EKONOMICKÁ UNIVERZITA V BRATISLAVE
FAKULTA HOSPODÁRSKEJ INFORMATIKY

Evidenčné číslo: 17300/B/2011/246 926 4522

ALGORITMIZÁCIA VYBRANÝCH PROBLÉMOV V JAZYKU C++

Bakalárska práca

2011 **Lukáš Chovančák**

EKONOMICKÁ UNIVERZITA V BRATISLAVE
FAKULTA HOSPODÁRSKEJ INFORMATIKY

ALGORITMIZÁCIA VYBRANÝCH PROBLÉMOV V JAZYKU C++

BAKALÁRSKA PRÁCA

Študijný program: Hospodárska informatika

Študijný odbor: 9.2.10 Hospodárska informatika

Školiace pracovisko: KAI FHI - Katedra aplikovanej informatiky FHI

Školiteľ: Ing., Ján Hanák, PhD., MVP.

Bratislava 2011 **Lukáš Chovančák**

Ekonomická univerzita v Bratislave
Fakulta hospodárskej informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Lukáš Chovančák
Študijný program: Hospodárska informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.10 Hospodárska informatika
Typ záverečnej práce: Bakalárska záverečná práca
Jazyk záverečnej práce: slovenský

Názov: Algoritmizácia vybraných problémov v jazyku C++

Anotácia: Analýza, návrh a implementácia parciálnych objektových riešení v jazyku C++.

Vedúci: Ing. Ján Hanák, PhD.
Katedra: KAI FHI - Katedra aplikovanej informatiky FHI
Vedúci katedry: doc. Ing. Gabriela Kristová, CSc.

Dátum zadania: 30.09.2010

Dátum schválenia: 19.10.2010

doc. Ing. Gabriela Kristová, CSc.
vedúci katedry

.....
študent

.....
Vedúci

ABSTRAKT

CHOVANČÁK, Lukáš: *Algoritmizácia vybraných problémov v jazyku C++*. – Ekonomická univerzita v Bratislave. Fakulta Hospodárskej informatiky; katedra aplikovanej informatiky. – Vedúci záverečnej práce: Ing., Ján Hanák, PhD., MVP. – Bratislava: FHI EU, 2011, počet strán 33s.

Cieľom záverečnej práce bolo ozrejmiť postupy a spôsoby algoritmizácie v jazyku C++ za použitia pokročilých techník v objektovo orientovanej paradigme programovania.
Práca je rozdelená do troch kapitol.
Prvá kapitola je venovaná súčasnému stavu riešenej problematiky doma a v zahraničí a histórií jazyka C++.
V ďalšej časti sa charakterizuje cieľ práce, metodika práce a metódy skúmania. Zoznamuje nás so samotným fungovaním jazyka a so stavebnými prvkami použitými na riešenie.
Záverečná kapitola sa zaoberá výsledkami práce a praktickými ukážkami algoritmizácie vybraných problémov.
Výsledkom riešenia danej problematiky je schopnosť efektívne algoritmizovať rôzne typy problémov pomocou sofistikovaných metód objektovo orientovaného programovania.

Kľúčové slová:
Algoritmizácia, objektovo orientované programovanie, pokročilé princípy programovania

ABSTRACT

CHOVANČÁK, Lukáš: *Algorithmization of chosen problems in language C++*. – Economic university in Bratislava. College of economic informatics; desk of applied informatics. – Leader of closing work: Ing., Ján Hanák, PhD., MVP. – Bratislava: FHI EU, 2011, number of pages 33p.

The goal of closing work was clarify pratcises of algorithmization in language C++ with using advanced techniques in object oriented programming.
The first chapter is devoted to actual condition of problem solution at home and externally and to history of language C++.
In next chapter is described the goal of this work, methodics of work and methods of research. She describes system funcioning of language C++ and with architectual elements that are to be used.
The last chapter deals with achievement of work and with applied examples of algorithmization of chosen problems.
Achievement of solution of this problem is the ability to effective algorithmization of various types of problems with sophistic methods of object oriented programming.

Key words:
Algorithmization, object oriented programming, advanced principle of programming

Čestné vyhlásenie

Čestne vyhlasujem, že záverečnú prácu som vypracoval samostatne a že som uviedol všetku použitú literatúru.

Dátum: **.....**

(podpis študenta)

Obsah:

ÚVOD 1

1 SÚČASNÝ STAV RIEŠENEJ PROBLEMATIKY DOMA A V ZAHRANIČÍ 2

1.1 História jazyka 2

1.2 Použitie jazyka C++ dnes 2

2 CIEĽ PRÁCE, METODIKA PRÁCE A METÓDY SKÚMANIA 4

2.1 Programovanie v jazyku C++ 4

2.2 Rozšírenia C++ 5

2.2.1 C++ triedy a objekty 5

2.2.2 Šablóny funkcií 7

2.2.3 Štandardné knižnice v C++ (STL - Standard Template Library) 8

3 VÝSLEDKY PRÁCE 15

3.1 Prvá praktická ukážka 15

3.2 Druhá praktická ukážka 17

3.3 Tretia praktická ukážka 20

3.4 Štvrtá praktická ukážka 27

3.5 Piata praktická ukážka 31

ZÁVER 33

ZOZNAM POUŽITEJ LITERATÚRY 34

ZOZNAM OBRÁZKOV 35

Úvod

V dnešnej dobe sa ľudia snažia všemožne si uľahčiť svoje životy. Chcú ich mať pokojné a bezpečné. Jedným z nástrojov, ktoré v širokej miere prispievajú k zvyšovaniu životnej úrovne ľudstva sú počítačové programy na ovládanie elektronických zariadení. Toto odvetvie sa neustále rozvíja, vývojári chcú stále prinášať lepšie a výkonnejšie algoritmy na zvládanie každodenných, ale aj príležitostných starostí bežného človeka. To činí túto problematiku nesmierne zaujímavou a príťažlivou. Napriek tomu, že programovanie je doménou skôr mladšej generácie, verím, že o neho majú záujem všetky ročníky. V nasledujúcom texte sa dozvieme, čo to vlastne programovanie je, predstavíme si jeho fungovanie, základné i niektoré pokročilé princípy a budeme sa snažiť vyriešiť časté problémy, s ktorými sa dnešný vývojár môže stretnúť. Zo širokého výberu programovacích jazykov si vyberieme jeden, ktorému sa budeme venovať, v našom prípade rozšírený a populárny hybridný jazyk C++ s jeho prekladačom od spoločnosti Microsoft, ktorý je súčasťou sofistikovaného vývojového prostredia Visual Studio 2010 Express Edition.

Preto názov tejto práce znie Algoritmizácia vybraných problémov v jazyku C++. Jej náplňou bude na vybraných častých problémových doménach ukázať ich možné, prípadne najvhodnejšie riešenie. Pre názornú demonštráciu budú uvedené úlohy rôznej náročnosti a pri ich algoritmizácii sa budeme snažiť odhaliť riešenia, ktoré čitateľom umožnia jednoduchú implementáciu týchto techník, pri programovaní vlastných projektov.

Práca obsahuje tri hlavné kapitoly. V prvej sa zoznámime s históriou zvoleného jazyka C++, dôvodom jeho vzniku, ako aj s jeho využitím vo svete. Druhá kapitola obsahuje popis metodík skúmania použitých v tejto práci, rozoberá cieľ práce a predstavuje teoretickú prípravu na algoritmizáciu v jazyku C++, popis jeho základných stavebných prvkov a použiteľných programových štruktúr aj s ich generickým zápisom. V podkapitole druhej časti je vysvetlené fungovanie celého jazyka. Tretia časť začína založením nového projektu a následne praktickými ukážkami, ktoré postupne zvyšujú svoju náročnosť.

1 Súčasný stav riešenej problematiky doma a v zahraničí

1.1 História jazyka

Jazyk C++ vznikol ako nadstavba štruktúrovaného jazyka C, ktorý bol vyvinutý v Bell Laboratories v 70. rokoch 20. storočia Dennisom M.Ritchiem za účelom prenositeľnosti a nezávislosti na hardware kvôli jeho projektu rozšírenia operačného systému UNIX. Jazyk C++ bol vytvorený v 80-tych rokoch 20. storočia v Bell Laboratories ako hybridný jazyk, v ktorom môžeme programovať štruktúrovane aj objektovo. Vyvinul ho Bjarne Stroustrup rozšírením jazyka C o objekty o čom svedčí aj názov, predpokladáme, že dve pluská predstavujú operátor inkrementácie ktorý v programovaní zvyšuje hodnotu danej premennej o jednotku. Jeho prvý názov bol C with Classes a prekladal sa pomocou špeciálneho preprocesora Cpre do pôvodného jazyka C. Neskôr sa jazyk premenoval na C++ a prekladač na Cfront. Nasledovala prvá komerčná verzia a postupné vylepšovanie prekladača, ktoré malo za následok aj vylepšovanie samotného jazyka pridaním komponentov, na ktoré sú vývojári v dnešnej dobe zvyknutí, ako napríklad výnimky, šablóny a menné priestory. Jeho prvý ISO štandard bol schválený v roku 1998. Od tej doby prešiel jeho kompilátor mnohými zmenami, napríklad pridaním preťažovania operátorov a funkcií a v najnovšej verzií aj pridaním takzvaných lambda funkcií a automaticky inferovanými dátovými typmi, ukotvenými aj v príslušných ISO normách.

1.2 Použitie jazyka C++ dnes

V dnešnej dobe je najznámejší prekladač od spoločnosti Microsoft, ale vlastný kompilátor má napríklad aj firma Borland. Kompilátor spoločnosti Microsoft je súčasťou programového balíka Visual Studio. Najnovšia verzia v dnešnej dobe je Visual Studio 2010, ktorá okrem kompilátora pre C/C++ obsahuje kompilátory jazykov C#, Visual Basic 2010 a F#.

Jazyk C++ je nasadzovaný predovšetkým do robustných projektov, z ktorých mnohé patria k takzvaným kritickým aplikáciám. Mnohí skúsení vývojári tvrdia, že zvládnutie jazyka C++ zabezpečuje úspešné algoritmizovanie všetkých úloh, ale aj jednoduchší prechod k inému programovaciemu jayzku. C++ v súčasnosti využíva najmä paradigmu objektovo orientovaného programovania a je spolu s jazykmi C a C# najrozšírenejším programovacím

jazykom, využívaným na tvorbu všetkých typov aplikácií, od počítačových hier, cez hospodárske programy, až po tvorbu operačných systémov. C++ je efektívny, ucelený, rýchly a prenositeľný jazyk. Spája tri výhodné štýly programovania dvadsiateho storočia:

- procedurálna paradigma reprezentovaná jeho predchodcom - jazykom C
- objektovo orientovaná paradigma reprezentovaná rozšírením o triedy
- programovanie pomocou predlôh podporované šablónami C++.

2 Cieľ práce, metodika práce a metódy skúmania

Cieľom práce je na algoritmizácií vybraných problémov demonštrovať základné princípy a postupy programovania v jazyku C++. Fragmenty zdrojových kódov budú zamerané na ukážky optimalizácie z hľadiska skrátenia exekučného času a zmenšenia nákladov na pamäťový priestor. Vo vybraných programových kódoch budú použité návrhové vzory a na ukážkach budú vysvetlené základné predpoklady objektovo orientovaného návrhu.

