EKONOMICKÁ UNIVERZITA V BRATISLAVE
FAKULTA HOSPODÁRSKEJ INFORMATIKY

Porovnanie exekučnej efektívnosti vyhľadávania štruktúrovaných dát uložených v poli a v jednosmernom lineárnom zozname v programe vytvorenom v jazyku C++

Diplomová práca

2022 **Lukáš Šury**

EKONOMICKÁ UNIVERZITA V BRATISLAVE
FAKULTA HOSPODÁRSKEJ INFORMATIKY

Porovnanie exekučnej efektívnosti vyhľadávania štruktúrovaných dát uložených v poli a v jednosmernom lineárnom zozname v programe vytvorenom v jazyku C++

Diplomová práca

Študijný program: Informačný manažment

Študijný odbor: Informačný manažment

Školiace pracovisko: Katedra aplikovanej informatiky

Vedúci záverečnej práce: Ing. Igor Košťál, PhD.

Bratislava 2022 **Lukáš Šury**

Čestné vyhlásenie

Čestne vyhlasujem, že som záverečnú prácu vypracoval samostatne a že som uviedol všetku použitú literatúru súvisiacu so zameraním bakalárskej práce.

V Trnave, 28. 04. 2022 .....

Podpis

Ekonomická univerzita v Bratislave
Fakulta hospodárskej informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Lukáš Šury
Študijný program: informačný manažment (Jednoodborové štúdium, inžinierský II. st., denná forma)
Študijný odbor: ekonómia a manažment
Typ záverečnej práce: Inžinierska záverečná práca
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Porovnanie exekučnej efektívnosti vyhľadávania štruktúrovaných dát uložených v poli a v jednosmernom lineárnom zozname v programe vytvorenom v jazyku C++

Anotácia: Diplomant v práci zanalyzuje možnosti ukladania štrukturovaných dát C ++ aplikáciou v jej úložisku a spôsoby vyhľadávania v týchto dátach metódami tejto aplikácie a porovná tieto možnosti a spôsoby s terajším spôsobom ukladania štrukturovaných dát inými aplikáciami a so spôsobmi vyhľadávania v týchto dátach metódami týchto iných aplikácií. V rámci diplomovej práce diplomant v programovacom jazyku C++ vytvorí oknovú aplikáciu, ktorá bude schopná uložiť si štruktúrované dáta do objektov statického poľa a do dátových častí dátových prvkov lineárneho jednosmerného zoznamu. C++ aplikácia pomocou metód s implementovaným vybratým vyhľadávacím algoritmom ponúkne používateľovi možnosť vyhľadávať podľa zvoleného kritéria v štruktúrovaných dátach uložených v poli objektov a v jednosmernom lineárnom zozname, pričom si každá metóda bude merať exekučné časy jednotlivých vyhľadávaní. Pomocou porovnania exekučných časov jednotlivých vyhľadávaní diplomant porovná exekučnú efektívnosť vyhľadávania podľa zvolených kritérií v rovnakých sadách štrukturovaných dát uložených v poli a v jednosmernom lineárnom zozname.

Vedúci: Ing. Igor Košťál, PhD.
Katedra: KAI FHI - Katedra aplikovanej informatiky FHI
Vedúci katedry: Ing. Mgr. Peter Schmidt, PhD.

Dátum zadania: 22.10.2018

Dátum schválenia: 23.10.2018 Ing. Mgr. Peter Schmidt, PhD.
vedúci katedry

ABSTRAKT

ŠURY, Lukáš: *Porovnanie exekučnej efektívnosti vyhľadávania štruktúrovaných dát uložených v poli a v jednosmernom lineárnom zozname v programe vytvorenom v jazyku C++* – Ekonomická univerzita v Bratislave. Fakulta hospodárskej informatiky; Katedra aplikovanej informatiky. – Vedúci záverečnej práce: Ing. Igor Košťál, PhD. – Bratislava: FHI, 2022, 66 s.

Cieľom diplomovej práce je zanalyzovať vybrané dátové štruktúry, ktoré sa používajú na ukladanie štruktúrovaných dát v programovacom jazyku C++. Následne vytvoriť program v jazyku C++ s implementovanými dátovými štruktúrami pole a jednosmerný lineárny zoznam, v ktorých budeme štruktúrované dáta vyhľadávať. Exekučné časy týchto vyhľadávaní budeme pomocou metód aplikácie merať a následne porovnávať. V prvej kapitole sa zaoberáme definíciou pojmu štruktúrované dáta, analýzou vybraných dátových štruktúr a ich implementáciou v jazyku C++. V poslednej časti kapitoly porovnávame vybraný program, ktorý ukladá štruktúrované dáta s programom, ktorý budeme vytvárať my. Druhá kapitola opisuje ciele práce a použitú metodiku. Tretia kapitola sa venuje tvorbe programu v jazyku C++ a implementácii dátových štruktúr pole a jednosmerný lineárny zoznam, v ktorých budeme vyhľadávať štruktúrované dáta, merať časy vyhľadávaní a porovnávať ich.

Kľúčové slová: dátové štruktúry, vyhľadávanie, pole, jednosmerný lineárny zoznam, jazyk C++

ABSTRACT

ŠURY, Lukáš: *Enforcement efficiency comparison, available resources stored in array and one-way line, which can be created in C ++* – University of economics in Bratislava. Faculty of economic informatics; Department of applied informatics. – Thesis supervisor: Ing. Igor Košťál, PhD. – Bratislava: FEI, 2022, 66 p.

The aim of the diploma thesis is to analyze selected data structures that are used to store structured data in the C ++ programming language. Then create a program in C ++ with implemented array and singly linked list in which we will search for structured data. We will measure and then compare the execution times of these searches using the methods of our application. In the first chapter we deal with the definition of the term structured data, analysis of selected data structures and their implementation in C ++. In the last part of the chapter we compare the selected program, which stores structured data with the program that we will create. The second chapter describes the objectives of the work and the methodology used. The third chapter deals with the creation of a program in C ++ and the implementation of data structures array and singly linked list in which we will search for structured data, measure search times and compare them.

Keywords: data structures, searching, arrays, singly linked list, C ++ language

Obsah

Úvod

Pri vytváraní programov má programátor okrem iného za úlohu optimalizovať program tak, aby fungoval čo najplynulejšie, najrýchlejšie s čo najnižším využitím pamäte ale popri tom musí zachovať požadovanú funkcionalitu a dodržať stanovené kritériá programu. Aby bolo možné tieto ciele dosiahnuť, je dôležité, aby poznal široké spektrum nástrojov programovacieho jazyka, vďaka ktorým je takáto optimalizácia možná.

V programovacom jazyku C++ existuje na výber veľká škála možností, akými je možné program optimalizovať pre rôzne potreby. Programátor môže šetriť pamäť napríklad vhodným výberom dátového typu alebo efektívnym využitím ukazovateľov na prácu s údajmi v pamäti alebo pomocou správne zvoleného algoritmu zvýšiť rýchlosť a výkonnosť programu.

V práci sa zameriame na jednotlivé spôsoby ukladania dát do vybraných dátových štruktúr a zanalyzujeme, ako sú rôzne operácie s dátami v týchto dátových štruktúrach časovo alebo pamäťovo efektívne. Pomocou vývojového prostredia *Visual Studio 2022* vytvoríme program v jazyku C++, ktorý bude porovnávať časovú efektívnosť vyhľadávania štruktúrovaných dát v poli a v jednosmernom lineárnom zozname. Vďaka tomuto testovaniu zistíme, ktorá z dvoch vybraných dátových štruktúr je vhodnejšia na prehľadávanie a na vyhľadávanie jednotlivých záznamov.

1 Súčasný stav riešenej problematiky doma a v zahraničí

1.1 Štruktúrované dáta

Pri práci s údajmi alebo pri analytike sa často diskutuje o pojmoch štruktúrované, neštruktúrované a pološtruktúrované údaje. Tieto tri formy údajov sa v súčasnosti stali relevantnými pre všetky typy obchodných aplikácií. Štruktúrované údaje existujú už nejaký čas a tradičné systémy sa stále spoliehajú na túto formu údajov. V posledných rokoch však došlo k rýchlemu nárastu tvorby pološtruktúrovaných a neštruktúrovaných zdrojov údajov. Výsledkom je, že čoraz viac firiem sa teraz snaží posunúť svoju analýzu na vyššiu úroveň zahrnutím všetkých troch foriem údajov. (Naeem, 2020)

Štruktúrované dáta sú informácie, ktoré boli naformátované a transformované do dobre definovaného dátového modelu. Nespracované údaje sa mapujú do vopred navrhnutých polí, ktoré sa potom dajú ľahko extrahovať a čítať cez SQL. Relačné databázy SQL pozostávajúce z tabuliek s riadkami a stĺpcami sú dokonalým príkladom štruktúrovaných údajov. Relačný model štruktúrovaných dát efektívne využíva pamäť, pretože minimalizuje redundanciu dát. To však tiež znamená, že štruktúrované údaje sú vzájomne prepojené a menej flexibilné.
(Naeem, 2020)

Štruktúrované dáta generujú ľudia aj stroje. Existuje mnoho príkladov štruktúrovaných údajov generovaných strojmi, ako sú údaje z pokladne, čiarové kódy a štatistiky z rôznych webov. Podobne každý, kto niekedy pracoval s údajmi, aspoň raz za život použil tabuľky, čo je klasický prípad štruktúrovaných údajov generovaných ľuďmi. Vďaka organizácii štruktúrovaných údajov je ich analýza jednoduchšia ako pri pološtruktúrovaných a neštruktúrovaných údajoch. (Naeem, 2020)

Údaje nemusia byť vždy štruktúrované alebo neštruktúrované; pološtruktúrované údaje alebo čiastočne štruktúrované údaje sú ďalšou kategóriou medzi štruktúrovanými a neštruktúrovanými údajmi. Pološtruktúrované údaje sú typom údajov, ktoré majú určité konzistentné a jednoznačné vlastnosti. Neobmedzuje sa na pevnú štruktúru, aká je potrebná pre relačné databázy. Pri takýchto dátach sa používajú metadáta alebo sémantické značky, aby boli lepšie spracovateľné a prehľadnejšie, stále však obsahujú určitú variabilitu a nejednotnosť. (Naeem, 2020)

Neštruktúrované dáta sú definované ako dáta prítomné v absolútne nespracovanej forme. Tieto údaje sa ťažko spracúvajú kvôli ich zložitému usporiadaniu a formátovaniu. Neštruktúrované údaje môžu mať rôzne formy a môžu prichádzať z rôznych zdrojov,

vrátane príspevkov na sociálnych sieťach, chatov, satelitných snímok, e-mailov alebo prezentácií. Na rozdiel od toho sa štruktúrované údaje riadia preddefinovanými modelmi údajov a dajú sa ľahko analyzovať. Príklady štruktúrovaných údajov zahŕňajú abecedne usporiadané mená zákazníkov a správne usporiadané čísla kreditných kariet. Po pochopení definície neštruktúrovaných údajov sa pozrime na niekoľko príkladov. (Naeem, 2020)

Pre lepšie pochopenie rozdielu môžeme použiť analógiu z bežného života. Predpokladajme, že existujú tri typy pracovných pohovorov: neštruktúrované, pološtruktúrované a štruktúrované pohovory. Pri pohovore v neštruktúrovanom formáte sú pokladané otázky výlučne osoby, ktorá vedie pohovor. Môže sa rozhodnúť, aké otázky chce položiť a v akom poradí budú položené. Populárne príklady neštruktúrovaných otázok zahŕňajú „povedzte nám niečo o sebe“ a „opíšte svoju ideálnu pracovnú pozíciu“. (Naeem, 2020)

Ďalším typom je štruktúrovaný pohovor. V takom prípade bude osoba, ktorá vedie pohovor striktne postupovať podľa scenára vytvoreného personálnym oddelením a rovnaký scenár bude použitý pre všetkých uchádzačov. (Naeem, 2020)

Tretím typom je pološtruktúrovaný pohovor. V takomto pohovore bude osoba, ktorá vedie pohovor kombinovať prvky neštruktúrovaného aj štruktúrovaného rozhovoru. Zahŕňa kvantitatívne a konzistentné prvky, podobne ako štruktúrovaný rozhovor. Zároveň však bude mať pohovor flexibilitu prispôsobenia otázok podľa situácie. (Naeem, 2020)

Na zopakovanie, hlavný rozdiel medzi neštruktúrovanými a pološtruktúrovanými údajmi je v tom, že neštruktúrované údaje nemajú žiadny vopred definovaný formát, zatiaľ čo pološtruktúrované údaje sú len čiastočne neštruktúrované. (Naeem, 2020)
V tabuľke č. 1 je na príklade znázornený rozdiel medzi štruktúrovanými, pološtruktúrovanými a neštruktúrovanými údajmi.

V tejto práci sa budeme zaoberať štruktúrovanými dátami, akým spôsobom sa dajú takéto dáta ukladať v aplikácii vytvorenej v jazyku C++ a akými spôsobmi sa dajú vyhľadávať jednotlivé údaje v týchto dátach.

Neštruktúrované dáta	Pološtruktúrované dáta	Štruktúrované dáta			
Univerztia má 2500 študentov. Marekove ID študenta je číslo jedna, má 21 rokov a má bakalársky titul. Davidove ID študenta je číslo dva, má 25 rokov a má inžiniersky titul. Monikine ID študenta je číslo tri, má 22 rokov a má bakalársky titul.	<Univerzita> <Student ID = "1"> <Meno> Marek </Meno> <Vek> 21 </Vek> <Titul> Bc. </Titul> </Student> <Student ID = "2"> <Meno> David </Meno> <Vek> 25 </Vek> <Titul> Ing. </Titul> </Student> </Univerzita>	**ID Študenta**	**Meno**	**Vek**	**Titul**
		1	Marek	21	Bc.
		2	David	25	Ing.
		3	Monika	22	Bc.

Tabuľka č. 1 – Porovnanie neštruktúrovaných, pološtruktúrovaných a štruktúrovaných dát (vlastné spracovanie)

1.2 Ukazovatele a pamäť

Aby sme lepšie pochopili, ako si C++ aplikácia ukladá údaje do dátových štruktúr a do pamäte, predstavíme si v tejto kapitole základné typy pamäte, s ktorou aplikácia pracuje a vysvetlíme si základy fungovania ukazovateľov, ktoré budeme potrebovať pri opise dátových štruktúr a vytváraní aplikácie. (vlastné spracovanie)

Kľúčom k pochopeniu ukazovateľov je pochopenie toho, ako sa spravuje pamäť v C++ programe. Ukazovatele obsahujú adresy v pamäti a ak nerozumieme tomu, ako je pamäť organizovaná a spravovaná, je ťažké pochopiť, ako fungujú ukazovatele. (Reese, 2013, s. 1)

Jazyk C++ podporuje pri práci s dátami tri rôzne spôsoby správy pamäte, ktoré sú závislé na použitom spôsobe alokácie pamäte: statické úložisko, automatické úložisko a dynamické úložisko, ktorému sa tiež hovorí hromada (*heap*). Každý z týchto typov úložiska si predstavíme:

1. Statické úložisko - tomuto typu úložiska sú priradené staticky deklarované premenné. Globálne premenné taktiež používajú túto oblasť pamäte. Prideľujú sa pri spustení programu a zostávajú v existencii až do ukončenia programu. Zatiaľ čo ku globálnym premenným majú prístup všetky funkcie, rozsah statických premenných je obmedzený na funkciu, v ktorej sú definované. (Reese, 2013, s. 2)
2. Automatické úložisko – bežným premenným definovaným vo vnútri funkcie hovoríme automatické premenné. Vznikajú automaticky pri vyvolaní funkcie, ktorá ich obsahuje a ich platnosť vyprší s ukončením funkcie. Tieto premenné sa ukladajú v automatickom úložisku. (Prata, 2013, s. 189)

3. Dynamické úložisko – Pri dynamickom udeľovaní pamäte, žiadame operačný systém, aby vyhradil časť pamäte na použitie pre náš program. Ak dokáže splniť túto požiadavku, vráti adresu tejto pamäte vašej aplikácii. Od tohto momentu môže aplikácia používať túto pamäť podľa vlastného uváženia. Keď aplikácia dokončí prácu s pamäťou, môže ju vrátiť späť do operačného systému, aby bola odovzdaná inému programu. (*Dynamic memory allocation with new and delete,* 2022)

   Na pridelenú pamäť odkazuje ukazovateľ. Rozsah pridelenej pamäte je obmedzený na ukazovateľ alebo ukazovatele ktoré na pamäť odkazujú. Pridelená pamäť existuje, pokiaľ nie je znovu uvoľnená. (Reese, 2013, s. 2)

Pochopením týchto typov pamäte nám umožní lepšie pochopiť, ako fungujú ukazovatele. Väčšina ukazovateľov sa používa na manipuláciu s údajmi v pamäti. Pochopenie toho, ako je pamäť rozdelená a organizovaná, objasní, ako ukazovatele manipulujú s pamäťou. (Resse, 2013, s. 2)

Všetky počítače majú pamäť nazývanú RAM (random access memory), ktorá je dostupná pre programy a môžu byť do nej uložené časti údajov behu programu. Táto pamäť je organizovaná do sekvenčných jednotiek nazývaných adresy pamäte (alebo skrátene adresy). Keď definujeme premennú, časť tejto pamäte sa vyhradí pre túto premennú. (*Introduction to fundamental data types,* 2021)

V C++ nie je povolený priamy prístup do pamäte. Namiesto toho pristupujeme k pamäti nepriamo cez objekt. V tomto prípade sa objektom nemyslí inštancia triedy. Objekt je oblasť úložiska (zvyčajne pamäť), ktorá má hodnotu a ďalšie súvisiace vlastnosti. Objekty môžu byť pomenované alebo nepomenované (anonymné). Pomenovaný objekt sa nazýva premenná a názov objektu sa nazýva identifikátor. Pristupovanie k pamäti pomocou objektov nám uľahčuje prácu, pretože sa nemusíme sa starať o to, kde sú údaje v pamäti skutočne umiestnené, k údajom pristupujeme pomocou vytvoreného objektu. (*Introduction to objects and variables,* 2021)

Premenná ukazovateľ uchováva adresu hodnoty v pamäti inej premennej, objektu alebo funkcie. Ukazovateľ je zvyčajne deklarovaný ako špecifický typ v závislosti od toho, na čo ukazuje, ako napríklad ukazovateľ na celé číslo. Ukazovateľ môže obsahovať adresu ľubovoľného dátového typu, ako je napríklad celé číslo, znak, reťazec alebo dokonca štruktúra či trieda. Avšak obsah ukazovateľa ničím nenaznačuje, na aký typ údajov sa ukazovateľ odkazuje. Ukazovateľ obsahuje iba adresu. (Resse, 2013, s. 3)

1.2.1 Deklarácia a inicializácia ukazovateľov

Hodnota premennej ukazovateľa je adresa. To znamená, že hodnota ukazovateľa odkazuje na iný pamäťový priestor, v ktorom sú zvyčajne uložené dáta. (Malik, 2009, s. 132)

