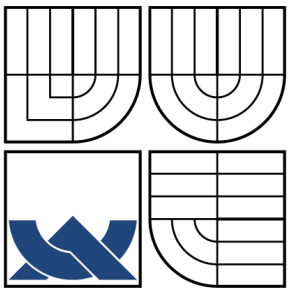
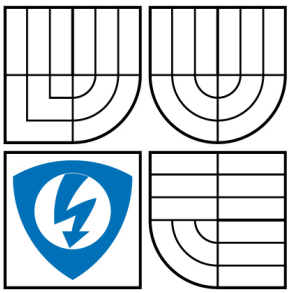


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

DEPARTMENT OF CONTROL AND INSTRUMENTATION

TVORBA ÚLOH PRO VÝUKU PŘEDMĚTU: PRAKTICKÉ PROGRAMOVÁNÍ V C++

C++ PROGRAMMING EXAMPLES FOR EDUCATION

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

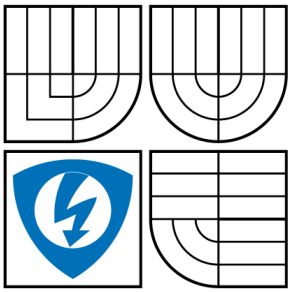
AUTOR PRÁCE — PAVEL ŠABATKA

AUTHOR

VEDOUCÍ PRÁCE — Ing. PETR PETYOVSKÝ

SUPERVISOR

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav automatizace a měřicí techniky

Bakalářská práce

bakalářský studijní obor
Automatizační a měřicí technika

Student:	Šabatka Pavel	***ID:***	83073
Ročník:	3	***Akademický rok:***	2007/2008

NÁZEV TÉMATU:

Tvorba úloh pro výuku předmětu: Praktické programování v C++

POKYNY PRO VYPRACOVÁNÍ:

1. Seznamte se s dostupnou literaturou a obsahem předmětu: Praktické programování v C/C++.
2. Navrhněte vhodné okruhy demonstračních úloh pro tento předmět.
3. Realizujte demonstrační úlohy pro tento předmět.
4. Prezentujte využití vytvořených demonstračních úloh ve výuce předmětu.

DOPORUČENÁ LITERATURA:

[1] Virius, M. : Jazyky C a C++, Grada 2006, ISBN 80-247-1494-9
[2] Virius, M. : Pasti a propasti jazyka C++ 2. vydání, Computer press 2005, ISBN 80-251-0509-1
[3] Richter, M. a kol. : Elektronické texty do kursu BPPC
[4] Prata, S. : Mistrovství v C++, Computer press 2004, ISBN 80-251-0098-7

Termín zadání:	1.2.2008	***Termín odevzdání:***	2.6.2008

Vedoucí práce: Ing. Petr Petyovský

prof. Ing. Pavel Jura, CSc.
předseda oborové rady

UPOZORNĚNÍ:

Vysoké učení technické v Brně

Fakulta elektrotechniky a komunikačních technologií

Ústav automatizace a měřicí techniky

Tvorba úloh pro výuku předmětu: Praktické programování v C++

Bakalářská práce

Specializace studia: Automatizace a měření

Student: Pavel Šabatka

Vedoucí práce: Ing. Petr Petyovský

Abstrakt:

Předmětem této práce je tvorba pomůcek pro začínající programátory, které usnadní studentům absolvování předmětu BPPC.

Teoretická část pojednává o programovacích jazycích C, C++ a jejich odlišnostech. Dále jsou zmiňovány kompilátory těchto jazyků, je zdůrazněno především vývojové prostředí MS Visual C++ 2005, ve kterém probíhá výuka předmětu BPPC a odlišnosti tohoto prostředí od standardů ISO. Jsou také probrány některé základní techniky programování v C++, které jsou často zmiňovány dále v práci.

Druhou částí je implementace knihoven pro kontrolu správnosti programů. Knihovna check.h kontroluje práci programátora s dynamicky alokovanou pamětí a se soubory. Knihovna adtcheck.h umožňuje ověření správnosti a vykreslení topologie abstraktních datových typů implementovaných programátorem. K těmto knihovnám byla vytvořena webová prezentace s jejich dokumentací.

Klíčová slova:

C/C++, programování, kurz BPPC, výuka, detekce úniku paměťových zdrojů, abstraktní datové typy

Brno University of Technology

Faculty of Electrical Engineering and Communication

Department of Control, Measurement and Instrumentation

C++ Programming examples for education

Thesis

Specialisation of study: Cybernetics, Control and Measurement
Student: Pavel Šabatka
Supervisor: Ing. Petr Petyovský

Abstract :

The purpose of this thesis is implementation utilities for beginners in programming. These utilities could make passing subject BPPC easier.

Theoretical part dissertates about programming languages C, C++ and about differences between them. There is also talked about compilers of these languages with accent for development environment MS Visual C++ 2005 and it's diversities form ISO standards. This path contains also some basic chapters of programming, which are often noticed later.

The second part is implementation libraries for checking programs. Library check.h checks programmer's using of dynamic allocated memory and files. Library adtcheck.h was designed for checking and printing topology of abstract data types, which were implemented by programmer. There was designed also web pages with documentation for both libraries.

Keywords:

C/C++, programming, subject BPPC, education, memory leak detection, abstract data types

ŠABATKA, P. *Tvorba úloh pro výuku předmětu: Praktické programování v C++*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2008. 68s.

Vedoucí bakalářské práce Ing. Petr Petyovský

P r o h l á š e n í

„Prohlašuji, že svou bakalářskou práci na téma "Tvorba úloh do předmětu „Praktické programování v C++“" jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.“

V Brně dne : Podpis:

Poděkování:

Chtěl bych poděkovat vedoucímu bakalářské práce Ing. Petru Petyovskému za jeho odbornou pomoc a cenné rady, které mi velice pomohly při realizaci bakalářské práce.

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM ZKRATEK

Zkratka/Symbol	Popis
ADT	Abstraktní datový typ, blíže viz kapitola 2.3.4
DDD	Display data debugger – nástroj pro odstraňování chyb v programech, viz kapitola 4.1
IDE	Integrované vývojové prostředí
IEC	Mezinárodní elektrotechnická komise
ISO	Mezinárodní organizace pro standardizaci
GCC	Kolekce kompilátorů se svobodnou licencí, viz kapitola2.2.1
GNU	Projekt zaměřený na vývoj software se svobodnou licencí

1. ÚVOD

Předmětem této práce je vytvoření demonstračních úloh a učebních pomůcek pro výuku předmětu Praktické programování v C++ (zkráceně BPPC).

Předmět BPPC je jedním z povinných předmětů bakalářského studijního programu na Fakultě elektrotechniky a komunikačních technologií a osnovu přednášek a cvičení má již vytvořenou. Ukazuje se ale, že studenti mívají s určitými oblastmi probíraných témat problémy. Předmětem této práce je určení problémových oblastí předmětu a dále návrh a vytvoření učebních pomůcek, které studentům umožní lépe pochopit probírané téma.

Praktickým obsahem práce je implementace výše zmiňovaných učebních pomůcek. Tyto pomůcky mají kontrolovat práci s dynamicky alokovanou pamětí a se soubory. Dále má být vytvořena pomůcka, která umožňuje kontrolu vytvořených abstraktních datových typů a jejich vykreslení.

Tyto pomůcky je nutno navrhnout s ohledem na jejich použitelnost pro výuku a otestovat na příkladech, se kterými se studenti setkají v rámci předmětu Praktické programování v C++.

2. PROGRAMOVÁNÍ V JAZYCÍCH C A C++

Tato kapitola se zabývá programovacími jazyky C a C++, tvorbou programu v něm a dále stručným úvodem o kurzu Praktické programování v C++.

2.1 PROGRAMOVACÍ JAZYKY

2.1.1 Jazyk C

Jazyk C byl vyvinut na počátku 70. let minulého století Dennisem Ritchiem v Bellových laboratořích AT&T ve Spojených státech Amerických. V krátké době se stal velmi oblíbený a začalo v něm vznikat množství programů. Do roku 1978 ale neexistoval žádný standard jazyka, takže různé překladače obsahovaly drobné odlišnosti. Teprve vydáním knihy „Programovací jazyk C“ od Briana Kernighana a Dennise Ritchieho vznikl první neoficiální standard C. Oficiální standard vznikl až po 11 letech, tzn. v roce 1989 v Americkém národním institutu pro standardizaci (zkráceně ANSI). Standard byl označen jako „X3.159-1989“, vžil se pro něj ale název ANSI C. Obsahoval specifikaci jazyka C s 47 klíčovými slovy a několik základních knihoven. V roce 1990 byl tento standard přijat i Mezinárodní organizací pro standardizaci pod označením „ISO/IEC 9899-1990“. O pět let později byl přijat dodatek rozšiřující standard o práci se širokými znaky (tzv. wide char).

V roce 1999 byl přijat nový standard „ISO/IEC 9899-1999“, často označovaný jako C99. Přinesl několik změn, jako například podporu inline funkcí, několik nových datových typů, nebo to, že proměnné mohou být deklarovány kdekoliv v programu. Úplnou implementaci tohoto standardu zatím neobsahuje žádný kompilátor, některé firmy, jako Microsoft a Borland, tuto implementaci nepodporují téměř vůbec, protože většinu vlastností C99 obsahuje C++. Navíc v některých případech nejsou tyto vlastnosti kompatibilní – např. datový typ C99 complex a stejnojmenná třída v C++.

Jazyk C je nízkoúrovňový procedurální jazyk. To znamená, že jeho příkazy přibližně odpovídají instrukcím procesoru. Zdrojový program v jazyce C je překládán do strojového kódu (kompilován), takže nepotřebuje žádnou runtime

podporu. Tyto jeho vlastnosti způsobují, že je běh programu velmi efektivní vzhledem k rychlosti jeho provádění a paměťovým nárokům.

Jazyk C obsahuje malé množství klíčových slov, většina složitějších funkcí zajišťují knihovny. Velké množství knihoven jazyka C je součástí jeho standardu.

Jazyk C je velmi oblíbený, zejména pro dobrou dostupnost překladačů a knihoven. V dnešní době již pro tvorbu aplikací nemá příliš velký význam. Protože je nízkoúrovňový, má stále velké uplatnění pro programování ovladačů zařízení, jader operačních systémů a podobných algoritmů, kde je kladen velký důraz na efektivitu.

Mezi hlavní nevýhody jazyka patří volný přístup do paměti a absence kontroly mezí polí. To může vést k přetečení a zásahu do dat, které nenáleží programu.

2.1.2 Jazyk C++

Autorem C++ je Bjarne Stroustrup z Bellových laboratoří AT&T. Tento jazyk byl vytvořen kvůli potřebě objektového přístupu pro tvorbu programů a byl vytvořen především na základě jazyka C. Ten byl vybrán pro svou oblíbenost a jednoduchost. Jméno C++ vzniklo spojením označení jazyka C a operátoru inkrementace ++.

Jazyk byl představen veřejnosti v roce 1983. První jeho verze nesla pojmenování C with classes a kombinovala vlastnosti jazyků C a Simula. Umožňovala objektově orientovaný návrh programu, program byl pak pomocí preprocesoru Cpre překládán do čistého C a teprve poté do strojového kódu. Další verze se již jmenovala C++ a byl pro ni vyvinut překladač CFront. Postupně byl vyvinut vlastní kompilátor jazyka C++, samozřejmě se vyvíjel i samotný jazyk – byly přidány výjimky, šablony a prostory jmen.

V roce 1991 vyšla kniha The Annotated C++ reference manual, která se stala základem pro návrh standardu jazyka C++. Standard pak byl oficiálně schválen v roce 1998. Tento standard se opírá o standard jazyka C z roku 1990.

Poslední standard jazyka C++ byl Mezinárodní organizací pro standardizaci přijat v roce 2003 pod označením INCITS/ISO/IEC 14882-2003, ve 3 dalších standardech z let 2006 a 2007 je definováno rozšíření některých knihoven.

Pro C++ existuje řada různých vývojových prostředí, např. Borland C++ Builder, Microsoft Visual C++ nebo KDeveloper či Eclipse pro platformu Linux.

Jazyk C++ ovlivnil další jazyky, jako například PHP, Java či Perl.

2.1.3 Kompatibilita C a C++

Jazyk C++ vznikl především z jazyka C a snaží se s ním být kompatibilní. Proto naprostá většina programů v C funguje i pod C++. Jazyk C++ definuje některá vlastní klíčová slova a operátory, obsahuje navíc množství dalších rozšíření pro objektově orientovaný návrh programu. To způsobuje, že program napsaný v C++ pravděpodobně v kompilátoru jazyka C přeložit nepůjde. V následující kapitole je popsána sémantika a syntaxe běžně se vyskytujících příkazů, které se v jazycích C a C++ odlišují:

2.1.3.1 Přístup k programování

Jazyk C je procedurální. To znamená, že programátor vytváří algoritmy, které dále zpracovávají data. Jazyk C++ umožňuje více různých způsobů návrhu programu, tedy nejen procedurální, jako v jazyce C. Podporuje i objektově orientovaný návrh, kdy programátor vytváří objekty a metody objektů, které popisují jejich vlastnosti. C++ podporuje i generické programování, ve kterém programátor vytváří obecné řešení problému bez ohledu na typ dat.

2.1.3.2 Direktivy preprocesoru

- Direktiva `#pragma` – tato direktiva umožňuje měnit specifické nastavení každého překladače. Různá prostředí mohou mít definovány různé výrazy. Standard C99 navíc obsahuje 3 vlastní direktivy.

  Pokud překladač direktivu nezná, pak ji dle standardu musí ignorovat.
- makro `__cplusplus` – v jazyce C++ je definováno. Díky tomu lze použít podmíněný překlad - uzavřením kódu mezi direktivy preprocesoru:

```
#ifdef __cplusplus
    kód v jazyce C++
#endif /* #ifdef __cplusplus */
```

Díky této konstrukci se program přeloží pouze v jazyce C++, při překladu kompilátorem podporujícím pouze C bude tento kód z programu odstraněn.

- makro `__func__` – vrací jméno nadřazené funkce, definováno pouze v C99 a v C++.

2.1.3.3 Proměnné

- V jazyce C++ je definovaný nový typ logické proměnné – `bool`, jehož hodnotami může být pouze `true` nebo `false`. Ve standardním jazyce C se pro tento účel obvykle používá typ celočíselné proměnné `int` a definovaná makra `TRUE` a `FALSE`. Tyto makra ale musí definovat programátor. S tím souvisí to, že logické výrazy v C vracejí typ `int`, zatímco v C++ vracejí `bool`.
- Typ char – jazyk C++ považuje za samostatný datový typ. Jazyk C ho považuje za synonymum pro `signed char` nebo `unsigned char` (který z těchto dvou to bude záleží na konkrétní implementaci nebo nastavení kompilátoru).
- Pro práci se širokými znaky – je nutno připojit zvláštní knihovny. V jazyce C je to knihovna `<wchar.h>`, v C++ je to pak `<cwchar>`.
- Práce s komplexními čísly – v ANSI C je nutné připojit knihovnu `<complex.h>`, v C++ je nutné připojit knihovnu `<complex>`, která obsahuje příslušnou šablonu, C99 definuje vlastní datové typy `float _Complex`, `double _Complex` a `long double _Complex`.
- Počet prvků pole – v ANSI C musí být zadán pouze konstantním výrazem, C++ umožňuje i zadání počtu pomocí proměnné s modifikátorem `const`, v C99 lze navíc lokální pole deklarovat i pomocí nekonstantního výrazu.
- Reference – C++ umožňuje deklarovat proměnnou jako referenci. Tzn. umožňuje konstrukci `typ &promenna;` .
- Modifikátor `const` – v C definuje proměnnou, kterou nelze měnit standardními prostředky, v C++ se jedná opravdu o konstantu, kterou měnit nelze.
- Modifikátor `restrict` – označení ukazatelů, které neukazují na stejný objekt. Tento modifikátor používá pouze C99 a slouží v překladači k dalším optimalizacím.

- Dynamická alokace pole – pro dynamickou alokaci pole se v C používají funkce `malloc`, `calloc` a `realloc`. Tyto funkce alokují paměť zadané velikosti. V C++ se pro alokaci používá operátor `new`, který navíc v případě alokace pole zvětší velikost alokované paměti o velikost typu `long` a do tohoto místa pak uloží údaj o počtu prvků pole.

2.1.3.4 Funkce, operátory, klíčová slova

- Používání prostorů jmen – jazyk C++ používá prostory jmen (`namespace`). Standardní funkce bývají v prostoru označovaném jako std. Jazyk C žádné podobné konstrukce nepodporuje.
- Přetěžování funkcí a operátorů – v jazyce C++ lze přetěžovat funkce, používat jmenné prostory, používat metody objektů stejných jmen. To znamená, že více funkcí může mít stejné jméno, musí mít ale jiné typy parametrů. O tom, která funkce nebo operátor budou použity v programu, rozhoduje překladač. Vyhodnocuje, která funkce nebo který operátor nejvíce vyhovuje typu parametrů zadaných při volání funkce. Teprve když nemůže najít vhodnou funkci, ohlásí překladač chybu. Jazyk C rozlišuje funkce pouze podle jména, v případě, že překladač najde 2 funkce se stejným jménem, ohlásí chybu.

