funkcí TableReadLine. Format je místo, kde se uloží formát. Pak následují 4 ukazatele na strukturu LineItem. Ty se využívají podle potřeby implementace pro plnění hodnotami a formátovými klíči. Každá položka má svůj key-klíč identifikátorů, type a aktuální value.

```
typedef struct LineItem {
    LineItem *read_origin;
    LineItem *next_read;
    LineItem *next_get;
    char *key;
    int type;
    char *value;

}LineItem;
```

```
typedef struct TableCtx {
    FILE *file;
    char *name;
    int size;
    int line_no;
    int number_line_table;
    char *format;
    LineItem *first_read;
    LineItem *first_get;
    LineItem *actual_get;
    LineItem **last_get;

}TableCtx;
```

## 5.1 Popis funkce TableNew

Funkce TableNew je první z funkcí, která se musí v rozhraní použít. Funkce pro vstup potřebuje zadat Název tabulky(TableName), kterou chceme nalézt a název souboru ze kterého chceme číst(FileName). Dále využívá možností struktury TableCtx do které zapíše. A vrátí hodnotu integer size. Prvním krokem co funkce udělá je, že otevře a otestuje zadaný soubor. Potom se čte line pomocí funkce read_line(viz 5.1.1). Aktualizuje se pořadí řádku v souboru. A poté se otestuje, zdali na aktuálním řádku nezačíná naše hledaná tabulka pomocí funkce. FindTable(viz 5.1.2). Dále se řetězci

line odebere přidělená paměť. Toto vše probíhá v cyklu do doby než je FindTable úspěšná.

Následuje přidělení paměti struktuře loc_table, do které se dále uloží název tabulky, formát vyseparovaný ve FindTable, aktualizuje se v ní momentální číslo řádku v soboru(pro případ že nastane chyba, aby se dala lépe dohledat). Nastaví se potřebné ukazatele ve struktuře na NULL. Poté se hledá v souboru hodnota size. Je ošetřeno, že před ní budou ležet mezery či celé prázdné řádky. Hledá se v cyklu, dokud nebude nalezena. Zapíše se do struktury, která poté ukazatelem na strukturu je poslána jako výstup TableNew. Poslední krok je průběh funkce Parse_Table_Format(viz 5.1.3).

### 5.1.1 read_line

Pokud soubor nalezne název tabulky začne číst řádek po řádku pomocí funkce read_line, která vyhazuje komentáře, spojuje řádky, které jsou ve vstupním souboru rozděleny lomítky a vrací pomocí ukazatele na ukazatel vždy kompletní řádek, kterému předtím přidělí pamět. Svojí integer hodnotou funkce říká, zda-li vše proběhlo v pořádku(to v případě že vrací 0) , nebo že nastala chyba při čtení (vrací 1), přetekla nám paměť(vrací 2), a nebo indikuje prázdný řádek (vrací 3). V TableNew se dále rozhoduje co s danou indikací dále. Zbavuje řádky všech přebytečných mezer či konců řádků.

### 5.1.2 FindTable

Tato funkce použije právě čtený řádek ukazatelem line a název tabulky, který byl na začátku v rozhraní zadán programátorem. Tyto dva řetězce tato funkce porovná pomocí funkce strstr(funkce datového typu string pro porovnávání řetězců) a pokud funkce uspěje, a nalezne název tabulky vyseparuje formát tabulky, který ořeže o přebytečné mezery a název dané tabulky a pomocí ukazatele na ukazatel ho pošle do TableNew. Každá položka formátu je uložena do struktury LineItema vytváří se spojový seznam, tyto struktury jsou propojeny pomocí next_read podle pořadí ve formátu. Na první položku ukazuje first_read. Ve strukturách LineItem plní next_read,key,type.

### 5.1.3 Parse_table_format

Funkce použije formát uložený ve struktuře. Rozčlení formát na jednotlivé klíče, které uloží vždy jako key do ukazatele LineItem->actual_get, a přiřadí ke každému klíči jeho typ, tím vytvoří spojový seznam. Pokud typ není uvedený, přiřadí se typ uknown(neboli 0). Typy se přiřazují v integer formě, tyto hodnoty jim přiřadí funkce Type_format, která pouze přiřazuje integer hodnoty definovaným typům. Pokaždé, když je takto actual_get naplněný posune se ve struktuře ve spojovém seznamu na další jako actual_get->next_get. Takto se posouvá a naplňuje hodnotami dokud se funkce nedostane nakonec řetězce formátu.