Počítač musí sledovať typ týchto dát, na ktoré sa ukazovateľ odvoláva. Napríklad adresa typu *char* vyzerá rovnako ako adresa typu *double* ale obe používajú rôzny počet bajtov a rôzne vnútorné formáty pre ukladanie hodnôt. (Prata, 2013, s. 166)

Preto, keď deklarujeme premennú ukazovateľa, špecifikujeme typ údajov hodnoty uloženej v pamäťovom priestore, na ktoré ukazuje premenná ukazovateľa. (Malik, 2009, s. 132)

Všeobecná syntax deklarácie ukazovateľa vyzerá teda nasledovne:

dátovýTyp *identifikátor;

C++ poskytuje na prácu s ukazovateľmi dva hlavné operátory – operátor &“a dereferenčný operátor (*). Operátor & je unárny operátor, ktorý vracia hodnotu jeho operanda. Dereferenčný operátor * je tiež unárny operátor, ktorý odkazuje na objekt, na ktorý ukazuje operand dereferenčného operátora (t. j. ukazovateľ). (Malik, 2009, s. 133 – 134)

Pre objasnenie toho, ako používanie ukazovateľov a s ním spojených operátorov funguje si uvedieme príklad. Máme vytvorenú premennú celočíselného dátové typu *pocet_bodov* a chceme vytvoriť ukazovateľ, ktorý bude uchovávať adresu tejto premennej. Jeho vytvorenie bude teda vyzerať takto: (vlastné spracovanie)

```
43      int pocet_bodov = 86;
44      int *p = &pocet_bodov;
```

Premenná *pocet_bodov* je dátového typu *int* a teda aj deklarácia ukazovateľa *p* musí určovať rovnaký dátový typ ako dátový typ hodnoty, ktorej adresu uchováva. Ako sme si už vysvetlili ukazovateľ môže obsahovať len adresu hodnoty a tú dostaneme pomocou unárneho operátora &, ktorý sme použili pri priraďovaní hodnoty ukazovateľu *p*. V tabuľke č. 2 je znázornenie uložených hodnôt premenných pocet_bodov a p a ich adries. Môžeme vidieť, že hodnota ukazovateľa *p* je totožná s adresou premennej *pocet_bodov*. (vlastné spracovanie)

Adresa v pamäti	Hodnota premennej	Názov premennej
0000003A1FEFF1E4	86	pocet_bodov
0000003A1FEFF208	0000003A1FEFF1E4	p

Tabuľka č. 2 – Hodnoty a adresy premenných (vlastné spracovanie)

Pomocou ukazovateľov je možné napísať rýchlejší a efektívnejší kód, pretože kompilátor môže jednoduchšie preložiť operáciu vykonanú pomocou ukazovateľov do strojového kódu rýchlejšie, než s inými dátovými štruktúrami. Mnoho dátových štruktúr je jednoduchšie implementovať pomocou ukazovateľov. (Reese, 2013, s. 3)

1.2.2 Alokovanie pamäte pomocou operátora new

Pretože už máme základnú predstavu o tom, ako ukazovatele fungujú, pozrieme sa, ako môžeme s ich pomocou implementovať dôležitú techniku objektovo orientovaného programovania, ktorou je alokovanie pamäte za behu programu. V predchádzajúcej časti sme inicializovali ukazovatele adresami premenných, ktoré mali takzvanú *pomenovanú* pamäť alokovanú v dobe prekladu kompilátorom. Ukazovateľ v takomto prípade poskytuje len druhé meno pre túto pamäť, ku ktorej by sme aj tak mohli pristupovať pod menom premennej. Pravá cena ukazovateľov sa prejaví pri alokovaní *nepomenovanej* pamäte za behu programu pre ukladanie hodnôt. K takto alokovanej pamäti je možné pristupovať len prostredníctvom ukazovateľov. V jazyku C++ je možné takto alokovať pamäť pomocou operátora „*new*“. (Prata, 2013, s. 170)

Alokovanie pamäte takýmto spôsobom si vyskúšame znovu na premennej *pocet_bodov*. Operátor jazyka C++ *new* zohráva pri tejto činnosti kľúčovú úlohu. Operátoru *new* musíme oznámiť, pre aký dátový typ chceme pamäť alokovať. *New* nájde blok správnej veľkosti a vráti jeho adresu. Túto adresu následne priradíme ukazovateľu. (Prata, 2013, s. 170

```
46      int* pocet_bodov = new int;
47      *pocet_bodov = 86;
```

Časť *new int* hovorí programu, že chce istú novú (new) pamäť vhodnú pre uloženie typu *int*. Operátor *new* používa predaný typ k určeniu počtu potrebných bajtov. Potom nájde vhodnú pamäť a vráti jej adresu. Vrátená adresa sa priradí premennej *pocet_bodov*, ktorá je deklarovaná ako typ ukazovateľa na *int*. Premenná *pocet_bodov* teda predstavuje adresu a **pocet_bodov* hodnotu uloženú na tejto adrese. (Prata, 2013, s. 170)

1.2.3 Uvoľnenie pamäte operátorom delete

Vo vybavení jazyka C++ pre správu pamäte je aj operátor *delete*, ktorý umožňuje vrátenie nepotrebnej pamäte systému. To je dôležitým krokom k čo najefektívnejšiemu využitiu pamäte. Vrátená alebo tiež uvoľnená pamäť môže byť znovu použitá inými časťami programu. Pre túto činnosť je používané kľúčové slovo *delete* nasledované ukazovateľom na blok pamäte, ktorý bol pôvodne alokovaný pomocou operátora *new*. (Prata, 2013, s. 172)

```
49      delete pocet_bodov;
```

1.3 Zložené dátové typy

V programovaní existuje veľa prípadov, kedy potrebujeme viac ako jednu premennú, aby sme mohli vyjadriť a uchovávať informácie o určitom objekte. Hoci sú základné dátové typy mimoriadne užitočné na priame použitie, nepokrývajú celý rozsah našich potrieb, keď začíname robiť zložitejšie veci. (*Introduction to compound data types*, 2022)

Povedzme, že chceme napísať program, kde potrebujeme ukladať informácie o zamestnancoch v spoločnosti. Mohlo by nás zaujímať sledovanie atribútov, ako je meno zamestnanca, priezvisko, dátum narodenia, adresa, pracovná pozícia, mzda, atď. (*Introduction to compound data types*, 2022)

Ak by sme na sledovanie všetkých týchto informácii použili nezávislé premenné, ich definovanie by vyzeralo nasledovne:

```
4     std::string meno;
5     std::string priezvisko;
6     int rokNarodenia;
7     int mesiacNarodenia;
8     int denNarodenia;
9     std::string adresa;
10    std::string pracovnaPozicia;
11    double mzda;
```

Tento prístup však prináša množstvo problémov. Po prvé, na prvý pohľad nie je jasné, či tieto premenné spolu skutočne súvisia alebo nie (je potrebné si prečítať komentáre alebo vidieť, ako sa používajú v kontexte). Po druhé, je potrebné spravovať 8 premenných. Ak by sme chceli zavolať funkciu, ktorá by s daným zamestnancom vykonala požadovanú operáciu, museli by sme jej odovzdať 8 argumentov v správnom poradí, čo by spôsobilo neporiadok v našich prototypoch funkcií a volaniach funkcií. A keďže funkcia môže vrátiť

iba jednu hodnotu, nemohla by v prípade potreby vrátiť všetky atribúty zamestnanca. V prípade, že by sme chceli uchovávať informácie o viac ako o jednom zamestnancovi, museli by sme definovať 7 ďalších premenných pre každého ďalšieho zamestnanca (z ktorých každá by vyžadovala jedinečné meno). To, čo teda potrebujeme, je nájsť nejaký spôsob, ako usporiadať všetky tieto súvisiace časti údajov dohromady, aby sa dali ľahšie spravovať. (*Introduction to compound data types,* 2022)

Jazyk C++ nám umožňuje vytvoriť zložené dátové typy. Tieto dátové typy sú vytvárané zo základných celočíselných typov a typov s pohyblivou desatinnou čiarkou. Najprepracovanejším zloženým typom je trieda, ktorá predstavuje základ objektovo orientovaného programovania a ktorej sa budeme venovať v neskoršej časti práce. Jazyk C++ podporuje aj niekoľko jednoduchších typov prevzatých z jazyka C. Napríklad pole môže obsahovať niekoľko hodnôt rovnakého typu. Ďalej C++ ponúka využitie ukazovateľov. Tieto premenné hovoria počítaču, kde sú údaje umiestnené v pamäti. V tejto kapitole preskúmame všetky vyššie uvedené zložené typy. (Prata, 2013, s. 131)

1.3.1 Dátový typ štruktúra

V predchádzajúcej časti sme zistili, že ak by sme chceli uložiť všetky požadované údaje o zamestnancovi, potrebujeme zložený dátový typ, do ktorého by sa dali uložiť informácie viacerých dátových typov. Odpoveďou na túto požiadavku je štruktúra jazyka C++. Štruktúra môže obsahovať položky viac než jedného dátového typu. Umožňuje zjednotiť dátovú reprezentáciu uložením všetkých informácií súvisiacich so zamestnancom do jednej štruktúrnej premennej. Štruktúra je užívateľsky definovaný dátový typ, ktorého dátové vlastnosti popisuje deklarácia štruktúry. (Prata, 2013, s. 152)

Definovanie štruktúry

Pri vytváraní štruktúry musíme najprv kompilátoru povedať, ako bude naša štruktúra vyzerať predtým, ako ju začneme používať. (vlastné spracovanie)

Príklad definície štruktúry v jazyku C++:

```
13  struct Zamestnanec
14  {
15      std::string meno;
16      std::string priezvisko;
17      int rokNarodenia;
18      int mesiacNarodenia;
19      int denNarodenia;
20      std::string adresa;
21      std::string pracovnaPozicia;
22      double mzda;
23  };
```

Kľúčové slovo *struct* kompilátoru hovorí, že definujeme štruktúru, ktorú sme nazvali zamestnanec. V zložených zátvorkách definujeme premenné, ktoré bude každá štruktúra *Zamestnanec* obsahovať. V tomto prípade budeme o zamestnancovi ukladať údaje ako meno, priezvisko, dátum narodenia, adresu, pracovnú pozíciu a jeho mzdu. (vlastné spracovanie)

Teraz, keď už máme vytvorenú definíciu štruktúry, môžeme vytvárať premenné typu *Zamestnanec* rovnako ako premenné jednoduchého dátového typu:

```
25  Zamestnanec jano;
26  Zamestnanec marian;
```

Prístup k jednotlivým premenným štruktúry zaisťuje operátor príslušnosti (.). Členské premenné štruktúry fungujú rovnako ako klasické premenné, môžeme im teda priraďovať hodnoty, porovnávať ich, vykonávať s nimi aritmetické operácie, atď. Jednou z najväčších výhod štruktúr je, že potrebujeme vytvoriť iba jeden nový názov premennej *struct*. (*Introduction to structs, members, and member selection*, 2022)

```
30      jano.mzda = 1500;
31      marian.meno = "Marian";
```

Vytvorením zloženej dátovej štruktúry *Zamestnanec* sme teda zabezpečili, že všetky sledované informácie o jednom zamestnancovi sú uložené v jednej zloženej premennej typu *struct*. S touto štruktúrou môžeme ďalej pracovať, môžeme ju napríklad predať funkcii ako argument alebo tiež vytvoriť funkciu, ktorá bude vracať štruktúru ako návratovú hodnotu. (vlastné spracovanie)

1.3.2 Triedy

V tradičnom programovaní sú programy v podstate zoznamy inštrukcií pre počítač, ktoré definujú údaje a potom s týmito údajmi pracujú prostredníctvom príkazov a funkcií. Údaje a funkcie, ktoré s týmito údajmi pracujú, sú samostatné entity, ktoré sa kombinujú, aby sa dosiahol požadovaný výsledok. Kvôli tomuto oddeleniu tradičné programovanie často neposkytuje veľmi intuitívne zobrazenie reality. Je na programátorovi, aby vhodným spôsobom spravoval a spájal vlastnosti (premenné) so správaním (funkciami). (*Welcome to object-oriented programming*, 2022)

Ako pri mnohých veciach, objektovo orientované programovanie možno najjednoduchšie pochopiť pomocou analógie. Všade okolo nás sú rôzne objekty: knihy, jedlo, budovy, nábytok... Objekty majú dve hlavné zložky: 1) zoznam relevantných vlastností (napr. hmotnosť, farba, veľkosť, pevnosť, tvar atď...) a 2) určité správanie, ktoré môžu vykazovať (napr. otvorenie, vykonanie určitej činnosti...). Tieto vlastnosti a správanie sú neoddeliteľné. (*Welcome to object-oriented programming*, 2022)

Objektovo orientované programovanie (OOP) nám poskytuje možnosť vytvárať objekty, ktoré spájajú tieto vlastnosti aj správanie do samostatného, opakovane použiteľného balíka. V jazyku C++ sú najdôležitejším nástrojom pre podporu objektovo orientovaného programovania triedy. Trieda definuje vlastnosti a správanie objektu. (Kirch-Prinz, Prinz, 2001, s. 243)

Definícia triedy špecifikuje názov triedy a mená spolu s dátovými typmi členov triedy. Definícia začína kľúčovým slovom *class*, za ktorým nasleduje názov triedy. Dátové členy a metódy sú potom deklarované v nasledujúcom bloku kódu. Dátové členy a členské funkcie môžu byť akéhokoľvek platného dátového typu, dokonca členom triedy môže byť aj iná predtým definovaná trieda. (Kirch-Prinz, Prinz, 2001, s. 247)

Na nasledujúcom príklade si ukážeme, ako vyzerá štruktúra z predchádzajúcej kapitoly prerobená do triedy. K údajom o zamestnancovi tiež pridáme aj jednu metódu, ktorá s nimi pracuje. (vlastné spracovanie)

```cpp
class Zamestnanec
{
public:
    std::string meno;
    std::string priezvisko;
    int rokNarodenia;
    int mesiacNarodenia;
    int denNarodenia;
    std::string adresa;
    std::string pracovnaPozicia;
    double mzda;

    void vypis_zamestnanca()
    {
        std::cout << "Meno: " << this->meno << std::endl;
        std::cout << "Priezvisko: " << this->priezvisko << std::endl;
        std::cout << "Datum narodenia: " << this->denNarodenia << ". "
                                         << this->mesiacNarodenia << ". "
                                          << this->rokNarodenia << std::endl;
        std::cout << "Adresa: " << this->adresa << std::endl;
        std::cout << "Pracovna pozicia: " << this->pracovnaPozicia << std::endl;
        std::cout << "Mzda: " << this->mzda << std::endl;
    }
};
```

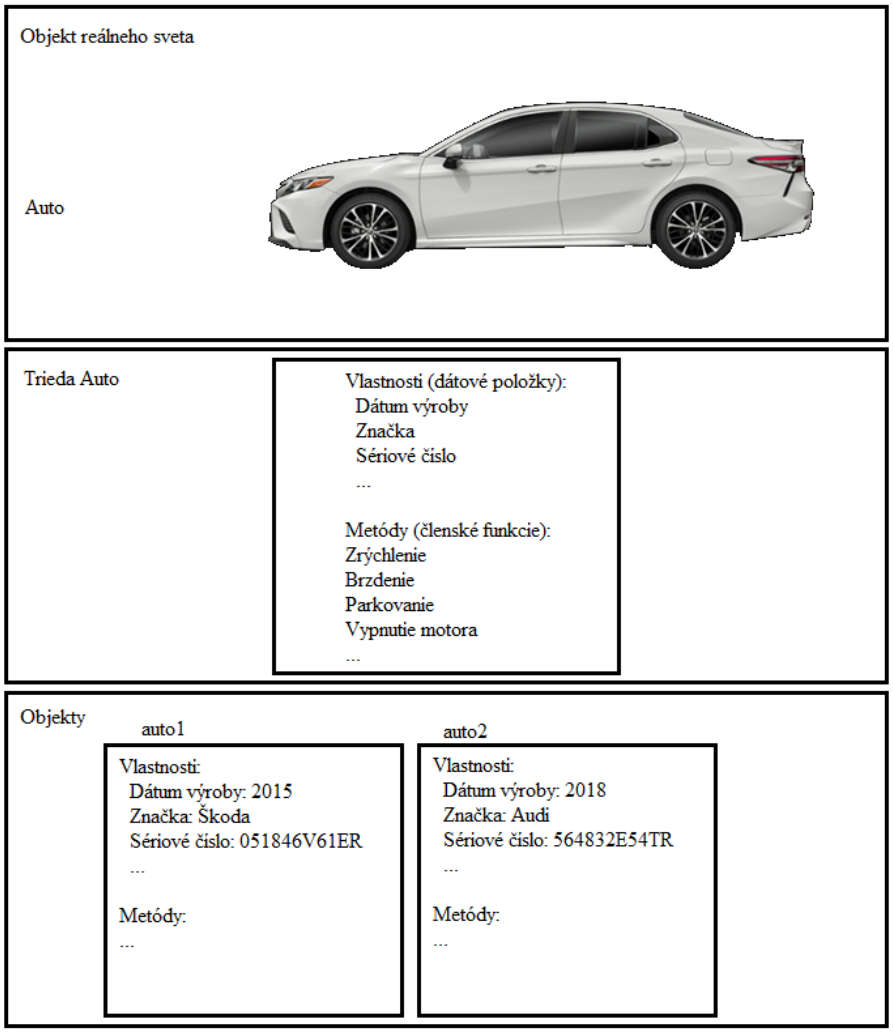
K členským premenným pribudla aj jedna metóda, ktorá po zavolaní vypíše všetky údaje o príslušnom zamestnancovi. Inštancia triedy sa vytvorí rovnako ako pri dátovom type štruktúra a k verejným členským premenným a metódam tiež pristupujeme pomocou operátora príslušnosti (.). (vlastné spracovanie)

```cpp
    Zamestnanec jano {"Jan", "Novak", 1982, 8, 27, "Kukucinova 5, Trnava", "Animator", 1720};
    jano.vypis_zamestnanca();
```

V uvedenom kóde sme vytvorili inštanciu triedy *Zamestnanec* a pomocou operátora príslušnosti sme zavolali jej členskú metódu *vypis_zamestnanca()*. Na obrázku č. 1 je výstup z programu po zavolaní tejto metódy. (vlastné spracovanie)

```
Meno: Jan
Priezvisko: Jan
Datum narodenia: 27. 8. 1982
Adresa: Kukucinova 5, Trnava
Pracovna pozicia: Animator
Mzda: 1720
```

Obrázok č. 1 – Výstup z programu (vlastné spracovanie)



Obrázok č. 2 – Koncept tried (vlastné spracovanie)

Keď definujeme triedu, špecifikujeme, či sú členovia triedy verejný alebo privátny. Privátny členovia triedy nie sú k dispozícii pre externý prístup. Ostatné funkcie programu môžu pristupovať k objektom triedy len pomocou verejných metód alebo dátových položiek triedy. Prístup k údajom o objekte je zriedkavo priamy, to znamená, že dátové položky (členské premenné) sú bežne deklarované ako privátne a potom sa čítajú alebo upravujú metódami s verejnými deklaráciami, aby sa zabezpečil správny prístup k údajom. Táto technika rozlišovania členov triedy na verejných alebo privátnych sa nazýva zapúzdrenie. Jedným z dôležitých aspektov tejto techniky je skutočnosť, že program nepotrebuje vedieť vnútornú štruktúru údajov v triede. (Kirch-Prinz, Prinz, 2001, s. 245)