  Při linkování programu kompilátor přejmenovává jednotlivé funkce dle typů jejich parametrů a návratových hodnot tak, aby jejich názvy byly jedinečné. Tomuto přejmenování říká name mangling. Způsob modifikace jména má každý kompilátor odlišný. Pro názornost je zde uveden příklad jmen funkce v kódu a po úpravě překladačem GCC 4.0:

```
void funkce(void) {}                        _Z6funkcev
void funkce(int i, const char *str) {}   _Z6funkceiPKc
```

  V případě překladu kódu v jazyce C kompilátorem v C++ mohou vznikat chyby tím, že překladač deklaraci funkce přejmenuje a při linkování ji nenajde. Deklarace takovýchto funkcí je proto nutno uzavřít do bloku označeného modifikátorem `extern "C" { deklarace }`. Tento modifikátor je součástí pouze jazyka C++, nikoliv součástí jazyka C. Proto je pro zajištění zpětné kompatibility nutno použít podmíněného překladu – celá konstrukce by pak

vypadala takto:

```
#ifdef __cplusplus
   extern "C" {
#enfif /* #ifdef __cplusplus */
   zde jsou uvedeny deklarace funkcí
#ifdef __cplusplus
   }
#enfif /* #ifdef __cplusplus */
```

- Návratový typ funkce – není-li zadán, ANSI C automaticky doplňuje návratový typ funkce jako int. C99 a C++ vyžadují vždy zadání typu.
- Deklarace `asm` – umožňuje zadat kód v asembleru, pouze v C99 v a C++
- Práce s výstupem – pro C je definována knihovna `<stdio.h>`. V C++ je ve standardní knihovně STL definovaná třída `<ios>`, ze které jsou dále zděděny třídy pro jednotlivé typy výstupu programu.
- Výjimky – klíčová slova pro ošetření nedefinovaných stavů programu. Práci s nimi definuje pouze C++.
- Šablony – C++ podporuje definici šablon objektů a funkcí (tzv. generické programování).

Pro oba jazyky existuje množství knihoven používající jiné funkce. Většina klíčových slov jazyka ale zůstává stejná a správně napsaný program v jazyce C by měl jít přeložit v C++, případně pouze s jeho malými úpravami.

2.2 KOMPILÁTORY JAZYKŮ C A C++

Kompilátor, nebo také česky překladač, je program, který z textu zdrojového programu vytváří binární soubor spustitelný počítačem.

Ve zdrojovém kódu je nejprve zkontrolováno, zda všechny znaky odpovídají základní množině znaků. V případě že ne, jsou nepovolené znaky nahrazeny. Zdrojový kód dále zpracovává preprocesor. Nejprve jsou odstraněny všechny logické přechody na další řádek a komentáře. Preprocesor dále zpracuje všechny direktivy preprocesoru a vkládá hlavičkové soubory. Vznikne tak čistý zdrojový kód. V další

části jsou symboly a klíčová slova nahrazena symboly překladače a tento kód analyzován. V případě, že neobsahuje chyby, je přeložen do zvláštního jazyka, vhodného k optimalizaci a je vytvořen relativní soubor. Při tom jsou vytvářeny funkce a instance dle šablon, pokud šablony překládaný projekt obsahuje. Poté probíhá fáze optimalizace, což je nejsložitější fáze překladu. V mnoha průchodech je projekt analyzován z hlediska toku dat, přístupu do paměti a práce s ní a probíhá vlastní optimalizace. Optimalizovaný projekt je přeložen do assembleru a uložen do objektového souboru s příponou .o nebo .obj v závislosti na použitém překladači. Odkazy v objektovém souboru ještě neobsahují skutečné adresy a jsou zapsány relativně. Na závěr probíhá fáze linkování, kdy jsou nahrazeny všechny relativní odkazy skutečnými adresami, jsou přidány odkazy na knihovní funkce, je přidán startovací kód a soubory jsou spojeny do spustitelného binárního programu.

V případě, že byla v programu nalezena syntaktická chyba nebo odkaz s neexistujícím cílem, je překlad přerušen a je vypsáno chybové hlášení (tzv. error), v případě menší chyby, která neohrozí běh programu je vypsáno varování (tzv. warning).

Překladačů jazyka C++ existuje celá řada, zde je uvedeno několik nejznámějších.

2.2.1 GCC

GNU je projekt zaměřený na tvorbu softwaru se svobodnou licencí. Jeho cílem byl vytvořit takovýto operační systém. To se nakonec podařilo a vznikl operační systém GNU/Linux (zkráceně pouze Linux). V rámci tohoto projektu vznikla mimo jiné i sada překladačů pojmenovaná jako GCC (z anglického GNU Compiler Collection), která obsahuje překladače pro C, C++, Objective C, Fortran, Javu a Adu. Vývoj těchto překladačů je tedy úzce svázán s platformou GNU/Linux.

Tento překladač lze použít v Linuxu přímo pomocí konzolové aplikace. K přeložení programu s jednou knihovnou by bylo třeba tří příkazů. Mohly by vypadat např. takto:

```
:~$ gcc -c -o modul1.o zdrojovy_soubor_1.c
:~$ gcc -c -o modul2.o zdrojovy_soubor_2.c
:~$ gcc modul1.o modul2.c -o programu
```

Tyto příkazy by bylo nutno zadávat pro každý překlad souboru zvlášť, což je poněkud nepraktické. Řešením je napsání tzv. makefile, tedy souboru, který obsahuje posloupnost příkazů k překladu celého projektu a který lze spouštět pouze pomocí příkazu `make`.

Překladač GCC používají také vývojová prostředí KDevelop pro OS Linux nebo Dev-C++ pro OS Windows. Lze také vytvořit spustitelný kód pro jinou platformu, než na které je provozováno vývojové prostředí, a to pomocí tzv. cross compileru. Lze tak vytvářet programy i pro platformy, pro které prozatím neexistuje kompilátor jazyka C.

Poslední dostupnou verzí je GCC 4.2.2, vydaná v říjnu 2007. Překladač GCC i obě výše zmiňovaná vývojová prostředí mají svobodnou licenci a jsou volně dostupné ke stažení.

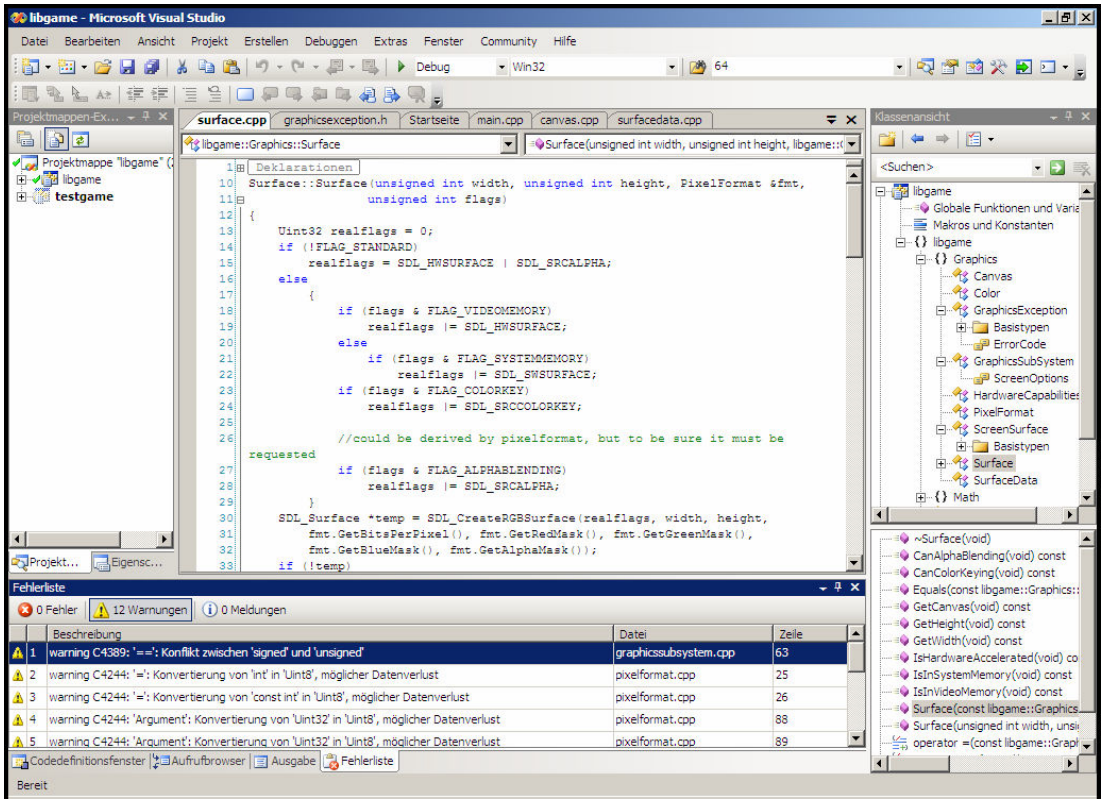
2.2.2 MS Visual C++

Jedná se o komplexní prostředí pro vývoj aplikací vydané společností Microsoft. Obsahuje nástroje pro editaci a správu zdrojového kódu, kompilátor, nástroje pro ladění programu a vlastní knihovny.

Visual C++ umožňuje tvorbu aplikací pro operační systém Microsoft Windows a pro aplikační rozhraní .NET framework. Zdrojový kód aplikace pro operační systém MS Windows jsou přeloženy do strojového kódu, aplikace pro .NET framework je přeložena do mezijazyka, který je dále interpretován pomocí .NET framework.

Oproti standardům ISO jazyků C a C++ obsahuje Microsoft Visual C++ další rozšíření. Jsou to například:

- Nová klíčová slova `__sptr` a `__uptr` pro konverzi 32 bitových a 64 bitových ukazatelů.
- Modifikátor `__declspec(dllexport)`, který říká kompilátoru, že funkce nemá být exportována z knihovny dll.



Obrázek 2-1 Náhled okna vývojového prostředí MS Visual C++

- Modifikátor `__resrtict`, který říká kompilátoru, že proměnná není dekarována v daném prostoru jmen.
- Různá nastavení preprocesoru, kompilátoru a linkeru.
- Některé funkce jazyka C byly označeny jako nebezpečné a zastaralé. Protože se jedná o standardní funkce jazyka C, jsou tyto změny popsány podrobněji v dalším textu:

Microsoft pro tyto funkce používá anglické slovo deprecated, které lze přeložit jako neschválené či zastaralé. Vývojáři proto tyto funkce předefinovali a pojmenovali je ve tvaru `jmeno_s`, kde jmeno je název původní funkce. Například funkce `srtcpy()` neověřuje, jak velká je velikost cílového bufferu. Mohli jste tedy zkopírovat 11 znaků do pole o velikosti 10, a proto mohlo dojít k přetečení a ztrátě dat. Nová funkce `strcpy_s()` má jako jeden z předávaných parametrů délku pole cílové destinace a ověřuje, zda je destinace pro řetězec dostatečně velká. V případě

přetečení vyvolá chybu. Je třeba ještě dodat, že parametr počet prvků destinace může být doplněn automaticky – v případě, že není zadán, MS Visual C++ doplní na místo tohoto argumentu makro `_countof(pole)`, které vrací počet prvků pole. Toto makro nahrazuje konstrukci

```
#define lenghtof(s) (sizeof(s) / sizeof(s[0]))
```

Na rozdíl od ní navíc kontroluje typ argumentu předávaného tomuto makru a v případě, že se nejedná o pole, ohlásí překladač chybu. Toto makro je definováno v Visual C++ 2005 a novějších verzích toho vývojového prostředí.

Většinou se jedná o některé funkce standardních knihoven C (anglicky C Run-Time library, zkráceně CRT). Důvody pro vytvoření těchto funkcí byly tyto:

- Ověřování velikosti bufferu, zabránění přetečení – toto se týká například funkcí pro práci s řetězci, jako je `strcpy()`.
- Ověřování parametrů funkcí – u všech funkcí důsledně ověřuje, jestli jsou zadané parametry v možném rozsahu hodnot a dále kontroluje korektnost v případě, že byl zadán jako parametr NULL.
- Kontrola syntaxe zadaných řetězců – funkce ověřují, zda jsou všechny znaky zadaného řetězce korektní. Toto se týká funkcí `printf()`, `fprintf()`, apod.
- Důsledné zjištění toho, že všechny funkce, které pracují s řetězci nebo je vytvářejí, vždy řetězec uzavírají znakem ”\0”.
- Bezpečnost souborového systému – nové funkce pro přístup k souborům využívají aplikační rozhraní WinAPI.
- Funkce je součástí standardní knihovny, ale není součástí oficiálního standardu ISO. Např. funkce `strnicp()`.

Funkce označené jako zastaralé v MS Visual C++ fungovaly, ale překladač při jejich výskytu v programu hlásil chybu. Pro práci s těmito funkcemi jsou definována některá makra:

- `#define _CRT_SECURE_NO_DEPRECATE 1`

  Při nastavení hodnoty makra 1 se nezobrazují hlášení warning vyvolaná použitím standardních funkcí, které Microsoft označil jako neschválené a zastaralé.

- `#define _CRT_NONSTDC_NO_DEPRECATE 1`
  Při nastavení hodnoty makra 1 se nezobrazují hlášení warning vyvolaná tím, že použitá funkce je ve standardní knihovně, a není popsána standardem ISO.
- `#define _CRT_SECURE_NO_WARNINGS 1`
  Při nastavení hodnoty makra 1 se nezobrazují hlášení warning související s bezpečností neschválených funkcí. Spojuje funkce předchozích 2 maker.
- `#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES 1`
  Při nastavení hodnoty makra 1 se automaticky nahradí všechny neschválené funkce funkcemi bezpečnými.

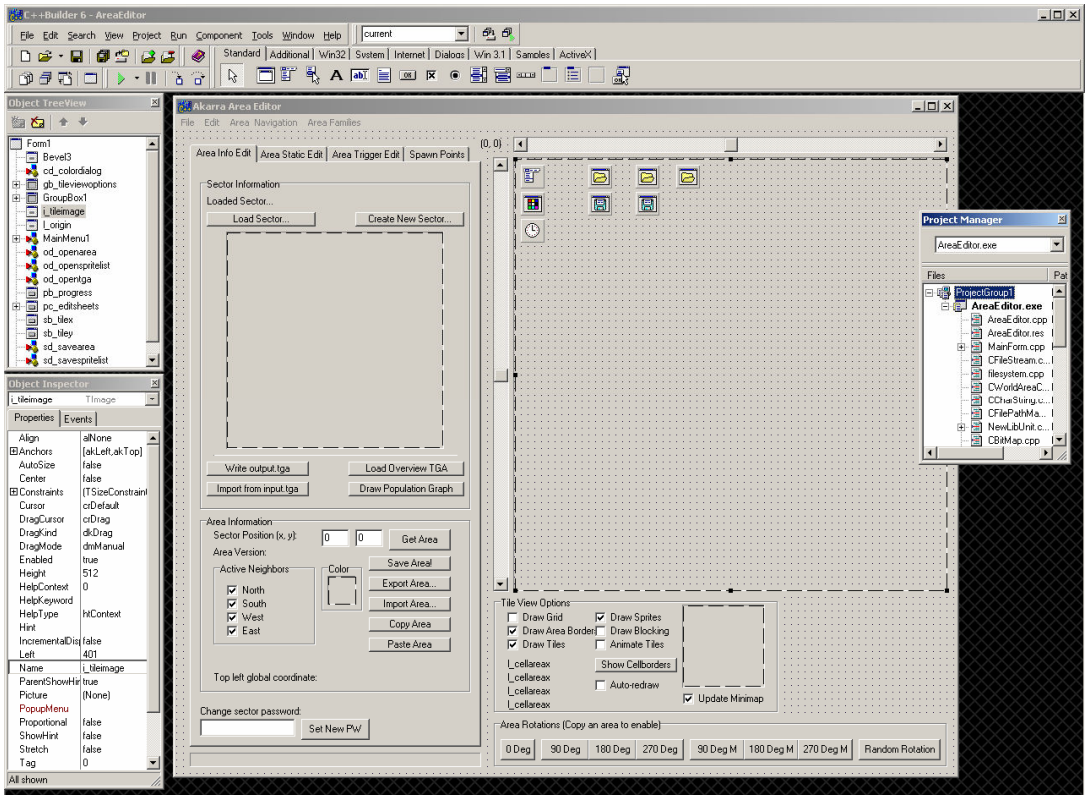
Zobrazování varovných hlášení lze zabránit i použitím direktivy preprocesoru `#pragma`.

Poslední dostupná verze tohoto vývojového prostředí je Visual C++ 2008 vydané v roce 2007. Jeho licence je placená a cena se pohybuje v desítkách tisíc korun. Studenti FEKT VUT mají k dispozici zdarma verzi MS Visual C++ 2005 a výuka předmětu BPPC probíhá právě v něm.

2.2.3 Borland C++ Builder

Je vývojové prostředí společnosti Borland určené k vývoji aplikací pro operační systém Windows a pro návrh webových aplikací. Má propracované prostředí pro tvorbu vzhledu aplikací, stačí pouze vybírat komponenty a myší je přetáhnout do okna aplikace. Tento způsob tvorby aplikace je pochopitelně výrazně rychlejší a jednodušší než psaní zdrojového kódu pro WinAPI. Samozřejmě umožňuje editaci a kompilování čistého kódu jazyků C a C++.

C++ Builder podporuje standard ISO C++ a ANSI C. Jeho vývojáři se ve velké míře těchto standardů drží, ale jednotlivé verze tohoto kompilátoru obsahují některé odchylky. Navíc obsahuje vývojové prostředí vlastní funkce a třídy. Cílem většiny z nich je zjednodušit práci s WinAPI, některé tvoří alternativu ke standardním funkcím. Jako příklad lze uvést funkce pro práci se soubory `LoadFromFile()` a `SaveToFile()`, které tvoří alternativu ke standardní knihovně C++ fstream. Používáním těchto funkcí společnosti Borland pochopitelně způsobuje nepřenositelnost zdrojového kódu na jiné platformy.



Obrázek 2-2 Náhled okna vývojového prostředí Borland C++ Builder

Poslední dostupná verze tohoto prostředí je C++ Builder 2007, která na rozdíl od verze z roku 2006 podporuje vývoj aplikací pro operační systém Microsoft Windows Vista. Licence je placená a stojí několik desítek tisíc korun. Společnost Borland nabízí na svých webových stránkách zdarma ke stažení 30 denní zkušební verzi.