Reálná podoba funkce TableNew se stručnými popiskami významu jednotlivých příkazů:

```
int TableNew(char *TableName,char *FileName,TableCtx **table_ctx,int
*size)
{
    int err,number_line_help=0,number_line=0;
    char *line,*format;
    FILE *fr;
    TableCtx *loc_table;

      if ((fr = fopen(FileName, "r")) == NULL) {
      printf("File %s wasn't open.\n", FileName); //open and test file
        return 1; }
      do{
   err=read_line(fr,&line,&number_line_help);   //reads line, err gets
   number_line+=number_line_help;          //counts actual number line
      if ((FindTable(TableName,line,&format)) != 0)
//find TableName if succes return 1
            break;
//if find breaks cycle do/while
        free(line);                              //unallocates line
    }while ((err==0)||(err==3));
// 0 means ok, 3 empty line, if err is something else its fail
            if (err==1)   {
                  printf("table didn't found\n");
        return 1;}
            if (err ==2) {
                  printf("Read line errror\n");
        return 1;}
      loc_table = (TableCtx*) malloc(sizeof(TableCtx));//allocates
loc_table struct TableCtx
            if (loc_table == NULL) {
      //test to allocate
                  printf("low memory\n");
        exit(1);}
    loc_table->name = (char *)malloc(sizeof(char)*strlen(TableName));
    strcpy(loc_table->name,TableName);
      //allocates memory at loctable for TableName
    loc_table->format = (char *)malloc(sizeof(char)*strlen(format));
      //allocates memory at loctable for format
```

```
    strcpy(loc_table->format,format);
      //format was separeted at fce FindTable
    loc_table->line_no = number_line;
      //writes into loctable actual reading line
      loc_table->number_line_table = 0;
    loc_table->first_get=NULL;
    loc_table->first_read=NULL;
       do {
      //cycle "do" find values size
    free(line);
    err=read_line(fr,&line,&number_line_help);
    number_line+=number_line_help;
    loc_table->line_no = number_line;
    if ((err!=0)&&(err!=3)){
    TableDestroy(loc_table);
    return 1;}
    }while (atoi(line)==0);
    loc_table->size = atoi(line);
    *size=atoi(line);
    loc_table->file = fr;
       //gets FILE pointer to loc_table->file
    free(line);

    *table_ctx = loc_table;
    parse_table_format(loc_table);
    return 0;
}
```

## 5.2 Popis funkce TableReadLine

Funkce nejprve přečte aktuální řádek pomocí read_line(viz. 1.2.1), a pokračuje dále pouze pokud aktuální řádek není prázdný. Zapíše se aktuální číslo řádku v souboru i v nalezené tabulce. Otestuje se, zdali je číslo řádku tabulky menší, než jaké udává hodnota size. Pokud tomu tak není, je vypsána varovná hláška. Poté se otestuje, zdali už není konec tabulky. Nastavíme si ukazatel na aktuální řádek. Začíná tvorba spojového seznamu v pořadí čtení, který se postupně posouvá v lokální struktuře actual_read pomocí next_read a ukládá do něj vždy jednu hodnotu value, kterou nalezne, a které se před uložením přidělí potřebná pamět. Tento cyklus se opakuje tolikrát, kolik bylo ve formátu nalezeno klíčů. Pokud počet hodnot a klíčů není stejný, napíše se pro operátora varovná hláška. V případě že cyklus probíhá poprvé je u něj ukazatel table->first_get == NULL a vytvoří se ukazatel na strukturu table->actual_get a table->last_get se nastaví na NULL, do které se zapíše při každém dalším průchodu. table->last_get=&(table->first_get).

Reálná podoba funkce TableReadLine se stručnými popiskami významu jednotlivých příkazů:

```
int TableReadLine(TableCtx *table) {
    int number_line_help,i,state;
    char *lines_readed, *line;
    char buff[MAX_LINE];
    LineItem * actual_read; // pointer to actual reads item



    while ((state=read_line(table-
>file,&line,&number_line_help))==3);//cut empty line
      ++table->number_line_table;
      // upgrading number line table
      if(table->number_line_table > table->size)
      //watch size and if size is less than actual
            printf("Number line of table is larger than size");
      //number line table
    if ((strstr(line,"$End")) != NULL)return 2;
// test $End -> konec , TableReadLine return 2
    table->line_no+=number_line_help;
    lines_readed=line;
      for (actual_read=table->first_read; actual_read != NULL;
actual_read=actual_read->next_read) {
        while (*lines_readed==' ')lines_readed++;
            for (i=0,buff[0]='\0';(*lines_readed!='
')&&(*lines_readed!='\0');i++,lines_readed++){
            buff[i]=*lines_readed;}
            while (*lines_readed==' ')lines_readed++;// cut spaces
        buff[i]='\0';
            printf(" pos: %d line: %s buff: %s key:
%s\n",i,line,buff,actual_read->key);
        if (buff[0]=='\0'){
                  printf("Miss values, fail is on line number
%d\n",table->line_no);
                       return 1;}
        actual_read->value=(char *)malloc(sizeof(char)*strlen(buff));
        strcpy(actual_read->value,buff);}
      if (table->first_get != NULL) {
      // if get list is ready, set actual_get to its first item
            table->last_get=NULL;
            table->actual_get=table->first_get;
    } else {
      // set last_get to build the get list
      table->last_get=&(table->first_get);
    }
    if(*lines_readed!='\0')printf("some values will be cut");
      return 0;
}
```

## 5.3 Popis funkce TableGetString

Možnost použití základního algoritmu. Prochází seznam table->first_read, hledá LineItem->key=my_key (strstr), když najde vrátí ukazatel na LineItem->value v proměné val.