Na začiatok sa ale zoznámime s vývojovým prostredím. Vybral som Visual Studio 2010 Express Edition od spoločnosti Microsoft, určené ako pre začínajúcich, tak aj pre pokročilých vývojárov. Ukážeme si, ako vytvoriť projekt programu v jazyku C++ a postupne budeme vylepšovať jednoduché zdrojové kódy, až do podoby čo najbližšej profesionálnym programom. Každý fragment kódu bude dôkladne vysvetlený z hľadiska sémantiky aj syntaxe a pre lepšie pochopenie budú kritické miesta podporené obrazovou dokumentáciu.

Pri skúmaní spôsobov algoritmizácie v jazyku C++ budú v nasledujúcom texte použité základné aj pokročilé stavebné prvky syntaxu tohoto hybridného jazyka. Priblížime si rôzne možnosti ich použitia a vysvetlíme si aj fungovanie niektorých prvkov, ktoré sú prebrané zo štruktúrovaného jazyka C.

Na začiatku sa však stručne zoznámime s fungovaním jazyka pre lepšie pochopenie spôsobu algoritmizácie.

2.1 Programovanie v jazyku C++

Pri programovaní v akomkoľvek programovacom jazyku je vždy na začiatku nutné si premyslieť postup riešenia problému. Až potom je možné tento postup začať programovať pomocou príslušného programovacieho jazyka v medziach jeho syntaxe. Rovnako je to aj pri jazyku C++, preto je výhodou mať pri programovaní v tomto jazyku dobre rozvinuté analytické myslenie, prípadne mať široké vedomosti z oblasti matematiky.

Program zapisovaný vo vývojovom prostredí sa nazýva algoritmus. Algoritmus je postupnosť krokov, ktorá je konečná, vyhodnotiteľná, deterministická a vracia nám určitý výsledok. Väčšina problémov s ktorými sa denne stretávame je algoritmizovateľná, to znamená, že sa dá vytvoriť všeobecný postup na ich riešenie.

C++ nám ponúka širokú škálu príkazov a operátorov, ktoré sú obsiahnuté v jeho príslušných knižniciach. Každý okruh takto preddefinovaných komponentov je obsiahnutý vo

vlastnej knižnici, čo je výhodou najmä pri robustnejších aplikáciách, pretože to skracuje dĺžku kódu a tým pádom aj dobu kompilácie a v konečnom dôsledku aj veľkosť aplikácie.

Príkazy sú vo veľkej miere anglické slová alebo nejaké ich logické spojenia. Takému jazyku človek rozumie, dokáže sa naučiť ako sa dajú také slová spájať. Počítač ale rozumie iba svojmu strojovému jazyku, čo je sled jednotiek a núl. O preklad zdrojového kódu jazyka C++ do strojového kódu sa starajú tri programy a to preprocesor, prekladač a linker. Preprocesor sa stará o predspracovanie kódu, nahrádza názvy knižníc vkladaných do kódu direktívou #include ich obsahom. Prekladač vykonáva konverziu zdrojového kódu v ktorom sú vykonané zmeny preprocesorom do strojového kódu, za predpokladu, že neobsahuje žiadne syntaktické chyby. Strojový kód uloží do objektového súboru s príponou .obj. Linker prideľuje k objektovému súboru kód z knižníc a vytvára spustiteľnú aplikáciu.

2.2 Rozšírenia C++

Jazyk C++ je niekedy nazývaný ako nadmnožina jazyka C, čo znamená, že C++ obsahuje všetky vlastnosti jazyka C a naviac niekoľko nových vlastných. Mimo to niektoré techniky a prvky s ktorými sa programátori zoznámili v C sú v jazyku C++ zmenené.

Príkazy, premenné, operátory používané v C sú funkčné aj v C++, niektoré sú upravené, umožňujú širšie použitie a sú výkonnejšie.

C++ je voči C rozšírený najmä o prvky objektovo orientovaného programovania s využitím všetkých jeho princípov - zapúzdrenie, dedičnosť a polymorfizmus a používanie šablón, ktoré sú základom generického programovania v C++.

2.2.1 C++ triedy a objekty

Trieda je dátovým typom. Inštancia je premennou danej triedy. Termín "objekt" v C++ označuje inštanciu danej triedy. Čiže trieda je popisom budúcej inštancie (objektu).

Trieda nám zabezpečuje využívanie prvkov obejktovo orientovanej paradigmy programovania.:

- zapúzdrenie.- ukrývanie dát a implementácie. Objekt je z vonkajšieho pohľadu "čiernou skrinkou" Je to samostatná časť programu, ktorý môžeme používať ale bez zásahu do vnútornej štruktúry. V okamihu keď chce nejaká časť programu použiť triedu v zmysle využitia objektu, nemusí poznať vnútornú štruktúru, stačí ak vie ako triede odovzdať dané údaje. Ona sama vykoná zmenu a vráti výsledok. Táto

operácia a ochrana dát sa spravidla uskutočňuje spôsobom verejných metód a súkromných, prípadne chránených členských premenných triedy. Prístup k verejným členom triedy nám zabezpečuje operátor bodka, alebo šípka.

- dedičnosť - nové objekty môžu vznikať na základe už existujúcich. Vzniká tak "špeciálna", zväčša rozšírená varianta základnej triedy. Potomok dedí všetky atribúty rodiča. Dedí aj všetky metódy rodiča s výnimkou konštruktorov, deštruktorov a priradení.
- polymorfizmus - rôzne objekty môžu zvonku vyzerať rovnako, ale napriek tomu plnia rôzne úlohy.

Na deklaráciu triedy sa používa kľúčové slovo "class". Deklarácia triedy je podobná deklarácii štruktúry (kľúčové slovo "struct"), avšak "class" vynucuje ukrývanie dát a implicitné rozhranie je "private". Štruktúra vytvorená kľúčovým slovom "struct" nepredpokladá ukrývanie dát a implicitným rozhraním je "public".

Trieda musí obsahovať svoj názov, členov triedy a to metódy a premenné, konštruktor a deštruktor.

Konštruktor je špeciálna členská funkcia bez návratovej hodnoty, ktorá ma rovnaký názov ako samotná trieda. Volanie konštruktora je vždy späté s vytvorením inštancie triedy. Využíva sa na nastavenie počiatočných vlastností objektu pri jeho vytváraní. Pokiaľ nevytvorí konštruktor programátor, prekladač implicitne vygeneruje kopírovací konštruktor daného objektu. Tento konštruktor iba vytvorí daný objekt. Väčšinou nie je pre potreby programátorov dostačujúci, preto sa zvykne konštruktor tvoriť explicitne. Často je vhodné ho aj preťažiť. Trieda môže mať viac konštruktorov.

Deštruktor má tiež názov zhodný s názvom triedy, ale je pred neho pridaný znak tilda (~). Deštruktor je vždy posledná volaná metóda pred zrušením inštancie. Volanie deštruktoru zabezpečí kompilátor automaticky. Jeho úlohou je obvykle uvoľniť všetky zdroje patriace inštancii. Deštruktor nemá návratovú hodnotu ani formálne parametre, preto nie je možné ho preťažiť. Deštruktor nie je povinný a je len jeden. Generický zápis deklarácie triedy je nasledovný:

```
class NazovTriedy {
protected:   //chránené členy triedy
private:     //súkromné členy triedy
public:      //verejné členy triedy
};
```

2.2.2 Šablóny funkcií

Dôležitou inováciou oproti jazyku C sú šablóny funkcií. Ide o generickú deklaráciu funkcie, alebo triedy nezávislú na typoch parametrov. Skutočná funkcia, alebo objekt sú potom vytvorené prekladačom na základe skutočných typov použitých parametrov pri volaní funkcie, resp. vytváraní objektu. Preto sa používanie šablón funkcií nazýva aj parametrické programovanie. Z jednej šablóny môže byť pri kompilovaní vytvorených viac funkcií s rozličnými typmi parametrov.

Niektoré črty použita šablón:

+ vysoká znovupoužiteľnosť kódu, t.j. zvýšenie produktivity a zníženie pravdepodobnosti výskytu chýb
+ ponuka parametrizovaných typov
+ umožňujú polymorfizmus v dobe kompilácie (bez záťaže počas behu programu)
– môžu zväčšovať veľkosť binárneho kódu
– predlžuje sa doba kompilácie
– generované chybové hlásenia pri kompilácii veľmi rozsiahle a neprehľadné

Kľúčovým slov pre vytvorenie šablóny je "template". Parametre šablóny sa uvádzajú v špicatých zátvorkách. Kľúčové slovo typename je zastupiteľné kľúčovým slovom class. Význam týchto kľúčových slov je rovnaký, v tomto prípade sú voľne zameniteľné. Je to veľmi výhodná metóda ak chceme vytvoriť funkciu, ktorá by mala vykonávať určitú operáciu pre viacero dátových typov. Namiesto programovania rovnakých funkcií vytvoríme jednu šablónu, z ktorej alternatívy pre rôzne dátové typy vytvorí prekladač. Šablónu je výhodné použiť aj v prípadoch, keď nevieme aké dátové typy bude funkcia používať. Namiesto skúmania týchto typov sa za nás o to postará prekladač, ktorý vytvorí konkrétnu inštanciu funkcie zo šablóny. Generický model šablónovej funkcie vyzerá nasledovne:

```
template < typename T >
návratový typ nazov_funkcie( T premenna, T premenna) {
      //telo funkcie
}
```

alebo:

```
template < class T >
návratový typ nazov_funkcie( T premenna, T premenna) {
      //telo funkcie
}
```

2.2.3 Štandardné knižnice v C++ (STL - Standard Template Library)

Efektívne programovanie v ľubovoľnom jazyku je priamo závislé na knižniciach. Knižnice umožňujú opakované použitie existujúceho kódu , riešia elementárne problémy pri práci s poliami, reťazcami, prúdmi, dynamickou pamäťou, implementujú elementárne algoritmy, pričom implementácia je "exception safe" (bezpečná voči výskytu výnimiek). Knižnice sú ľahko rozšíriteľné a sú rovnako rýchle ako ručne písaný kód.

Základnými komponentami STL sú funkčné objekty, generické algoritmy, kontajnery a iterátory.