2.3 VYBRANÉ KAPITOLY PROGRAMOVÁNÍ V C A C++

Kapitoly programování byly vybrány a popsány vzhledem k jejich dalšímu použití ve výkladu a praktické části této práce.

2.3.1 Alokace paměti, paměťové třídy proměnných

V každém programu je nutno použít proměnné. Pro tyto proměnné musí být v paměti vyhrazeno místo. Tento proces se jmenuje alokace paměti a rozděluje se na tzv. statickou a dynamickou alokaci. Při statické alokaci je nutno zadat velikost místa již při překladu programu a tato paměť je vždy uvolněna automaticky. Oproti tomu

velikost dynamicky alokované proměnné je vyhodnocena až za běhu programu a její uvolnění musí obstarat programátor. Dynamická alokace je podrobně popsána v kapitole 2.3.2.

Každá proměnná má paměťovou třídu, která určuje její životnost, oblast její viditelnosti a její umístění v paměti. Paměťovou třídu proměnné určuje modifikátor, který může být uveden v deklaraci proměnné ještě před jejím typem. Konstrukce vypadá tedy takto (nepovinné části jsou označeny indexem nep):

```
modifikátor_nep typ_proměnné název_proměnné = hodnota_nep;
```

tedy například:

```
static int i=50;
```

Jazyky C a C++ rozlišují několik paměťových tříd. Jsou to:

- **`auto`** – je paměťová třída, které používají proměnné implicitně, tzn. jestliže není jako modifikátor není použit žádný jiný modifikátor. Paměť se pro takovou proměnnou alokuje automaticky na začátku bloku, kde je proměnná deklarována a na konci tohoto bloku je tato paměť automaticky uvolněna.
  Pokud při inicializaci není uvedena hodnota proměnné explicitně, zůstává hodnota v přidělené paměti stejná jako před přidělením, tzn. hodnota nemusí být nula.
  V případě, že je proměnná v bloku typu auto a má stejné jméno jako jiná globální proměnná, je vždy globální proměnná zastíněna proměnnou typu auto.
- **`static`** – je paměťová třída, která určuje, že je proměnná viditelná pouze v bloku, kde je deklarována. Na rozdíl od třídy `auto` je ale tato proměnná viditelná po celou dobu běhu programu. Tzn. při prvním vstupu do bloku je v paměti alokováno místo a toto místo a hodnota v něm zůstávají zachovány do konce programu. Na jeho konci je tato proměnná automaticky uvolněna.
  Pokud při inicializaci není uvedena hodnota proměnné explicitně, je automaticky nastavena jako nula.
- **`register`** – tento modifikátor lze použít pro optimalizaci programu. Doporučuje překladači, že daná proměnná má být uložena v registrech procesoru, což zrychluje přístup k ní. Z důvodů omezené velikosti registrů procesoru není možné uložit do nich každou proměnnou označenou `register`. O tom, jestli bude

proměnná uložena do registrů či ne, rozhoduje překladač. Tuto třídu je vhodné použít pro proměnné, které se při běhu programu velice často používají.

Vlastnosti proměnných této třídy jsou až na několik výjimek stejné jako vlastnosti proměnných třídy `auto`. Tuto třídu lze použít pouze pro lokální proměnné.

- **`volatile`** – tento modifikátor lze použít pro optimalizaci programu. Při použití této třídy nebude překladač proměnnou optimalizovat. Tuto třídu je vhodné použít zejména tam, kde se očekává změna proměnné pomocí externích událostí, tedy např. přerušením.
- **`const`** – tento modifikátor určuje konstantní proměnou, tedy proměnnou, jejíž obsah nelze měnit. Hodnota takovéto proměnné lze nastavit pouze při inicializaci nebo pomocí ukazatele na proměnnou.

  Modifikátor `const` lze použít i v kombinaci s jinými modifikátory paměťových tříd.

2.3.2 Dynamická alokace paměti

Statická alokace paměti je omezena tím, že již při překladu musí být uvedena velikost alokované paměti. To s sebou nese nevýhody, jako například v případě, že neznáme velikost proměnné, nezbývá než alokovat obrovský prostor tak, aby se do něj určitě proměnná vešla. V případě, že to neuděláme, může nastat situace, kdy se některá data do proměnné nevejdou a hrozí jejich ztráta, navíc může nastat i přepsání a ztráta dat v paměti za proměnnou.

Proto je prakticky ve všech složitějších programech vhodné používat tzv. dynamickou alokaci paměti. Velikost dynamicky alokované proměnné nemusí být známa při překladu a stačí, když je určena proměnnou až při běhu programu. Pro práci s touto pamětí je třeba mít vytvořeny ukazatele, tzn. proměnné, které obsahují adresu alokované paměti. Na konci programu jsou tyto ukazatele odstraněny. Pro alokaci paměti používá jazyk C funkce `void* malloc(size_t)` a `void* calloc(size_t, size_t)`, které alokují paměť požadované velikosti, dále funkci `void* realloc (void *, size_t)`, která mění velikost alokované paměti, a funkci `void free (void*)`, která paměť uvolňuje. Jazyk C++ používá pro dynamické alokování paměti operátory `void* operator new`

`(size_t)` a pro alokaci polí `void* operator new [] (size_t)`. Pro uvolnění paměti pak operátory `void operator delete (void*)` a `void operator delete [] (void *)`.

V případě, že paměť, na kterou směřoval ukazatel, nebyla do konce programu řádně uvolněna, zůstává alokována dál, i když už program neběží. V operační paměti se tak vytváří tzv. memoryleaky – místa, která sice již nepoužívá žádný program, ale nebyla uvolněna, a tak nemohou být dále používána. Tyto memoryleaky zpomalují chod počítače a lze je odstranit teprve restartováním počítače. V případě opakované alokace, například v cyklu, dostává program stále další paměťové zdroje a může dojít k zaplnění velké části paměti. Proto je nutné vždy v programu alokovanou paměť uvolňovat.

2.3.3 Práce se soubory

Pro práci se soubory používá jazyk C funkce `fopen()` a `fclose()`, jazyk C++ pak třídu fstream. V případě zápisu do souboru se znaky, které se mají uložit, ukládají nejprve do bufferu, a teprve když je buffer plný, zapíše se jeho obsah do souboru. Tento zápis nastává i při uzavření souboru. To znamená, že pokud soubor neuzavřeme, hrozí ztráta dat, které jsou v bufferu a ještě nebyly zapsány do souboru. V paměti také vzniká memory leak. Operační systémy mají také omezení pro počet souborových handlů pro každý proces v jednom okamžiku. I když lze implicitní nastavení tohoto počtu měnit, je nutné soubory uzavírat, aby bylo možné otevírat další a nedocházelo k chybám programu.

2.3.4 Úvod do abstraktních datových typů

Abstraktní datový typ určuje způsob ukládání dat a práce s nimi. Je reprezentován množinou hodnot, které mohou data nabývat a dále množinou operací, které lze s daty provádět. Typickými operacemi abstraktních datových typů jsou např. funkce pro vložení dat a pro jejich mazání.

Výhodou abstraktních datových typů je to, že poskytují obrovský uživatelský komfort – při jejich používání stačí deklarovat daný datový typ a dále pracovat s několika funkcemi pro vkládání, mazání či vyhledávání dat. Uživatel se vůbec nemusí starat o samotnou implementaci uživatelských funkcí nebo správu paměti.

Implementace ADT navíc může být změněna a nahrazena bez ovlivnění funkčnosti programu, stačí pouze zachovat názvy funkcí.

Generický abstraktní datový typ je typ ADT, kdy je jasně specifikována pouze množina operací nad datovým typem, ale není definován množina možných hodnot. Typickým příkladem způsobu implementace generických abstraktních datových typů je použití šablon v jazyce C++.

Použití ADT je velmi častým problémem, proto se staly některé nejčastěji používaní generické abstraktní datové typy součástí STL, tj. standardní knihovny jazyka C++, která je součástí každého kompilátoru tohoto jazyka. Tyto typy se nazývají datové kontejnery. Mezi ně patří lineární seznam `<list>`, `<vector>`, zásobník `<stack>`, fronta `<queue>`, obousměrně vázaná fronta `<deque>` a fronta řazená dle priority `<priority_queue>` a datový typ `complex` umožňující práci s komplexními čísly. Mezi další velice významné ADT patří například stromy, nebo graf. Některé uvedené typy jsou popsány v následující kapitole.

2.3.5 Základní ADT

Pro implementaci ADT se obvykle používá několik různých postupů. Lze například vytvořit třídu, jejíž datová složka tvoří data ADT a pomocí členských funkcí je k datům přistupováno. Příkladem tohoto přístupu je třída `complex` knihovny STL. Dalším možným postupem je vytvoření pole ukazatelů, jimž jsou pomocí operátorů přiřazovány adresy na vkládané prvky. Pomocí tohoto postupu lze implementovat např. zásobník. Nejčastějším způsobem implementace je ukládání dat do struktur, které obsahují ukazatele na další prvky. Pomocí tohoto způsobu lze implementovat např. seznam, strom, apod.

Pro přístup k hodnotám lze použít klíč (`key`), tzn. každému prvku se přiřadí klíč, pomocí něhož lze vyhledávat data. Datové typy, které používají tento přístup, se označují jako asociativní. Příkladem využití takovéhoto postupu je tvorba databáze videokazet, kde má každá kazeta svoje číslo. Knihovna STL obsahuje asociativní datové kontejnery množina (`set`) a mapa (`map`).

Typické algoritmy nad ADT jsou kontruktor, funkce pro vložení prvku a jeho odstranění a dále vyhledání prvku. Další algoritmy jsou závislé na konkrétním datovém typu.

V dalších bodech jsou popsány některé ADT, jejichž správnost kontroluje knihovna adtcheck.h nebo jsou použity při její implementaci.

2.3.5.1 Seznam

Seznam je lineární datový typ. To znamená, že každý prvek s výjimkou počátečního a koncového má jeden prvek, který mu předchází a jeden, který ho následuje. Základem seznamu je datový typ struktura, jejíž součástí je jeden nebo dva ukazatele na strukturu stejného typu. Těmto ukazatelům jsou pak přiřazeny adresy následujícího, popř. předchozího prvku. Deklarace tohoto typu může vypadat například takto:

```
typedef struct TPrvek {
    Typ DATA;              // data uložená v seznamu
    TPrvek *Ukazatel_na_dalsi; // ukazatel na další
    };
```

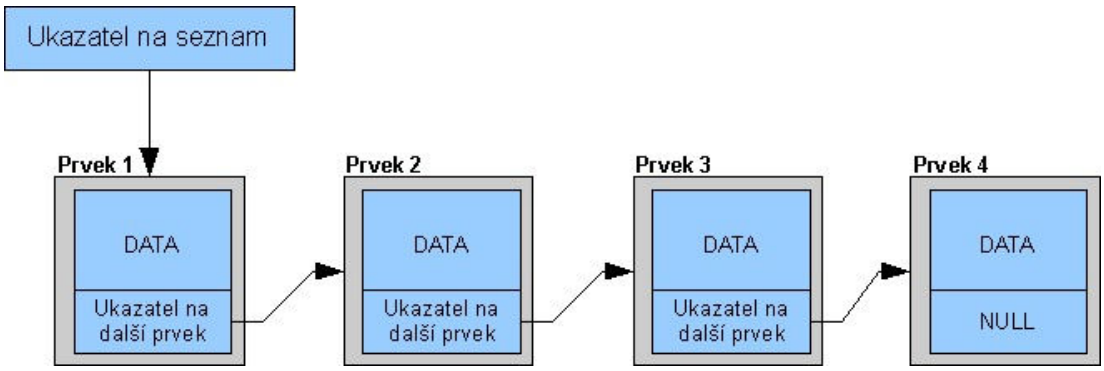
Datovou složkou může být libovolný typ, tzn. i seznam. Prvků může být libovolný počet, jediným omezením je fyzická paměť počítače.

Pro přístup k datům je nutné mít ukazatel na první prvek, v některých případech se používá i ukazatel na prvek poslední. Pomocí ukazatelů v dalších prvcích lze procházet všemi prvky seznamu.

Podle topologie a počtu ukazatelů ve struktuře prvku lze rozdělit seznamy na několik základních typů:

2.3.5.2 Jednosměrně vázaný lineární seznam

Tento typ seznamu je složen z prvků, které obsahují pouze jeden ukazatel, a to na následující prvek. Ukazatel posledního prvku musí být nulový. Znázornění principu jednosměrně vázaného lineárního seznamu je na obrázku 2-3.



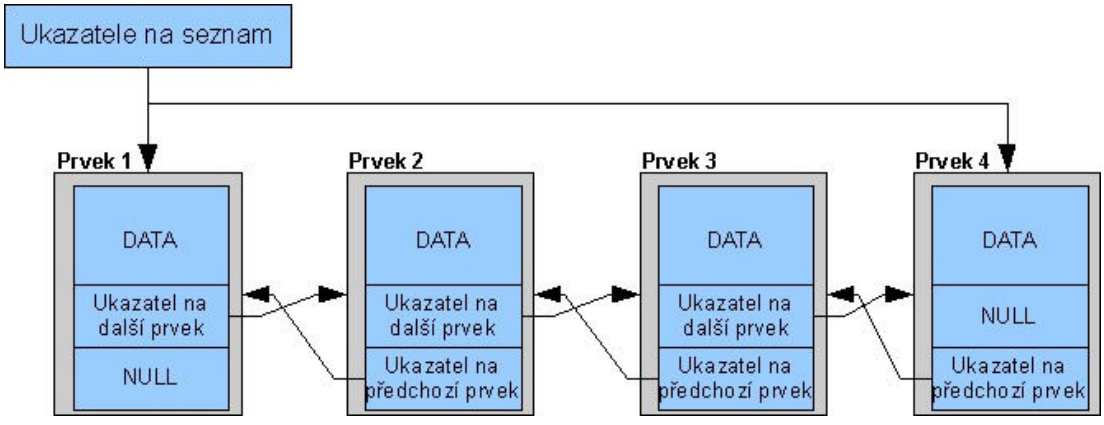
Obrázek 2-3 Jednosměně vázaný lineární seznam

Implementace jednosměrně vázaného lineárního seznamu je také součástí standardní knihovny STL, nazývá se list.

2.3.5.3 Obousměrně vázaný lineární seznam

Tento typ seznamu je složen z prvků, které obsahují dva ukazatele, a to na předchozí a následující prvek. Nulové musí být dva ukazatele – ukazatel na prvek před prvním a na prvek za posledním. Obvykle se pro práci se seznamem uchovávají ukazatele na první a poslední prvek. Vyhledávání tak nemusí vždy začínat zepředu a pokud jsou prvky seznamu řazeny, může se tak vyhledávání značně urychlit.

Ukázku obousměrně vázaného lineárního seznamu znázorňuje obrázek 2-4.



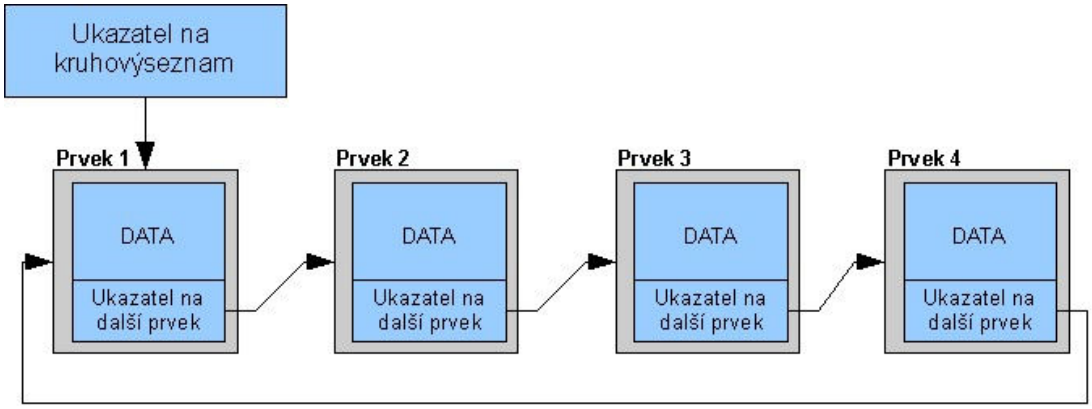
Obrázek 2-4 Obousměrně vázaný lineární seznam

2.3.5.4 Kruhový seznam

Tento typ seznamu vychází z jednosměrně vázaného lineárního seznamu, poslední prvek na rozdíl od něj obsahuje ukazatel na první. Pozice prvního prvku se

může libovolně měnit. To umožňuje usnadnit implementaci, může to ale také vést k potížím při stanovování podmínek procházení seznamem.

Struktura kruhového seznamu je na obrázku 2-5.



Obrázek 2-5 Kruhový seznam

2.3.6 Strom

Strom je hierarchická datová struktura, kde pro každý prvek lze určit, které prvky jsou v hierarchii výše a které níže. Každý prvek obsahuje alespoň 2 odkazy na prvky níže. Prvek nejvýše v topologii se nazývá kořen, uzel nejníže se nazývá list. Ostatní prvky se nazývají uzly.