Tato rutina přečte položku name z aktuální řádky. Pole vracených hodnot jsou automaticky alokována na požadovanou velikost. Pokud je next_get == NULL tak se projde celý seznam na actual_get a hledá se shoda klíče, když se najde tak se nastaví next_get == NULL a pokračuje se stejně.

Optimalizovaný algoritmus, pokud otestuju table->actual_get->key == my_key (strstr), pro bezpečnost, pokud by neplatilo, použije se zakladni algoritmus. Vrátí value, nastaví table->actual_get=table->actual_get->next_get. Pro inicializaci get seznamu je potřeba ukazatel na ukazatel na LineItem "last_get", který na začátku ukazuje na table->actual_get. A potom na položku next_get posledního LineItem čteného programem. Table New nastavila table->first_get na NULL, a tudíž zde můžeme využít podmínky. ReadLine v případe first_get==NULL nastaví last_get=&first_get. Naopak, když je first_get platný ukazatel, nastaví last_get=NULL. funkce TableGetString pak v pripade last_get!=NULL tvori seznam.

Reálná podoba funkce TableGetString se stručnými popiskami významu jednotlivých příkazů:

```
int TableGetString(TableCtx *table,const char *my_key, char ** val) {
 LineItem *my_actual_get;
      if (table->last_get==NULL) {
            // get list is ready
            if (strstr(table->actual_get->key,my_key)!= NULL){
                *val=table->actual_get->value        table-
>actual_get=table->actual_get->next_get;
                 return 1;//succes
            }
      }
      for (my_actual_get = table->first_read; my_actual_get != NULL ;
my_actual_get=my_actual_get->next_read){
            if (strstr(my_actual_get->key,my_key)!= NULL){
                  if (table->last_get != NULL) {
                        // we bulid get list
                        if (my_actual_get->next_get != NULL ||
&(my_actual_get->next_get) == table->last_get) {
// check previous items and the last item in the get list aginst
//my_actual_get
                  printf("Repeated get operation for the same key!");
                        return 0;
                  }
```

```
                        *table->last_get=my_actual_get;
// point the last item in get list to the new one
                        table->last_get=&(my_actual_get->next_get);
                    }
                    *val=my_actual_get->value;
// move  actual_get->next_get
                    table->actual_get=my_actual_get->next_get;
                    return 1;//succes
                }
            }

        printf("bad name key");
      return 0;
} //fail
```

Kromě TableGetString se můžou v Interface použít i funkce TableGetReal, nebo TableGetInt, které fungují stejně jako TableGetString a vlastně jejich tělo tato funkce tvoří, ale ještě navíc otestují, jestli dané hodnoty odpovídají požadovanému formátu Real nebo integer.

# 6 Závěr

Byl navržen formát pro Flow123d, který umožňuje kompatibilitu s různými verzemi programu. Dále byla navžena knihovna pro čtení dat v tomto formátu a knihovna byla implementována v jazyce C. Implementace byla odzkoušena testy na odstraňování komentářů a správného rozdělování řádek, poté testem na hledání tabulky v souboru s více tabulkami. Dále bylo odzkoušeno čtení více tabulek v jednom programu (otestování funkcí TableNew a TableDestroy). Byl proveden test čtení podle formátu, neboli pořadí čtení v programu jiného než na vstupu, pro stejný program dva různé vstupy.

Další potřebné kroky pro zvýšení použitelnosti této implementace je rozšíření formátu o datové typy Array, Enum, Var a začlenění do zdrojového kódu programu Flow123d, které se v této práci nestihli realizovat

Pro realizaci této bakalářské práce bylo nutné nabít teoretických znalostí programovacího jazyka C. Po mých zkušenostech s programovacími jazyky pascal a delphi na mě jazyk C ze začátku působil velice dobře. Veškeré chování a formátové rysy tohoto jazyka mi přišli velice logické. Jednoduché definice proměnných bez zbytečných znaků navíc, stejně tak i přiřazovací příkazy. Jednoduchá struktura bloku funkce bez zbytečných výrazů navíc. Naprosto logické sestavování řídících struktur, podmínek. Takto nadšený jsem byl až do doby než se narazil na problematiku ukazatelů (pointrů), netvrdím, že se v tu chvíli náhled na tento programovací změnil, protože pointer je velice elegantní řešení v případech, kdy už většinou není jiné volby. Avšak z mého pohledu je tato kapitola nejvíc stěžejním bodem, na který jsem já v jazyce C narazil a proto jsem mu věnoval místo i této práci. Nemyslím, že základní princip je bůhví jak složitý, ale použít pointry ve složitějších situacích tak, aby se chovali přesně tak, jak je potřeba je někdy docela oříšek, protože i přesto, že už se někdy tváří, jako že je vše v pořádku, ne vždy tomu tak je a je těžké bez kvalitního překladače s dobrou vizualizací tyto problémy dohledat.

# Odkazy

[1] M. Hokr, O. Severýn, D. Tondr : Potucky_zprava_MM_05

[2] Pavel Herout: Učebnice jazyka C, 2006, KOOP

[3] Domovská stránka programu GMSH http://www.geuz.org/gmsh/ k datu 15.5.2009