2.2.3.1 Funkčné objekty

Sú to objekty, ktoré sa správajú ako funkcie, sú volané ako funkcie, avšak voči funkciám majú niekoľko výhod (môžu obsahovať ďalšie členské funkcie a atribúty, každý funkčný objekt môže mať svoj vlastný typ, funkčné objekty sú obyčajne rýchlejšie ako funkcie). Aby bolo možné použiť objekt ako funkciu, jeho trieda musí obsahovať deklaráciu operátora "()" [3]:

```
class X {
     public:
       //define "function call" operator
       return-value operator() (arguments) const;
       ...
};
```

Teraz je možné použiť objekt z takejto triedy ako funkciu:

```
X fo;
...
fo(arg1, arg2);    //call operator () for function object fo
```

C++ štandardné knižnice obsahujú preddefinované funkčné objekty, ktoré pokrývajú základné operácie. Príklady:

- aritmetické operácie:

`plus<int>()A,B)`	- sčítanie celých čísel A a B
`minus<int>()(A,B)`	- odčítanie celých čísel A a B
`multiplies<int>()(A,B)`	- násobenie celých čísel A a B
`divides<int>()(A,B)`	- delenie celých čísel A a B
`modulus<int>()(A, B)`	- celočíselný zvyšok po delení
`less<int>()(A, B)`	- porovnanie "menší"

`less_equal<int>()(A, B)` - porovnanie "rovný"

`greater<int>()(A, B)` - porovnanie "väčší"

`greater_equal<int>()(A, B)` - porovnanie "väčší, alebo rovný"

`not_equal_to<int>()(A, B)` - porovnanie "rôzny"

- bindery (funkčné adaptéry) - umožňujú kombinovať rozličné funkčné objekty:

  `bind1st(op, value)` - konštruuje unárny funkčný objekt z binárnej funkcie "op" a zviaže s ním jej prvý parameter "value"

  `bind2nd(op, value)` - konštruuje unárny funkčný objekt z binárnej funkcie "op" a zviaže s ním jej druhý parameter "value"

  `not1(op)` - konštruuje funkčný objekt z negácie unárnej funkcie "op"

  `not2(op)` - konštruuje funkčný objekt z negácie binárnej funkcie "op"

2.2.3.2 Generické algoritmy

STL poskytuje viacero štandardných algoritmov pre prácu s elementami. Tieto algoritmy ponúkajú základné služby ako vyhľadávanie, triedenie, kopírovanie, prerozdelenie, modifikácia a číselné spracovanie. Algoritmy nie sú členskými funkciami kontajnerov, ale sú to globálne funkcie, ktoré pracujú s iterátormi. Výhodou takejto koncepcie je, že algoritmy nie sú implementované zvlášť pre každý kontajner, ale sú implementované len raz pre akýkoľvek typ kontajnera.

Príklady algoritmov:

`min_element(iterator_begin, iterator_end)` - vráti iterátor na najmenší element z rozsahu elementov begin - end

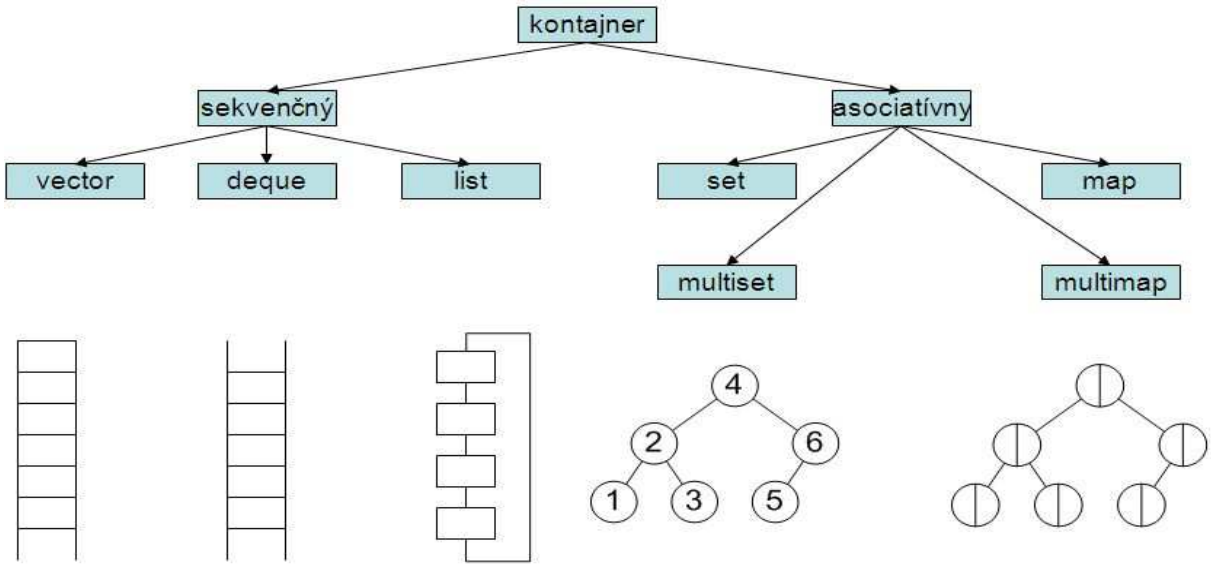
`max_element(iterator_begin, iterator_end)` - vráti iterátor na najväčší element z rozsahu elementov begin - end

`count_if(iterator_begin, iterator_end, predikat)` - vráti počet elementov z rozsahu begin-end, ktoré vyhovujú podmienke "predikat"

`replace_if(iterator_begin,iterator_end, predikat, const T& nova_hodnota)` - zmení elementy zrozsahu begin - end, ktoré vyhovujú predikátu "predikat" na novú hodnotu "nova_hodnota"

2.2.3.3 Kontajnery

Sú objekty na uchovanie (zapamätanie) iných objektov. Obsahujú metódy pre prístup k elementom. Každý objekt typu kontajner má obvykle asociovaný objekt typu "iterátor", ktorý sa používa na iteráciu cez elementy kontajnera. STL ponúka rôzne druhy kontajnerov:



obr 1 rozdelenie kontajnerov

2.2.3.3.1 Sekvenčné kontajnery

Sú to zoradené súbory elementov, v ktorých má každý element určitú pozíciu. Táto pozícia závisí na čase a mieste vloženia elementu, ale nezávisí na jeho hodnote. Napríklad po vložení niekoľkých elementov do súboru jeden za druhým majú tieto elementy presné poradie rovnaké v akom boli vložené. STL ponúka niekoľko preddefinovaných sekvenčných kontajnerov:

- `vector` - ukladá elementy do dynamického poľa. Umožňuje náhodný prístup k elementom, čo znamená, že je možné pristupovať ku každému elementu priamo cez korešpondujúci index. Pridávanie a mazanie elementov na konci poľa je veľmi rýchle. Vkladanie elementov na začiatok a do "stredu" je pomalé, pretože všetky nasledujúce elementy sa musia posunúť aby vytvorili priestor pre vkladaný element.
- `deque` - "double-ended queue" - je to dynamické pole, ktoré je implementované tak, že môže rásť oboma smermi, čiže vkladanie elementov na začiatok a na koniec zoznamu je rýchle. Vkladanie do "stredu" zoznamu je rovnako ako u vektora pomalé, pretože už vložené elementy sa musia presúvať.

- `list` - je implementovaný ako obojsmerne spájaný zoznam elementov. To znamená, že každý element v zozname má pridelený vlastný segment pamäte a má odkaz na predchádzajúci a nasledujúci element. Zoznam neposkytuje náhodný prístup. Na prístup ku každému elementu je potrebné prejsť odkazy od prvého až po element predchádzajúci požadovanému elementu. Skok medzi dvoma elementami zaberá konštantný čas, čiže čas na prístup k určitému elementu lineárne rastie s jeho vzdialenosťou od začiatku zoznamu. Výhodou tohto kontajnera je rýchle vkladanie a mazanie elementov.

2.2.3.3.2 Asociatívne kontajnery

Asociatívne kontajnery automaticky zoraďujú elementy podľa určitých triediacich kritérií. Tieto kritériá nadobúdajú formu funkcie, ktorá porovnáva hodnoty elementov, alebo kľúčov. Implicitné kontajnery porovnávajú elementy, alebo kľuče operátorom "<" (menší). Na definovanie porovnávacích kritérií je možné použiť vlastnú porovnávaciu funkciu.

Táto skupina kontajnerov je najčastejšie implementovaná ako binárne stromy. Každý element (uzol) má jedného rodiča a dvoch potomkov. Všetci predkovia vľavo majú menšiu hodnotu, predkovia vpravo väčšiu hodnotu. Asociatívne kontajnery sa odlišujú podľa druhu elementov, ktoré podporujú a podľa toho, ako manipulujú s duplikátmi.

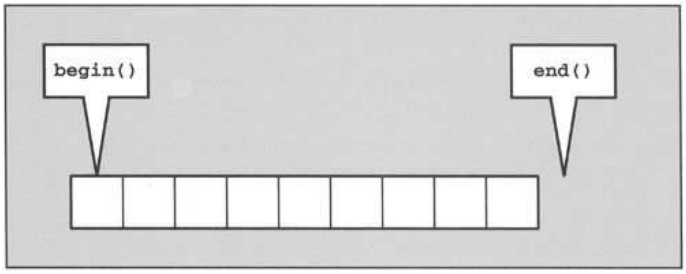
- `set` - elementy sú zoradené podľa ich hodnoty. Každý element sa môže v zozname vyskytovať len raz, čiže duplikáty nie sú povolené
- `multiset` - rovnaký kontajner ako `set`, avšak duplikáty sú povolené. To znemená, že multiset môže obsahovať elementy s rovnakou hodnotou
- `map` - obsahuje elementy, ktoré sú dvojicami kľúč - hodnota. Každý element má kľúč, ktorý je základom pre triediace kritérium a hodnotu elementu. Každý element sa môže v zozname vyskytovať len raz.
- `multimap` - rovnaký kontajner ako `map, avšak` duplikáty sú povolené. Čiže multimap môže obsahovať elementy, ktoré majú rovnaký kľúč.

2.2.3.4 Iterátory

Sú objekty, ktoré obvykle "ukazujú" na iné objekty. Umožňujú iteráciu (navigovanie) cez elementy zoznamu. Sú používané na iteráciu v zozname objektov. Ak iterátor ukazuje na jeden element rozsahu, potom je možné inkrementovať ho tak, aby ukazoval na nasledujúci

element rozsahu. Každý iterátor reprezentuje určitú pozíciu v kontajneri. Všetky kontajnerové triedy poskytujú základné členské funkcie, ktoré im umožňujú používať iterátory na navigáciu cez ich elementy. Najdôležitejšie z týchto funkcií sú:

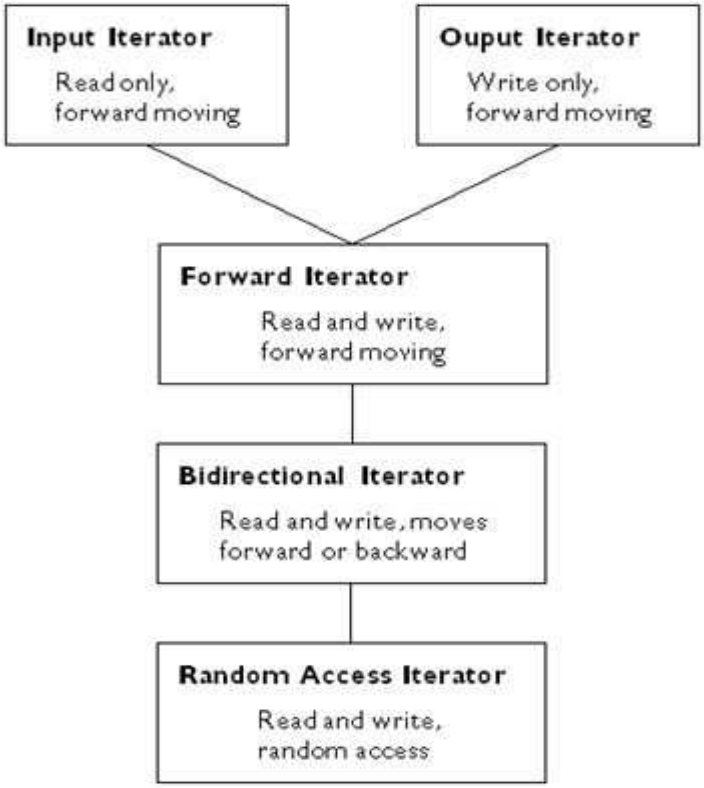
- `begin()` - vráti iterátor, ktorý reprezentuje začiatok elementov v kontajneri (pozícia prvého elementu)
- `end()` - vráti iterátor, ktorý reprezentuje koniec elementov v kontaineri. Je to pozícia za posledným elementom



obr 2 funkcie begin() a end() pre kontajner

V prázdnom kontajneri sa návratová hodnota `begin()` rovná návratovaej hodnote `end()`.