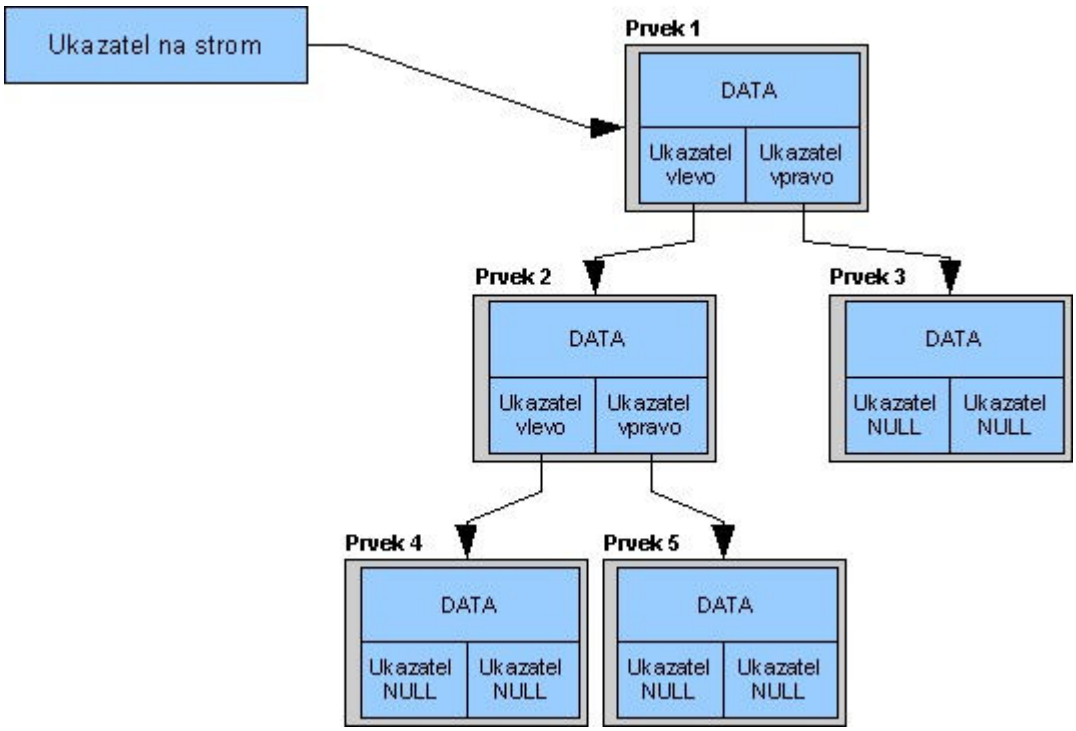
Pro algoritmy implementované nad datovým typem strom se velmi často používá rekurze, tzn. volání funkce z jejího těla.

Používání stromu je velmi výhodné zejména kvůli vyhledávání – doba vyhledávání prvku v seznamu je lineárně závislá na počtu prvků, zatím co strom má tuto závislost logaritmickou. Pro velký počet prvků je tedy vyhledávání výrazně rychlejší.

Struktura stromu je znázorněn na obrázku 2-6.

2.3.7 Vector

Vector je datový kontejner standardní knihovny STL. Jedná se o šablonu třídy, což znamená, že typ ukládaných dat si může zvolit programátor dle svých potřeb. Data jsou ukládána do jednorozměrného pole. Třída vector obsahuje množství funkcí pro práci s daty, čímž výrazně zmenšuje čas potřebný k tvorbě programů.



Obrázek 2-6 Topologie stromu

2.4 PŘEDMĚT PRAKTICKÉ PROGRAMOVÁNÍ V C++

2.4.1 O předmětu BPPC

Předmět Praktické programování se vyučuje v zimním semestru 2. ročníku bakalářského studijního programu na Fakultě elektrotechniky a komunikačních technologií na VUT v Brně.Vychází ze znalostí programování v jazyce C, které měli studenti získat v předmětu BPC2. Cílem je, cituji z anotace, „Získání dovedností v oblasti objektového programování“. Přesněji se zaměřuje na tvorbu objektově orientovaných programů, dědění, streamy, tvorbu šablon a práci se standardními knihovnami.

Pro výuku se v BPPC používá vývojové prostředí Microsoft Visual C++.

2.4.2 Problémy studentů v BPPC

Ukazuje se, že některé oblasti programování a objektového návrhu programů činí studentům obvykle potíže. Jsou to především:

- alokace a korektní uvolňování paměti;
- uzavírání otevřených souborů;
- získání konkrétní představy o abstraktních datových typech, jako je například lineární seznam nebo binární strom, dále získání praktických zkušeností při práci s těmito typy;
- potíže s pochopením funkce objektu, jeho použitím a návrhem objektově orientovaného programu;
- nepochopení, co je předávání parametrů, a osvojení si způsobů předávání parametrů.

Pro pomoc s odstraněním prvních dvou problémů byla implementována knihovna `<check.h>`, v dalším výkladu je pro ni použit i termín checker. Pro pomoc s navrhováním a realizací abstraktních datových typů byla pak navržena knihovna `<adtcheck.h>`.

3. KNIHOVNA CHECK

Cílem práce bylo vytvořit knihovnu, která by kontrolovala programátorovy chyby při dynamické alokaci paměti a při práci se soubory. Nutné bylo, aby programátor nemusel dělat změny v běžné syntaxi jazyka C a všechny kontroly byly zajištěny automaticky. Dalším požadavkem na tuto knihovnu bylo, aby ovládání bylo co nejednodušší.

3.1 NÁSTROJE PRO KONTROLU ALOKOVANÉ PAMĚTI

Kontrola alokované paměti je běžným problémem při vývoji a ladění programů. Pro různé platformy proto byly vytvořeny různé nástroje, které práci s pamětí kontrolují. Lze jmenovat například:

- **Valgrind** pro platformu GNU/Linux

  Tento nástroj je komplexnější než ostatní uvedené nástroje. Umožňuje kontrolu paměti již přeloženého programu, to znamená, že původní kód programu mohl být napsán v libovolném programovacím jazyce. Mimo kontroly uvolnění paměti ovšem také kontroluje další chyby při práci s pamětí, jako přístup do paměti, uvolňování již uvolněné paměti, používání neinicializované proměnné apod.

  Programátor pomocí Valgrindu spustí kontrolovaný program. Valgrind vytvoří virtuální procesor, který přiděluje programu paměť a kontroluje přístup do ní. Na konci programu pak vypisuje statistiku chyb. Pro kontrolu programu je vhodné přeložit ho s pomocnými ladícími informacemi pro debugger, tzn. přidat do příkazu pro překlad parametr `-g`. Ve výpise se pak objevuje i přesná lokace chyb a místa, kde byla alokována neuvolněná paměť.

  Výhodami Valgrindu jsou jeho komplexnost, jednoduchost a dále to, že pro kontrolu programu nemusí programátor nijak zasahovat ho zdrojového kódu.

  Nevýhodami tohoto nástroje je svázanost s konkrétní platformou.

- **CleanupStack** – pro C++ v Symbian OS, dříve EPOC

  Tento nástroj je určen pro vývoj aplikací pro operační systém Symbian, který se používá zejména pro mobilní telefony. Jedná se nástroj použitelný pouze pro překladače podporující jazyk C++.

Nástroj obsahuje třídu, která vytváří seznam objektů, pro které byla alokována paměť. Programátor pomocí metod třídy CleanupStack `PushL()` a `Pop()` přidává do tohoto seznamu ukazatele na tyto objekty. V případě, že na konci programu není všechna dynamicky alokovaná paměť korektně uvolněna, bude uvolněna automaticky.

Zajímavou vlastností je způsob ošetření nebezpečí chyby samotného CleanupStacku v případě nedostatku paměti – vždy má alokovánu paměť pro jeden prvek navíc. To znamená, že v případě, kdy při přidávání ukazatele na objekt do seznamu dojde k zaplnění paměti a chybě, má tedy ještě místo pro zaznamenání ukazatele na objekt. V takovémto případě je před alokací další paměti nutno jinou paměť uvolnit, a tím vznikne i volné místo v seznamu objektů CleanupStacku.

Nevýhodou tohoto nástroje je jeho vázanost na konkrétní platformu. Další nevýhodou je, že programátor musí do aplikace přidávat příkazy pro zařazení a odstranění objektu ze seznamu objektů.

- Dále např. placené nástroje **BoundsChecker** nebo **Deleaker** pro Visual C++ společnosti Microsoft.

Přesnou dokumentaci všech uvedených nástrojů lze nalézt na internetu. Jejich vlastností je ale vždy vázanost na konkrétní program nebo platformu.

Navržená knihovna checker je napsána pro překlad na různých platformách a byla překládána a testována v Microsoft Visual C++ 2005, spuštěném v operačním systému Windows XP SP2, a dále překládána překladači GCC 4.2.2 pod operačním systémem GNU/Linux Kubuntu.

3.2 POPIS CHOVÁNÍ KNIHOVNY CHECK.H

Při kontrole jednotlivých příkazů se navržená knihovna chová takto:

3.2.1.1 Jazyk C

Knihovna podporuje kontrolu používání funkcí `malloc()`, `calloc()` a `realloc()`. Dále kontroluje používání funkce `free()`. V případě, že je pomocí této funkce uvolňována paměť, která nebyla programu přidělena, zahlásí checker

chybu a program ukončí. V případě, že je pomocí funkce `free()` uvolňována paměť, která byla alokována pomocí operátoru `new`, vypíše checker varování. Knihovna dále kontroluje, zda byla všechna dynamicky alokovaná paměť správně uvolněna.

Knihovna podporuje kontrolu používání funkcí `fopen()` a `fclose()` a kontroluje, zda byly všechny soubory, které byly otevřené pomocí těchto funkcí, na konci programu korektně uzavřeny.

3.2.1.2 Jazyk C++

Knihovna podporuje kontrolu používání operátorů `new` a `delete`. V případě, že operátoru `new` není přidělena paměť, je generována chyba. Pokud je pomocí operátoru `delete` uvolňována paměť, která byla alokována pomocí funkce `malloc()` nebo `calloc()`, je vypsáno varování.

Knihovna nepodporuje kontrolu práce se soubory pomocí standardní knihovny jazyka C++ STL.

3.3 PRINCIP KNIHOVNY CHECK.H

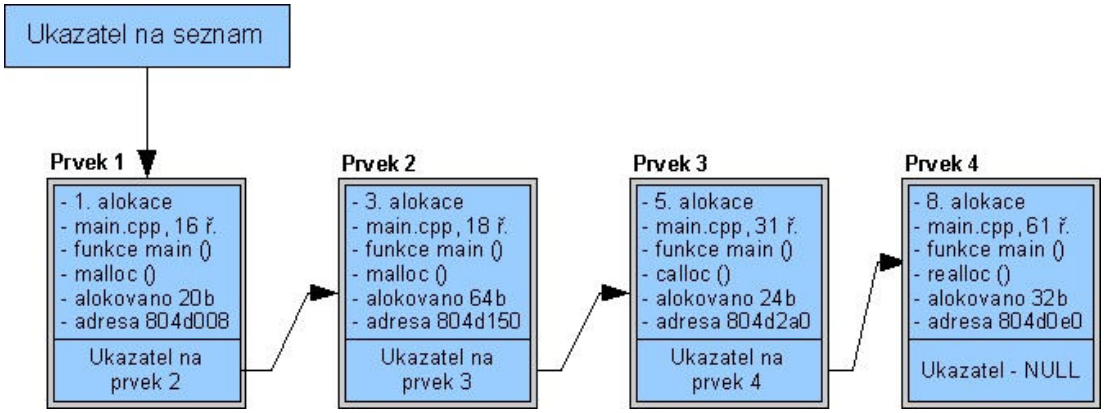
Kontrolu správnosti používání funkcí jazyka C zajišťují makra, která přejmenovávají názvy funkcí a zároveň do jejich těl vkládají předdefinovaná makra `__LINE__`, `__FILE__` a `__func__`, která vracejí řádek a jméno souboru a jméno nadřazené funkce, ve kterém jsou tato makra umístěna. V knihovně checker je pak deklarace a definice funkcí, které se po změně názvu volají.

Kontrola dynamické alokace pomocí operátorů C++ je řešena přetížením globálních operátorů `new` a `delete`, kterým jsou dodány makrem další parametry.

Pro alokovanou paměť a pro otevřené soubory je pak z předaných údajů vytvořen jednosměrně vázaný lineární seznam. Jako hlavičky seznamů jsou užity definované struktury `tUnit` pro alokaci paměti a `tUnitFile` pro kontrolu práce se soubory. Tyto struktury obsahují přehled použití jednotlivých funkcí v programu, velikost alokované paměti apod. Dále pak obsahují ukazatel na první prvek seznamu a na pomocný prvek „act“. Vlastní seznamy jsou pak tvořeny definovanými strukturami `tList` a `tListFile`. V nich jsou uchovávány informace

o konkrétním volání funkce nebo operátoru, tzn. řádek a soubor, kde byly operátory použity, nadřazená funkce apod.

Při alokaci paměti nebo otevření souboru se uloží do lineárního seznamu záznam, při uvolňování paměti nebo uzavření souboru se příslušný záznam odstraňuje. Při tom je kontrolováno, jestli programátor uvolňuje všechnu paměť, která mu byla přidělena a jestli nekombinuje funkce jazyka C a operátory jazyka C++ pro totéž paměťové místo. Na obrázku 3-1 je znázornění způsobu uložení dat.



Obrázek 3-1 Znázornění uložení dat v knihovně check.h

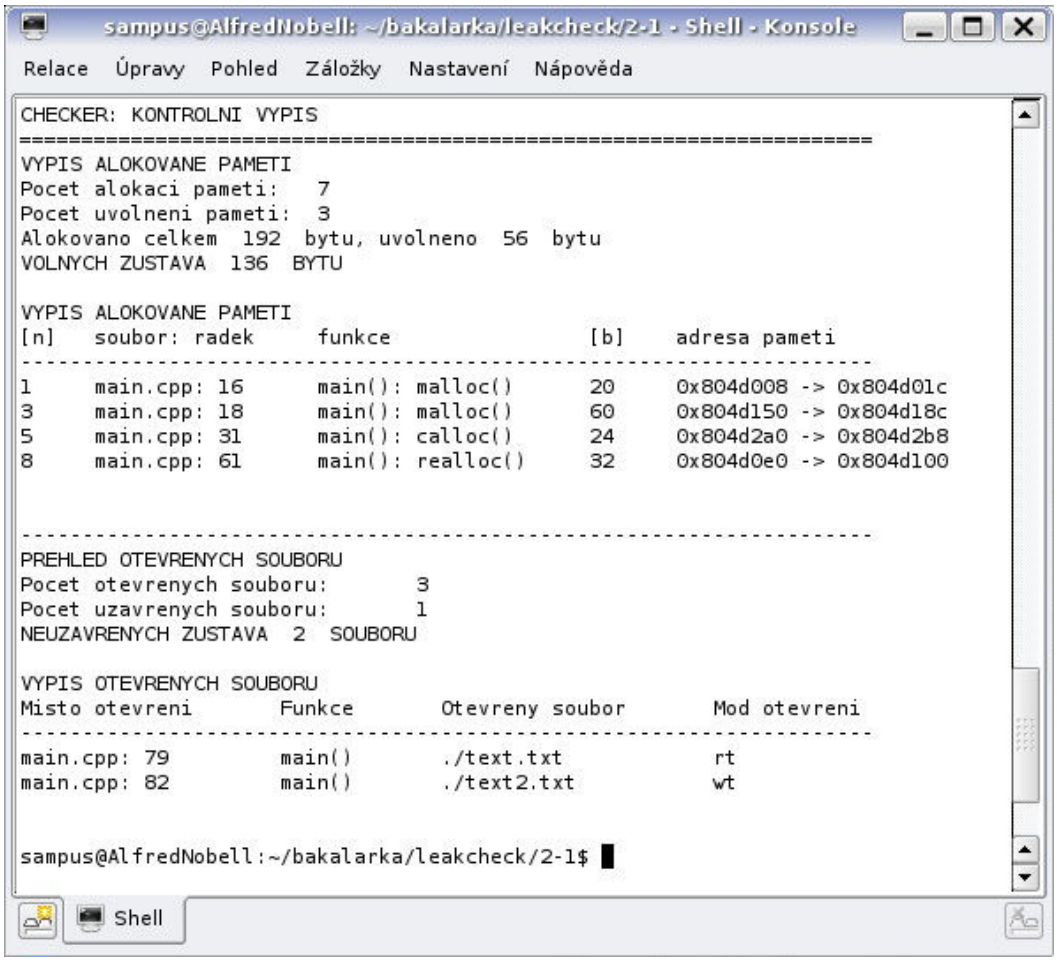
Funkce `exit()` je funkce, která se volá automaticky na konci programu a volá se vždy. Pomocí funkce `atextit()` je přidán do funkce `exit()` ukazatel na funkci `checkdel()`. Tato funkce zajišťuje výpis checkeru na konci programu. Dále pak projde oba seznamy a uvolní paměť na místech, která jsou ve zbylých záznamech. Program se tak navenek chová správně a nezanechává v paměti memory leaky.

Tabulka 1: Seznam maker pro výpis knihovny check.h

Makro	Význam makra
`__n__`	výpis pořadí alokace
`__SOUBOR_RADEK__`	výpis souboru a řádku
`__funkce__`	výpis nadřazené funkce
`__b__`	výpis velikosti alokované paměti
`__adresa_pameti__`	výpis počáteční a koncové adresy alokované paměti

Pro vypisování je definována funkce `print_list()`. Tato funkce v závislosti na předaných parametrech může vypisovat obsahy obou dvou seznamů. To, které sloupce se budou při výpisech zobrazovat, lze ovlivnit definicí maker v hlavičkovém souboru `<check.h>`. Tato makra jsou uvedena v tabulce 1. V případě, že jsou tato makra definována, sloupce tabulek se zobrazují.

Tabulkový výpis funkce `print_list()` byl navržen tak, aby se změnou počtu mezer mezi sloupci sám přizpůsoboval šířce konzole. Nejdříve funkce zjistí počet znaků nejdelších řetězců ve všech údajích, které se budou vypisovat a podle jejich celkového počtu pak nastaví šířku vzdálenosti mezi sloupci. Toto zvyšuje přehlednost výpisu a omezuje možnost zalomení řádku tabulky na více řádků na konzoli. Samozřejmě je ošetřen případ, kdy je šířka nadpisu v záhlaví tabulky větší, než šířka všech údajů v tabulce kontrolního výpisu.