V objektovo orientovanom programovaní je zapuzdrenie proces uchovávania podrobností o tom, ako je objekt implementovaný, skrytý pre používateľov objektu. Namiesto toho používatelia objektu pristupujú k objektu cez verejné rozhranie. Týmto spôsobom môžu používatelia používať objekt bez toho, aby museli chápať, ako je implementovaný. Hoci

požiadavka, aby používatelia triedy používali verejné rozhranie, sa môže zdať náročnejšia ako priame poskytovanie verejného prístupu k členským premenným, v skutočnosti to poskytuje veľké množstvo užitočných výhod, ktoré pomáhajú podporovať opätovnú použiteľnosť triedy. (*Access functions and encapsulation,* 2022)

Na rozlíšenie, či sú členovia triedy verejný alebo privátny používame kľúčové slová *public* (pre verejných členov) a *private* (pre privátnych členov). V nami vytvorenej triede „Zamestnanec“ môžeme vidieť, že pre zjednodušenie sú všetci členovia triedy verejný. To nám umožnilo inicializáciu všetkých premenných pri vytváraní inštancie triedy. Ak by sme však chceli dodržať princíp zapúzdrenia, dátové položky triedy by sme mali deklarovať ako privátne a vytvoriť verejné metódy, ktoré by boli prístupné zvyšným častiam programu, vďaka ktorým by sme mohli nastavovať hodnoty dátových položiek a pristupovať k ich hodnotám. (vlastné spracovanie)

1.4 Dátové štruktúry

1.4.1 Polia

V časti o dátovom type *struct* sme zistili, že tento dátový typ je dobrý pre prípad, ak chceme modelovať jeden objekt, ktorý má veľa rôznych vlastností. To však nie je vhodné v prípade, ak chceme sledovať viacero súvisiacich inštancií niečoho. Pre takýto prípad sú vhodné polia. (*Arrays (Part I)*, 2022)

Pole je forma dát, ktorá môže obsahovať niekoľko hodnôt rovnakého typu. Každá hodnota predstavuje samostatný prvok poľa a počítač všetky prvky ukladá za sebou do pamäti. (Prata, 2013, s. 131)

Pole ako zložený typ

Pole nazývame zloženým typom, pretože je založené na inom dátovom type. Nemôžeme jednoducho deklarovať, že niečo je poľom; vždy to musí byť pole nejakého určitého typu. Neexistuje žiadny všeobecný typ poľa. Namiesto toho máme mnoho konkrétnych typov polí, ako napríklad pole typu *char* alebo ako v našom príklade, pole typu *Zamestnanec*. (Prata, 2013, s. 132)

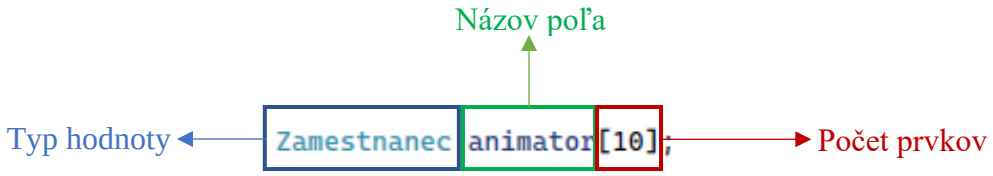
Predstavme si prípad, keď by sme potrebovali zaznamenať údaje o zamestnancoch, ktorý pracujú na animáciách. Ak by bolo vo firme 10 animátorov, znamenalo by to, že by sme museli alokovať 10 takmer identických premenných. Pre znázornenie použijeme zložený dátový typ *Zamestnanec*, ktorý sme vytvorili v predošlej časti: (vlastné spracovanie)

```
33      Zamestnanec animator1;
34      Zamestnanec animator2;
35      //...
36      Zamestnanec animator10;
```

Polia nám poskytujú oveľa jednoduchší spôsob, ako to urobiť. Pole je vytvorené pomocou deklaračného príkazu a jeho deklarácia by mala obsahovať tri informácie:

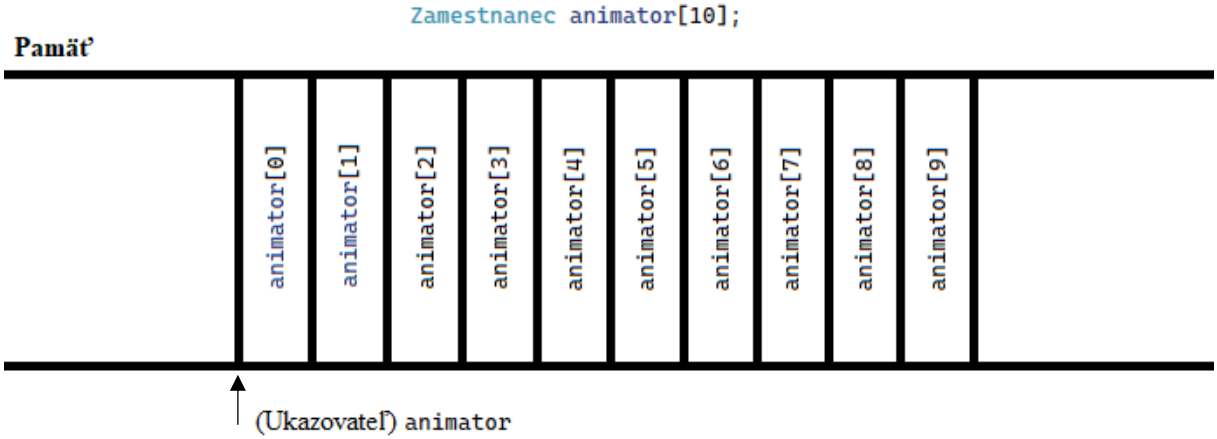
- Typ hodnoty, ktorá má byť uložená do každého prvku
- Názov poľa
- Počet prvkov v poli

V jazyku C++ to spravíme úpravou deklarácie premennej pridaním hranatých zátvoriek, ktorá obsahuje počet prvkov. (Prata, 2013, s. 131-132)



Obrázok č. 3 – Deklarácia poľa *animator* (vlastné spracovanie)

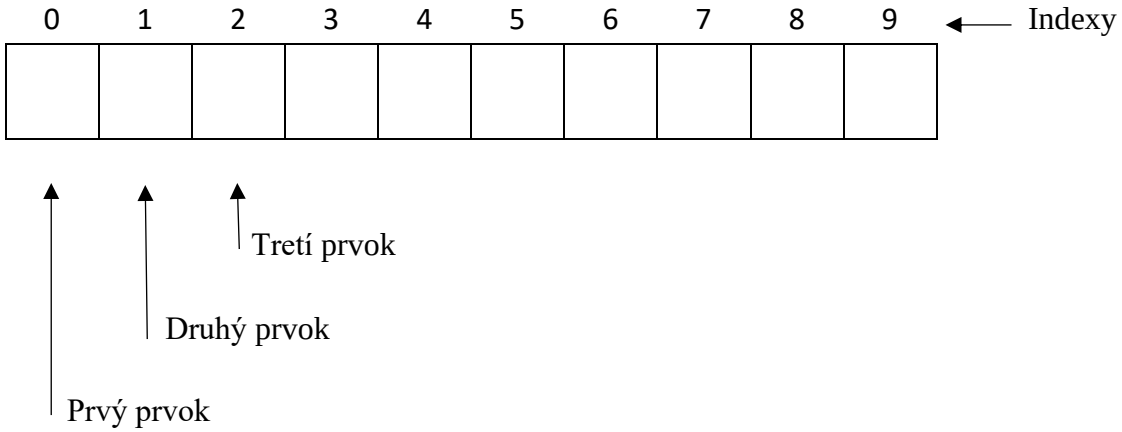
Statické pole je také pole, ktorého dĺžka je známa v čase kompilovania. Na obrázku č. 3 sme deklarovali statické pole s názvom „*animator*“ s jeho dĺžkou 10. Deklarovaním poľa týmto spôsobom program vyžaduje priestor vhodný pre uloženie desiatich dátových typov „*Zamestnanec*“ a vytvorí sa ukazovateľ *animator*, ktorý ukazuje na miesto v pamäti, kde je uložený prvý prvok (premenná *animator* je zároveň aj názov poľa aj ukazovateľ na prvý prvok v poli). Polia uložené v pamäti musia byť súvislé, to znamená, že všetky prvky poľa musia byť v pamäti uložené súvisle za sebou. (*Back To Basics: C++ Containers*, 2021)



Obrázok č. 4 – Uloženie poľa v pamäti (vlastné spracovanie)

Uloženie prvkov v pamäti za sebou prináša mnoho užitočných vlastností polí, napríklad to, že k prvkom poľa je možné pristupovať jednotlivo. To je realizované pomocou indexov, ktorými sú jednotlivé prvky očíslované. Číslovanie poľa v jazyku C++ začína hodnotou nula. To znamená, že ak je dĺžka poľa *N*, počet prvkov v poli je N-1. (Prata, 2013, s. 132)

```
40      animator[0].meno = "Martin";
41      animator[2].rokNarodenia = 1987;
```



Obrázok č. 5 – Vytvorenie poľa (vlastné spracovanie)

Kompilátor nekontroluje, či sú použité indexy platné. Neprekáža mu ani keď priraďujeme hodnotu neexistujúcemu prvku. Takéto priradenie ale môže spôsobiť problémy za behu programu, možné je aj poškodenie dát alebo kódu a môže dokonca dôjsť aj k ukončeniu

programu. Za používanie platných indexov je teda zodpovedný programátor. (Prata, 2013, s. 132)

Keďže majú statické polia pridelenú pamäť v čase kompilovania, stretávame sa s dvoma obmedzeniami. Statické polia nemôžu mať svoju dĺžku na základe vstupu používateľa alebo na základe inej hodnoty vypočítanej počas behu programu. Druhé obmedzenie je, že statické polia majú pevnú dĺžku, ktorú nemožno zmeniť. V mnohých prípadoch sú tieto obmedzenia problematické. C++ našťastie podporuje aj druhý typ poľa známy ako dynamické pole, v ktorom je možné veľkosť poľa nastaviť aj za behu programu. (*Arrays (Part I)*, 2022)

Dynamické polia

V kapitole „Ukazovatele a pamäť" sme si ukázali alokovanie a uvoľnenie pamäte pre jednoduché premenné pomocou operátorov *new* a *delete*. Avšak pri takom malom množstve dát ako je jedna premenná nie je využitie týchto operátorov až tak efektívne. (vlastné spracovanie)

Operátor „new" sa obvykle používa pri veľkom množstve dát, ktorá sa nachádza napríklad v poliach, reťazcoch alebo štruktúrach. V taktom prípade je použitie operátora „*new*" užitočné. (Prata, 2013, s. 173)

V neskoršej časti práce budeme vytvárať program s využitím poľa, ktorého veľkosť bude závisieť od počtu záznamov v súbore, z ktorého bude načítavať údaje. Ak by sme toto pole vytvorili pomocou statického poľa, nevedeli by sme zabezpečiť, že nami zadaný počet prvkov bude vyhovovať vstupom zo súboru. Buď by sa mohlo stať, že vytvorené pole bude príliš veľké a zaberie zbytočne veľa pamäte alebo by bolo pole príliš malé a nezmestili by sa doňho všetky záznamy zo vstupného súboru. (vlastné spracovanie)

Alokácia poľa behom prekladu sa nazýva statickou väzbou, to znamená, že do programu začlenené behom prekladu. Pomocou operátora „*new*" môžeme ale pole vytvoriť za behu programu, keď ho potrebujeme alebo vytvorenie preskočiť, pokiaľ ho nepotrebujeme alebo môžeme tiež za behu programu zadať jeho veľkosť. Vytváranie poľa za behu programu je označované ako dynamická väzba. Takto vytvorené pole sa nazýva dynamickým poľom. (Prata, 2013, s. 173)

Vytvorenie dynamického poľa

V C++ je vytvorenie dynamického poľa jednoduché, stačí operátoru „*new*" oznámiť typ prvkov poľa a ich požadovaný počet. (Prata, 2013, s. 173)

Rozdiel od statického poľa je však v tom, že požadovaný počet prvkov nemusí byť konštanta. (vlastné spracovanie)

```
58      std::cout << "Zadajte cele kladne cislo: ";
59      int velkost{};
60      std::cin >> velkost;
61
62      Zamestnanec *zamestnanci = new Zamestnanec[velkost];
```

Ako môžeme vidieť, do hranatých zátvoriek pre zadávanie veľkosti poľa sme vložili premennú, ktorej hodnotu zadá používateľ za behu programu. K prvkom dynamického poľa sa pristupuje úplne rovnako ako k prvkom statického poľa. (vlastné spracovanie)

```
65      zamestnanci[0].meno = "Andrej";
66      zamestnanci[0].priezvisko = "Novak";
```

Pri uvoľňovaní pamäte poľa vytvoreného pomocou operátora „new“ musíme použiť alternatívny tvar operátora „*delete*“, ktorý označuje uvoľňovanie poľa.

```
64      delete[] zamestnanci;
```

Prítomnosť hranatých zátvoriek hovorí programu, že má uvoľniť z pamäte celé pole, nie len prvok, na ktorý ukazuje ukazovateľ. (Prata, 2013, s. 173-174)

1.4.2 Lineárne zoznamy

Využitie poľa v programe je veľmi užitočné a rýchle. Umožňuje napríklad prelistovanie alebo vypísanie prvkov v lineárnom čase v závislosti od ich počtu čo je skvelá vlastnosť. Avšak vkladanie a odstraňovanie prvkov z poľa sú potenciálne drahé na výkon, v závislosti od toho, ktorý prvok vkladáme alebo odstraňujeme. V najhoršom prípade je potrebné vložiť prvok na pozíciu 0 (inými slovami na začiatok zoznamu), čo si vyžaduje posunutie celého poľa o jedno miesto dopredu, aby sa uvoľnilo miesto, a vymazanie prvého prvku vyžaduje posunutie všetkých prvkov v zozname o jedno miesto dozadu, takže najhorší prípad pre tieto operácie je O(N). V priemere je teda potrebné presunúť polovicu zoznamu pre obe operácie. Lineárny zoznam nám umožňuje vkladanie aj odstraňovanie prvkov v konštantnom čase. (Weiss, 2013, s. 78)

Existuje veľa situácií, keď v zozname nie je treba vkladať alebo mazať prvky v strede alebo na začiatku zoznamu. Prvky sa pridávajú a mažú len na konci zoznamu. V takom prípade je vhodnou implementáciou pole. Ak je však vkladanie a mazanie prvkov potrebné

v celom zozname a najmä na začiatku zoznamu, pole nie je dobrou voľbou. Nasledujúca časť sa zaoberá alternatívou: jednosmerným lineárnym zoznamom. (Weiss, 2013, s. 79)

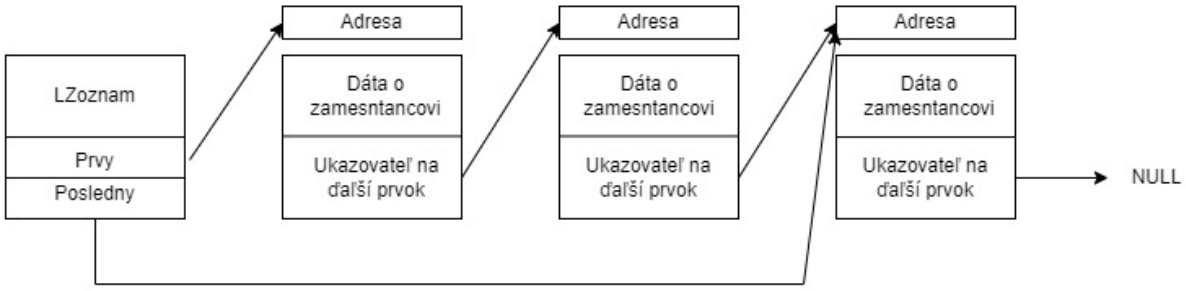
Jednosmerný lineárny zoznam

Aby sme sa vyhli lineárnym nákladom na vkladanie a mazanie, musíme zabezpečiť, aby prvky zoznamu neboli uložené súvisle, pretože inak bude potrebné presunúť celé časti (množiny prvkov) zoznamu.

Jednosmerný lineárny zoznam je kolekcia komponentov nazývaných uzly. Každý uzol (okrem posledného) obsahuje adresu nasledujúceho uzla. Každý uzol v jednosmernom lineárnom zozname má teda dve zložky: jednu na uloženie relevantných informácií (t. j. údaje) a jednu na uloženie adresy, nazývanej odkaz, nasledujúceho uzla v zozname. Adresa prvého uzla v zozname je uložená na samostatnom mieste, nazývanom hlavička. (Malik, 2009, s. 266)

Na implementáciu jednosmerného lineárneho zoznamu v programe jazyka C++ je napríklad vhodné použiť buď štruktúru alebo triedu. My si konkrétne fungovanie jednosmerného lineárneho ukážeme prostredníctvom nami vytvorenej triede *Zamestnanec*. Jej inštancie budú tvoriť lineárny zoznam. Túto triedu ale musíme upraviť a pridať ďalší dátový člen – ukazovateľ *dalsi*. Tento ukazovateľ bude obsahovať adresu nasledujúceho uzla, v tomto prípade adresu nasledujúceho objektu typu „*Zamestnanec*“ uloženom v jednosmernom lineárnom zozname. Každý takýto objekt v zozname bude zároveň uzlom. Tiež si vytvoríme aj novú, triedu s názvom *LZoznam*, ktorá bude mať dva dátové členy, ukazovatele na prvý a posledný prvok zoznamu. Objekt tejto triedy bude slúžiť ako hlavička jednosmerného lineárneho zoznamu, ktorá obsahuje adresu prvého (v našom prípade aj posledného) objektu v zozname. V triede sa tiež nachádzajú metódy na vkladanie, mazanie alebo na pristupovanie k prvkom. Vytvoríme si inštanciu tejto triedy s názvom *zoznam* a vložíme do neho tri objekty triedy *Zamestnanec*. Každý objekt *Zamestnanec* má svoju adresu. (vlastné spracovanie)

Na obrázku č. 6 je graficky znázornené ako sú pomocou ukazovateľov jednotlivé objekty jednosmerného lineárneho programu poprepájané.