Iterátory sa podľa svojich vlastností a možností, ktoré poskytujú rozdeľujú do niekoľkých kategórií:



obr 3 kategórie iterátorov

2.2.3.4.1 Input iterator

Input iterátor vie postupovať len vpred od `begin()` až k `end()`cez jednotlivé elementy zoznamu s prístupom len na čítanie. Tento iterátor môže čítať elementy len raz.

Takmer všetky iterátory majú schopnosti input iterátora. Typickým príkladom input iterátora je iterátor, ktorý číta štandardný vstup (napr. klávesnica).

Dva input iterátory sa rovnajú, ak zaujímajú rovnakú pozíciu. Input iterátor sa používa v objektoch istream.

2.2.3.4.2 Output iterator

Output iterátor môže postupovať len vpred od `begin()`až po `end()`cez jednotlivé elementy s prístupom pre zápis. Nie je možné použiť output iterátor viackrát v tom istom rozsahu elementov.

Typickým príkladom output iterátora je iterátor, ktorý zapisuje na štandardný výstup (monitor). Output iterátor sa používa v objektoch ostream.

2.2.3.4.3 Forward iterator

Je to kombinácia Input a Output iterátora. Iterátory tejto kategórie majú všetky vlastnosti input iterátorov a väčšinu vlastností output iterátorov. Narozdiel od predchádzajúcich kategórií iterátorov môže Forward iterátor odkazovať na ten istý element v zozname a procese viackrát.

2.2.3.4.4 Bidirectional iterator

Je to forward iterátor, ktorý poskytuje možnosť spätnej iterácie cez elementy. Čiže táto kategória iterátorov poskytuje "decrement operator" na pohyb smerom od konca zoznamu k začiatku.

2.2.3.4.5 Random Access Iterator

Random Access Iterator je bidirectional iterátor, ktorý poskytuje náhodný prístup k elementom. Poskytuje operátory pre "iterátorovú aritmetiku" (podobná pointerovej - smerníkovej aritmetike).

Random access iterátory sa používajú v nasledovných objektoch a typoch:

- kontajnery s náhodným prístupom (`vector, deque`)
- reťazce (`string, wstring`)
- bežné polia (smerníky na prvky)

3 Výsledky práce

3.1 Prvá praktická ukážka

V prvej praktickej ukážke kódu budeme hľadať počet prvočísel nachádzajúcich sa v intervale, ktorý začína nulou a končí číslom, ktoré zadá používateľ vrátane tohto čísla. Ďalej budeme chcieť zistiť, ako dlho toto hľadanie trvalo, pre názornú ukážku neoptimálnosti. Jazyk C++ nám ponúka širokú škálu možností, ako túto úlohu splniť, prvá ukážka bude upozorňovať na časté chyby a nedostatky kódu, ktoré odstránime v druhej ukážke.

Prvý krok je premyslieť si spôsob riešenia takéhoto matematického problému. Interval začína nulou, ale samotná nula prvočíslom nie je, takisto ani jednotka, takže začíname rátať až od čísla dva. Najjednoduchšou možnosťou je testovanie každého čísla zo zadaného intervalu. Neznamená to, že je táto možnosť najlepšia, ale napadla nás ako prvá tak ju skúsime použiť. Ďalšia otázka je ako dané číslo testovať. Prvočísla sú deliteľné iba jednotkou a sebou samým, pri delení akýmkoľvek iným číslom nám ostane nejaký zvyšok. Na to použijeme operátor modulo (ozn. “%“), ktorý nám vracia zvyšok po delení. Na rátanie exekučného času použijeme funkciu GetTickCount z knižnice windows.h. Celý program je možné pre jeho jednoduchosť vložiť do jedinej funkcie a to funkcie main, ktorá musí byť v jazyku C++ prítomná.

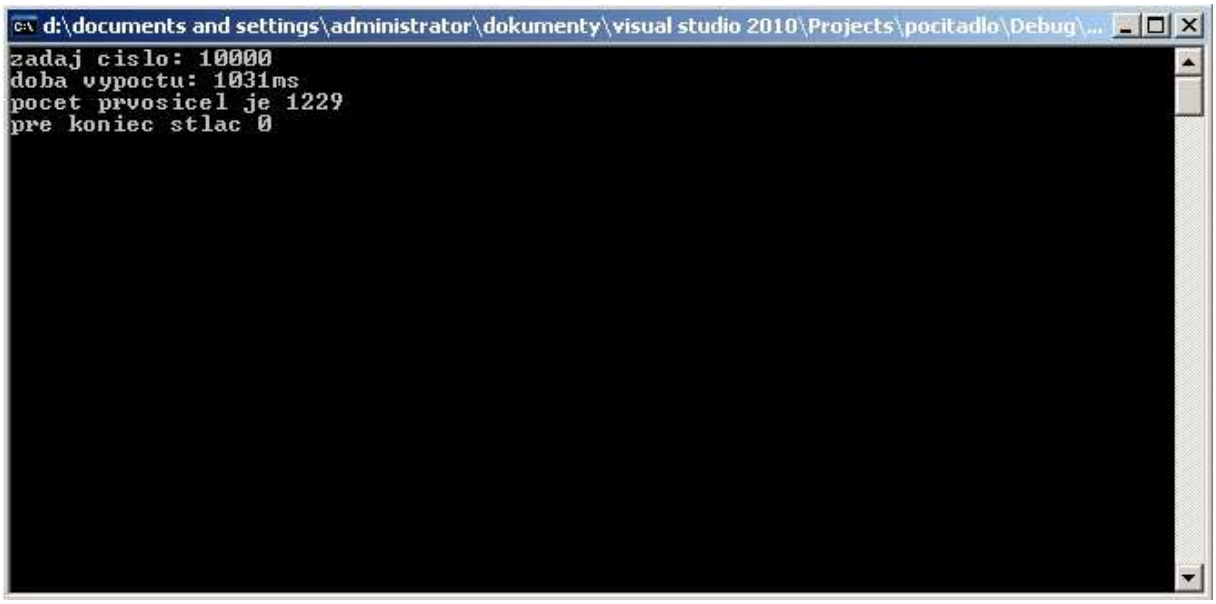
```
#include <iostream>      //hlavičkový súbor s popisom IO objektov
#include <windows.h>     //hlavičkový súbor s deklaráciou funkcie pre výpočet
                         //exekučného času
using namespace std;     //použitie štandardného menného priestoru


int main()                                  //začiatok funkcie main
{
    unsigned cislo,i,j,pocitadlo,           //deklarácia premenných
             cas,cas2,pocet=0,koniec;
    cout<<"zadaj cislo: ";
    cin>>cislo;
    cas2=GetTickCount();                    //začiatok merania času
    for(i=1;i<=cislo;i++)                   //začiatok cyklu hľadania
    {
        pocitadlo=0;
        for(j=1;j<=i-1;j++)                 //vnorený cyklus
        {
            if(i%j==0)
                pocitadlo++;
        }
        if(pocitadlo==1)
        {
            pocet++;
        }
```

```
    }
      cas=GetTickCount();                    //koniec merania času
      cout<<"doba vypoctu: "<<cas-cas2<<"ms"<<endl;
      cout<<"pocet prvosicel je "<<pocet<<endl;
      cout<<"pre koniec stlac 0"<<endl;
      cin>>koniec;
    return koniec;
}
```

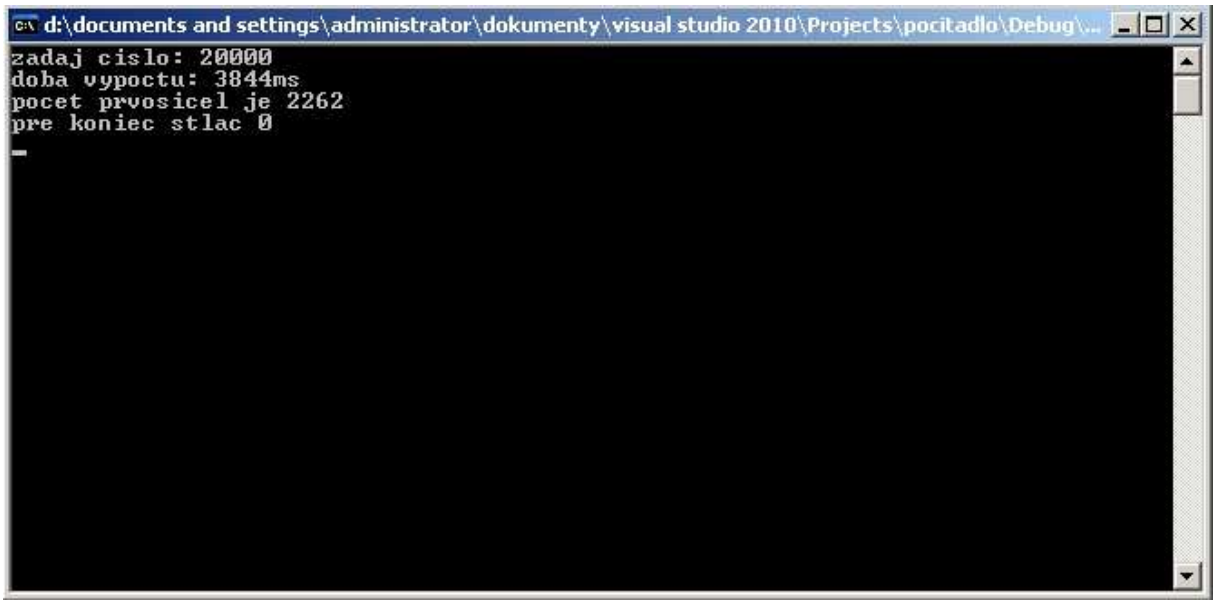
Tento program plní zadanú úlohu, ale nevyužíva objektovo orientovanú paradigmu programovania, ktorá je pre C++ napriek jeho hybridnosti špecifická. Pri takýchto krátkych kódoch možno nemá zmysel objekty používať, no pri väčších projektoch sú nenahraditeľné, preto je dobré si na ne zvykať a snažiť sa ich logicky využívať aj pri malých aplikáciách. Objekty plnia aj bezpečnostnú funkciu, ide o abstrakciu, ktorá zapuzdruje a chráni obsah premenných, ku ktorým sa dá pristupovať iba pomocou metód, preto odstraňujú možnosť náhodného prepísania hodnoty premennej.