Obrázek 3-2 Příklad kontrolního výpisu checkeru

3.4 POUŽÍVÁNÍ KNIHOVNY CHECK.H

Knihovnu checker tvoří dva soubory – hlavičkový soubor check.h a zdrojový soubor check.c. Pro kontrolu správnosti checkrem je třeba přikopírovat oba uvedené soubory k překládanému projektu. Dále je třeba checker přeložit společně s projektem. V MS Visual C++ lze připojit soubory k projektu pomocí tlačítka v hlavním menu „Add new item“ a nabídky „Add existing item“. V projektech s automaticky generovaným makefile to znamená pouze připojit tyto soubory k projektu, v jiných případech je třeba do makefile doplnit příkaz pro připojení souborů knihovny do projektu.

Posledním krokem je připojení hlavičkového souboru k programu direktivou preprocesoru `#include`. Tzn. mít v souboru, který má být kontrolován nebo v jeho hlavičkovém souboru příkaz `#include "check.h"`.

Checker obsahuje 3 uživatelské funkce, které lze volat přímo z těla kontrolovaného programu. Tyto funkce jsou spolu s jejich významem uvedeny v tabulce 2.

Tabulka 2: Seznam uživatelských funkcí knihnvy check.h

Funkce	Význam funkce
`void memory_stat (void);`	vypíše aktuální stav alokované paměti, tj. přehled a případně tabulku s výpisem paměti, alokované v dané chvíli
`void file_stat (void);`	vypíše aktuální počet otevřených a uzavřených souborů, checker umí kontrolovat pouze soubory otevřené dle standardů C, kontrolu vstupních a výstupních proudů jazyka C++ nepodporuje
`void stat (void);`	slučuje obě dvě předchozí funkce

Checker sám vždy na konci programu zkontroluje stav alokované paměti a otevřených souborů a vypíše o tom hlášení. V případě, že v programu zůstala neuvolněná paměť nebo neuzavřený soubor, programátor je na to upozorněn

výpisem. Checker dále zajistí uvolnění alokované paměti a korektní uzavření všech souborů, takže celý program se chová správně a nezanechává v paměti memory leaky.

V hlavičkovém souboru je možno změnou některých definovaných maker změnit vlastnosti checkeru. Makrem `CHECKER_OUT` je definován výstup výpisu checkeru. Implicitně je definován jako standardní chybový výstup, tzn. hodnota makra bude `stderr`. Dalšími makry lze ovlivnit obsah vypisovaných tabulek. Jedná se o makra jako `__funkce__`, `__SOUBOR_RADEK__` atd. Je li makro definováno, příslušný sloupec se bude zobrazovat.

3.5 ZAJÍMAVÉ PROBLÉMY ŘEŠENÉ PŘI IMPLEMENTACI

Při implementaci bylo nutno odstranit množství problémů souvisejících s odlišným chováním překladačů na různých platformách. Většina těchto problémů je popsána v kapitole o programování v jazycích C a C++. Jednalo se o tyto problémy a řešení:

- Překladač MS Visual C++ hlásil při používání standardních funkcí C varování, že jsou tyto funkce zastaralé. Tento problém je podrobněji popsán v kapitole 2.2.2 o vývojovém prostředí MS Visual C++. Pro odstranění varovných hlášení bylo definováno makro `#define _CRT_SECURE_NO_DEPRECATE 1`.
- Makra definovaná pro přejmenování funkcí a pro přidání parametrů funkcí přepisují všechna nalezená definovaná slova od své definice v souboru až po jeho konec nebo po direktivu `#undef`. Za určitých okolností by tedy mohla nastat situace, kdy by byla přepsána i deklarace funkcí v původních hlavičkových souborech – např. `<stdlib.h>` nebo `<malloc.h>`. Pro odstranění tohoto problému bylo nutno připojit do hlavičkového souboru checkeru všechny knihovny, které obsahují nahrazované funkce, a to ještě před definicí maker pro přejmenování. Kód souboru check.c je pak proti přepsání funkcí chráněn direktivou preprocesoru `#undef`.
- Operátory jazyka C++ nejsou definovány v jazyce C. Při překladu v C++ je deklarováno makro `__cplusplus`. Kód v jazyce C++ je uzavřen mezi direktivy

preprocesoru `#ifdef __cplusplus` a `#endif`, které zajišťují, že se funkce napsané v C++ při překladu pro jazyk C ve výsledném programu vůbec neobjeví.

- Problémy s name manglingem – ten je blíže popsán v kapitole 2.1.3.4. Pro vyřešení tohoto problému bylo nutno uzavřít deklarace knihovních funkcí jazyka C mezi výrazy `extern "C"{deklarace funkcí}`. Tento příkaz je uzavřen mezi výše zmiňovanými direktivami preprocesoru `#ifdef __cplusplus` a `#endif`. Díky tomu je respektován rozdíl deklarace funkcí v hlavičkových souborech v C a C++.
- Přetěžování globálního operátoru `delete` – podle standardu ISO/IEC může být operátoru `delete` předáván jeden parametr typu `void*` nebo mohou být předávány dva operátory, z nichž první je typu `void*` a druhý typu `const std::nothrow_t&`. Není definováno přidávání dalších parametrů tomuto operátoru (narozdíl od operátoru `new`, jehož specifikace je v bodě 5.3.4, paragrafu 11 standardu ISO/IEC 14882:1998).

  Z tohoto důvodu nelze operátor `delete` přetížit a přidat mu další parametry. K volání definované verze globálního operátoru dochází i při uvolňování paměti mimo soubory, ke kterým byla připojena knihovna check, na rozdíl od operátoru `new` nelze určit, ve kterém místě programu se operátor `delete` volá. Docházelo tak k nesprávným chybovým výpisům knihovny check i pro paměť, která nebyla zaznamenána do seznamu, tj. například při použití třídy pro práci se soubory `fstream`, která si alokuje vlastní paměť.

  Proto bylo rozhodnuto, že v situaci, kdy knihovna narazí na paměť, která byla alokována a není v seznamu paměti, nebude knihovna vypisovat chybu a neukončí program, ale bude pouze vypsáno varovné hlášení.

  Jiným řešením tohoto problému mohla být definice funkce alternativní k operátoru `delete`, jako např. `my_delete()`. Ta by uvolňovala paměť v kontrolované části programu, a volání operátoru `delete` by nezpůsobovalo žádné hlášení. Používáním takovéto alternativní funkce by se ale začínající programátoři mohli naučit nesprávné návyky, proto bylo zvoleno výše popsané řešení.

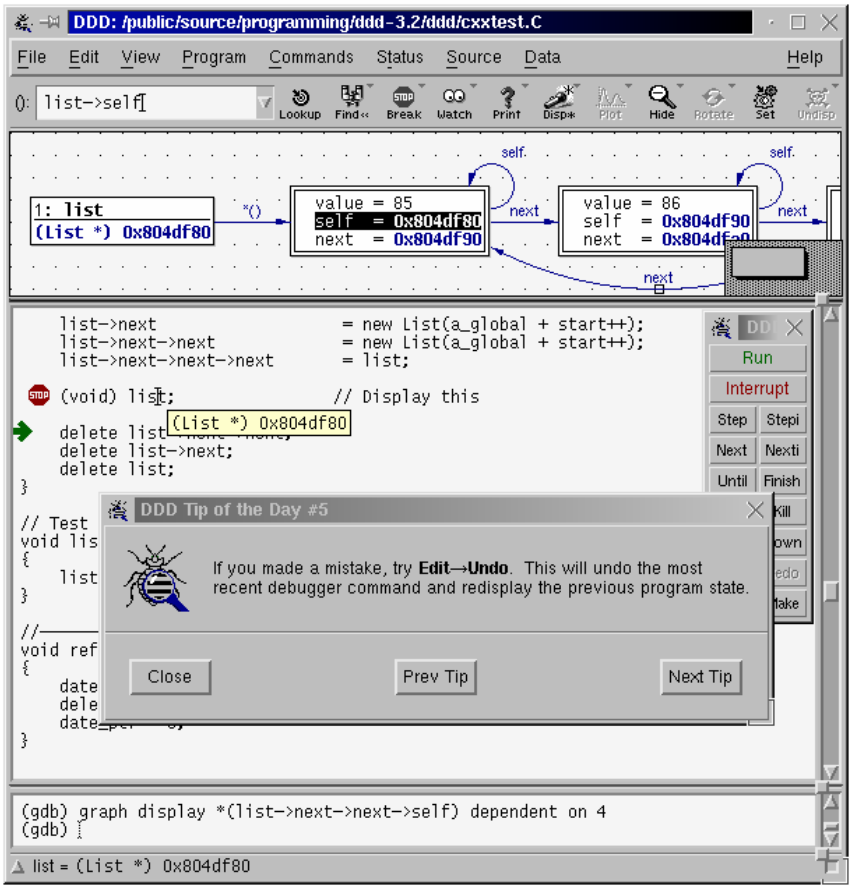
4. KNIHOVNA ADTCHECK.H

Ve cvičení předmětu BPPC je jednou z úloh tvorba lineárně vázaného seznamu, který patří mezi abstraktní datové typy. Zvládnutí tvorby tohoto seznamu, ale i jiných podobných typů patří mezi základní programátorské dovednosti.

Při návrhu knihovny adtcheck.h byl vytyčen její účel, tj. kontrola práce programátora při práci s abstraktními datovými typy. Při tom by ale měla knihovna zachovávat jednoduchost ovládání a měla by ovlivňovat napsaný program minimálně.

4.1 EXISTUJÍCÍ NÁSTROJ PRO VIZUALIZACI DAT - DDD

Existujícím nástrojem pro ladění programů a vizualizaci dat je DDD – Display data debugger. Tento nástroj patří k projektu GNU, je tedy bezplatný. Je však vázán na platformu GNU/Linux nebo Unix.



Obrázek 4-1 Náhled okna nástroje DDD

Pro kontrolu programu pomocí DDD je třeba program přeložit s pomocnými ladicími informacemi pro debugger, tj. je nutné do příkazu překladu přidat parametr `-g`. Přeložený program je pak přidán jako parametr při spouštění nástroje DDD. Při ladění je třeba otevřít zdrojové kódy programu, DDD je tak schopen detekovat chování aplikace pro každou část zdrojového kódu. Tento debugger umožňuje kontrolu i binárního kódu bez připojení zdrojových souborů, možnosti ladění se tak ale omezí.

Nástroj DDD umožňuje pozastavit běh programu, zobrazit a měnit data uložená v proměnných, pomáhá odhalit příčiny případného pádu programu. DDD má nástroj pro vizualizaci obsahu proměnných, pokud se jedná o ukazatel, umožňuje navíc vykreslení jeho cíle spolu s šipkou. Lze tak postupně názorně rekonstruovat celou datovou strukturu.

Nástroj DDD umožňuje kontrolu aplikací pro několik jazyků, např. C, C++, Python, Pearl, Java, nebo Bash.

4.2 ROZSAH KONTROLY KNIHOVNY ADTCHECK.H

V rámci cvičení předmětu BPPC mají studenti implementovat jednosměrně vázaný lineární seznam a jako nepovinnou úlohu mají naprogramovat třídu pro práci s binárním stromem. Funkčnost knihovny byla navržena především s ohledem na obvyklé problémy, které mají studenti při zvládnutích těchto úloh. Kromě jednosměrně vázaného lineárního seznamu podporuje ale knihovna také kontrolu obousměrně vázaného a kruhového seznamu.

Obsahem kontroly je ověření správnosti topologie ADT. Jedná se tedy především o kontrolu ukazatelů na další prvky seznamu, tj. ověření, že ukazatel obsahuje buď adresu prvku struktury, nebo hodnotu `NULL`.

Další fází kontroly je ověření správnosti struktury ADT. Jedná se zejména o testování, jestli struktura neobsahuje cyklus. Dále je testováno, zda-li lze dosáhnout všech registrovaných prvků.

Při návrhu knihovny bylo také uvažováno o kontrole práce s operátory, o této problematice je dále hovořeno v kapitole 4.5.

4.3 PRINCIP FUNGOVÁNÍ KNIHOVNY

Knihovnu tvoří soubor `<adtcheck.h>`. V něm je definována šablona objektu `template<TClass, TClass> adtcheck;`, která zajišťuje veškeré kontrolní funkce. Je zde definováno i několik pomocných struktur. Třída `adtcheck` je definována jako šablona, a díky tomu ho lze používat pro libovolné druhy datových typů.

Uživatelské rozhraní třídy tvoří tři metody. Dvě z nich slouží pro manipulaci s prvky ve struktuře a jedna je určena pro jejich kontrolu. Pro zařazení prvku do seznamu kontrolovaných je definována metoda `registruj()`. Tato metoda je přetížená a lze ji volat s různým počtem parametrů. Hlavička této metody vypadá takto:

```
void registruj (TPrvek*, TTyp_dat**, TPrvek*,…);
```

Prvním parametrem musí být vždy adresa registrovaného prvku, druhým parametrem ukazatel na ukazatel na data, která jsou uložena ve struktuře. Dále následují ukazatele na ukazatele dalších prvků, které mohou být jeden, dva nebo tři. Použití ukazatele na ukazatel je v těchto případech nutné – použitím prostého ukazatele by se jednalo o předání parametrů hodnotou a nebylo by možné načítat aktuální odkazy v ukazatelích po změnách v ADT.

Registrované adresy prvků a dat v ADT se ukládají do datového kontejneru typu `vector` pojmenovaného do definované šablony struktury. Deklarace této proměnné vypadá takto:

```
std::vector<TSeznam_adr<TPrvek, TData> > adresy;
```

Každému prvku je přiděleno jedinečné číslo, které je uloženo spolu s ním. Do dalšího kontejneru typu vector jsou ukládány adresy ukazatelů. Deklarace tohoto vektoru vypadá takto:

```
std::vector<TSeznam_uk<TPrvek> > ukazatele;
```

Každé adrese ukazatele je přiděleno stejné číslo, jaké má příslušný registrovaný prvek. Díky tomu lze vždy jednoznačně určit, které ukazatele náleží kterým prvkům.

Pro odebrání prvku ze seznamu je definována metoda `odeber()`. Tato metoda má vždy jen jeden parametr, a to adresu odebíraného prvku. Hlavička této metoda vypadá takto:

```
void odeber (TPrvek*);
```

Při volání funkce je ve vektoru adres odstraněn příslušný záznam a podle příslušného čísla jsou nalezeny a smazán také ukazatele tohoto prvku.

Kontrola uloženého ADT je spouštěna metodou `zkontroluj()`, jejíž hlavička vypadá takto:

```
void zkontroluj(void);
```

Kontrola ADT je probíhá v několika fázích. V první je ověřováno, jestli všechny registrované ukazatele obsahují platné hodnoty, tj. buď adresu některého registrovaného prvku, nebo hodnotu `NULL`, která označuje prázdný ukazatel. V případě, že tento test nenalezne chybu, přechází se k další fázi a je rozpoznáváno, o jaký datový typ se jedná. Je vyhodnocován počet ukazatelů na další prvky a jejich obsah. Následují kontroly pro daný typ ADT a výpis prvků. Všechny vypsané prvky jsou zaznamenávány do seznamu. Díky němu je pak kontrolováno, zda-li struktura neobsahuje cyklus, a zda-li byly správně vypsány všechny prvky v ADT. Výpis se v případě poškozené struktury nemusí vždy podařit, knihovna ale nabízí výpis prvků s jejich ukazateli. Tento výpis navíc lokalizuje a označí místo předpokládané chyby. Kontrolní výpis knihovny pro binární strom je naznačen na obrázku 2-1.

4.4 POUŽÍVÁNÍ KNIHOVNY ADTCHECK.H

Knihovnu `adtcheck.h` tvoří pouze jeden soubor. Při používání knihovny je třeba zkopírovat uvedený soubor k překládanému projektu. Dále je nutné přeložit knihovnu společně s projektem. V MS Visual C++ lze připojit soubory k projektu pomocí tlačítka v hlavním menu „Add new item“ a nabídky „Add existing item“. V projektech s automaticky generovaným makefile to znamená pouze připojit soubor k projektu, v jiných případech je třeba do makefile doplnit příkaz pro připojení souborů knihovny do projektu.