Obrázok č. 6 – Lineárny zoznam *LZoznam* (vlastné spracovanie)

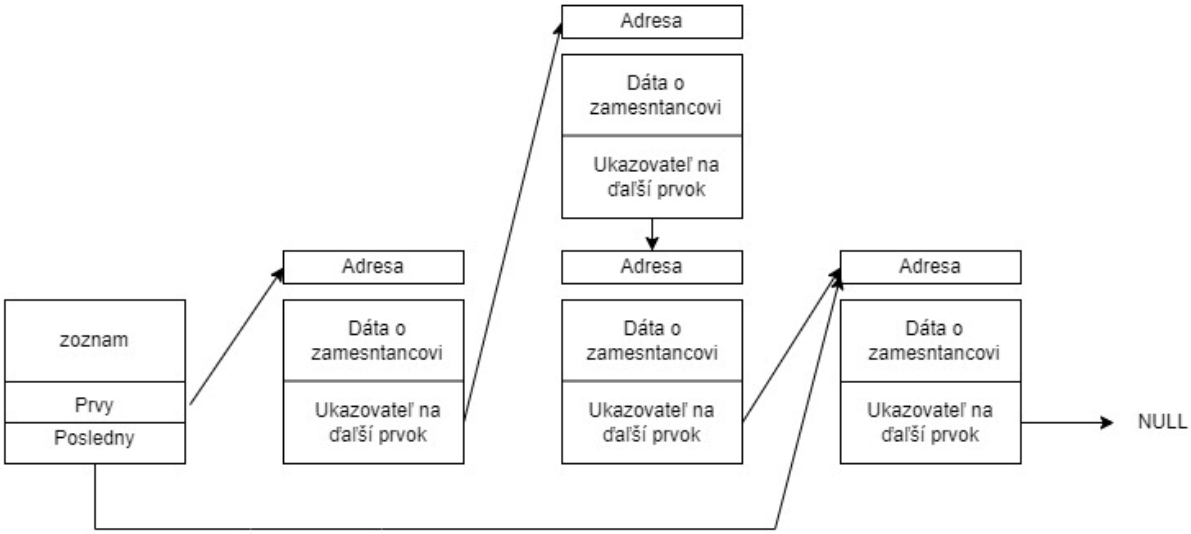
Na adresu prvého a posledného prvku ukazujú ukazovatele objektu *zoznam*. Každý ďalší prvok zoznamu obsahuje okrem dátach o zamestnancovi aj adresu nasledujúceho prvku. Posledný prvok neukazuje na žiadnu adresu ďalšieho prvku ale na takzvaný nulový ukazovateľ. To zabezpečí, že po použití hodnoty tohto ukazovateľa program nebude vykazovať neočakávané správanie. Inak by táto hodnota bola úplne náhodná adresa. (vlastné spracovanie)

Vkladanie, odstraňovanie prvkov zo zoznamu sa musia vykonávať pomocou nami vytvorených funkcií v triede *LZoznam*. Keďže tieto funkcie nebudeme pri vytváraní nášho programu používať, ich podrobným opisom sa nebudeme zaoberať. Namiesto toho si krátko popíšeme a ukážeme na diagramoch, ako sa tieto činnosti vykonávajú, aby sme získali základný prehľad o jednosmernom lineárnom zozname. Zdrojový kód na nasledujúcej strane pridáva do triedy Zamestnanec ukazovateľ *dalsi* ukazujúci na nasledujúci prvok zoznamu, vytvorenie triedy *LZoznam* a vytvorenie jej inštancie *zoznam*. (vlastné spracovanie)

```
26  class Zamestnanec
27  {
28  public:
29      std::string meno;
30      std::string priezvisko;
31      int rokNarodenia;
32      int mesiacNarodenia;
33      int denNarodenia;
34      std::string adresa;
35      std::string pracovnaPozicia;
36      double mzda;
37      Zamestnanec* dalsi;
38
39      void vypis_zamestnanca();
40  };
41
42  class LZoznam
43  {
44  private:
45      Zamestnanec* prvy = NULL;
46      Zamestnanec* posledny = NULL;
47  public:
48      void Vloz_na_koniec(Zamestnanec z);
49      void Vloz_na_zaciatok(Zamestnanec z);
50      void Zmaz_prvok(Zamestnanec z);
51  };
52
53  LZoznam zoznam;
```

Vkladanie prvkov do jednosmerného lineárneho zoznamu

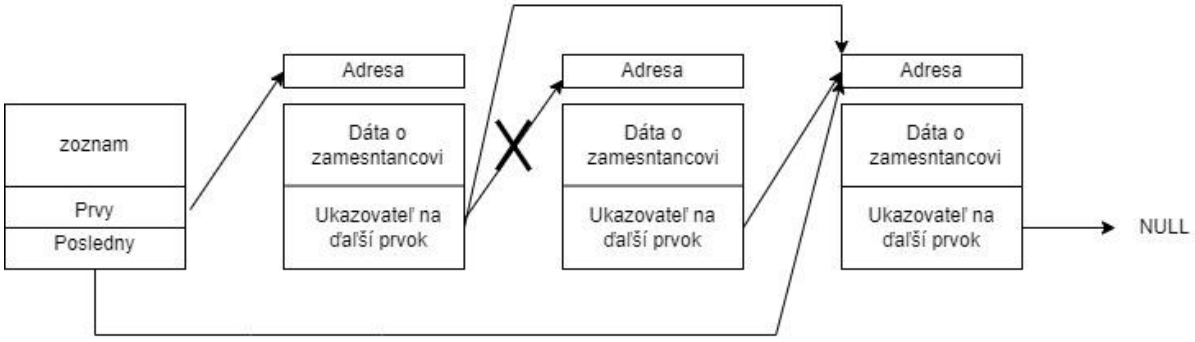
Vloženie prvku sa vykoná tak, že do jeho ukazovateľa sa skopíruje adresa prvku, ktorý bol doteraz na jeho mieste a predchádzajúcemu prvku sa zmení ukazovateľ na adresu ukazujúcu na nový prvok. (vlastné spracovanie)



Obrázok č. 7 – Vloženie prvku do zoznamu (vlastné spracovanie)

Odstraňovanie prvkov jednosmerného lineárneho zoznamu

Pri odstraňovaní prvku postupujeme tak, že prvku, ktorý predchádza tomu prvku, ktorý chceme odstrániť, zmeníme ukazovateľ na adresu tak, aby ukazoval na prvok nasledujúci po prvku, ktorý odstraňujeme. (vlastné spracovanie)



Obrázok č. 8 – Odstraňovanie prvku do zoznamu (vlastné spracovanie)

Odstránenie posledného prvku v zozname je zložitejšie, pretože musíme nájsť prechádzajúci prvok, zmeniť jeho ukazovateľ na adresu na nulový ukazovateľ a potom aktualizovať ukazovateľ na adresu, ktorý udržiava posledný uzol v hlavičke zoznamu. V

klasickom jednosmernom lineárnom zozname, kde každý uzol uchováva adresu na svoj nasledujúci uzol, ukazovateľ na posledný uzol neposkytuje žiadne informácie o predposlednom uzle. Zjavná myšlienka zachovania tretieho ukazovateľa na predposledný uzol nefunguje, pretože aj ten by sa musel počas odstraňovania aktualizovať. Namiesto toho by musel každý uzol udržiavať prepojenie na predchádzajúci uzol v zozname. Typ takéhoto zoznamu sa nazýva obojsmerný lineárny zoznam. (Weiss, 2013, s. 80)

1.4.3 Binárne vyhľadávacie stromy a B-stromy

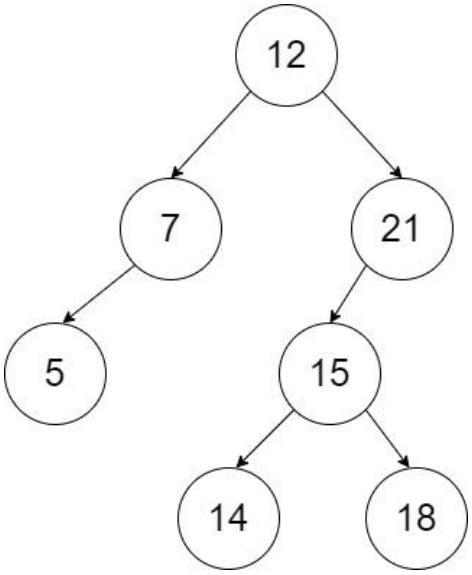
Pri organizovaní údajov je najvyššou prioritou programátora usporiadať ich tak, aby vkladanie, mazanie a vyhľadávanie (vyhľadávanie) bolo rýchle. Ukázali sme si, ako ukladať a spracovávať dáta v poli. Pretože pole je dátová štruktúra s priamym prístupom k prvkom, je možné pri ich správnom usporiadaní (napríklad zoradenie) použiť vyhľadávací algoritmus na efektívne nájdenie a získanie prvku zo zoznamu. Vieme však, že ukladanie údajov do poľa má svoje obmedzenia. Napríklad vkladanie položiek (najmä ak je pole zoradené) a odstraňovanie prvkov môže byť veľmi časovo náročné, najmä ak je veľkosť poľa veľmi veľká, pretože každá z týchto operácií vyžaduje presun údajov. Na urýchlenie vkladania a odstraňovania položiek môžeme použiť lineárne zoznamy. Vkladanie a vymazávanie prvkov v lineárnom zozname nevyžaduje žiadny presun údajov; jednoducho upravíme niektoré ukazovatele v zozname. Jednou z nevýhod lineárnych zoznamov je však to, že musia byť spracované postupne. To znamená, že ak chceme vložiť alebo odstrániť položku alebo jednoducho vyhľadať konkrétnu položku v zozname, musíme začať hľadať v prvom uzle zoznamu. Táto kapitola popisuje, ako dynamicky organizovať údaje, aby bolo vkladanie, odstraňovanie a vyhľadávanie položiek efektívnejšie. (Malik, 2009, s. 600)

Keďže sa jedná a celkom zložité štruktúry a pri vytváraní nášho programu a meraní časov nebudeme tieto dátové štruktúry používať, nebudeme si ukazovať ich praktickú implementáciu v jazyku C++ a budeme sa im venovať len okrajovo na pochopenie základov ich fungovania. Dôležité pre nás bude, aby sme sa s týmito dátovými štruktúrami oboznámili a získali prehľad o ich fungovaní, pretože aplikácia, s ktorou našu aplikáciu budeme porovnávať využíva na uchovávanie štruktúrovaných dát práve štruktúru B-Strom. (vlastné spracovanie)

Binárne vyhľadávacie stromy

Binárne vyhľadávacie stromy (BST) sú veľmi jednoduché na pochopenie. Pri ich vytváraní vždy začíname vytvorením koreňového uzla s hodnotou *x*, z ktorého vychádzajú dva ďalšie *podstromy*. Ľavý *podstrom* obsahuje uzly s hodnotami menšími ako *x* a pravý

hodnoty väčšie alebo rovné ako *x*. Z každého uzla môžu vychádzať najviac dva. Uzly, ktoré nemajú žiadne *podstromy* sa nazývajú listy. (Barnett a Tongo, , s. 19)

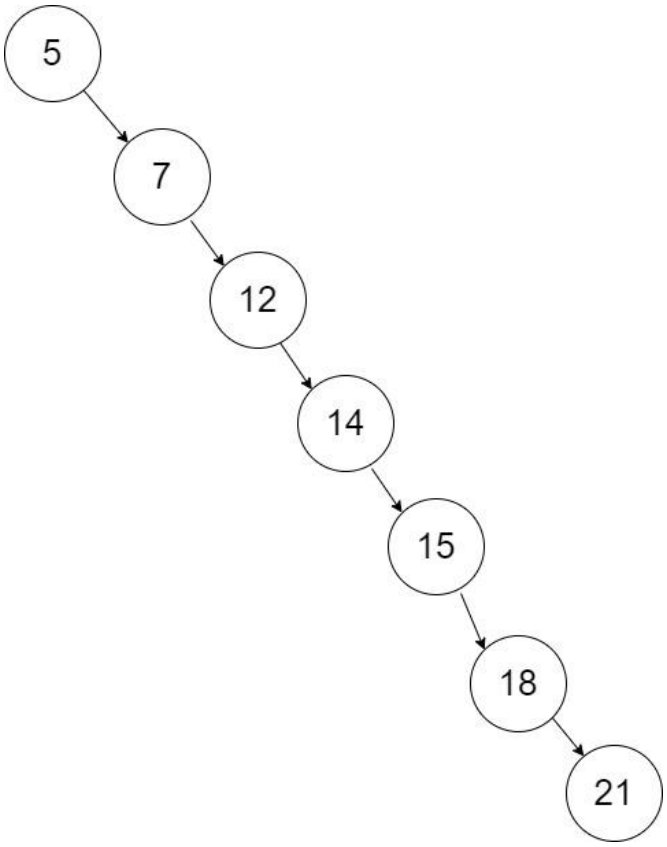


Obrázok č. 9 – Binárny vyhľadávací strom (vlastné spracovanie)

Binárny vyhľadávací strom na obrázku č. 9 má koreňový uzol s číslom 12. Naľavo z neho vychádza *podstrom* s menšou hodnotou a napravo s väčšou hodnotou. Uzly 5, 14 a 18 sú listami, pretože nemajú žiadne *podstromy*. (vlastné spracovanie)

Takéto usporiadanie nám pomáha veľmi rýchlo vyhľadávať údaje. Povedzme, že potrebujeme vyhľadať uzol 22. Začíname s porovnávaním s koreňovým uzlom. 22 je viac ako 12, takže sa posunieme po pravej vetve stromu k uzlu 21. 22 je viac ako 21, takže sa znovu presunieme po pravej vetve. Avšak z uzlu 21 žiadna pravá vetva nevedie, preto vieme, že uzol 22 v našom strome neexistuje. Tieto čísla, pomocou ktorých sa uzly pridávajú, mažú alebo vyhľadávajú sa nazývajú kľúče. (vlastné spracovanie)

Aby bolo vyhľadávanie v binárnom vyhľadávacom strome efektívne, musíme ho udržiavať vyvážený. Vyvážené vyhľadávacie stromy majú logaritmickú časovú zložitosť vyhľadávania. Na obrázku č. 10 je znázornený nevyvážený binárny vyhľadávací strom. Jeho hodnoty sú rovnaké ako na predošlom binárnom vyhľadávacom strome, no inak usporiadané. Ak by sme v tomto vyhľadávacom strome chceli vyhľadať uzol 22, museli by sme prejsť všetkými siedmimi uzlami a porovnať ho s nimi. V predošlom strome nám stačili len dve porovnávania. Časová zložitosť vyhľadávania v nevyváženom vyhľadávacom strome sa zmenila na lineárnu, ktorá je menej efektívna ako logaritmická. (vlastné spracovanie)



Obrázok č. 10 – Nevyvážený binárny vyhľadávací strom (vlastné spracovanie)

B-Stromy

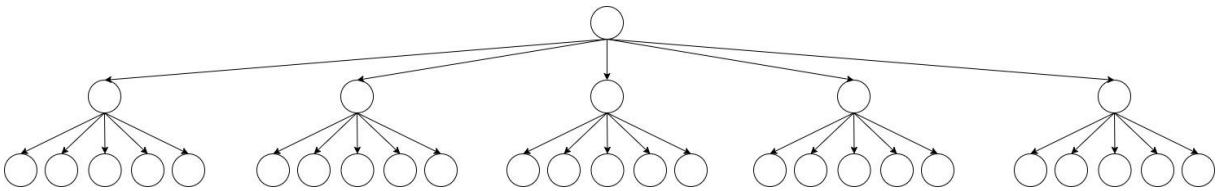
Doteraz sme predpokladali, že do hlavnej pamäte počítača môžeme uložiť celú dátovú štruktúru. Predpokladajme však, že máme viac údajov, než sa zmestí do hlavnej pamäte a v dôsledku toho musí byť štruktúra údajov uložená na disku. Prístupy k diskom sú neuveriteľne drahé na čas aj na množstvo výpočtov procesora. (Weiss, 2013, s. 168)

V takýchto prípadoch je dôležité znížiť počet prístupov na disk. Ak by sa nám podarilo pri vkladaní, mazaní alebo vyhľadávaní znížiť počet prístupov na disk napríklad na polovicu, čas chodu sa skráti na polovicu. (Weiss, 2013, s. 168)

Predpokladajme, že máme na disku 10 000 000 záznamov, v ktorých musíme vyhľadať určitý záznam. Nevyvážený binárny vyhľadávací strom je v takomto prípade veľmi neefektívny. V najhoršom prípade má lineárnu hĺbku (ako na obrázku č. 10), a preto môže vyžadovať 10 000 000 prístupov na disk. Vo vyváženom binárnom vyhľadávacom strome by v priemere takáto operácia potrebovala v priemere len 25 prístupov na disk. (Weiss, 2013, s. 168)

Pre efektívne vkladanie, odstraňovanie a vyhľadávanie prvkov v binárnych vyhľadávacích stromoch musia byť prvky vyvážené. Jedným z riešení ako udržovať binárne vyhľadávacie stromy vyvážené je použiť samovyvažovacie vyhľadávacie stromy. Po vložení prvku do takéhoto stromu sa strom automaticky prispôsobí tak, aby zostal vyvážený. Jednou z implementácií samovyvažovacieho stromu je B-Strom. (vlastné spracovanie)

Ak chceme znížiť počet prístupov na disk na veľmi malú konštantu, napríklad tri alebo štyri prístupy, riešenie je intuitívne jednoduché: potrebujeme viac ako dve rozvetvenia. Ak máme viac rozvetvení, máme menšiu výšku. Zatiaľ čo dokonalý binárny strom s 31 uzlami má päť úrovní, strom s piatimi rozvetveniami a s 31 uzlami má iba tri úrovne, ako je znázornené na obrázku č. 11. (Weiss, 2013, s. 169)



Obrázok č. 11 – Strom s piatimi rozvetveniami (vlastné spracovanie)

S rastúcim vetvením sa hĺbka znižuje. Zatiaľ čo úplný binárny strom má výšku približne $\log_2 N$, úplný strom s M rozvetveniami má výšku približne $\log_M N$. Vyhľadávací strom s M rozvetveniami môžeme vytvoriť takmer rovnakým spôsobom ako binárny vyhľadávací strom. V binárnom vyhľadávacom strome potrebujeme jeden kľúč, aby sme sa rozhodli, ktorú z dvoch vetiev si vyberieme. V strome s M rozvetveniami potrebujeme $M - 1$ kľúčov, aby sme sa rozhodli, ktorú vetvu zvoliť. Aby bola táto schéma efektívna aj v najhoršom prípade, musíme zabezpečiť, aby bol vyhľadávací strom M-ary nejakým spôsobom vyvážený. V opačnom prípade by sa podobne ako binárny vyhľadávací strom mohol zvrhnúť na lineárny zoznam (ako na obrázku č. 10). V skutočnosti chceme ešte prísnejšie podmienky vyváženia. To znamená, že nechceme, aby sa vyhľadávací strom s M rozvetveniami zvrhol dokonca na binárny vyhľadávací strom. Jedným zo spôsobov, ako to implementovať, je použiť B-strom. B-strom rádu M je strom s M rozvetveniami s nasledujúcimi vlastnosťami:

1. Každý uzol má najviac M potomkov

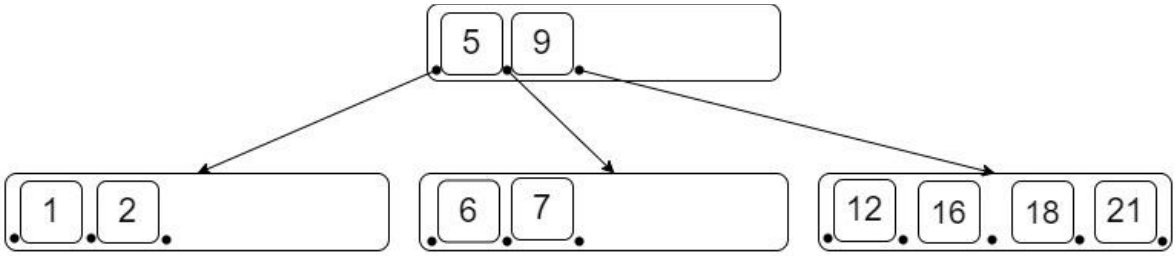
2. Nelistové uzly s M potomkami obsahujú M-1 kľúčov