Druhý nedostatok je spôsob ukončenia programu. Posledným príkazom by mal byť `return 0;`. Pri programovaní v C++, ale aj v akomkoľvek inom programovacom jazyku platí pravidlo, že ak program ukončí svoju činnosť úspešne, vráti volajúcemu prostrediu (inému programu, alebo operačnému systému) celočíselnú hodnotu rovnú nule. Ak dôjde počas vykonávania programu k chybe a program nemohol úspešne vykonať svoju úlohu, vráti akúkoľvek inú hodnotu. Nie je vhodné nechať rozhodnutie, či program prebehol správne na používateľovi. Typický používateľ je pohodlný a aj keď má programom odporúčané, ako ho ukončiť, očakáva, že program sa ukončí podľa jeho želania aj inými spôsobmi.

Tretím a posledným väčším nedostatkom je, že napriek správnosti výsledku, je jeho výpočtová zložitosť neoptimálna. Nie je nutné testovať každé číslo z intervalu, čo nám dokazuje matematika. Napríklad nie je nutné testovať čísla, ktoré sú násobkami prvočísel. Takéto čísla majú určite viac deliteľov ako jednotku a seba samého, pretože sú násobkami iných čísel. Tieto zbytočné testovania nám predlžujú exekučný čas a zaťažujú procesor. Dôkazom toho je aj výstup:



obr 4 doba vykonávania algoritmu pre rozsah 1 až 10000



obr 5 doba vykonávania algoritmu pre rozsah 1 až 20000

Ako vidno vo výpisoch, exekučný čas je naozaj dlhý a nerastie lineárne ale rýchlejšie. Z tohoto dôvodu je nutná optimalizácia aplikácie z pohľadu výhodnejšieho využitia výpočtovej kapacity.

3.2 Druhá praktická ukážka

Druhá praktická ukážka optimalizuje prvý program z pohľadu zavedenia objektovo orientovanej paradigmy programovania a to hlavne zapúzdrenia počítajúceho mechanizmu do

triedy, zatraktívnenia aplikácie pre používateľa a skrátenia časovej a pamäťovej zložitosti.

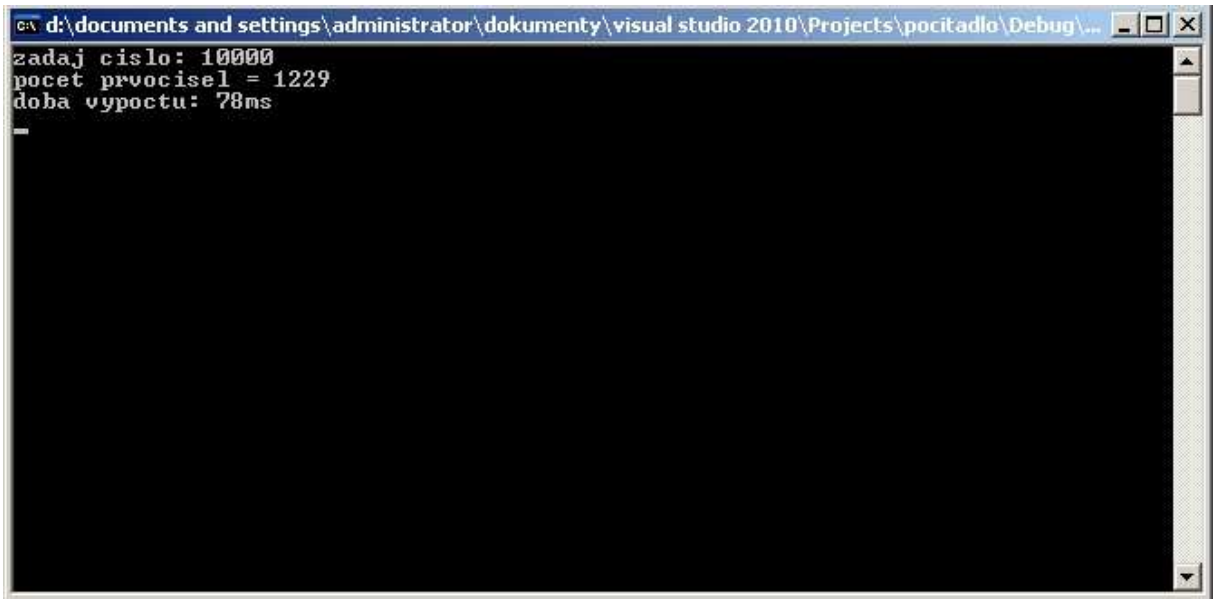
Program bude obsahovať jednu triedu obsahujúcu jednu metódu, rátajúcu počet prvočísel. Mechanizmus testovania čísel je zoptimalizovaný. Pri skúmaní čísla nie je toto číslo delené všetkými hodnotami, ale len hodnotami menšími ako jeho vlastná odmocnina. Nad hodnotou odmocniny sa nachádzajú násobky čísel s hodnotou nižšou ako odmocnina, preto nie je nutné delenie aj týmito číslami. Táto metóda značne skracuje exekučný čas a výpočtovú zložitosť programu.

Triedu chápeme ako objekt, ak by nejaký používateľ mal prístup k našej triede, mohol by vďaka nej zisťovať počet prvočísel bez toho, aby vedel aký mechanizmus je v nej implementovaný. Stačilo by aby jej odovzdal hodnotu, ktorá predstavuje hornú hranicu skúmaného intervalu. Mechanizmus a údaje ktoré trieda obsahuje sú v triede zapúzdrené. To znamená, že sú chránené pred náhodnými alebo nežiadúcimi zmenami zo strany používateľov.

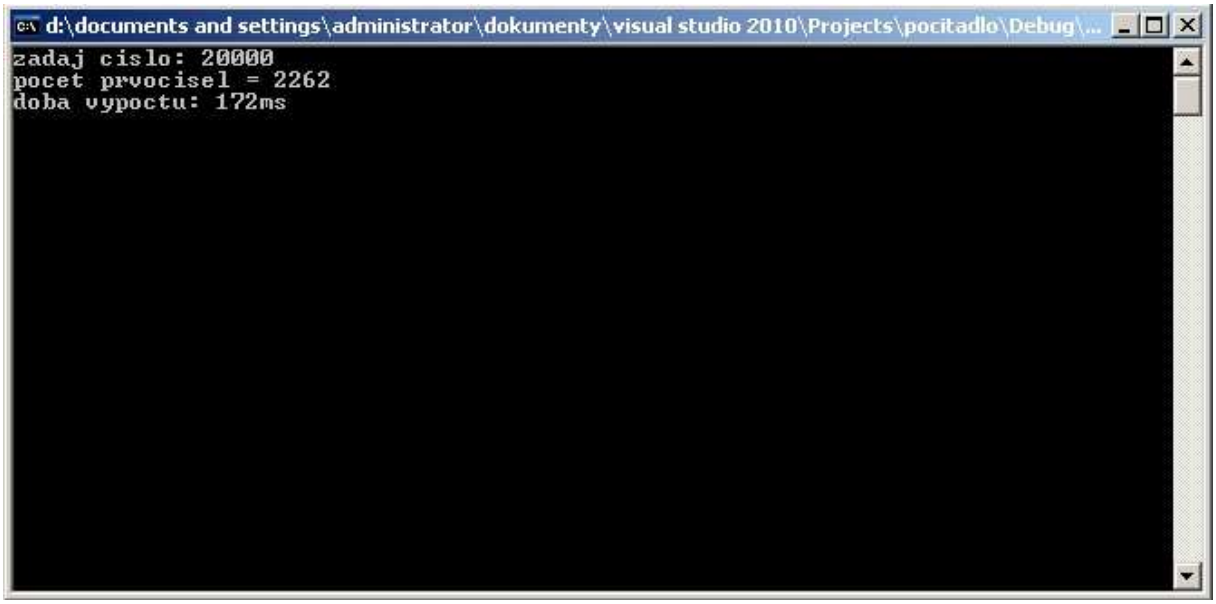
```
#include <iostream>
#include <windows.h>
#include <cmath>                //popis objektov pre matematické operácie

using namespace std;

class Pocitadlo {                           //začiatok triedy
protected:
      unsigned int pocitadlo,pocet;

public:                                     //metóda, ktorá ráta prvočísla
      Pocitadlo() {                         //konštruktor
            pocet = 0;
      }

      int ratac(int ccislo) {
            int j,c;
            double i;
            for(i = 2;i <= ccislo;i++) {
                  pocitadlo = 0;
                  for(j = 1;j <= (sqrt(i));j++) {
                        c=(int)i;           //explicitná typová konverzia
                        if (c % j == 0)
                              pocitadlo++;
                  }
                  if(pocitadlo==1) {
                        pocet++;
                  }
            }
            return pocet;
      };
};

int main()
{
      unsigned int cas2;
      Pocitadlo single;
      int n;
      cout<<"zadaj cislo: ";
```

```
        cin>>n;
        cas2 = GetTickCount();
        cout << "pocet prvocisel = " << single.ratac(n) << "\n";
        cout<<"doba vypoctu: "<< GetTickCount() - cas2 <<"ms"<<endl;
        return 0;
}
```

Rátanie exekučného času je ponechané vo funkcií main, čas je meraný od chvíle zavolania metódy objektu po jej úspešné ukončenie. Vo výstupe je zrejmé, že optimalizáciou nášho algoritmu sme zabezpečili niekoľkonásobné skrátenie exekučného času:



obr 6 doba vykonávania algoritmu pre rozsah 1 až 10000



obr 7 doba vykonávania algoritmu pre rozsah 1 až 20000

Z výstupov je zrejmé obrovské šetrenie zdrojov, kým pre interval (0,10000> bol pri neoptimalizovanom algoritme exekučný čas 1031ms, pri optimalizovanom iba 78ms, čo je

viac ako 13-násobne kratší čas, pri intervale (0,20000> bol čas prvého programu 3844ms, druhého iba 172ms, to je viac ako 22-násobne menej. Z toho vyplýva, že čím väčší je interval, tým viac času nám optimalizácia ušetrí.

Pri vytváraní optimálneho algoritmu sme sa stretli s problémom získania odmocniny pre cyklus, ktorý inkrementuje deliteľov testovaných čísel. Tento cyklus deliteľa zvyšuje až po hodnotu, rovnú odmocnine testovaného čísla. Na odmocnenie čísla sme použili funkciu sqrt zo štandardnej knižnice jazyka C++ pre matematické operácie s názvom cmath. Táto funkcia dokáže odmocniť iba číslo s dátovým typom double, čo by nebol až taký problém, ak by sme o pár krokov ďalej netestovali skúmané číslo hodnotou, aktuálne uloženou v premennej, z ktorej potrebujeme odmocninu pre hornú hranicu cyklu. Na túto hodnotu je aplikovaný binárny operátor modulo, ktorý nám vracia zbytok po celočíselnom delení, takže pracuje iba s premennými, ktorých dátový typ je int, čiže celé číslo. Z celého čísla ale nedokážeme dostať odmocninu, pri určovaní po akú hodnotu sa má cyklus vykonať. Ako teda takýto zákerný problém riešiť? Odpoveďou na otázku je explicitná typová konverzia. Vďaka nej dokážeme pretypovať väčšiu hodnotu double, na menšiu hodnotu integer aj za cenu straty údajov. V tomto prípade je však nutné si na pretypovanú hodnotu vytvoriť novú premennú, pretože cyklus bude mať určite minimálne jednu iteráciu a odmocňovať, ako aj testovať hodnotu operátorom modulo je nutné v každej iterácií príslušných vnorených cyklov.