Je nutné připojit hlavičkový soubor k programu direktivou preprocesoru `#include`. Tzn. je nutné mít v souboru, který obsahuje definici funkcí nad ADT nebo v jeho hlavičkovém souboru, příkaz

```
#include "adtcheck.h".
```

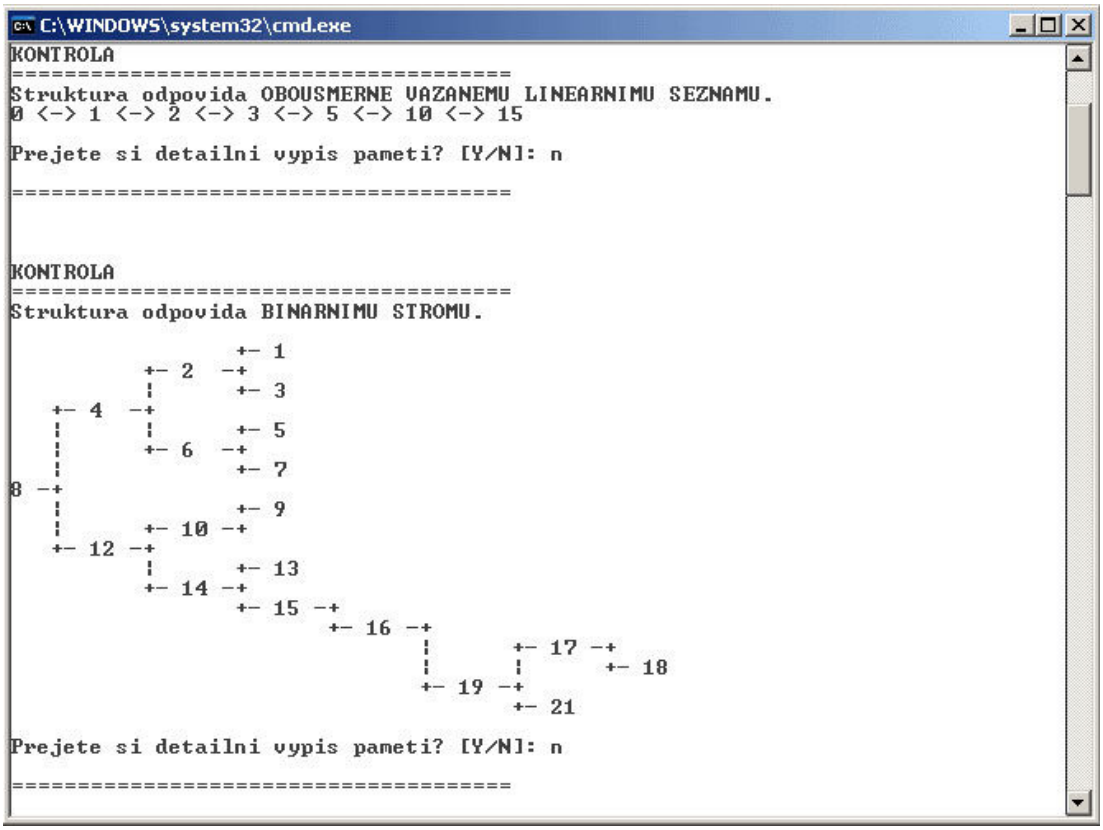
Knihovna umožňuje kontrolu datových typů jednosměrně a obousměrně vázaný lineární seznam, kruhový seznam a binární strom. Pro kontrolu datového typu slouží šablona třídy `adtcheck`. V souboru s definicí funkcí nad ADT je nutné vytvořit proměnnou této třídy. Prvním parametrem šablony je typ objektu prvku ADT, druhým je datový typ, který je v ADT uložen. Deklarace proměnné může vypadat takto:

```
adtcheck<TPrvek, int> kont;
```

V proměnné třídy `adtcheck` je vytvářen seznam všech prvků ADT. Pro práci jsou definovány 3 metody, které je nutné použít. Jsou to:

- **void** `registruj (TPrvek*, TTyp_dat**, TPrvek*);`
  void `registruj (TPrvek*, TTyp_dat**, TPrvek*, TPrvek*);`
  metoda, která registruje prvek do seznamu prvků. Tuto metodu je nutné volat pro každý prvek v ADT, v jiném případě bude docházet k chybovým hlášením knihovny. Prvním parametrem musí být vždy adresa registrovaného prvku, druhým parametrem ukazatel na ukazatel na data, která jsou uložena ve struktuře. Dále následují ukazatele na ukazatele dalších prvků, které mohou být jeden nebo dva. Příklady registrace prvků:
  ```
  kont.registruj(NewUk, &NewUk->data, &NewUk->Uk_ dalsi);
  kont.registruj(Ptr, &Ptr->data, &Ptr->Uk1, &Ptr->Uk2);
  ```
- **void** `odeber (TPrvek*);` – metoda, která odebere prvek s danou adresou ze seznamu kontrolovaných prvků;
- **void** `zkontroluj(void);` – metoda, která, která spouští kontrolní mechanizmus nad kontrolovanou datovou strukturou.

Knihovna zkontroluje správnost ADT na základě zadaných ukazatelů. Pokud je v pořádku, pokusí se strukturu vykreslit. Všechny výpisy jsou směřovány na standardní chybový výstup.



Obrázek 4-2 Příklad kontrolního výpisu knihovny adtcheck.h

4.5 ZAJÍMAVÉ PROBLÉMY ŘEŠENÉ PŘI IMPLEMENTACI

Při implementaci knihovny bylo třeba vyřešit některé problémy a upravit funkce knihovny dle technických možností překladače MS Visual C++:

- Použití šablon - většina omezení knihovny byla způsobena faktem, že třída `adtcheck` je šablonou. Překladač předkládá metodu šablony až ve chvíli, kdy narazí na její použití. Překladač tedy dříve nemůže vyřešit odkazy na tyto metody. To způsobuje například problém s deklarací globálních objektů pomocí šablon, tj. objektů s klíčovým slovem `extern`, kdy lze deklarovat takovýto objekt, ale překladač neumí vyřešit odkazy na jeho metody.

  Důsledkem toho nebylo možné implementovat kontrolu práce s operátory nebo využít možnosti přetížení globálního operátoru `delete` pro automatické odstraňování prvků ze seznamu kontrolovaných ukazatelů knihovny. Pro tyto funkce by byl nutný přístup k seznamu prvků, který by ale musel být globální, což

vzhledem k uvedeným faktům nelze zajistit.

I přes uvedené problémy je použití šablon velmi výhodné, neboť umožňuje práci s libovolnými datovými typy.

- Organizace knihovny – norma jazyka C++ neumožňuje oddělit deklaraci a definici funkcí šablon. Obojí musí být umístěno v hlavičkovém souboru, který se tak stává velmi nepřehledný. Tento problém řeší dle standardu klíčové slovo `export`, které oddělení deklarace a definice umožňuje. Toto klíčové slovo ale není v současné době překladači jazyka C++ podporováno. Knihovnu `adtcheck.h` tak tvoří pouze jeden soubor, kde jsou uvedeny deklarace i definice tříd a jejich metod.
- Měření délky řetězce libovolného typu – pro zajištění správného formátování výpisů bylo nutno vyřešit problém, jak měřit počet znaků proměnné libovolného datového typu. Tento problém byl vyřešen pomocí proudu `stringstream`, který umí pracovat s libovolnými datovými typy a dále pomocí řetězce `string`, který pomocí metody `length()` dokáže určit počet znaků řetězce. Zdrojový kód funkce pro měření počtu znaků je tento:

```
unsigned int pocet_znaku(TData promenna) {
      std::stringstream ss;
      std::string str;
      ss << promenna;   // konverze promenne do promenne strem
      str = ss.str();   // konverze streamu na rezezec
      return str.length();// vraci delku retezce
      }
```

5. ZÁVĚR

Předmětem práce je návrh a implementace pomůcek pro předmět Praktické programování v C++, který se učí na Fakultě elektrotechniky a komunikačních technologií v Brně. Bylo vycházeno z předpokladu, že studenti mají základní znalosti programování v jazyce C. Úvod práce obsahuje základní fakta o programování v jazycích C a C++. Jsou zde shrnuty i rozdíly mezi těmito programovacími jazyky. Dále je popsáno několik překladačů a vývojových prostředí pro jazyky C a C++, se kterými se mohou studenti setkat. Důraz je kladen především na vývojové prostředí MS Visual C++, které se používá pro výuku předmětu BPPC, a jeho vztah ke standardům přijatých organizací ISO.

V rámci práce byla prostudována osnova předmětu BPPC, byly analyzovány části, kde mívají studenti obvykle problémy. Z těchto závěrů byl navrženy dvě pomůcky, které studentům pomohou s pochopením látky předmětu, tj. knihoven pro jazyk C a C++. Praktickou částí byla implementace navržených pomůcek.

Knihovna `check.h` kontroluje práci programátora s dynamicky alokovanou pamětí v jazycích C i C++ a se soubory v jazyce C. Rozšíření funkčnosti pro práci se soubory pro standardní knihovnu STL jazyka C++ se ukázalo jako problematické – implementace této knihovny je velmi komplexní a pro zajištění úplné kompatibility se všemi metodami pro práci se soubory by bylo nutné předefinovat několik rozsáhlých tříd, což přesahuje rámec této práce. Knihovna `check.h` byla otestována na demonstračních příkladech přejatých ze cvičení kurzu BPPC a byla ověřena její bezchybná funkčnost. Knihovna i testovací projekty jsou součástí přiloženého CD.

Knihovna `adtcheck.h` kontroluje implementaci abstraktních datových typů, se kterými se studenti setkávají při výuce předmětu BPPC. Knihovna umožňuje zobrazování jednotlivých datových typů, a také výpisy registrovaných prvků, a to včetně obsahu jejich ukazatelů. Testování bylo prováděno na všech ADT, které knihovna podporuje, testy byly prováděny pro různé způsoby návrhu struktur, tj. funkcionální, zapouzdření ve struktuře i plně objektový návrh. Pro všechny uvedené způsoby funguje knihovna bezchybně. Knihovna i testovací projekty jsou součástí přiloženého CD.

Obě knihovny byly navrženy s ohledem na jejich složitost a použitelnost při výuce a byly testovány na příkladech přejatých z předmětu BPPC.

Na závěr byly vytvořeny webové stránky obsahující odkazy na obě uvedené knihovny a podrobné návody k jejich použití. Dále na nich lze nalézt demonstrační projekty, na kterých byly uvedené knihovny testovány. Tyto stránky jsou součástí přiloženého CD.

6. SEZNAM POUŽITÉ LITERATURY

[1] Virius Miroslav: Jazyky C a C++ kompletní kapesní průvodce programátora, nakladatelství Grada, 2006

[2] Bruce Eckel: Myslíme v jazyku C++, nakladatelství Grada, 2000

[3] Petr Šaloun: Programovací jazyk C, nakladatelství Neokortex, 1999

[4] Piotr Wróblewski: Algoritmy, Datové struktury a programovací techniky, nakladatelství Computer Press, 2004

[5] Václav Kadlec: Učíme se programovat v Borland C++ Builder a jazyce C++, nakladatelství Computer Press, 2004

[6] ISO/IEC 14882:1998: Standard jazyka C++

[7] Jan Němec: http://www.linuxsoft.cz - kurz programování v jazyce C++

[8] Petr Bílek: http://www.sallyx.org/sally/c - kurz programování v jazycích C a C++

[9] Jose María Gómez Vergara: http://www.gnu.org/software/ddd/ - internetová dokumentace projektu DDD

[10] Cerion Armour-Brown, Jeremy Fitzhardinge a kolektiv: http://valgrind.org/ - internetová dokumentace projektu Valgrind

[11] http://cs.wikipedia.org/wiki/C++

[12] http://msdn2.microsoft.com - dokumentace pro vývojáře v MS Visual C++

[13] http://www.cplusplus.com - dokumentace jazyka C++ a knihovny STL

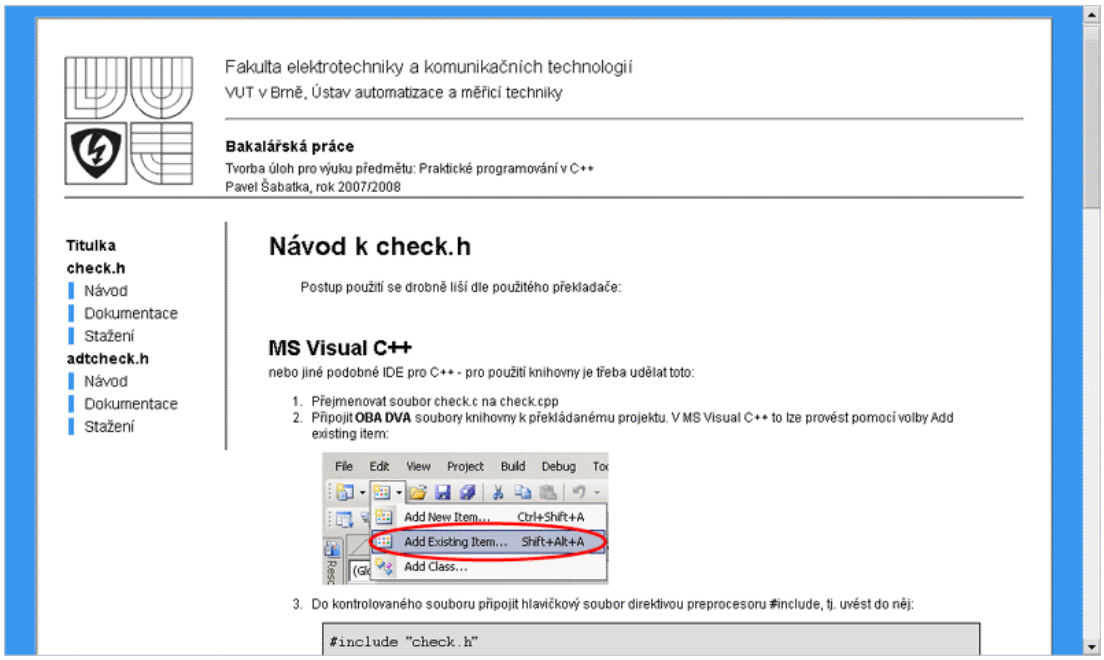
7. PŘÍLOHY

SEZNAM PŘÍLOH

OBSAH CD

- Bakalářská práce v elektronické podobě
- Knihovny check.h a adtcheck.h
- Návody k použití
- Testovací projekty v MS Visual C++ 2005
- Webová prezentace obou knihoven

Příloha A: Náhled webové prezentace knihoven check.h a adtcheck.h

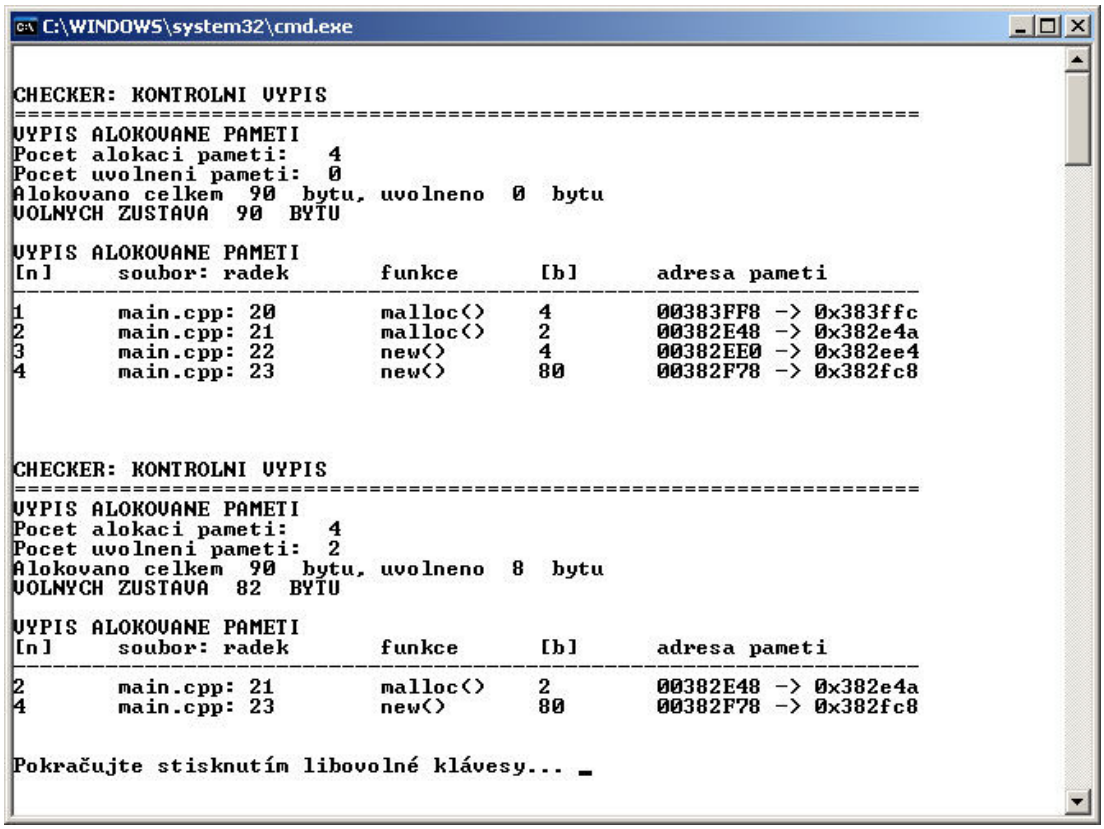


Příloha B: Testovací úloha knihovny check.h – alokace paměti

Základní operace při práci s pamětí. V příkladu jsou záměrně vytvořeny chyby, které jsou označeny. Testovací soubor `main.cpp` vypadá takto:

```
#include "check.h"
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <fstream>

using namespace std;

int main(void) {

      int i;

/*****************************
*   testovani new a delete    *
*****************************/
      int *polePtr1, *polePtr3;
      char *polePtr2;
      float *polePtr4;

      polePtr1 = (int *) malloc(sizeof(int));
      polePtr2 = (char *) malloc(2 * sizeof(char));
      polePtr3 = new int;
      polePtr4 = new float[20];

      memory_stat();          // kontrolni vypis

                              // zapis do alokovaneho pole
      for (i=0; i<20; i++) polePtr4[i] = 2^i;

      free (polePtr1);
      // free (polePtr2);     // CHYBA - TATO PAMĚŤ NEBUDE UVOLNĚNA
      delete polePtr3;
      // delete[] polePtr4;   // CHYBA - TATO PAMĚŤ NEBUDE UVOLNĚNA



      /*********************************************
      * konec testovani, nechavam neuvolnenou pamet, *
      * coz checker vypise jako chybu, a korektne    *
      * ukonci program                               *
      *********************************************/


      return 0;

}
```

Po všech alokacích v testovacím programu volám funkci `memory_stat()`, která vypíše aktuální stav paměti. Na konci programu je automaticky vypsán stav alokované paměti. Okno testovacího programu s kontrolní knihovnou `check.h` je na obrázku 7-1.