3. Koreň má aspoň dva uzly ak nie je listovým uzlom

4. Všetky nelistové uzly (okrem koreňa) majú aspoň M/2 potomkov

5. Všetky listy sú na rovnakej úrovni

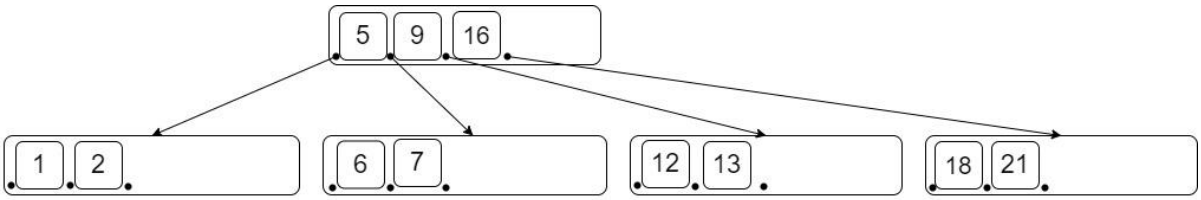
Pridávanie kľúčov do B-Stromu si ukážeme na príklade. (Weiss, 2013, s. 169)



Obrázok č. 12 – B-Strom (vlastné spracovanie)

Na obrázku č. 12 vidíme B-Strom rádu 5, pretože jeden uzol môže mať najviac 5 potomkov a 4 (M-1) kľúče. Obdĺžniky predstavujú uzly a čísla predstavujú kľúče. Do tohto stromu chceme vložiť kľúč s hodnotou 13. Na začiatku postupujeme ako pri vkladaní prvku do binárneho vyhľadávacieho stromu. Kľúč začneme porovnávať s hodnotami v koreňovom uzle. 13 je viac ako 5 a tiež viac ako 9, preto sa posúvame po vetve napravo od kľúča 9. Po pravej vetve sme sa posunuli do uzla druhého stupňa, kde vkladaný kľúč opäť začneme porovnávať. 13 je viac ako 12 a menej ako 16, vyzerá teda, že sme našli miesto pre náš kľúč. Avšak jeden uzol nemôže obsahovať viac ako 4 kľúče, preto sa musí rozdeliť na menšie časti. Tretiu úroveň vytvoriť nemôžeme, pretože by sme nezachovali pravidlo, že všetky listy (uzly bez potomkov) musia byť na rovnakej úrovni. Preto z hodnôt 12, 13, 16, 18 a 21 vypočítame medián, ktorý presunieme do koreňa. Medián z uvedených hodnôt je 16. Tento kľúč presunieme do koreňa. Teraz, keď koreň obsahuje 3 kľúče, musí mať aj štyroch potomkov M, pretože platí, že kľúčov v nelistových uzloch je vždy M-1. Uzol sa teda rozdelí na dva, jeden s hodnotami menšími ako 16 vychádzajúci naľavo od kľúča 16 a druhý s hodnotami väčšími ako 16 vychádzajúci napravo od kľúča 16. (vlastné spracovanie)

Vyhľadávanie v B-Strome prebieha rovnako ako vo vyhľadávacom binárnom strome. Program porovná hodnoty, ak je hľadaná hodnota väčšia ako hodnota kľúčov, presunie sa na uzol po vetve doprava, ak menšia, tak doľava. (vlastné spracovanie)



Obrázok č. 13 – B-Strom po vložení prvku (vlastné spracovanie)

1.5 Kontajnery knižnice STL

Okrem pristupovania k prvkom pomocou indexov alebo pomocou aritmetických operácií s ukazovateľmi neexistuje žiadny iný spôsob interakcie s poľami. Ak by sme chceli prvky v poli usporiadať, vymeniť medzi sebou alebo vykonať s nimi iné operácie, museli by sme sami vytvoriť požadované funkcie. Je to v podstate len hromada dát, ktoré vyžadujú manuálnu interakciu zo strany programátora. (*Back To Basics: C++ Containers*, 2021)

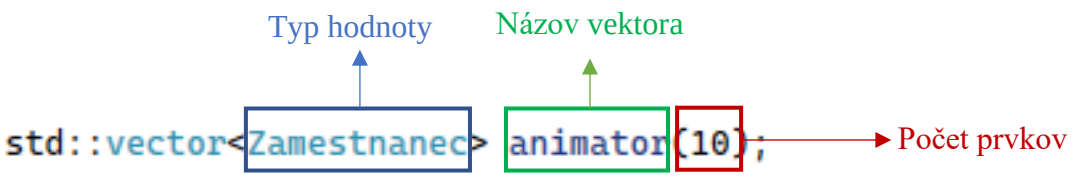
Jazyk C++ je vybavený štandardnou knižnicou šablón (STL – Standard template library), ktorá obsahuje kolekciu šablón reprezentujúcich kontajnery, iterátory, funkčné objekty a algoritmy. Pomocou týchto nástrojov môžeme vykonávať rôzne úlohy, napríklad zoradenie poľa alebo nájdenie určitej hodnoty v poli. Vďaka STL knižnici teda môžeme vytvárať rôzne kontajnery vrátané polí, lineárnych zoznamov a vykonávať rôzne operácie ako prehľadávanie alebo triedenie. (Prata, 2013, s. 848)

V tejto kapitole sa pozrieme na vybrané kontajnery STL knižnice, ktoré budeme v práci používať aj neskôr.

1.5.1 Šablónová trieda vector

Kontajner *vector* ukladá a spravuje svoje objekty v dynamickom poli. Je definovaný v hlavičkovom súbore *vector*, ktorý musíme zahrnúť do svojho kódu, ak chceme tento kontajner používať. (Malik, 2009, s. 211)

Deklarácia objektu šablóny *vector* je zobrazená na obrázku č. 14.

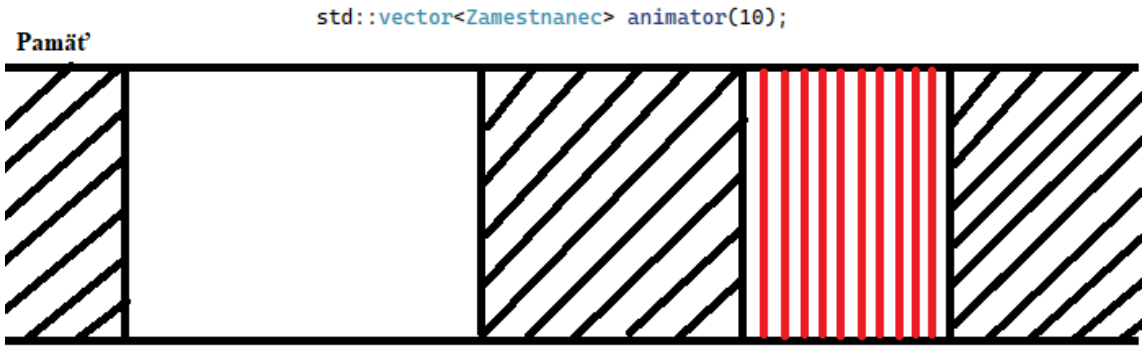


Obrázok č. 14 – Deklarácia *vectora* „*animator*“ (vlastné spracovanie)

Keďže *vector* poskytuje funkcie dynamického poľa, stará sa aj o správu vlastnej pamäte. To znamená, že môžeme pomocou neho vytvárať polia, ktorých veľkosť je nastavená za behu, no výhodou je, že na rozdiel od klasických dynamických polí nemusíme explicitne prideľovať a udeľovať pamäť pomocou *new* a *delete*. (vlastné spracovanie)

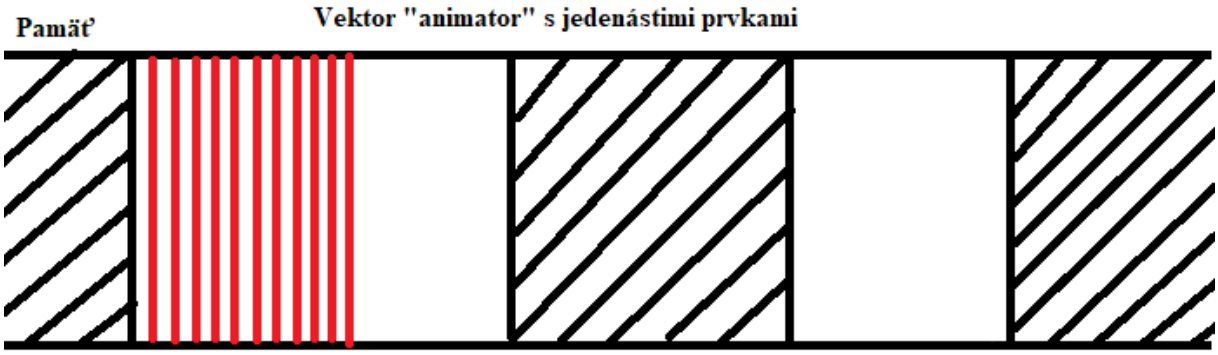
Taktiež nemusíme počas deklarácie stanoviť počet prvkov, čiže časť *Počet prvkov* znázornená na obrázku č. 14 je nepovinná. (vlastné spracovanie)

Počet prvkov nemusíme určovať ani pri jeho písaní programu ani pri jeho spustení. Pokiaľ program bude mať dostatok pamäte, bude sa veľkosť *vector*u zväčšovať podľa potreby. (Prata, 2013, s. 852)



Obrázok č. 15 – Znázornenie vektora v pamäti (vlastné spracovanie)

Vyšrafovaná plocha na obrázku č. 15 predstavuje obsadenú pamäť, ktorá nie je pre program dostupná a biela plocha predstavuje voľnú pamäť. Vytvorením *vector*a *animator* s desiatimi prvkami nájde program vhodne veľký blok voľnej pamäte a vloží doň *vector* s prvkami súvisle vedľa seba. Prvky sú na obrázku znázornené zvislými čiarami. Ak by sme chceli do *vector*a pridať ďalší prvok, už by sa nevošiel do bloku pamäte, v ktorom je *vector* uložený a porušila by sa súvislosť prvkov, ktorá je veľmi dôležitá, pretože nám umožňuje rýchly prístup k prvkov pomocou indexov. *Vector* vie našťastie rozoznať, že už sa jeho prvky viac nemôžu rozširovať a preto nájde pamäť, ktorá je väčšia ako tá súčasná, do ktorej skopíruje svoj celý obsah. (*Back To Basics: C++ Containers*, 2021)



Obrázok č. 16 – Premiestnenie *vectora* v pamäti (vlastné spracovanie)

Vector sa teda premiestnil do väčšieho bloku voľnej pamäte, v ktorom je priestor na pridávanie nových prvkov. S premiestňovaním *vector*ov v pamäti však prichádza jeden problém, na ktorý si programátor musí dávať pozor. Predstavme si, že máme vytvorený ukazovateľ, ktorý držal adresu v pamäti jedného z pôvodných desiatich prvkov. Teraz po premiestnení *vector*a ukazovateľ ukazuje na adresu, ktorej hodnota je neznáma a použitie takéhoto ukazovateľa má za následok nedefinovaný a neželaný výsledok. (*Back To Basics: C++ Containers*, 2021)

Po vytvorení objektu triedy „*vector*“ môžeme k jednotlivým prvkom pristupovať tak ako pri klasických poliach pomocou operátora „[]“ alebo pomocou takzvaného iterátora, ktorý si predstavíme v tejto časti. (vlastné spracovanie)

Výhody vectora

Všetky kontajnery knižnice STL obsahujú určité základné metódy, medzi ktorá patrí metóda *size()* vracajúca počet prvkov v kontajnery, metóda *swap()* meniaca obsah dvoch kontajnerov, metóda *begin()* vracajúca iterátor odkazujúci na prvý prvok v kontajnery a metóda *end()* vracajúca iterátor, ktorý odkazuje na miesto bezprostredne za posledným prvkom kontajneru. (Prata, 2013, s. 851)

Iterátor je zovšeobecnený ukazovateľ, ktorý slúži na identifikáciu prvku v kontajneri, na ktorý ukazuje. Môže byť ukazovateľom alebo objektom, pre ktorý sú definované operácie používané pre ukazovatele, napríklad dereferencovanie (pomocou operátora *) alebo inkrementovanie hodnoty (operátor ++). Iterátor deklarujeme nasledujúcim spôsobom: (Prata, 2013, s. 851)

```
43    std::vector<Zamestnanec>::iterator i; //iterátor i
44    i = animator.begin(); //i ukazuje na prvý prvok
45    (*i).meno = "Martin"; //dereferencuje i a hodnotu priradí prvku, na ktorý ukazuje
46    ++i; //i ukazuje na nasledujúci prvok
```

Ako môžeme vidieť, iterátor sa chová ako ukazovateľ. Metódy begin() a end() môžeme využiť napríklad v cykle, ak by sme chceli zobraziť všetky prvky v kontajneri.

```
48      for (i = animator.begin(); i != animator.end(); i++)
49          std::cout << (*i).meno << std::endl;
```

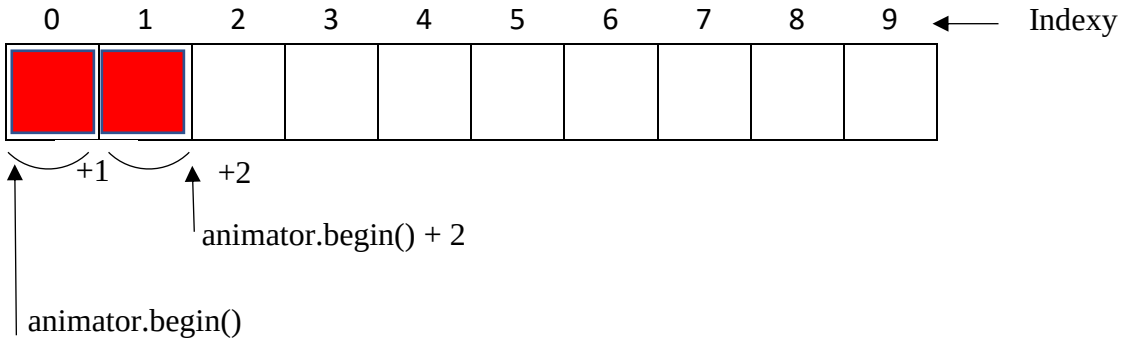
V cykle *for* sme iterátor *i* nastavili, aby ukazoval na prvý prvok. V každom cykle sa vypíše meno objektu, na ktorý práve ukazuje iterátor *i*, ktorý sa bude inkrementovať, až kým jeho hodnota nebude totožná s miestom bezprostredne za posledným prvkom *vector*a *animator*. (vlastné spracovanie)

Šablónová trieda *vector* obsahuje tiež aj niekoľko metód, ktoré majú iba niektoré kontajnery knižnice STL. Jednou z užitočných metód je metóda *push_back()*, ktorá pridáva prvok na koniec *vector*u. Pritom dohliada na správu pamäte, takže veľkosť *vector*u sa zväčšuje podľa pridávaných prvkov. (Prata, 2013, s. 852)

Vector využíva tiež metódu *erase()*, ktorá ruší stanovený rozsah. Má dva parametre – iterátory definujúce rozsah, ktorý má byť zrušený. Prvý iterátor ukazuje na začiatok bloku, zatiaľ čo druhý ukazuje o jednu pozíciu ďalej, ako je jeho koniec. (Prata, 2013, s. 852)

```
48      animator.erase(animator.begin(), animator.begin() + 2);
```

Príkazový riadok vyššie vymaže z *vector*a prvé dva prvky. Pre lepšie pochopenie si blok ohraničený dvoma iterátormi ukážeme na obrázku č. 17. (vlastné spracovanie)



Obrázok č. 17 – Znázornenie metódy *erase()* (vlastné spracovanie)

Farebne odlíšené štvorce predstavujúce prvky sú súčasťou bloku, ktorý bude metódou *erase()* zrušený. (vlastné spracovanie)

1.5.2 Šablónová trieda forward list

V kapitole o lineárnych zoznamoch sme zistili, že vkladanie a odstraňovanie prvkov sa vykonáva v konštantnom čase, keďže sa prvky zoznamu nemusia presúvať. Lineárne zoznamy však nemajú žiadne vstavané funkcie pre prácu s prvkami. Pre manipuláciu s jednotlivými prvkami však musíme sami vytvárať jednotliví funkcie na pridávanie a odstraňovanie prvkov alebo prechádzanie prvkami v zozname. Šablónová trieda forward list nám ale okrem iných zabezpečuje túto funkcionalitu pre jednosmerný lineárny zoznam. Preto ju budeme pri vytváraní nášho programu používať ako vhodný, rýchly a efektívny spôsob vytvorenia jednosmerného lineárneho zoznamu (vlastné spracovane).