Poslednou zvláštnosťou nášho algoritmu je neštandardný začiatok cyklu for od hodnoty dva. Najčastejšie cyklus tohoto typu začína od nuly alebo jednotky, no v našom prípade by to znamenalo testovať navyše hodnotu, ktorá prvočíslom nie je. Jednotka spĺňa podmienku testovania, no nie je prvočíslom. Začiatok testovania intervalu od čísla dva šetrí exekučný čas na testovanie jednotky.

3.3 Tretia praktická ukážka

Už vieme vytvoriť vlastnú triedu v jazyku C++. To ale nie je jediná možnosť algoritmizácie, ktorá neporušuje objektovo orientovanú paradigmu programovania. Pri riešení mnohých úloh je efektívnejšie použiť už vytvorené objekty, ktoré sú obsiahnuté v knižniciach jazyka C++, namiesto toho, aby sme vytvárali triedu, aká už existuje. Vlastné triedy sú vhodné na riešenie netypických problémov, no ak potrebujeme v našom kóde vykonať operáciu, ktorá sa často vyskytuje v mnohých algoritmoch, je skoro isté, že takúto operáciu už niekto pred nami vyriešil a zapuzdril do objektu, ktorý je obsiahnutý v niektorej knižnici. Pre urýchlenie a skvalitnenie programátorskej činnosti teda stačí, vyhľadať si vhodný objekt

a poskytnúť mu vstupnú hodnotu, ktorú spracuje a vráti nám výsledok bez toho, aby sme sa zdržiavali s vytváraním nového algoritmu.

V tretej praktickej ukážke sa zoznámime s riešením problémov pomocou preddefinovaných knižničných objektov jazyka C++. Zvýšime náročnosť zadanej úlohy na hľadanie prvočísel v tom zmysle, že budeme chcieť na výstupe vidieť nie len počet prvočísel z intervalu, ale aj ich hodnoty vypísané na štandardný výstup. Takáto úloha pomerne náročná na celkovú algoritmizáciu, ale použitie knižničných objektov nám ju veľmi uľahčí.

Hodnoty nájdených prvočísel si musíme kvôli výpisu niekde ukladať - zapamätať. Možno vás ako prvé napadá pole, ale v C++ existuje viacero vhodných a robustných tried na ukladanie dát z ktorých sa pre takúto úlohu javí najvhodnejším kontajner typu vector.

Aj samotný spôsob prehľadávania intervalu musíme upraviť, aby sme dokázali nájdené čísla v našom vektore meniť. Vektor sa naplní celým zadaným intervalom a čísla, ktoré nie sú prvočíslami sa budú nahrádzať znakom, ktorý určuje, že nejde o prvočíslo. Využijeme tu algoritmus nazývaný Eratostenovo sito. Jeho postup je nasledovný:

- Vyberie prvé číslo z radu, čiže dva, pretože jednotka a nula nie sú prvočísla, tak ich ignorujeme
- Označí dvojku za prvočíslo a zo zoznamu odstráni všetky jeho násobky
- Vyberie číslo tri zo zoznamu, a zistí, či je to prvočíslo
- Označí trojku za prvočíslo a zo zoznamu odstráni všetky jeho násobky
- Vyberie ďalšie číslo zo zoznamu, čo je päť, zisťuje či je to prvočíslo
- Označí päť za prvočíslo a zo zoznamu odstráni všetky jeho násobky
- Vyberie nasledujúce číslo v zozname a testuje ho...

Takto algoritmus pokračuje, až kým nenarazí na číslo, ktoré je väčšie ako odmocnina hornej hranice intervalu, pričom toto číslo označí za prvočíslo. V tomto momente sú všetky nasledujúce čísla, ktoré v zozname ostali, prvočísla.

```
#include <iostream>
#include <vector>        //hlavičkový súbor triedy vector
#include <cmath>
#include <algorithm>     //hlavičkový súbor popisujúci algoritmy STL knižnice

using namespace std;

#define VYHODENE 0       //znak, nahradzujúci neprovočíslo
```

```
template < typename T >  //šablóna pre vytvorenie funkcie pre výpis vektora
void vypis( T zac, T kon) {
      for ( ;zac != kon ;++zac ) {
            std::cout << *zac << " ";
      };
      std::cout << endl;
}


int main() {
 int maxcislo;
 int pocet_prvkov;
 int i;
 int pom;
 cout << "zadaj cislo vacsie ako 2: ";
 cin >> maxcislo;
 if (maxcislo < 2) {
  cout << "mensie ako 2" << endl;
  return 1;
 }
 pocet_prvkov = maxcislo - 1;
 vector<int> v;
 vector<int>::iterator v_iterator_i;
 for (i = 0; i < pocet_prvkov; ++i)
  v.push_back(i + 2);
 vypis(v.begin(), v.end());
 for (v_iterator_i = v.begin(); v_iterator_i != v.end(); ++v_iterator_i)
 {
  if (*v_iterator_i != VYHODENE)
  {
   if (*v_iterator_i <= sqrt((double)maxcislo))
   {
    pom = *v_iterator_i;
    replace_if(v_iterator_i+1,v.end(),not1(bind2nd(modulus<int>(),pom)),
                     VYHODENE);
   }
   else
    break;
   vypis(v.begin(), v.end());
  }
 }
 cout << "pocet prvocisel=" << count_if(v.begin(),v.end(),bind2nd(greater<int>(),
                                      0)) << endl;
 return 0;
}
```

Náročnosť tohoto algoritmu je zo sémnatickej stránky pomerne vysoká, preto sa v jeho analýze budeme venovať jeho uceleným častiam osobitne. Využívame algoritmy z knižnice Standard Template Library, z ktorej pochádza aj náš objekt vektor. Príkaz

```
vector<int> v;
```

vytvorí objekt vector s prvkami celočíselného dátového typu int, ktorý ale neobsahuje žiadne hodnoty, následne príkazom

```
vector<int>::iterator v_iterator_i;
```

vytvoríme objekt iterátora na prvky vektora v. Po vytvorení týchto objektov napĺňame vektor vo for cykle celými číslami od 2 po zadanú hranicu intervalu. Hodnoty pridávame postupne na koniec vektora. Prvá hodnota, ktorou sa vektor naplní, sa zapíše na jeho koniec. Keďže bol

vektor na začiatku prázdny, nič pred touto hodnotou nie je, takže sa stáva prvým prvkom vektora. Nasledujúci riadok vypíše celý vektor. Dostávame sa k najzaujímavejšiemu fragmentu zdrojového kódu a to k samotnému testovaniu, či sa jedná o prvočíslo, ktoré je uskutočnené v nasledujúcom cykle:

```
for (v_iterator_i = v.begin(); v_iterator_i != v.end(); ++v_iterator_i)
{
 if (*v_iterator_i != VYHODENE)
 {
  if (*v_iterator_i <= sqrt((double)maxcislo))
  {
   pom = *v_iterator_i;
   replace_if(v_iterator_i+1,v.end(),not1(bind2nd(modulus<int>(),pom)),
              VYHODENE);
  }
  else
   break;
  vypis(v.begin(), v.end());
 }
}
```

Ak hodnota prvku na ktorú ukazuje iterátor nie je označená príznakom "VYHODENE" a súčasne ak hodnota prvku na ktorú ukazuje iterátor je menšia alebo rovná odmocnine najväčšieho čísla z intervalu, tak sa vykonajú príkazy v tele vnoreného rozhodovacieho príkazu if. Do pomocnej premennej pom sa priradí hodnota prvku, na ktorú ukazuje iterátor.

V druhom kroku vykonávania tela rozhodujúceho príkazu máme zapísaný zložený príkaz:

```
replace_if(v_iterator_i+1,v.end(),not1(bind2nd(modulus<int>(),pom)),VYHODENE);
```

Prvý a druhý parameter príkazu sú začiatok a koniec intervalu. Tretím prametrom je unárny predikát binárnej funkcie modulus pre typ integer. Modulus je binárna funkcia, ktorú v tomto prípade potrebujeme použiť ako unárny predikát. Na príslušnú konverziu a priradenie druhého parametra sa používa funkcioa bind2nd, ktorá nám mení binárnu funkciu modulus na unárnu, vďaka čomu ju môžeme použiť ako unárny predikát a odovzdať mu druhý parameter, v našom prípade pomocnú premennú pom. Funkcia not1 nám výsledok modulu neguje v zmysle logickej hodnoty. To znamená, že ak je výsledok funkcie modulus rovný nule, výsledok výrazu bude po negácií pravdivá hodnota true, ak bude výsledok nenulový, po jeho znegovaní sa nám vráti nepravdivá hodnota false a aktuálna hodnota sa prepíše na príznak (hodnotu) VYHODENE.

Za predpokladu, že je hodnota prvku na ktorú ukazuje iterátor väčšia ako odmocnina najväčšieho čísla intervalu, opustíme cyklus vďaka príkazu break.

Po opustení tela vnoreného rozhodovacieho príkazu sa program ocitne v tele nadradeného rozhodovacieho príkazu a vypíše celý interval so zmenenými hodnotami neprvočísel na VYHODENE, v našom prípade nuly. Tento cyklus sa opakuje, kým sa nedostane na koniec vektora, pričom obsahuje aj aktuálny výpis. Vypisovať vektor po každom vynulovaní násobkov konkrétneho prvočísla nám umožňuje naša šablóna funkcie výpis. Prekladač podľa šablóny definovanej pred telom funkcie main vytvorí funkciu pre používaný dátový typ. Vytvorí sa iba jedna inštancia tejto funkcie, pretože pri každom volaní sú jej odovzdávané parametre toho istého dátového typu.

```
void vypis( T zac, T kon) {
      for ( ;zac != kon ;++zac ) {
            std::cout << *zac << " ";
      };
      std::cout << endl;
}
```

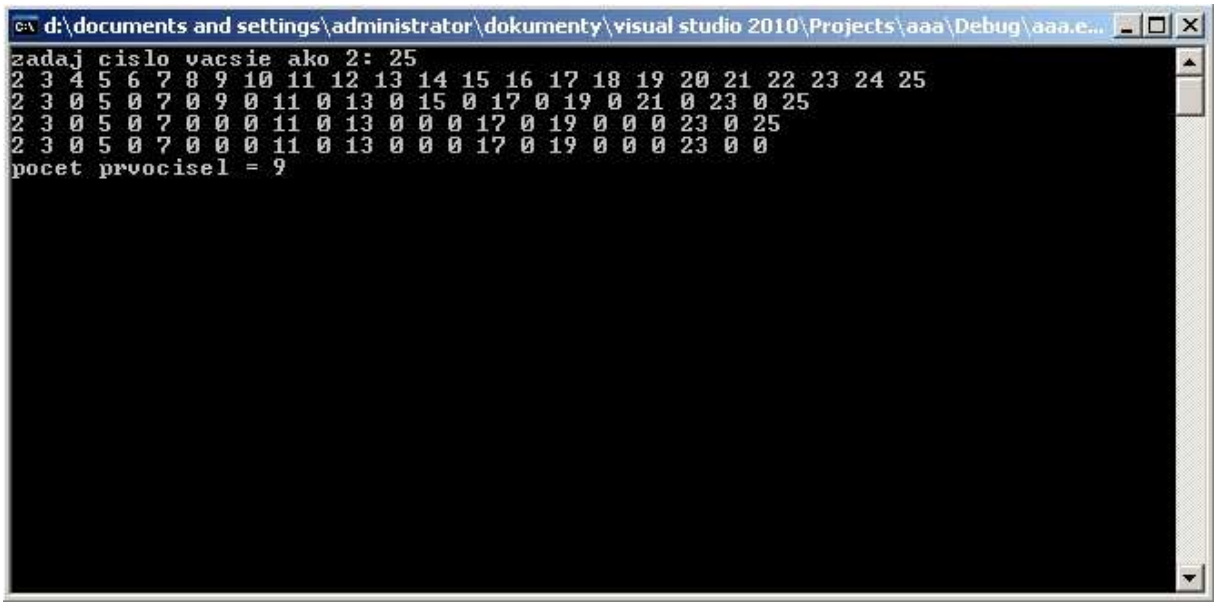
Šablóny funkcií, prípadne aj objektov, sú mocným nástrojom jazyka C++. Sú základom generického programovania, t.j. programovania, ktoré nie je závislé na type premenných.