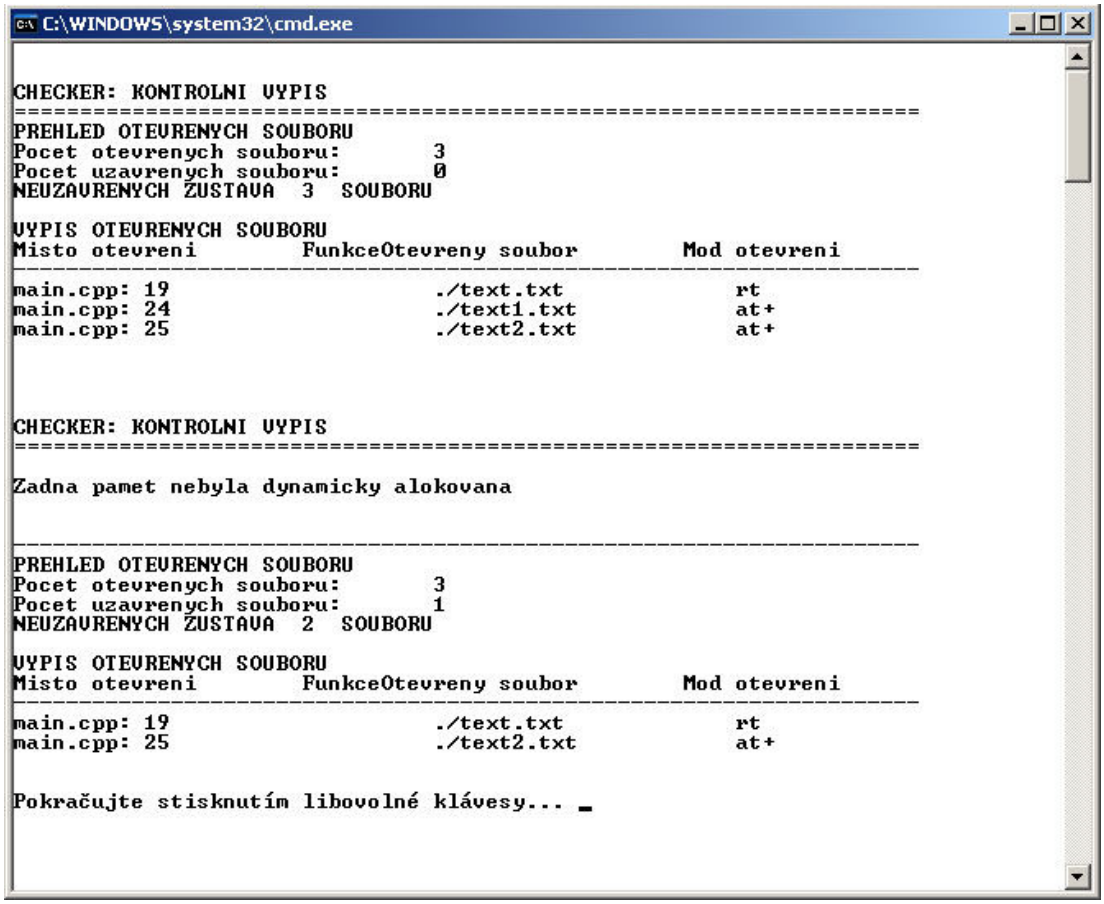
Obrázek 7-1 Kontrolní výstup knihovny check.h pro test alokace

Příloha C: Testovací úloha knihovny check.h – práce se soubory

Základní operace při práci se soubory. V příkladu jsou záměrně vytvořeny chyby, které jsou označeny. Testovací soubor `main.cpp` vypadá takto:

```
#include "check.h"
#include <stdio.h>
#include <iostream>

using namespace std;

int main (void) {

/*****************************
* testovani otevreni souboru *
*****************************/

      FILE *in = NULL;
      FILE *out1 = NULL, *out2 = NULL;
      char znak;

      in   = fopen ("./text.txt", "rt");
      if (in == NULL) {
            return 0;
            }
      out1 = fopen ("./text1.txt", "at+");
      out2 = fopen ("./text2.txt", "at+");

      file_stat();

      znak = fgetc (in);                  // zkopirovani souboru
      while (znak != EOF) {
            fputc (znak, out1);
            fputc (znak, out2);
            znak = fgetc (in);
            }
      // fclose(in);          // CHYBA - TENTO HANDLE NEBUDE UVOLNĚN
      fclose(out1);
      // fclose(out2);        // CHYBA - TENTO HANDLE NEBUDE UVOLNĚN

      /************************************************
      * konec testovani, nechavam neuzavrene soubory, coz *
      * checker vypise jako chybu, ale chyby opravi a     *
      * korektne ukonci program                           *
      ************************************************/

      return 0;

      }
```

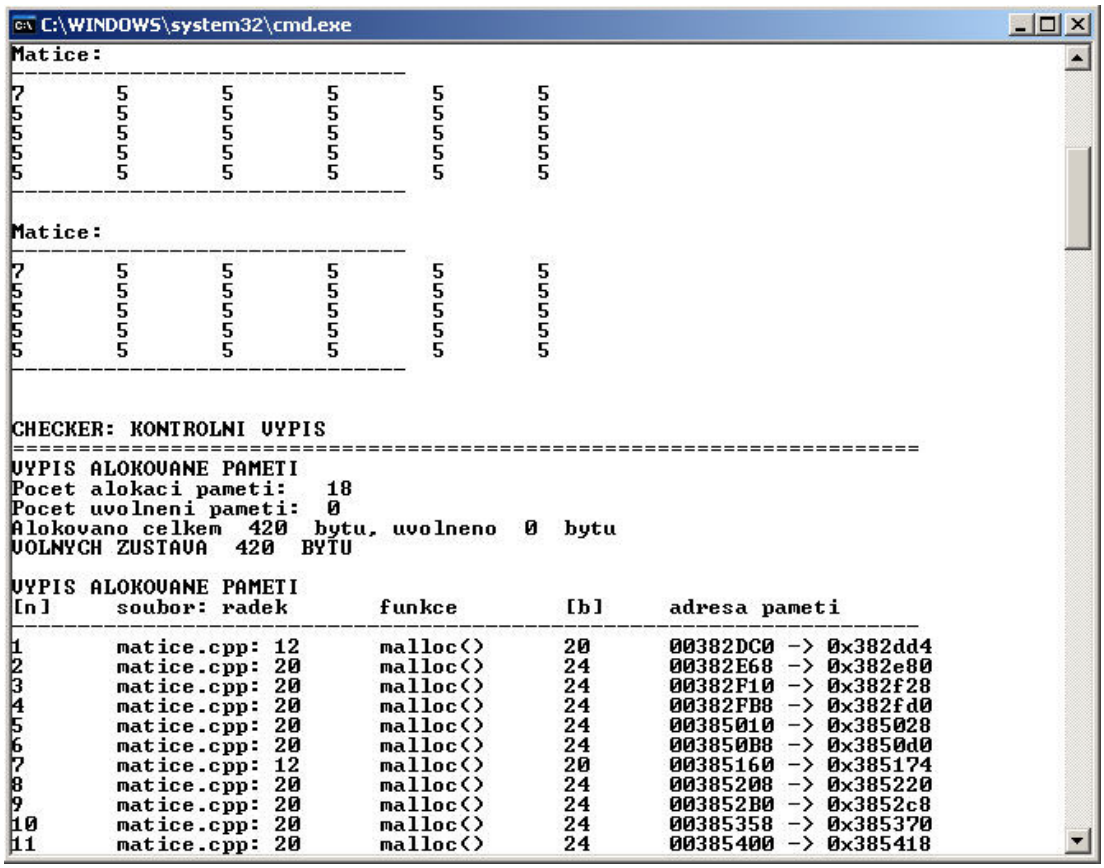
Po otevření všech souborů v testovacím programu je volána funkce `file_stat()`, která vypíše aktuální informace o otevřených souborech. Na konci programu je automaticky vypsán stav otevřených souborů. Okno testovacího programu s kontrolní knihovnou `check.h` je na obrázku 7-2.



Obrázek 7-2 Kontrolní výstup knihovny check.h pro test práce se soubory

Příloha D: Testovací úloha knihovny check.h – práce s maticemi

Implementace funkcí pro práci s maticemi. Tento příklad je součástí cvičení předmětu BPPC. Zdrojové soubory a projekt v MS Visual C++ je na přiloženém CD. Kontrolní výpis knihovny `check.h` pro tuto úlohu je na obrázku 7-3.

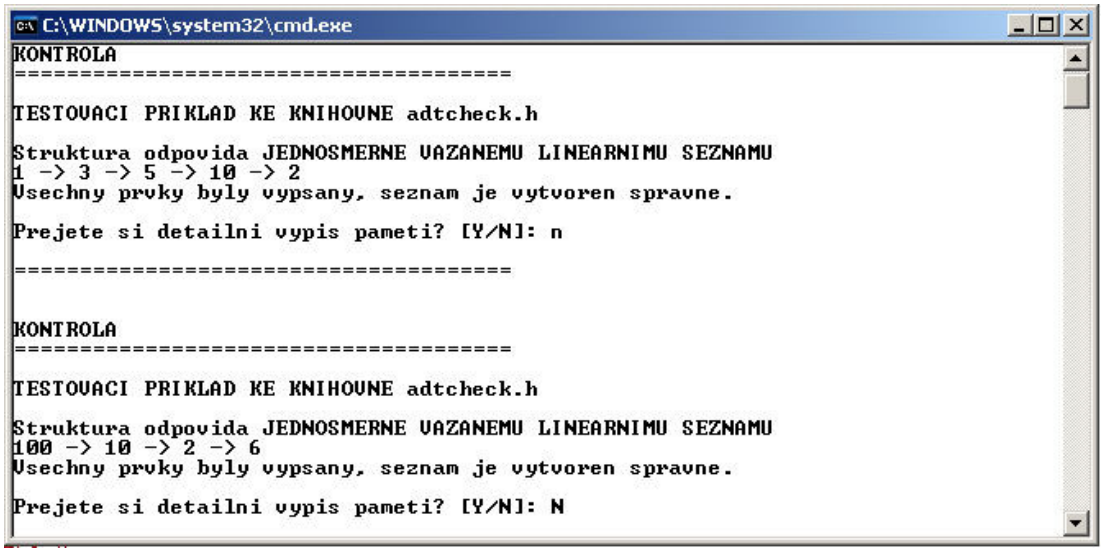


Obrázek 7-3 Kontrolní výstup knihovny check.h pro příklad práce s maticemi

Příloha E: Testovací úloha knihovny adtcheck.h – lineární seznam

Implementace lineárního seznamu a jeho kontrolní výpis. Zdrojový soubor `main.cpp` a kontrolní výstup knihovny `adtcheck.h` vypadají takto:

```
#include "adt.h"

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

using namespace lin_seznam_1sm;
      LSPrvek* seznam = NULL;

      LS_insert(&seznam, 1);        // vlozeni prvku s hodnotou 1
      LS_insert(&seznam, 3);        // vlozeni prvku s hodnotou 3
      LS_insert(&seznam, 5);        // vlozeni prvku s hodnotou 5
      LS_insert(&seznam, 10);       // vlozeni prvku s hodnotou 10
      LS_insert(&seznam, 2);        // vlozeni prvku s hodnotou 2

      LS_print(seznam);             // vypis a kontrola seznamu

      LS_insert(&seznam, 6);        // vlozeni prvku s hodnotou 6
      LS_change(&seznam, 3, 100);   // zmena hodnoty prvku z 3 na 100
      LS_dispose(&seznam, 1);       // smazani prvku s hodnotou 1
      LS_dispose(&seznam, 5);       // smazani prvku s hodnotou 5

      LS_print(seznam);             // vypis a kontrola seznamu

      LS_dispose_list(&prvni);      // odstraneni seznamu

      return 0;
}
```

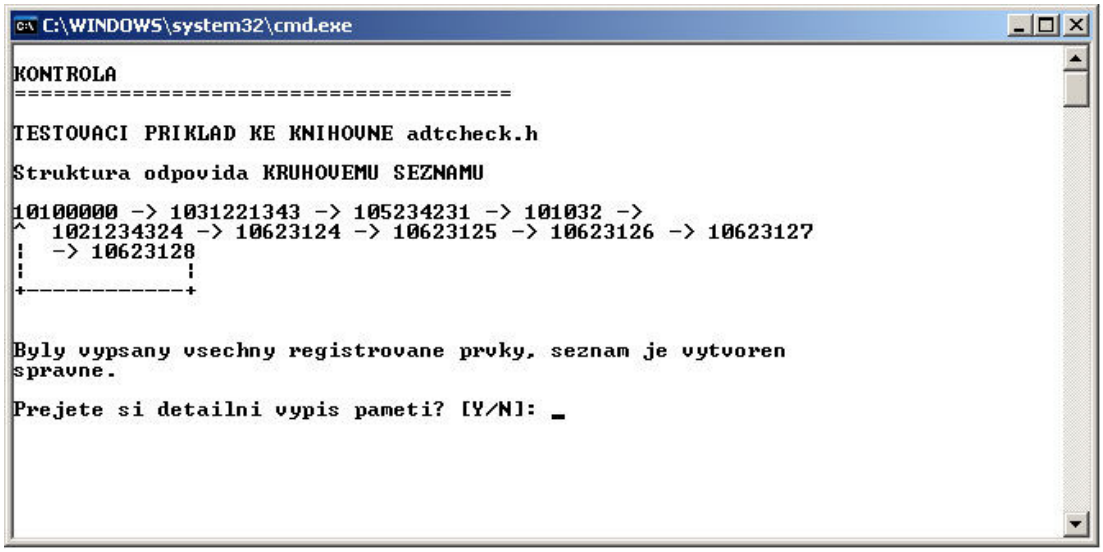


Obrázek 7-4 Kontrolní výstup knihovny adtcheck.h pro lineární seznam

Příloha F: Testovací úloha knihovny adtcheck.h – kruhový seznam

Implementace kruhového seznamu a jeho kontrolní výpis. Zdrojový soubor `main.cpp` a kontrolní výstup knihovny `adtcheck.h` vypadají takto:

```
#include "adt.h"

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

	using namespace seznam_kruh;
	LSPrvek* prvni_k = NULL;

	LS_insert(&prvni_k, 10100000);
	LS_insert(&prvni_k, 1031221343);
	LS_insert(&prvni_k, 105234231);
	LS_insert(&prvni_k, 101032);
	LS_insert(&prvni_k, 1021234324);
	LS_insert(&prvni_k, 10623124);
	LS_insert(&prvni_k, 10623125);
	LS_insert(&prvni_k, 10623126);
	LS_insert(&prvni_k, 10623127);
	LS_insert(&prvni_k, 10623128);

	LS_print(prvni_k);

	LS_dispose_list(&prvni_k);
	return 0;
}
```

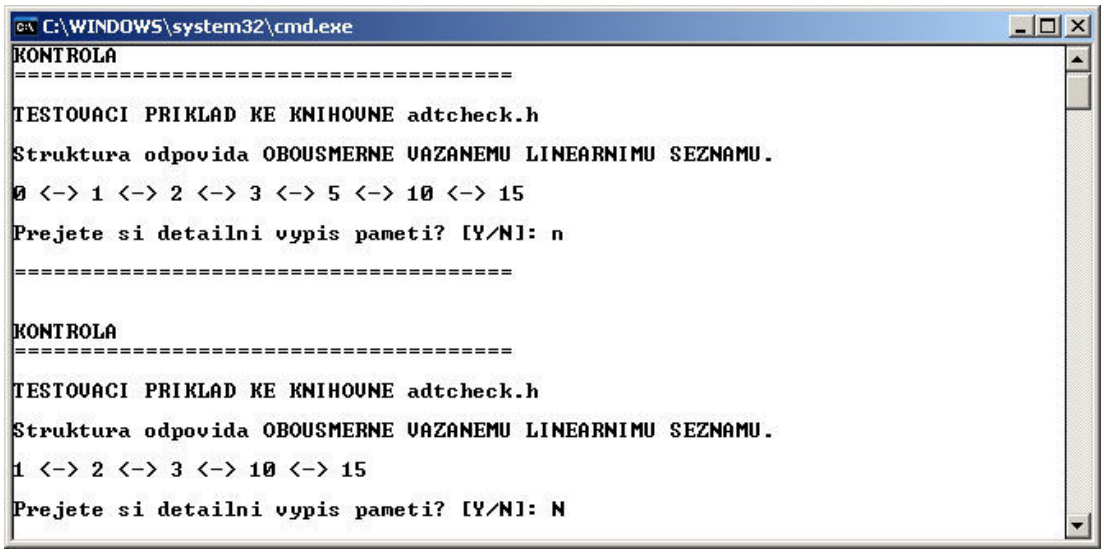


Obrázek 7-5 Kontrolní výstup knihovny adtcheck.h pro kruhový seznam

Příloha G: Testovací úloha knihovny adtcheck.h – obousměrně vázaný seznam

Implementace obousměrně vázaného seznamu a jeho kontrolní výpis. Seznam automaticky řadí prvky dle velikosti. Zdrojový soubor `main.cpp` a výstup knihovny `adtcheck.h` vypadají takto:

```
#include "adt.h"

int main (int argv, const char* argc[]) {
	using namespace lin_seznam_2sm;
	LS2List L;
	LS2_init (&L);                  // inicializace seznamu

	LS2_insert(&L, 2);              // vlozeni prvku s hodnotou 2
	LS2_insert(&L, 10);             // vlozeni prvku s hodnotou 10
	LS2_insert(&L, 5);              // vlozeni prvku s hodnotou 5
	LS2_insert(&L, 3);              // vlozeni prvku s hodnotou 3
	LS2_insert(&L, 15);             // vlozeni prvku s hodnotou 15
	LS2_insert(&L, 0);              // vlozeni prvku s hodnotou 0
	LS2_insert(&L, 1);              // vlozeni prvku s hodnotou 1

	LS2_print(&L);                  // kontrolni vypis

	LS2_dispose(&L, 5);             // odstraneni prvku s hodnotou 5
	LS2_dispose(&L, 0);             // odstraneni prvku s hodnotou 0

	LS2_print(&L);                  // kontrolni vypis

	LS2_dispose_list(&L);           // zruseni celeho seznamu
	return 0;
}
```