Šablónová trieda *forward list* (deklarovaná v hlavičkovom súbore *forward_list*) predstavuje jednosmerný lineárny zoznam. Každý prvok okrem prvého je spojený prvkom predchádzajúcim, čo znamená že môžeme zoznamom prechádzať jedným smerom – dopredu. Hlavný rozdiel medzi *vectorom* a *forward listom* je rovnaký ako rozdiel medzi poľom a jednosmerným lineárnym zoznamom a to ten, že forward list vkladá a odoberá prvky (okrem posledného prvku) v konštantnom čase. Trieda *vector* kladie dôraz na rýchly prístup k jednotlivým prvkom (možný vďaka priamemu prístupu), zatiaľ čo *forward list* kladie dôraz na rýchle vkladanie a odoberanie prvkov. (Prata, 2013, s. 879)

Trieda forward list nie je na rozdiel od *vectora* reverzibilným zoznamom, dá sa prechádzať len jedným smerom. Nepodporuje ani zápis používaný pre pole ani priamy prístup. Iterátor triedy *forward list* na rozdiel od iterátoru triedy *vector* ukazuje aj po vložení prvku alebo odstránení prvku do kontajnera na rovnaký prvok. Toto konštatovanie si musíme objasniť. Predpokladajme iterátor ukazujúci na piaty prvok kontajneru triedy *vector*. Potom na začiatok kontajneru vložíme prvok. Všetky ostatné prvky sa musia presunúť, aby sa vytvorilo miesto, takže po vložení obsahuje piaty prvok hodnotu, ktorá bola v štvrtom prvku. Iterátor teda ukazuje na rovnaké miesto, ale na iné dáta. Pri vložení nového prvku do objektu triedy forward list však k presunu existujúcich nedôjde. Zmení sa len informácia o väzbách na jednotlivé prvky. Iterátor ukazujúci na určitý prvok naňho bude ukazovať stále, ale môže byť spojený s inými prvkami ako predtým. (Prata, 2013, s. 879)

1.6 Porovnanie nami vytváranej aplikácie s aplikáciou *Jorani*

V tejto kapitole sa zameriame na porovnanie aplikácie, ktorú budeme vytvárať s inou aplikáciou, ktorá má podobnú funkcionalitu. Porovnáme, do akých dátových štruktúr budú aplikácie dáta ukladať a akým spôsobom v nich budú vyhľadávať. (vlastné spracovanie)

Na porovnávanie som vybral aplikáciu *Jorani*, keďže sa jedná o *open source* aplikáciu voľne dostupnú na *jorani.org* a mohol som nahliadnuť na spôsob, akým aplikácia funguje. Jedná sa o aplikáciu na riadenie ľudských zdrojov, do ktorej sa napríklad dajú ukladať údaje o zamestnancoch, rozdeliť organizáciu do rôznych divízií, oddelení alebo tímov alebo ponúka funkcionalitu na efektívnejšie plánovanie práce. Keďže aj naša aplikácia bude ukladať údaje o zamestnancoch, porovnáme si základné rozdiely medzi aplikáciami. (vlastné spracovanie)

Prvý rozdiel je v štruktúre samotných aplikácii. Naša aplikácia bude oknová Windowsová aplikácia s jednostupňovou architektúrou. Aplikačná aj dátová vrstva našej aplikácie sú spojené a našu aplikáciu je možné nainštalovať a spustiť na jednom počítači. *Jorani* je webová aplikácia, ktorá na ukladanie dát využíva SQL databázu na strane servera. (vlastné spracovanie)

Druhým rozdielom sú použité dátové štruktúry na ukladanie štruktúrovaných dát. V našej aplikácii budeme štruktúrované dáta ukladať do poľa a jednosmerného lineárneho zoznamu. Aplikácia Jorani využíva na ukladanie dát SQL databázu. SQL server ukladá dáta do 8 kilobajtových súborov vytvárajúcich dátovú štruktúru B-Strom. Ako sme si povedali v predchádzajúcich kapitolách, polia a jednosmerné lineárne zoznamy majú lineárnu časovú zložitosť vyhľadávania. B-Strom má logaritmickú časovú zložitosť vyhľadávania. To znamená, že aplikácia *Jorani* vyhľadáva v štruktúrovaných dátach niekoľkonásobne rýchlejšie, ako naša aplikácia. (vlastné spracovanie)

2 Cieľ práce, metodika práce a metódy skúmania

Po predstavení najpoužívanejších vybraných dátových štruktúr v C++ bude hlavným cieľom práce vytvoriť program, v ktorom vytvoríme dynamické pole a jednosmerný lineárny zoznam. Tieto dátové štruktúry program naplní inštanciami tried (objektami) so štruktúrovanými dátami o fiktívnych zamestnancoch z používateľom vybraného vstupného textového súboru. Fiktívne dáta o zamestnancoch vygenerujeme v online dostupnom generátore štruktúrovaných dát. Pomocou algoritmu lineárneho vyhľadávania budeme v týchto dátových štruktúrach vyhľadávať záznamy podľa zvoleného atribútu a program exekučné časy týchto vyhľadávaní zapíše do výstupného textového súboru. Vyhľadávať nebudeme len prvý zhodný prvok ale vyhľadáme všetky prvky, ktorý majú vybraný atribút zhodný s hľadaným výrazom. Vyhľadávanie budeme spúšťať niekoľkokrát pri rôznych objemoch dát. Následne budeme spriemerované exekučné časy vyhľadávaní porovnávať a na základe výsledkov zistíme, v ktorej z dvoch vybraných dátových štruktúr bolo vyhľadávanie štruktúrovaných dát lineárnym vyhľadávacím algoritmom efektívnejšie, teda ktorá z týchto dátových štruktúr je na vyhľadávanie vhodnejšia. (vlastné spracovanie)

Program vytvoríme vo vývojovom prostredí Visual Studio 2022 prostredníctvom projektovej šablóny *MFCApp*. Na vytvorenie dynamického poľa použijeme šablónovú triedu *vector* a na vytvorenie jednosmerného lineárneho zoznamu šablónovú triedu *forward_list*. Štruktúrované dáta o zamestnancoch vygenerujeme pomocou online generátora dostupnom na webe www.mockaroo.com. Tieto dáta uložíme do textového súboru, pričom jednotlivé záznamy oddelíme čiarkami. Vďaka tomu bude program vedieť tieto dáta načítať a uložiť jednotlivé záznamy o fiktívnych zamestnancoch ako inštancie triedy a následne tieto inštancie uložiť do vybraných šablónových tried. Po uložení údajov v dátových štruktúrach vyberieme atribút, podľa ktorého budeme jednotlivé záznamy vyhľadávať a zadáme zvolené kritérium vyhľadávania. (vlastné spracovanie)

Aby sme zabezpečili čo najmenej skreslené výsledky, každé vyhľadávanie spustíme 10 krát a výsledky vyhľadávaní spriemerujeme. Porovnávať budeme priemerné časy. Program budeme spúšťať po zostavení pre účely vydania programu (*release build*), pretože zostavovanie programu pre účely ladenia (*debug build*) zvyčajne vypína optimalizáciu programu a táto optimalizácia môže mať významný vplyv na výsledky. Výsledky tiež môžu byť ovplyvnené procesmi, ktoré systém vykonáva na pozadí, preto pri meraniach budeme mať všetky procesy náročné na výkon procesora, pamäť alebo pevný disk vypnuté a budeme sa snažiť pre všetky merania zabezpečiť rovnaké podmienky. (vlastné spracovanie)

Vyhľadávania budeme spúšťať a porovnávať na počítači so 64-bitovou verziou operačného systému Windows 10, 16 GB RAM a procesorom AMD Ryzen 5 3600 6-Core Processor 3.60 GHz. (vlastné spracovanie)

3 Výsledky práce a diskusia

3.1 Vytvorenie programu

Na vytvorenie programu použijeme projektovú šablónu *MFC App*. Táto projektová šablóna slúži na vytváranie oknových aplikácií pre operačný systém Windows s možnosťou komplexného používateľského rozhrania. (vlastné spracovanie)

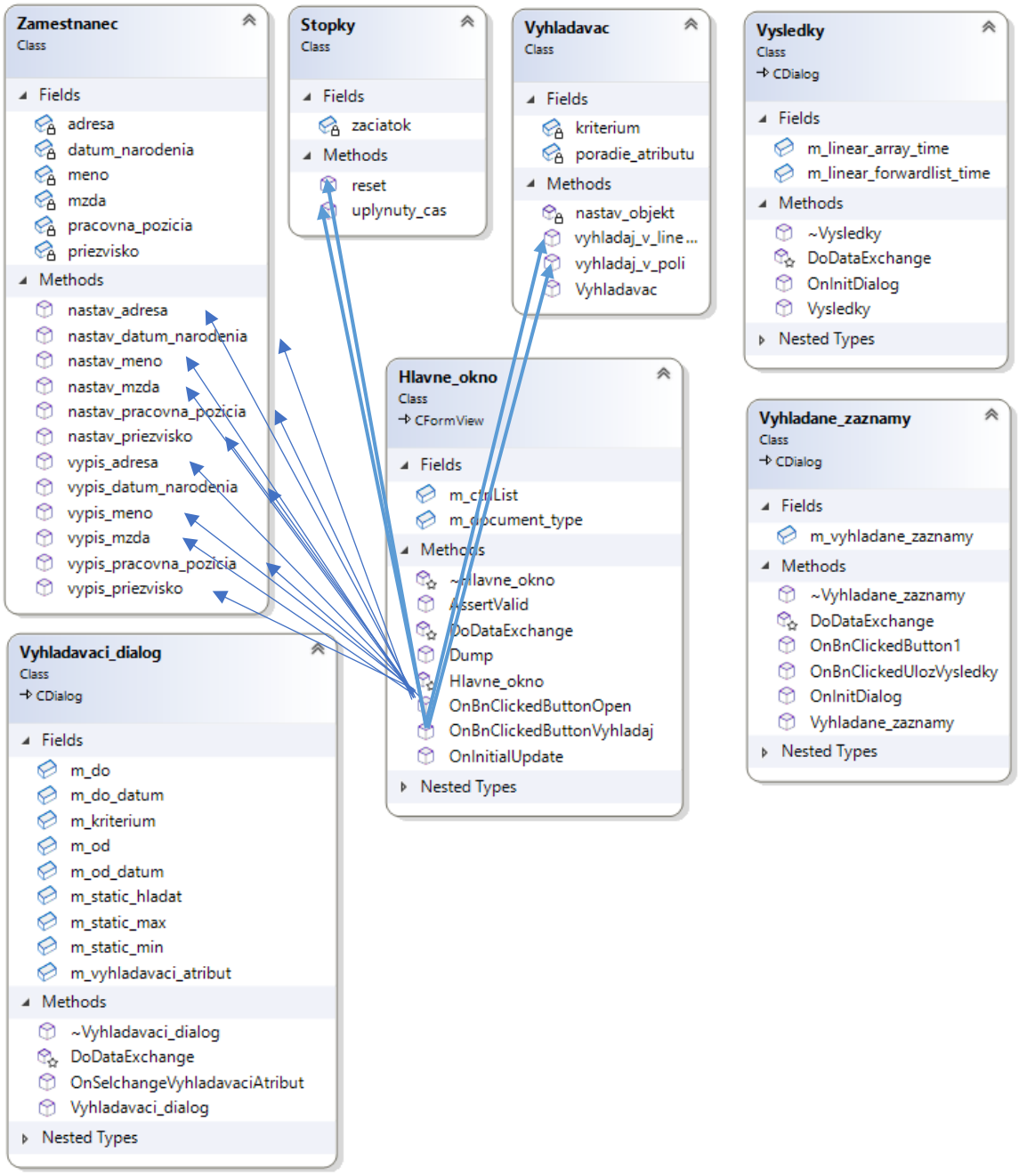
3.2 Vytvorenie tried

Po vytvorení projektovej šablóny si ako prvé vytvoríme 3 základné triedy potrebné pre chod nášho programu. Deklarácie týchto tried uložíme v hlavičkovom súbore *Premenne.h*, ktorý vytvoríme. Do tohto hlavičkového súboru budeme ukladať všetky deklarácie premenných, tried a funkcií. Vždy, keď tieto premenné, triedy alebo funkcie budeme potrebovať v nejakom súbore v našom programe, stačí nám len tento hlavičkový súbor zahrnúť direktívou *#include* do požadovaného súboru. Preprocesor direktívu nahradí obsahom zahrnutého súboru, čím sa obsah stane pre požadovaný súbor dostupný. Definície týchto premenných, tried a funkcií budeme vytvárať v súbore *Premenne.cpp*. (vlastné spracovanie)

Pri tvorbe programu budeme ešte vytvárať ďalšie triedy pre dialógové okná našej aplikácie. Týmto triedam sa budeme venovať neskôr pri opise dialógových okien.

3.3 Triedna architektúra aplikácie

Na obrázku č. 18 je diagram tried našej aplikácie, kde sú šípkami znázornené jednotlivé volania metód metódami iných tried. Všetky metódy troch nami vytvorených tried *Zamestnanec*, *Stopky* a *Vyhladavac* sú volané metódami triedy *Hlavne_okno*, ktorá obsahuje metódy pre rôzne ovládacie prvky hlavného okna. Je to preto, pretože načítanie údajov o zamestnancoch, vyhľadávanie v poli a v jednosmernom lineárnom zozname a meranie exekučných časov sa vykonáva cez hlavné okno. Ostatné dialógové okná slúžia len na zobrazenie a zapísanie týchto výsledkov a ich triedy volajú len svoje vlastné metódy. Podrobnejšie informácie si o vybraných dôležitých metódach a ich volaniach povieme v nasledujúcich podkapitolách venujúcim sa triedam. (vlastné spracovanie)



Obrázok č. 18 – Diagram tried
(vlastné spracovanie pomocou Visual Studia 2022)

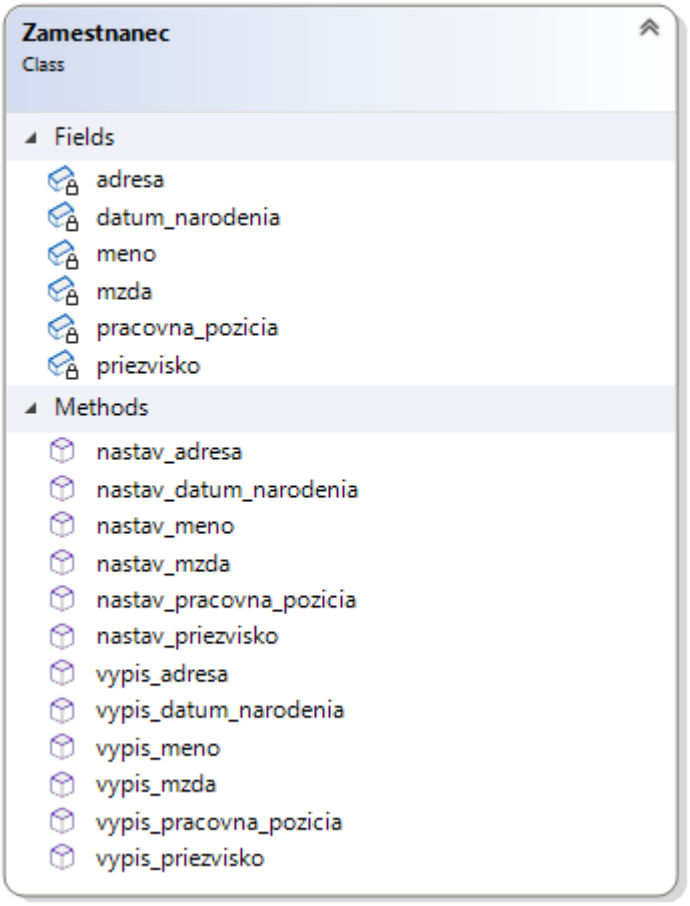
3.3.1 Trieda Zamestnanec

Prvá trieda, ktorú vytvoríme bude obsahovať údaje o zamestnancovi. Program bude vytvárať inštancie tejto triedy a ukladať do nich údaje o zamestnancoch, ktoré načíta zo vstupného súboru. O zamestnancovi budeme ukladať tieto údaje: meno, priezvisko, dátum narodenia, adresa, pracovná pozícia, mzda. Všetky tieto údaje uložíme do premenných, ktoré budú dátovými členmi funkcie. Pre zachovanie princípu zapúzdrenia budú všetky tieto premenné privátne. Ostatné časti programu budú k týmto premenným pristupovať len pomocou

verejných metód, ktoré si v triede vytvoríme. Ku každej premennej vytvoríme metódu, ktorá bude slúžiť na nastavenie hodnoty príslušnej premennej a tiež metódu, ktorá bude slúžiť na vrátenie hodnoty príslušnej premennej. Objekty tejto triedy uložíme do šablóny *vector* s názvom „*zamestnanci*“, ktorú použijeme ako vhodný spôsob pre vytvorenie dynamicky alokovaného poľa a tiež do šablóny *forward_list*, ktorú použijeme na vytvorenie jednosmerného lineárneho zoznamu. (vlastné spracovanie)

```
91    std::vector<Zamestnanec> zamestnanci;
92    std::forward_list<Zamestnanec> z_for;
93    std::vector<Zamestnanec> vyhladany_zamestnanci;
```

Vector vyhladany_zamestnanci budeme používať na ukladanie tých objektov, ktoré splnili požiadavky vyhľadávacieho kritéria. Vzhľadom nato, že tento *vector* používame len na zobrazenie vyhľadaných záznamov a nie na porovnávanie výkonnosti, nie je potrebné vyhľadané objekty ukladať zvlášť aj do jednosmerného lineárneho zoznamu, stačí ich mať uložené v jednej dátovej štruktúre. (vlastné spracovanie)



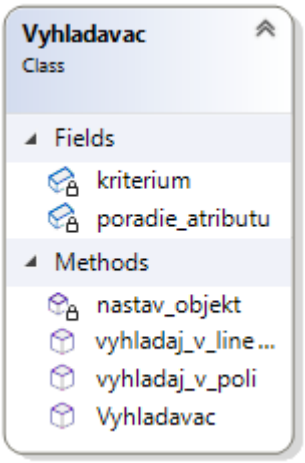
Obrázok č. 19 – Dátové členy a metódy triedy *Zamestnanec*
(vlastné spracovanie pomocou Visual Studia 2022)

3.3.2 Trieda Vyhladavac

Druhou nami vytvorenou triedou bude trieda *Vyhladavac*, ktorej metódy budú slúžiť na vyhľadávanie v poli a na vyhľadávanie v jednosmernom lineárnom zozname. Obsahuje dva dátové členy, prvý na uloženie používateľom zadaného kritéria na vyhľadávanie a druhý na uloženie atribútu, podľa ktorého záznam vyhľadávame. Metódy tejto triedy sú implementované algoritmy lineárneho vyhľadávania. Prvá porovnáva používateľom zadané kritérium s vybraným atribútom všetkých prvkov v poli *zamestnanci,* vyberie tie objekty, ktorým sa vybraný atribút zhoduje so zadaným kritériom a uloží ich do *vector*a *vyhladany_zamestnanci*. Druhá metóda dané kritérium porovná s prvkami jednosmerného lineárneho zoznamu *z_for* a vyhovujúce objekty tiež uloží do *vector*a *vyhladany_zamestnanci*. Medzi jednotlivými vyhľadávaniami sa tento *vector* vyprázdni, čiže metódy vždy ukladajú vyhovujúce objekty do prázdneho *vector*a. (vlastné spracovanie)

Metódu *nastav_objekt(Zamestnanec elm)* využívajú obi dve vyhľadávacie metódy. Pomocou nej pridávajú do *vector*a *vyhladany_zamestnanci*. Konkrétne fungovanie si

vysvetlíme pri popise vyhľadávajúcich metód. Keďže túto metódu využívajú iba členské metódy triedy *Vyhladavac*, nastavili sme ju ako privátnu, pretože nie je potrebné, aby k nej mali prístup ostatné časti programu. (vlastné spracovanie)



Obrázok č. 20 – Dátové členy a metódy triedy *Vyhladavac*
(vlastné spracovanie pomocou Visual Studia 2022)

Vyhľadávacie metódy

Obi dve vyhľadávacie metódy sú implementáciou lineárneho vyhľadávacie algoritmu. Každý jeden prvok od prvého po posledný sa porovná s hľadanou hodnotou. Namiesto toho, aby sa vyhľadávanie pri nájdení prvého prvku so zhodným vybraným atribútom zastavilo, funkcie tento prvok uložia do *vector*a *vyhladany_zamestnanec* a pokračujú ďalej v hľadaní ďalších vyhovujúcich prvkov, až kým neprídu na koniec poľa alebo jednosmerného lineárneho zoznamu. Tým nájdeme všetky vyhovujúce prvky. (vlastné spracovanie)

Telo metódy *vyhladaj_v_poli()* je zobrazené na obrázku č. 21. Každý atribút zamestnanca má priradené svoje číslo. Číslo atribútu, podľa ktorého chceme vyhľadávať zadaný výraz sa uloží do premennej *poradie_atributu*. Vďaka príkazu „*switch*“, ktorý nám umožňuje vetvenie kódu, program vykoná len tú časť kódu, ktorá zodpovedá príslušnému číslu atribútu.