Posledný príkaz programu spočíta počet prvočísel vďaka funkcií:

```
count_if(v.begin(), v.end(), bind2nd(greater<int>(), 0))
```

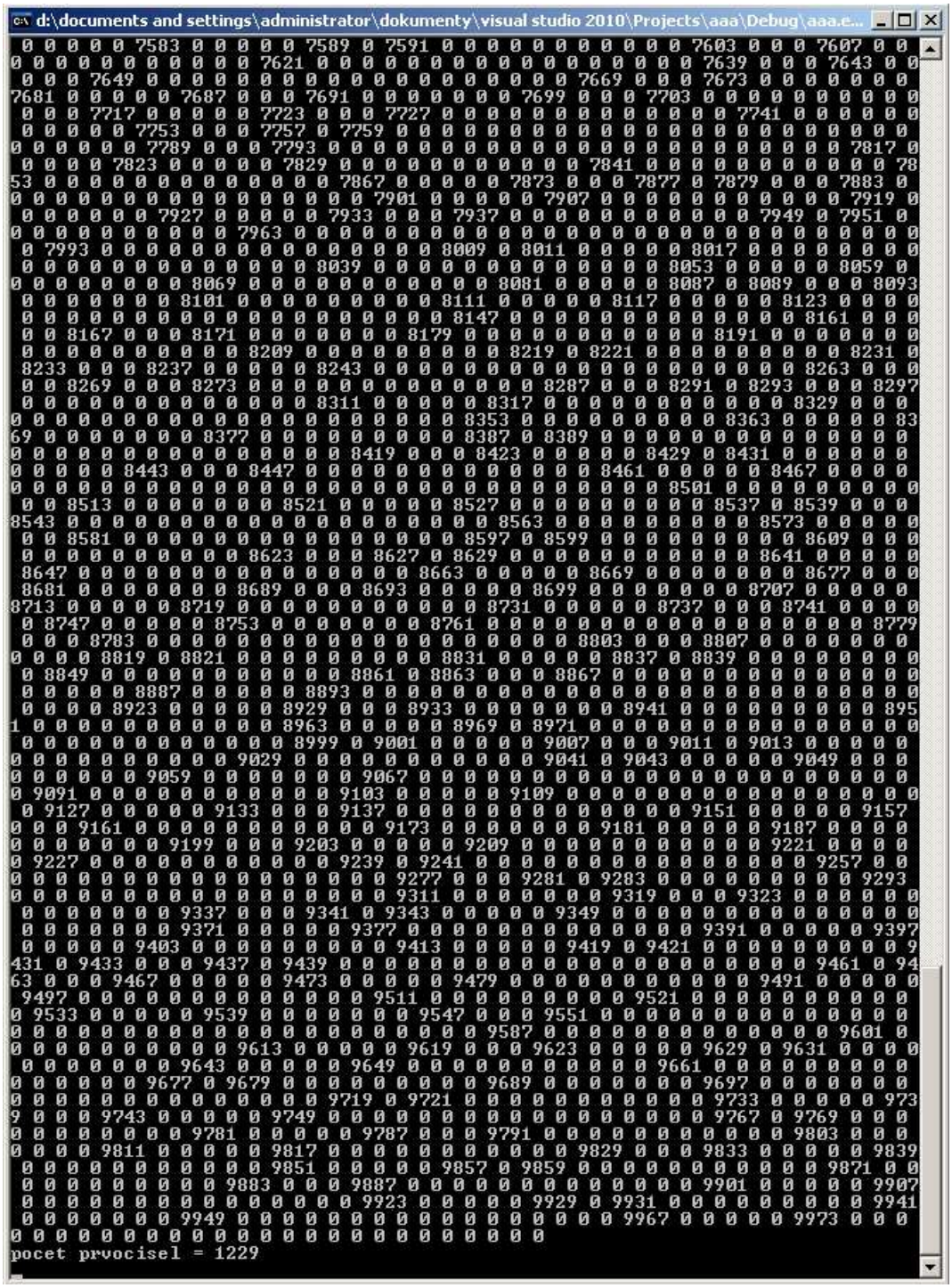
Táto funkcia spočíta všetky čísla v intervale, ktoré sú väčšie ako nula, tak dostaneme počet prvočísel. Opäť sme museli binárnu funkciu greater "konvertovať" pomocou funkcie bind2nd na unárny predikát a tomu odovzdať druhý parameter, s ktorým má porovnávať čísla v intervale.

Výstup programu vyzerá nasledovne:



obr 8 priebeh vykonávania algoritmu v rozsahu 1 až 25

Z výstupu je zrejmé, že vďaka výpisu intervalu v cykle, vidno celý priebeh postupu nulovania neprvočísel. Pri malých intervaloch je to dobrá metóda, pretože vidíme, či algoritmus pracuje podla našich predstáv. Ale pokiaľ by sme chceli skúmať väčší interval, výpis by začal byť neprehľadný, nie len kôli jeho násobnosti, ale aj pre množstvo vyradených (v našom prípade vynulovaných) neprvočísel. Pri skúmaní intervalu do čísla desaťtisíc by sa nám vo výstupe zobrazilo niekoľkonásobne viac čísel, viď obrázok:



obr 9 priebeh vykonávania algoritmu v rozsahu 1 až 10000

Na odstránenie tejto neprehľadnosti budeme v ďalšej ukážke vyžadovať, aby sa nám vo výpise zobrazil iba zoznam prvočísel z intervalu bez akéhokoľvek postupu nulovania násobkov nájdených prvočísel a bez znaku vyhodenia čísel, ktoré nespĺňajú podmienku prvočíselnosti.

3.4 Štvrtá praktická ukážka

V štvrtej praktickej ukážke mierne modifikujeme predchádzajúci kód v zmysle odstránenia zbytočných dát z výstupu. Vo výpise budeme vyžadovať iba zoznam prvočísel a počet prvočísel v intervale. Myšlienka algoritmu prehľadávania ostane neporušená, zmenou bude vymazávanie vyhodených (v našom prípade vynulovaných) prvkov z vektora a následný výpis vektora, ktorý ale bude obsahovať iba nenulové prvky. Táto operácia značne skráti a sprehľadní výstup programu.

```
#include <iostream>
#include <vector>
#include <math.h>
#include <algorithm>
#include <functional>

using namespace std;

template < typename T >
void vypis( T zac, T kon) {
      for ( ;zac != kon ;++zac ) {
            std::cout << *zac << " ";
      };
      std::cout << endl;
}

int main() {
 int maxcislo;
 int i;

 cout << "zadaj cislo vacsie ako 2: ";
 cin >> maxcislo;
 if (maxcislo < 2) {
  cout << "mensie ako 2" << "\n";
  return 1;
 }
 vector<int> v;
 vector<int>::iterator v_iterator_i;
 vector<int>::iterator v_iterator_koniec;

 for (i = 0; i < maxcislo - 1; ++i)
  v.push_back(i + 2);

 v_iterator_i = v.begin();
 v_iterator_koniec = v.end();
 while (v_iterator_i != v_iterator_koniec)
 {
  if (*v_iterator_i <= (sqrt(double(maxcislo))))
  {
   v_iterator_koniec=remove_if(v_iterator_i+1,v_iterator_koniec,
                               not1(bind2nd(modulus<int>(), *v_iterator_i)));
  }
  else
   break;
  ++v_iterator_i;
 }
 vypis(v.begin(), v_iterator_koniec);
 cout << "pocet prvocisel = " << v_iterator_koniec - v.begin() << endl;
```

```
 return 0;
}
```

Napriek rovnakej myšlienke prehľadávania vektora je v tejto ukážke zmenená vnútorná implementácia testovania prvkov. Cyklus for je nahradený cyklom while, ktorý nám v tomto prípade ponúka komfortnejšie riešenie daného problému a ďalšie šetrenie pamäte a, pre používateľa najdôležitejšie, skrátenie exekučného času programu. Túto zmenu nám umožnilo mazanie vynulovaných prvkov vektora. Vymazanie síce prebehne až po prejdení celého vektora, ale cyklus zabezpečuje, že vynulované prvky sú posúvané na koniec vektora. Táto operácia nám zabezpečuje skracovanie prehľadávanej časti vektora obojsmerne. Od začiatku sa posúvame cyklom a koniec sa nám približuje posúvaním vyhodených násobkov prvočísel za najväčšiu hodnotu intervalu.

Okrem pôvodných objektov vektora a iterátora máme vytvorený ďalší objekt iterátor_koniec, ktorý ukazuje na pozíciu za posledným prvkom intervalu uloženým vo vektore.