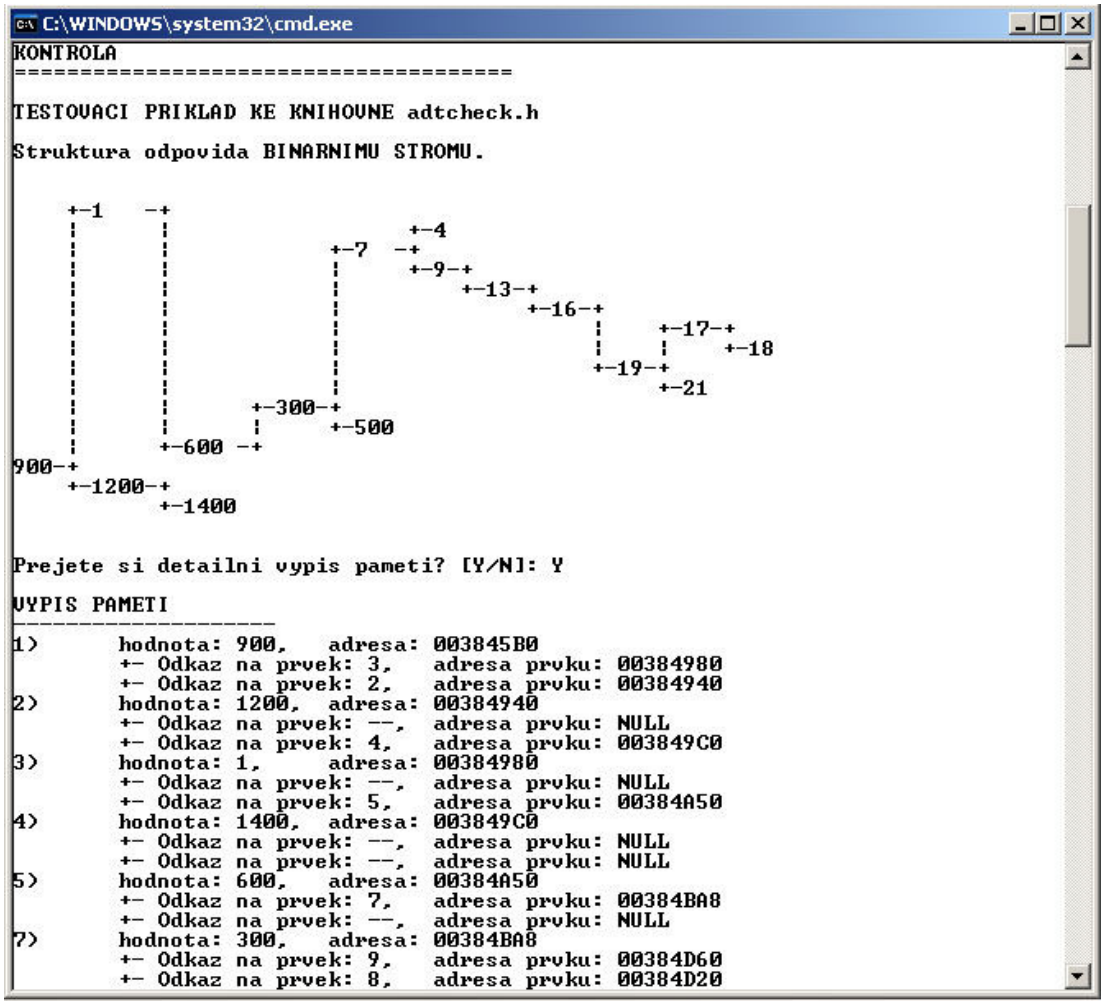


Obrázek 7-6 Kontrolní výstup knihovny adtcheck.h pro obousměrně vázaný seznam

Příloha H: Testovací úloha knihovny adtcheck.h – binární strom

Implementace binárního stromu a jeho kontrolní výpis. Zdrojový soubor `main.cpp` a výstup knihovny `adtcheck.h` vypadají takto:

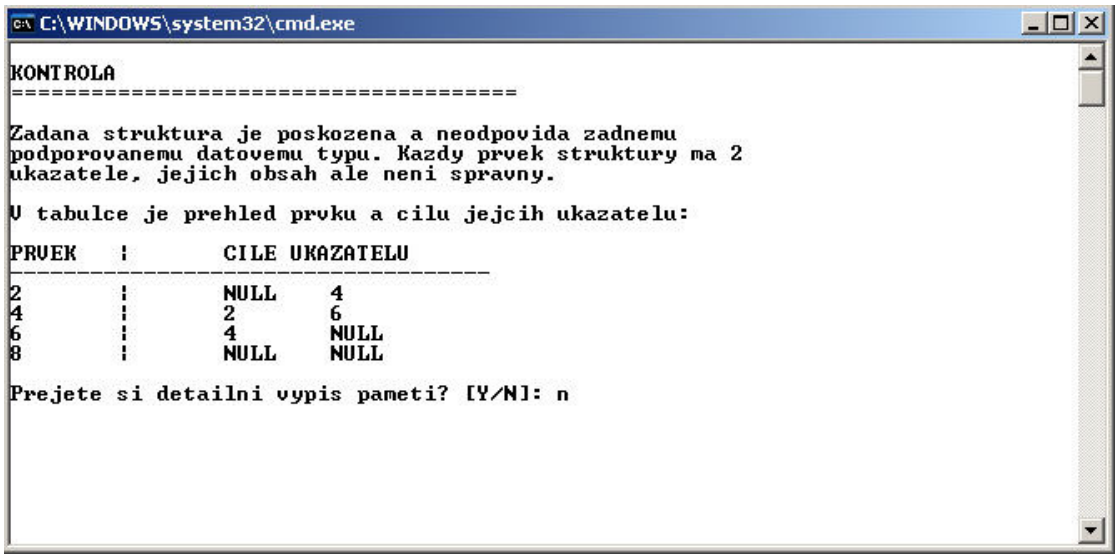
```
#include "adt.h"

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

	using namespace BT;

	BS strom;

	strom.insert(900);
	strom.insert(1200);
	strom.insert(2);
	strom.insert(1400);
	strom.insert(600);
	strom.insert(1);
	strom.insert(300);
	strom.insert(500);
	strom.insert(7);
	strom.insert(9);
	strom.insert(13);
	strom.insert(15);

	strom.print();          // kontrolni vypis

	strom.insert(16);
	strom.insert(19);
	strom.insert(21);
	strom.insert(17);
	strom.insert(18);
	strom.dispose(15);
	strom.insert(4);
	strom.dispose(2);

	strom.print();          // kontrolni vypis

	strom.dispose_list();
	return 0;
}
```



Obrázek 7-7 Kontrolní výstup knihovny adtcheck.h pro binární strom

Příloha I: Testovací soubor

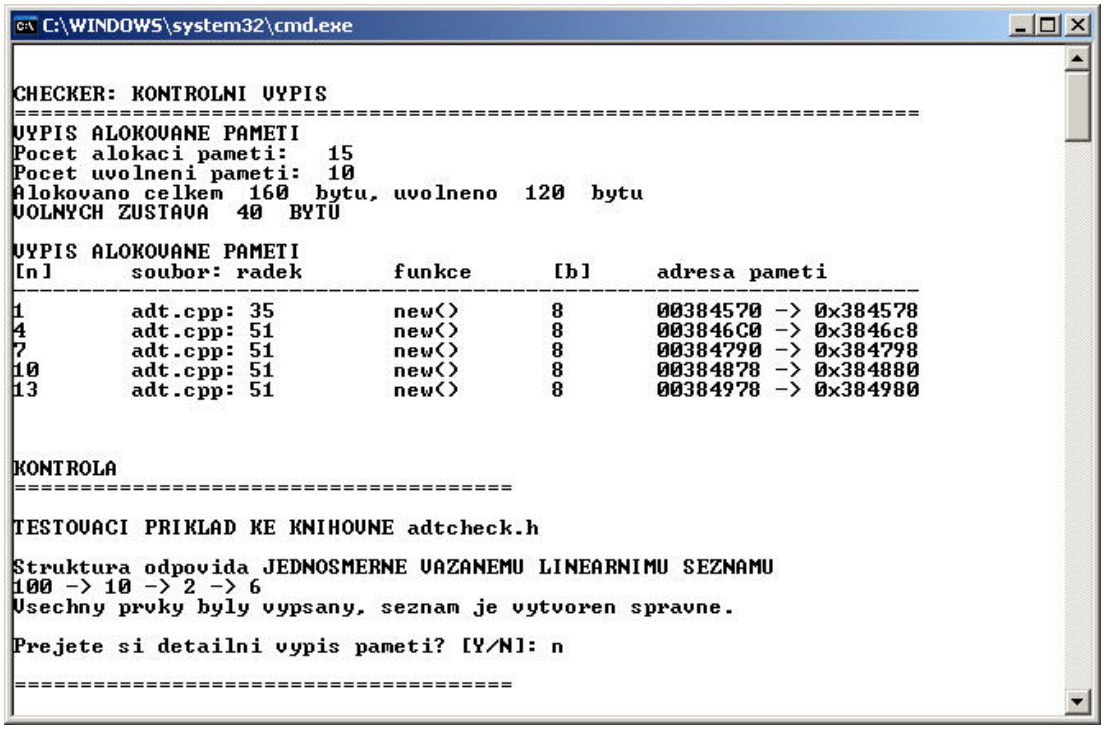
Testovací projekt pro modifikaci datových typů, tvorbu chyb v nich a jejich kontrolní výpis. Výpis je na obrázku 7-8. Celý projekt je součástí přiloženého CD.



Obrázek 7-8 Kontrolní výstup knihovny adtcheck.h pro poškozenou strukturu

Příloha J: Test kompatibility knihoven check.h a adtcheck.h – lineární seznam

Implementace lineárního seznamu. Zvládnutí tohoto příkladu je součástí cvičení předmětu BPPC. Kontrolní výstupy knihoven `check.h` a `adtcheck.h` jsou na . Projekt tohoto testu je vypálen na přiloženém CD.



Obrázek 7-9 Test kompatibility knihoven check.h a adtcheck.h

Příloha K: Popis souboru check.h

MAKRA:

`_CRT_SECURE_NO_DEPRECATE 1` - makro zamezující překladači MS Visual C++ zobrazovat chybová hlášení

`CHECKER_OUT stderr` – přesměrování výstupu knihovny
`__DELKA_NAZVU__ 50` – nastavení maximální délky názvu souboru
`__DELKA_NAZVU_FCE__ 20` – nastavení maximální délky názvu funkce
`__n__` - zobrazování sloupce „pořadí alokace“ při kontrolním výpisu
`__SOUBOR_RADEK__` - zobrazování sloupce „jméno souboru“ při kontrolním výpisu
`__funkce__` - zobrazování sloupce „jméno funkce“ při kontrolním výpisu
`__b__` - zobrazování sloupce „alokovaná velikost“ při kontrolním výpisu
`__adresa_pameti__` - zobrazování sloupce „adresa paměti“ při kontrolním výpisu

MAKRA FUNKCÍ A OPERÁTORŮ:

```
#define malloc(x) my_malloc(x, __LINE__, __FILE__, __func__)
#define calloc(x, y) my_calloc(x, y, __LINE__, __FILE__, __func__)
#define realloc(x, y) my_realloc(x, y, __LINE__, __FILE__, __func__)
#define free(s) my_free(s, __LINE__, __FILE__, __func__)
#define fopen(x, y) my_fopen(x, y, __LINE__, __FILE__, __func__)
#define fclose(s) my_fclose(s, __LINE__, __FILE__, __func__)
#define new new (__LINE__, __FILE__, __func__)
```

UŽIVATELSKÉ FUNKCE:

```
extern void stat (void)          - vypíše aktuální alokovanou paměť a otevřené soubory
extern void memory_stat (void);  - vypíše aktuální alokovanou paměť
extern void file_stat (void);    - vypíše aktuální otevřené soubory
```

PŘETÍŽENÉ FUNKCE A OPERÁTORY:

```
extern void* my_malloc (size_t, int, const char[], const char[]);
extern void* my_calloc (size_t, size_t, int, const char[], const
       char[]);
extern void* my_realloc (void*, size_t, int, const char[], const
       char[]);
extern void my_free (void*, int, const char[], const char[]);
extern FILE* my_fopen (const char [], const char [], int, const
       char[], const char[]);
extern int my_fclose (FILE*, int, const char[], const char[]);

extern void* operator new (size_t, int, const char[], const char[]);
extern void* operator new[] (size_t, int, const char[], const
       char[]);
extern void operator delete (void *) throw();
extern void operator delete[] (void *) throw();
```

Příloha L: Popis souboru adtcheck.h

MAKRA:

```
_MAX_DELKA 60                  - nastavení maximálního počtu znaků na jednom řádku
_ZNAKU_MEZI_UZLY_STROMU 5      - počet znaků mezi uzly stromu při výpise
```

TŘÍDY A STRUKTURY:

```
template<class TPrvek, class TData> class adtcheck
                               - třída obsahující kontrolní mechanizmy
template<class TPrvek, class TData> class TSeznam_adr;
                               - třída pro uložení adres registrovaných prvků a adres výpisů
template<class TPrvek> class TSeznam_uk;
                               - třída pro uložení adres registrovaných ukazatelů
```

PROMĚNNÉ TŘÍDY ADTCHECK:

```
std::vector<TSeznam_adr<TPrvek, TData> > adresy
                               - vektor registrovaných prvků
std::vector<TSeznam_uk<TPrvek> > ukazatele
                               - vektor registrovaných ukazatelů
int pocet_ukazatelů            - počet ukazatelů registrovaných s předchozím prvkem
```

VEŘEJNÉ METODY TŘÍDY ADTCHECK:

```
adtcheck();                    - konstruktor třídy
virtual ~adtcheck();           - destruktor třídy
void registruj (TPrvek* ptr, TData *a, TPrvek **Ptr1);
                               - přetížená metoda pro registraci prvku ke kontrole
void registruj (TPrvek* ptr, TData *a, TPrvek **Ptr1, TPrvek
              **Ptr2);         - přetížená metoda pro registraci prvku ke kontrole
void odeber (TPrvek* ptr);
                               - odebrání prvku ze seznamu registrovaných ke kontrole
void zkontroluj (void);        - metoda spouštějící kontrolní mechanismus
```

PRIVÁTNÍ METODY TŘÍDY ADTCHECK:

```
unsigned int get_max_id(void); - vrátí největší ID registrovaného prvku
void registruj_ukazatel(unsigned int id, TPrvek **ptr);
                - zaregistruje ukazatel do vectoru ukazatelů
unsigned int je_reg_adresa (void *ptr);
                - vrací true jestliže je prvek s danou adresou již registrován, jinak false
unsigned int je_reg_ukazatel (TPrvek **ptr);
                - vrací true jestliže je ukazatel s danou adresou již registrován, jinak false
void chybove_hlaseni(std::string s, bool ukoncit = false);
                - vypíše řetězec chybového hlášení, případně ukončí program
void vrat_dalsi(unsigned int id_predka, int *dalsi1, int *dalsi2);
                - nasaví druhý a třetí parametr jako index dalších prvků ve vectoru adresy
void vrat_predchozi(unsigned int id, int *predchozi);
```

- nastaví druhý parametr jako index předchozího prvku ve vektoru adresy; není-li předchozí, nastaví na -1

```
void vrat_predchozi(unsigned int id, int *predchozi1, int
              *predchozi2, int *predchozi3);
```
- nastaví druhý až čtvrtý parametr jako indexy předchozích prvků ve vektoru adresy; není-li předchozí, nastaví na -1

```
void vrat_predchozi(unsigned int id, int *predchozi1, int
              *predchozi2, int *predchozi3, int *predchozi4);
```
- nastaví druhý až pátý parametr jako indexy předchozích prvků ve vektoru adresy; není-li předchozí, nastaví na -1

```
void vypis_pameti (void);
```
- vypíše přehled prvků a jejich ukazatelů v dané chvíli

```
void vypis_poskozene_struktury (void);
```
- pokusí se vypsat nesprávně vytvořený datový typ

```
bool spravne_ukazatele (void);
```
- ověří, zda-li jsou obsahy a počty ukazatelů správné

```
inline unsigned int pocet_znaku(TData promenna);
```
- vrátí počet znaků předané proměnné

```
unsigned int hloubka (unsigned int id_root);
```
- vrací hloubku stromu

```
unsigned int hloubka (unsigned int id_root, unsigned int id_uzel);
```
- vrací hloubku uzlu ve stromu

```
unsigned int delka_vypisu (unsigned int id_root);
```
- vrací délku výpisu stromu

```
bool je_v_seznamu (std::list<unsigned int> *seznam, unsigned int
              hodnota);
```
- ověří, je-li zadaná hodnota uložena v seznamu, vrací bool

```
void zapis_uzel (std::vector<std::string> *pole, unsigned int
              id_root, std::list<unsigned int> *vypsane_prvky, int
              level, int x, std::vector<unsigned int> *y);
```
- zapíše uzel stromu do předaného pole, funkce je rekurzivní

```
std::string to_string (TData data);
```
- konvertuje proměnnou na řetězec

```
void strom_levels_y (std::vector<unsigned int> *y, unsigned int
              id_root, unsigned int levels, unsigned int level_act);
```
- vrací počty znaků do začátku jednotlivých úrovní stromu

```
void strom_optimalizace_vypisu (std::vector<std::string> *pole);
```
- optimalizuje výpis binárního stromu

```
void tisk (std::string s, int mezer_za = 0, int mezer_pred = 0);
```
- vytiskne řetězec přizpůsobený maximální šířce, ta je definována makrem _MAX_DELKA