Pomocou slučky *for* program prejde do radu každý jeden objekt uložený v poli *zamestnanci* od prvého po posledný a porovná používateľov zadaný vstup (kritérium vyhľadávania) s hodnotou zvoleného atribútu týchto objektov. Vždy, keď nájde zhodu medzi kritériom a hodnotou zvoleného atribútu, uloží celý objekt „*Zamestnanec*“ do *vector*a *vyhladany_zamestnanci*.

Druhá vyhľadávacia metóda *vyhladaj_v_linearnom_zozname* funguje rovnako ale zadané kritérium neporovnáva s objektami uloženými vo *vector*e ale v jednosmernom lineárnom zozname vytvorenom pomocou šablónovej triedy *forward_list*. (vlastné spracovanie)

```
137  void Vyhladavac::vyhladaj_v_poli()
138  {
139      dlzka = 0;
140      switch (poradie_atributu)
141      {
142      case 0:
143          for (auto& elm : zamestnanci)
144          {
145              if (kriterium == elm.vypis_meno())
146              {
147                  dlzka++;
148                  vyhladany_zamestnanci.resize(dlzka);
149                  nastav_objekt(elm);
150              }
151          }
152          break;
153      case 1:
154          for (auto& elm : zamestnanci) { ... }
163          break;
164      case 2:
165          for (auto& elm : zamestnanci) { ... }
174          break;
175      case 3:
176          for (auto& elm : zamestnanci) { ... }
185          break;
186      case 4:
187          for (auto& elm : zamestnanci) { ... }
196          break;
197      case 5:
198          for (auto& elm : zamestnanci) { ... }
207          break;
208      default:
209          AfxMessageBox(_T("Zaznam sa v zozname nenachadza!"));
210          break;
211      }
```

Obrázok č. 21 – Metóda *vyhladaj_v_poli()* (vlastné spracovanie)

Na uloženie objektu do *vector*a *vyhladany_zamesntanci* vyhľadávacie metódy zavolajú privátnu metódu *nastav_objekt(Zamestnanci elm)*. Má jeden parameter, objekt typu *Zamestnanec*. Všetky dátové členy tohto objektu vloží do *vector*a. Keďže sú tieto dátové členy privátne, nemá k nim priamy prístup. Hodnotu im priradí pomocou verejných metód.

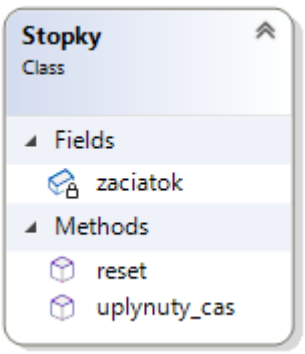
Táto metóda je veľmi užitočná, pretože zabraňuje viacnásobnému opakovaniu kódu. Inak by sme jej telo museli vložiť do každej vetvy príkazu *switch* v oboch vyhľadávacích funkciách. (vlastné spracovanie)

```
127  void Vyhladavac::nastav_objekt(Zamestnanec elm)
128  {
129      vyhladany_zamestnanci[dlzka - 1].nastav_meno(elm.vypis_meno());
130      vyhladany_zamestnanci[dlzka - 1].nastav_priezvisko(elm.vypis_priezvisko());
131      vyhladany_zamestnanci[dlzka - 1].nastav_datum_narodenia(elm.vypis_datum_narodenia());
132      vyhladany_zamestnanci[dlzka - 1].nastav_adresa(elm.vypis_adresa());
133      vyhladany_zamestnanci[dlzka - 1].nastav_pracovna_pozicia(elm.vypis_pracovna_pozicia());
134      vyhladany_zamestnanci[dlzka - 1].nastav_mzda(elm.vypis_mzda());
135  }
```

Obrázok č. 22 – Metóda *nastav_objekt()* (vlastné spracovanie)

3.3.3 Trieda Stopky

Ďalšia vytvorená trieda bude slúžiť na meranie uplynutého času od určitého momentu počas spustenia aplikácie. Obsahuje jeden dátový člen *zaciatok* typu *time_point*, reprezentujúci bod v čase. Tento dátový typ je definovaný v hlavičkovom súbore <*chrono*>, ktorý je súčasťou štandardnej knižnice C++. Pri vytvorení objektu tejto triedy je premenná *zaciatok* hneď inicializovaná na aktuálny časový bod. Môžeme teda povedať, že premenná *zaciatok* má pri inicializácií rovnakú hodnotu, ako čas, v ktorom bol vytvorený objekt triedy *Stopky*. Ďalej má trieda dve metódy, *reset()* a *uplynuty_cas()*. Metóda reset po zavolaní zmení hodnotu premennej *zaciatok* na aktuálny časový bod. Po jej zavolaní už teda premenná nemá rovnakú hodnotu, ako čas vytvorenia objektu triedy *Stopky* ale čas zavolania funkcie *reset()*. Metóda *uplynuty_cas()* odpočíta hodnotu premennej *zaciatok* od súčasného časového bodu. Inými slovami, vypočíta čas uplynutý od poslednej hodnoty premennej *zaciatok* po časový bod, kedy bola táto metóda zavolaná v milisekundách. Týmto spôsobom dokáže objekt tejto triedy merať časové úseky v priebehu aplikácie. (vlastné spracovanie)



Obrázok č. 23 – Dátové členy a metódy triedy *Stopky*
(vlastné spracovanie pomocou Visual Studia 2022)

3.4 Tvorba dialógových okien

Po vytvorení tried, poľa a jednosmerného lineárneho zoznamu, ktoré sa starajú o výpočty a ukladanie dát musíme vytvoriť aj nejaké používateľské rozhranie, vďaka ktorému bude môcť používateľ aplikáciu ovládať. V našom programe budú tvoriť používateľské rozhranie štyri dialógové okná: Hlavné okno, vyhľadávací formulár, dialógové okno so zobrazenými vyhľadanými záznamami a dialógové okno s exekučnými časmi vyhľadávania. Ku všetkým dialógovým oknám musíme tiež vytvoriť ich vlastnú triedu. Pozadie tvorby dialógového okna aj triedy vykoná knižnica MFC. My len v dialógovom editore poskytovanom vývojovým prostredím Visual Studio 2022 rozvrhneme veľkosť a ovládacie prvky okna a pri vytváraní triedy zadáme jej názov a z ktorej rodičovskej triedy chceme dediť. My budeme všetky naše triedy dediť z rodičovskej triedy *Cdialog*. Po zadaní týchto údajov sa automaticky vytvorí hlavičkový súbor s triedou a súbor zdrojového kódu s príponou *.cpp,* oba s nami zvoleným názvom triedy. (vlastné spracovanie)

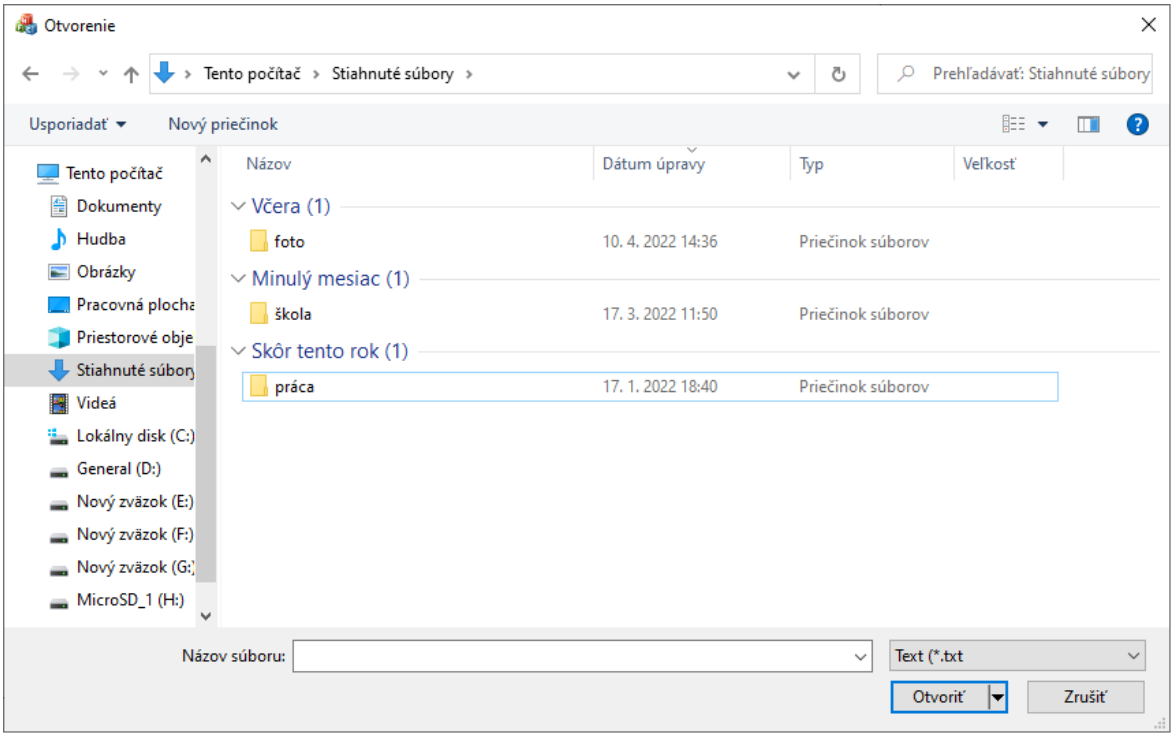
V každom dialógovom okne budú rôzne ovládacie prvky, vďaka ktorými budeme môcť program ovládať. K týmto prvkom je možné podľa potreby priradiť buď premenné alebo funkcie nazývané *Event Handler*. *Event Handler* je funkcia, ktorá je zavolaná vždy pri určitej interakcii s ovládacím prvkom. Keď program zaregistruje určitú udalosť (v preklade „*event*“) spojenú s ovládacím prvkom, zavolá príslušný *Event Handler,* ktorý sa vykoná. Udalosť môže byť rôzny spôsob interakcie, napríklad kliknutie ľavého tlačidla myši, podržanie tlačidla, dvojklik, prejdenie z kurzorom nad určitým ovládacím prvkom. Ak používateľ niektorú z týchto udalostí vykoná, program zavolá náležitý *Event Handler*, do ktorého môžeme napísať kód podľa potreby. Deklarácia *Event Handlera* je uložená v hlavičkovom súbore triedy dialógového okna, ktorému patrí a definícia je uložená v súbore zdrojového kódu triedy. (vlastné spracovanie)

3.4.1 Hlavné dialógové okno

Hlavné dialógové okno je prvé okno, ktoré sa zobrazí po spustení aplikácie. Obsahuje tri ovládacie prvky: Control List, tlačidlo *Otvor súbor* a tlačidlo *Vyhľadať.*

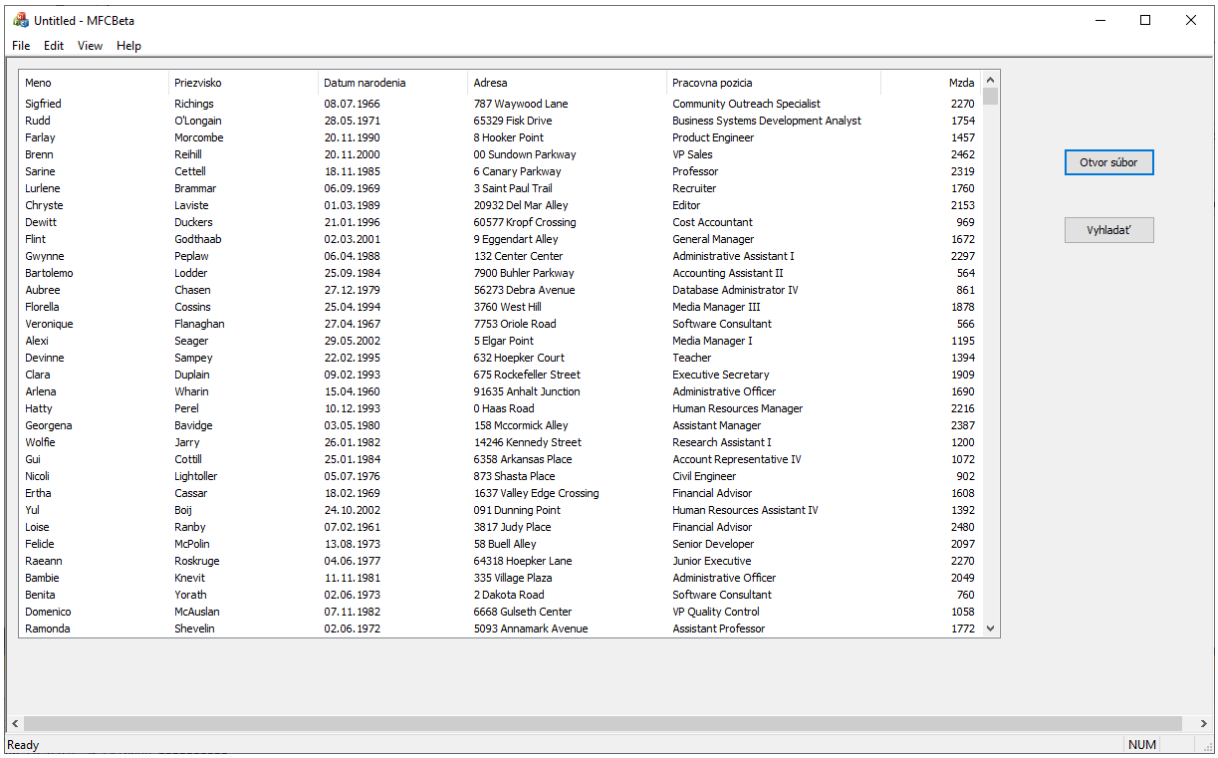
Control List je ovládací prvok, ktorý slúži na zobrazenie zoznamu položiek. Jedná sa teda v podstate o tabuľku, v ktorej vytvoríme hlavičku s názvami stĺpcov (v našom prípade s názvami atribútov), nastavíme počet stĺpcov a do ktorej načítame položky, ktoré sa v nej zobrazia. V našom prípade budú týmito položkami objekty triedy *Zamestnanec*.

Tlačidlo *Otvor súbor* zavolá *Event Handler*, ktorý vytvorí inštanciu triedy *CfileDialog*, ktorá je súčasťou MFC knižnice. Objektom tejto triedy je dialógové okno, v ktorom používateľ môže vybrať súbor, ktorý program otvorí. Na obrázku č. 24 je toto dialógové okno znázornené.



Obrázok č. 24 – Dialógové okno triedy *CFileDialog* (vlastné spracovanie)

Keďže sme si určili, že údaje o zamestnancoch bude program čítať z textových súborov, obmedzili sme zobrazovanie súborov v tomto dialógovom okne len na súbory s príponou *.txt*. Po vybraní textového súboru a kliknutí na tlačidlo *Otvoriť* program prečíta obsah súboru. Aby program správne uložil dáta do objektov tried, musí byť každý záznam o zamestnancovi v jednom riadku a atribúty musia byť oddelené čiarkou. Program si potom uloží jednotlivé riadky súboru do *vectora riadky*. V jednotlivých riadkoch hľadá čiarky, ktoré sú nastavené ako oddeľovacie znaky, vďaka ktorým vie rozlíšiť jednotlivé atribúty. Tieto atribúty potom vloží objektom triedy *Zamestnanec* uložených vo *vectore zamestnanci* a v jednosmernom lineárnom zozname *z_for* vytvorenom pomocou šablónovej triedy *forward_list*. Vzhľadom nato, že obe tieto štruktúry využívajú dynamické alokovanie pamäte, vytvorí program len toľko prvkov, koľko potrebuje, to znamená podľa počtu riadkov vo vstupnom súbore. Okrem toho sa jednotlivé objekty načítajú do interaktívneho prvku *Control List*. (vlastné spracovanie)



Obrázok č. 25 – Hlavné dialógové okno (vlastné spracovanie)

Tlačidlo *Vyhľadať* slúži na vytvorenie nového dialógového okna vyhľadávací formulár. Ak program ešte nenačítal do poľa a jednosmerného lineárneho zoznamu, nemá v čom vyhľadávať údaje a vypíše upozornenie, že zoznam objektov je prázdny. (vlastné spracovanie)

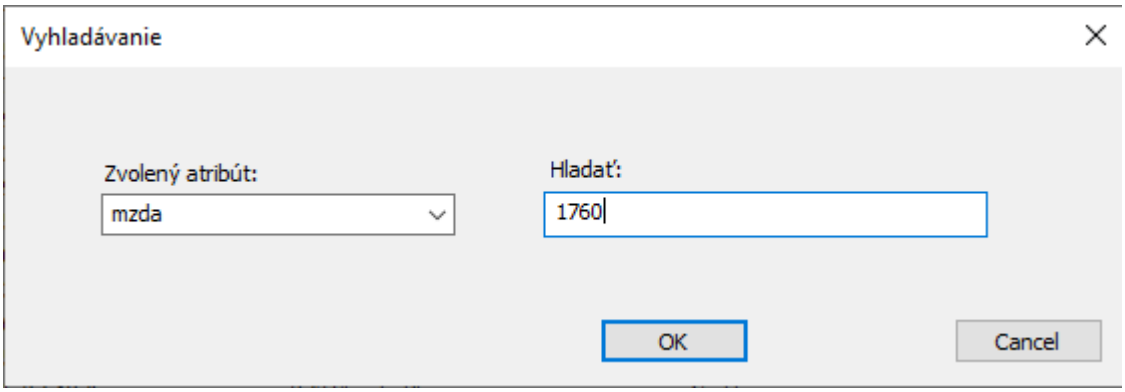
3.4.2 Vyhľadávací formulár

Toto dialógové okno obsahuje dva hlavné ovládacie prvky: *Combo Box* a *Edit Control*. *Combo Box* sa často nazýva aj rozbaľovací zoznam, ktorý obsahuje predvolené hodnoty a používateľ môže po jeho rozbalení jednu z týchto hodnôt vybrať. V našom *Combo Boxe* sa nachádza zoznam atribútov. Používateľa teda vyberie atribút, v ktorom chce vyhľadať zadanú hodnotu. Hodnoty zvoleného atribútu u jednotlivých objektov bude potom program porovnávať so vstupom používateľa. Ak chce teda vyhľadávať objekty napríklad podľa atribútu *mzda*, vyberie tento atribút z *Combo Boxu*. (vlastné spracovanie)

Na zadávanie vstupu od používateľa slúži druhý hlavný ovládací prvok okna – *Edit Control*. Jedná sa o obdĺžnikové okienko, ktoré používateľovi umožňuje zadávať text, v tomto prípade nejakú hodnotu, ktorá sa uloží do premennej vytvorenej k tomuto *Edit*

Controlu. Hodnotu bude potom program porovnávať s hodnotami vybraného atribútu pri všetkých objektoch. (vlastné spracovanie)