Hlavná časť nášho programu sa skrýva v cykle while:

```
while (v_iterator_i != v_iterator_koniec)
{
 if (*v_iterator_i <= (sqrt(double(maxcislo))))
 {
  v_iterator_koniec=remove_if(v_iterator_i+1,v_iterator_koniec,
                              not1(bind2nd(modulus<int>(), *v_iterator_i)));
 }
 else
  break;
 ++v_iterator_i;
}
```

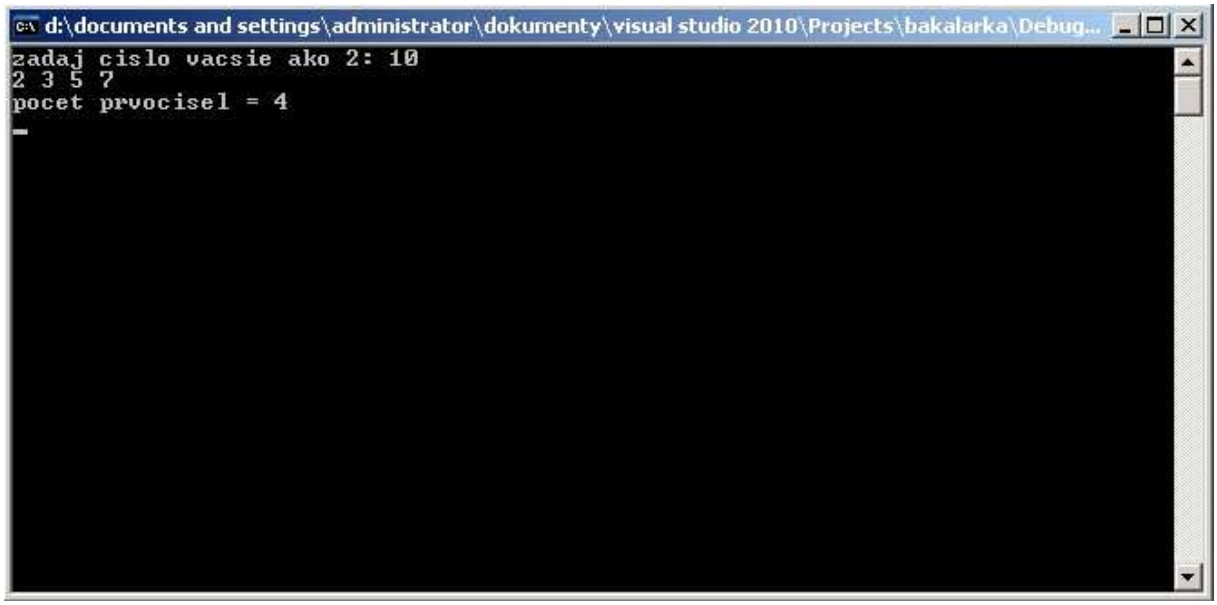
Cyklus bude prebiehať, kým bude hodnota iterátora rôzna od hodnoty iterátora ukazujúceho na koniec vektora, alebo kým nebude cyklus prerušený príkazom break. V tele cyklu sa nachádza jeden rozhodujúci príkaz. Na rozdiel od predchádzajúcej ukážky nie je potrebný zložený rozhodujúci príkaz, pretože neprvočíselné hodnoty nie je potrebné meniť na inú hodnotu, ktorá bola v treťom algoritme nulová. Telo rozhodovacieho príkazu sa vykoná, ak hodnota na ktorú ukazuje iterátor bude menšia, nanajvýš rovná odmocnine z najväčšej hodnoty intervalu.

V nasledujúcom riadku máme zložený výraz. Ide o priradenie hodnoty nového konca vektora do iterátora ukazujúceho na koniec. Funkcia remove_if násobky skúmaného prvočísla nevymaže, iba ich presunie na koniec vektora. To znamená, že za iterátorom ukazujúcim na koniec vektora budú hodnoty, ktoré nespĺňajú podmienky prvočíselnosti. Časť s funkciou modulus bola vysvetlená v predchádzajúcej praktickej ukážke. Nie je nutné používať žiadnu

pomocnú premennú ako v predchádzajúcom prípade, pretože iterátory v rozhodujúcom príkaze sú lokálne a neporušia nám hodnoty iterátorov mimo tela rozhodovacieho príkazu. Po vykonaní príkazov v tele cyklu sa posunieme na ďalšie číslo intervalu. Ak sa dostaneme na číslo, ktoré je väčšie alebo rovné odmocnine najväčšej hodnoty intervalu a toto číslo je označené za prvočíslo, opúšťame cyklus príkazom break, pretože všetky hodnoty, ktoré sa nachádzajú vo vektore od začiatku až po iterátor ukazujúci na koniec sú prvočísla.

Za telom cyklu sa nachádza výpis nájdených prvočísel pomocou funkcie vytvorenej prekladačom vďaka šablóne funkcie vypis. Počet prvočísel sa skúma veľmi jednoduchým spôsobom a to odčítaním hodnoty iterátora ukazujúceho na koniec a funkcie začiatku vektora.

Vďaka týmto úpravám sa výpis programu stal prehľadnejší:



obr 10 výpis zoznamu prvočísel po použití remove_if v rozsahu 1 až 10

obr 11 výpis zoznamu prvočísel po použití remove_if v rozsahu 1 až 10000

Drobnými zmenami v tele algoritmu, sme dokázali sprehľadniť aj výpis veľkého intervalu a vypísania všetkých jeho prvočísel.

3.5 Piata praktická ukážka

Posledná praktická ukážka nám bude prezentovať inováciu v jazyku C++, ktorá je novinkou posledného ISO štandardu. Jedná sa o takzvaný lambda výraz. Tento výraz nám umožňuje definovať anonymnú funkciu. Jeho použitie je časté pri paralelnom programovaní. V tejto parktickej ukážke bude prezentovaný na jednoduchom probléme.

Budeme požadovať zrátanie počtu párnych čísel v používatelovi zadanom intervale.

Generický model:

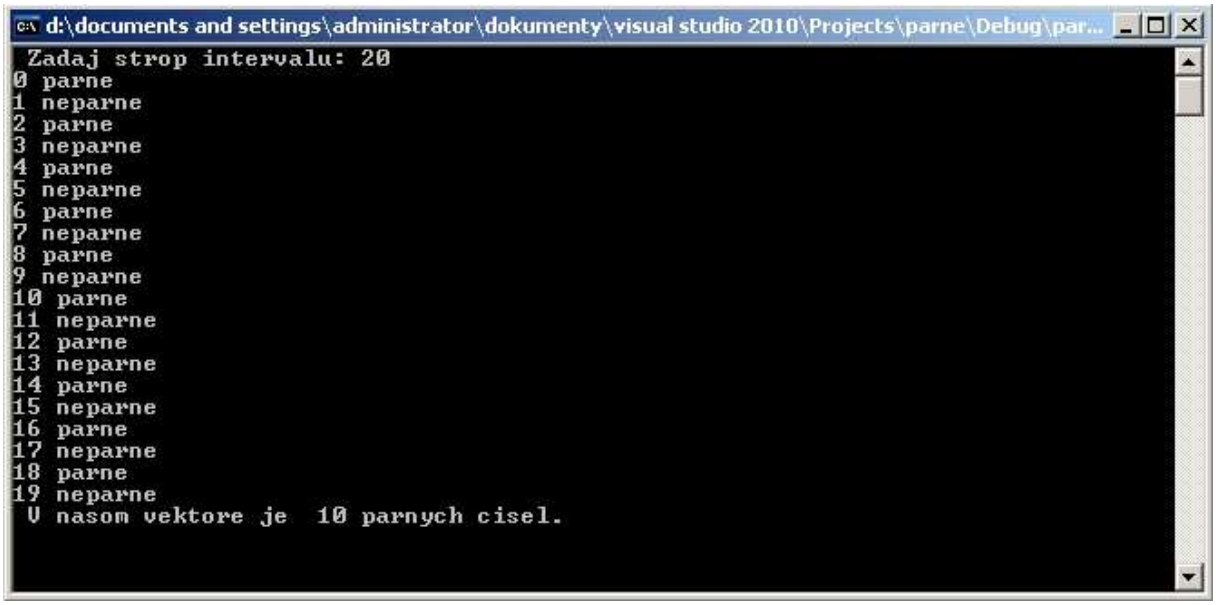
[inicializacnySymbol](formalneParametre)

{

telo λ-výrazu

}

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

int main()
{
   int maxcislo;
   cout << " Zadaj strop intervalu: ";
   cin >> maxcislo;

   vector<int> v;

   for (int i = 0; i < maxcislo; ++i)
   {
      v.push_back(i);
   }


   int parne = 0;
   for_each(v.begin(), v.end(), [&parne] (int n) {
      cout << n;

      if (n % 2 == 0)
      {
               cout << " parne " << endl;
               parne++;
      }
      else
      {
         cout << " neparne " << endl;
      }
   });


   cout << " V nasom vektore je  " << parne << " parnych cisel." << endl;
   return 0;
}
```



obr 12 výpis počtu párnych čísel po použití anonymnej funkcie

Táto ukážka dokonale demonštruje anonymnú funkciu, ktorá nám vyráta počet párnych čísel v objekte vektor a zároveň vypíše všetky prvky intervalu a označí párne a nepárne čísla. Táto novinka skrýva nevídané možnosti vo svete moderného programovania.

Záver

Tvorba počítačových programov je v posledných rokoch veľmi široký obor. Človek sa s nimi stretáva denne, pričom v mnohých prípadoch si to ani neuvedomuje. Algoritmizácia prenikla do mnohých oblastí nášho života z dôvodu skvalitnenia životnej úrovne ľudstva a uľahčenia mnohých úkonov. V domácnostiach nám programovateľné sportebiče pomáhajú zvládať každodenné starosti, pri cestovaní nám navigácie a výbava automobilov pomáha bezpečnejšie prekonávať veľké vzdialenosti a nezabúdajme ani na to, že nás bavia v podobe dnes veľmi obľúbených počítačových hier.

V tejto práci boli ukázané niektoré techniky, vďaka ktorým sa dajú vytvoriť prevratné programy. Mňa osobne veľmi obohatila práca so štandardnou knižnicou jazyka C++ STL. Keď človek dokáže zvládnuť základné a niektoré pokročilé princípy programovania v jazyku C++, je pripravený na tvorbu programov, ktoré mu uľahčia mnoho starostí a prinesú veľa zábavy.

Zoznam použitej literatúry

1. Knihy / Monografie

KENT, J. 2009. C++ bez předchozích znalostí. Brno : Computer Press a.s., 2009. 254 s. ISBN 978-80-251-2411-6.

PROKOP, J. 2009. Algoritmy v jazyku C a C++. Praha : Grada Publishing a.s., 2009. 153 s. ISBN 978-80-247-2751-6.

PRATA, S. 2007. Mistrovství v C++.3.vyd. Brno : Computer Press a.s., 2007. 1119 s. ISBN 978-80-251-1749-1.

HANÁK, J. 2010. Inovácie v jazykoch Visual Basic 2010, C# a C++. Brno : Artax a.s., 2010. 60 s. ISBN 978-80-87017-08-1.

JOUSUTTIS, N. 1999. C++ Standard Library: A Tutorial and Reference. USA : Addison Wesley., 1999. 832 s. ISBN 0-201-37926-0.

NENADÁL, K. – VÁCLAVÍKOVÁ, D. 1991. Turbo C :popis jazyka. Praha : Grada a.s., 1991. 263 s. ISBN 80-85424-08-8.

ANDREWS, M. 1997. Programujeme v jazyce C++. Praha : Computer Press a.s., 1997. 387 s. ISBN 80-85896-91-5..

Zoznam obrázkov

obr 1 rozdelenie kontajnerov 10
obr 2 funkcie begin() a end() pre kontajner 12
obr 3 kategórie iterátorov 12
obr 4 doba vykonávania algoritmu pre rozsah 1 až 10000 17
obr 5 doba vykonávania algoritmu pre rozsah 1 až 20000 17
obr 6 doba vykonávania algoritmu pre rozsah 1 až 10000 19
obr 7 doba vykonávania algoritmu pre rozsah 1 až 20000 19
obr 8 priebeh vykonávania algoritmu v rozsahu 1 až 25 25
obr 9 priebeh vykonávania algoritmu v rozsahu 1 až 10000 26
obr 10 výpis zoznamu prvočísel po použití remove_if v rozsahu 1 až 10 29
obr 11 výpis zoznamu prvočísel po použití remove_if v rozsahu 1 až 10000 30
obr 12 výpis počtu párnych čísel po použití anonymnej funkcie 32