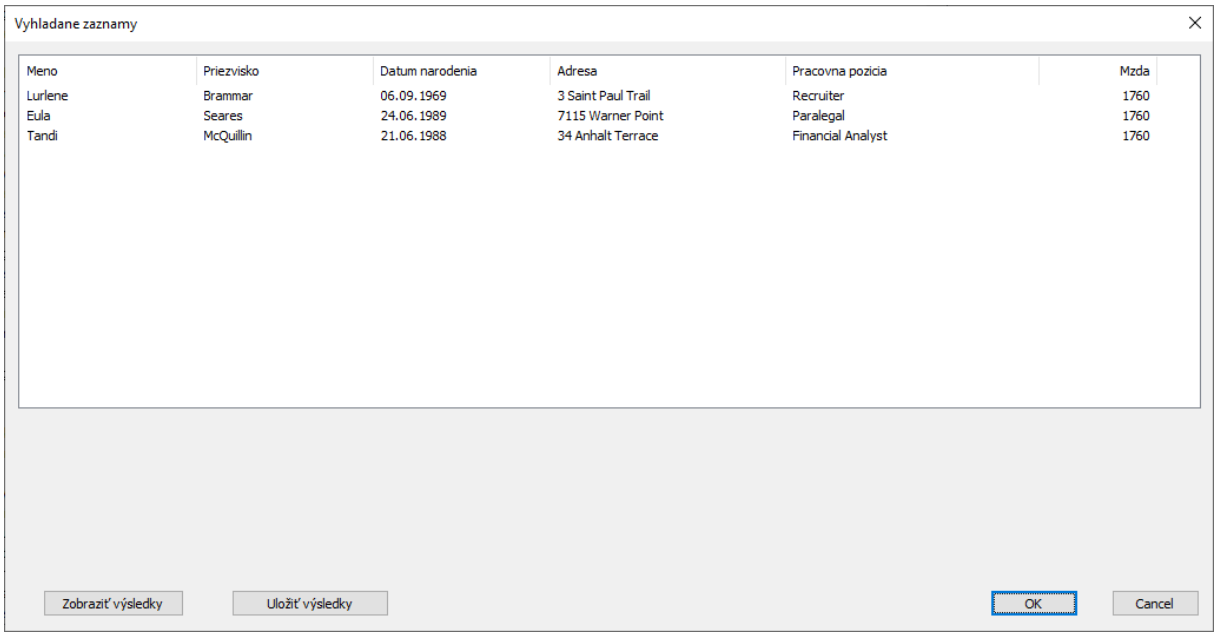
Vyhľadávací formulár ďalej obsahuje dve tlačidlá, tlačidlo *Cancel* slúži na zavretie dialógového okna a vrátenie sa na hlavné dialógové okno. Druhé tlačidlo *OK* slúži na potvrdenie zadaného vstupu a zvoleného atribútu z rozbaľovacieho zoznamu. Po potvrdení sa v prípade, že program našiel záznam, ktorého vybraný atribút sa zhoduje s používateľovým vstupom, otvorí ďalšie dialógové okno s vyhľadanými záznamami. Ak program nenájde žiadny zhodný záznam, zobrazí používateľovi okno s upozornením, že sa v zozname nenašiel žiadny vyhovujúci záznam. (vlastné spracovanie)



Obrázok č. 26 – Vyhľadávací formulár (vlastné spracovanie)

3.4.3 *Dialógové okno so zobrazenými vyhľadanými záznamami*

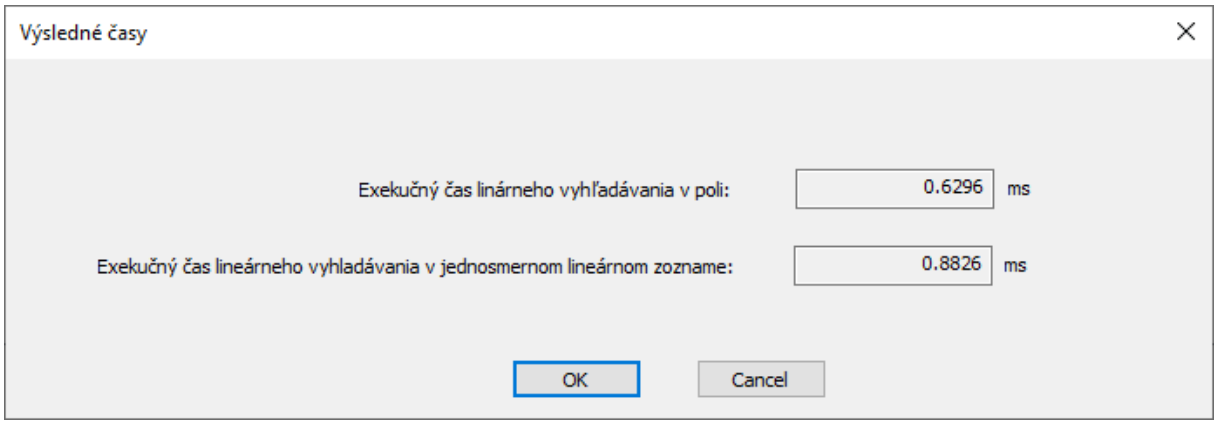
Predposledné dialógové okno, ktoré sme vytvorili obsahuje *Control List*, v ktorom program zobrazí vyhľadané záznamy. Ďalej sa v ňom nachádzajú štyri tlačidlá. Tlačidlo „*Zobraziť výsledky*“ otvorí nové dialógové okno, v ktorom sú zobrazené exekučné časy, koľko milisekúnd trvalo programu vyhľadať objekty v dynamickom poli a koľko milisekúnd v jednosmernom lineárnom zozname. Tlačidlo *Uložiť výsledky* vytvorí výstupný dátový súbor s príponou *.dat* s časovou pečiatkou v názve, do ktorého uloží exekučné časy vyhľadávaní, dátum a čas vyhľadávania a počet prehľadaných záznamov. Posledné dve tlačidlá *OK* a *Cancel* vrátia používateľa späť na hlavné dialógové okno. (vlastné spracovanie)



Obrázok č. 27 – Dialógové okno so zobrazenými vyhľadanými záznamami (vlastné spracovanie)

3.4.4 *Dialógové okno s exekučnými časmi vyhľadávania*

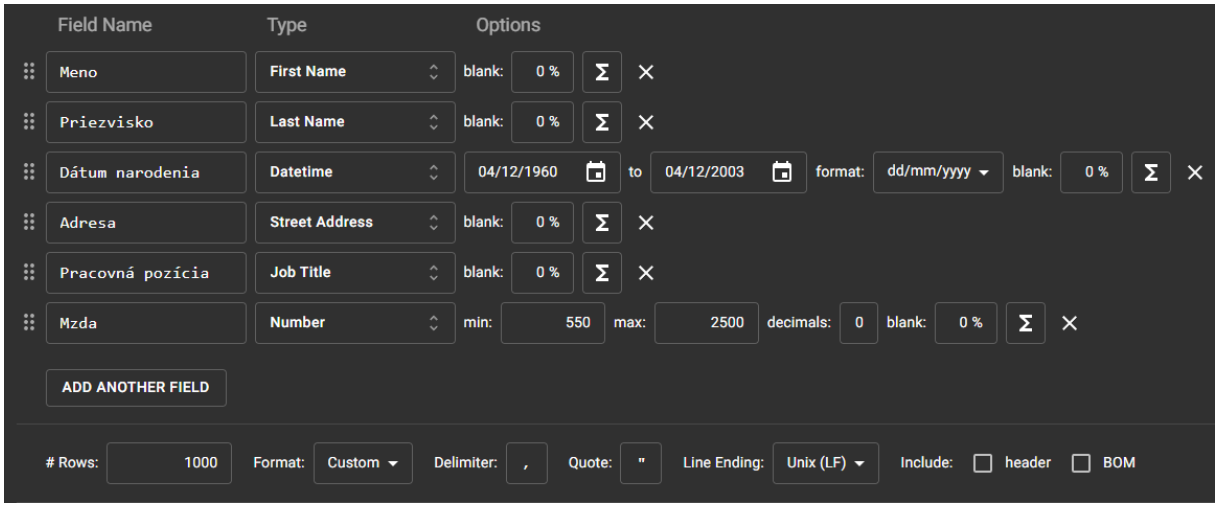
Posledné dialógové okno, ktoré sme vytvorili má dva interaktívne prvky *Edit Control*, ktoré sú prepnuté do režimu „len na čítanie“, pretože do nich nič nevpisujeme, program do nich vypíše exekučné časy vyhľadávania objektov v poli a v jednosmernom lineárnom zozname v milisekundách. Tlačidlá *OK* a *Cancel* vrátia používateľa do dialógového okna so zobrazenými vyhľadanými záznamami. (vlastné spracovanie)



Obrázok č. 28 – Dialógové okno s exekučnými časmi vyhľadávania (vlastné spracovanie)

3.5 Vygenerovanie štruktúrovaných dát

Na vygenerovanie štruktúrovaných dát o fiktívnych zamestnancoch sme použili generátor dát dostupný online na webe *www.mockaroo.com*, ktorý vyhovuje našim účelom, pretože dokáže priamo vygenerovať štruktúrované dáta do riadkov v textovom súbore a oddeliť jednotlivé atribúty čiarkami. Do generátora sme teda zadali mená požadovaných atribútov a pri každom atribúte sme zadali typ údajov, ktorý sa má generovať. Pri dátume narodenia sme zadali, aby sa dátumy generovali len od roku 1960 po rok 2003 vo formáte *dd.mm.yyyy*. Mzdu sme ohraničili na interval od 550 do 2500 eur. Formát súboru sme vybrali *Custom*, oddeľovací znak nastavili čiarku a súbor stiahli. Keďže nám generátor bez registrácie a plateného účtu nedovolil vygenerovať viac ako tisíc riadkový súbor a potrebujeme 10 000 záznamov, dáta sme vygenerovali 10 krát a spojili do jedného textového súboru. Exekučné časy vyhľadávaní budeme ale porovnávať pri rôznych objemoch dát, preto sme časť záznamov tiež nakopírovali do viacerých textových súborov so 100, 1 000, 5 000, a 8 000 záznamami, ktoré budeme postupne v programe otvárať a vyhľadávať v nich. Vytvorené súbory pomenujeme *Vstupne_data_XXX*, kde namiesto znakov *X*, napíšeme, koľko daný súbor obsahuje záznamov. Napríklad súbor s 10 000 záznamami sa teda bude volať *Vstupne_data_10000*. (vlastné spracovanie)



Obrázok č. 29 – Generovanie dát (vlastné spracovanie)

3.6 Spustenie programu

Po vytvorení všetkých tried a funkcií potrebných pre chod nášho programu a po vytvorení používateľského rozhrania pre jeho ovládanie môžeme program skompilovať a spustiť. Ako sme si povedali, kvôli lepšej optimalizácii a menej skresleným výsledkom program

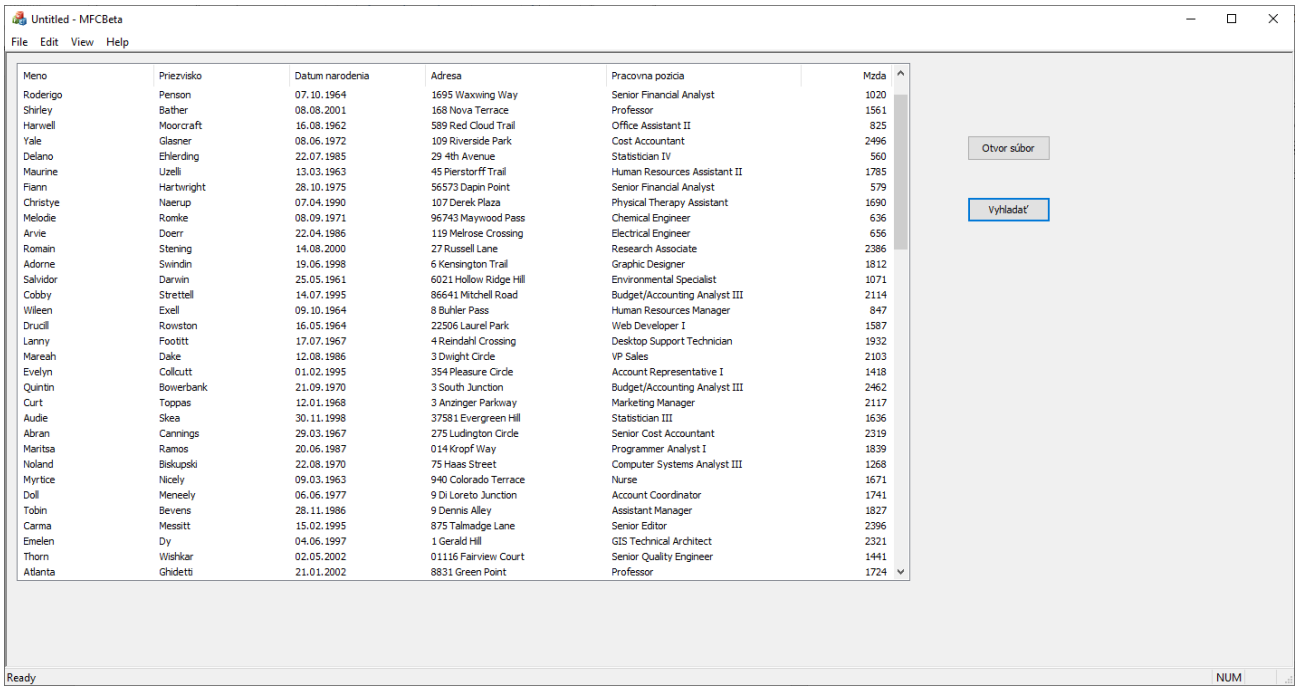
zostavíme v móde *release* určenom pre konečné vydanie programu. Vo *Visual Studiu 2022* vyberieme z horného panelu nástrojov režim *release* a riešenie zostavíme výberom karty *Build* z horného menu a kliknutím na možnosť *Build Solution* z rolovacieho zoznamu. Ak zostavenie a skompilovanie programu prebehlo úspešne, v projektovom adresári v priečinku *Release* sa vytvorí spustiteľný súbor s príponou *.exe* s cestou MFCBeta\x64\Release\MFCBeta.exe. Pomocou tohto súboru budeme náš program spúšťať ako klasickú Windowsovú oknovú aplikáciu. Po úspešnom spustení programu máme všetko pripravené na porovnávanie exekučných časov. (vlastné spracovanie)

3.7 Meranie exekučných časov vyhľadávania v dynamickom poli a v jednosmernom lineárnom zozname

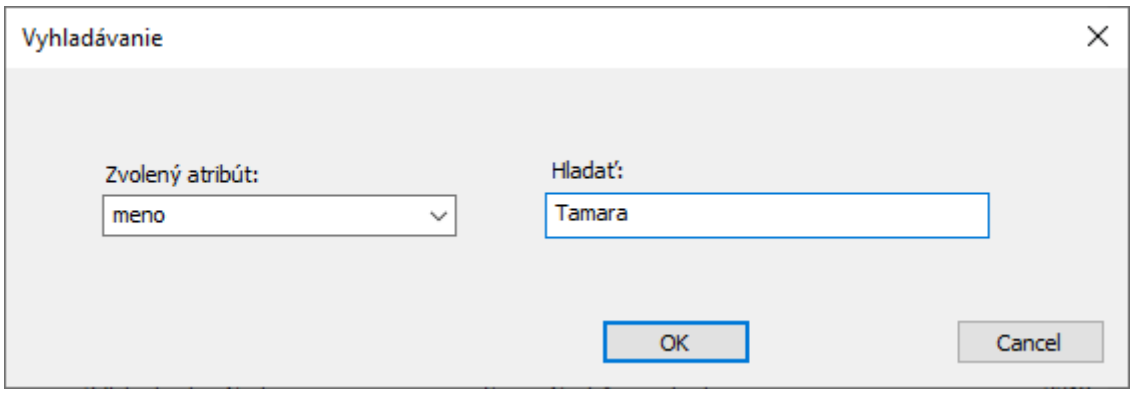
Hlavným cieľom tejto práce je zistiť, či je časovo efektívnejšie vyhľadávanie štruktúrovaných dát v programe vytvorenom v jazyku C++ uložených v dátovej štruktúre pole alebo v jednosmernom lineárnom zozname. Vyhľadávať budeme pomocou implementovaného algoritmu lineárneho vyhľadávania. Pole sme vytvorili pomocou šablónovej triedy *vector* a jednosmerný lineárny zoznam pomocou šablónovej triedy *forward_list*. Časy vyhľadávaní nám zmeria program a z výsledkov vyvodíme záver, v ktorej z vybraných dátových štruktúr dokáže program rýchlejšie vyhľadávať. (vlastné spracovanie)

Meranie exekučných časov začneme od najmenšieho objemu dát a postupne budeme objem dát zvyšovať až na konečných 10 000 záznamov. Každé meranie vykonáme 10 krát a výsledky meraní spriemerujeme. Po vykonaní meraní zhrnieme spriemerované výsledky do tabuľky a porovnáme ich. (vlastné spracovanie)

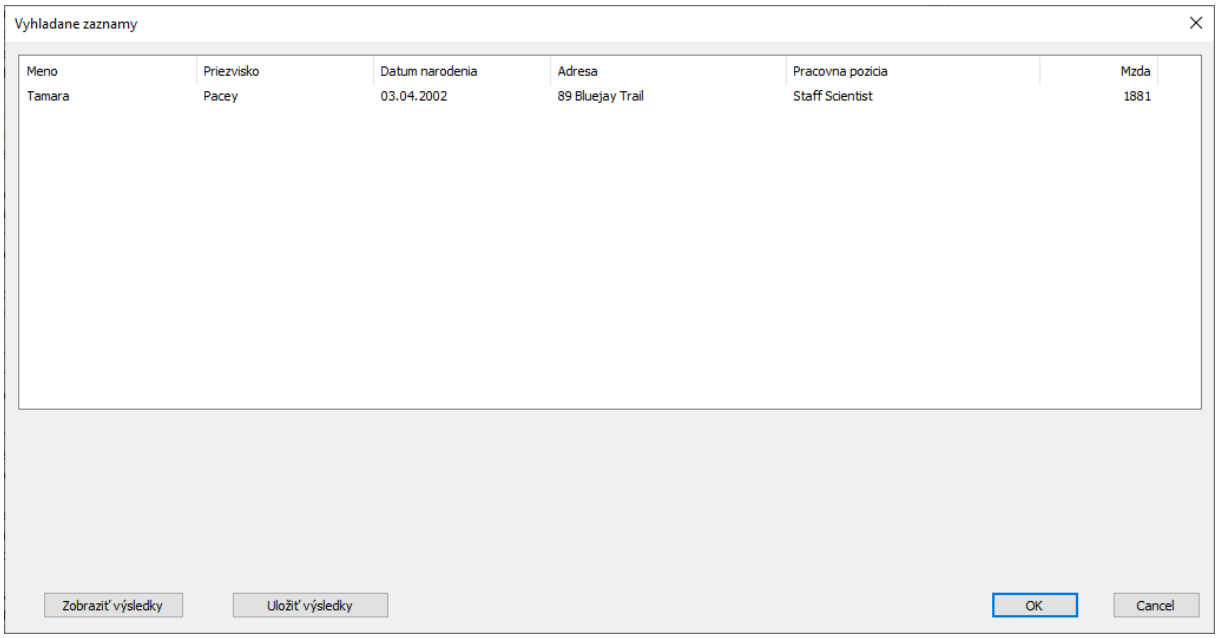
Ako prvý si teda načítame do programu vstupný súbor obsahujúci iba 100 záznamov. V hlavnom dialógovom okne, ktoré sa nám otvorí hneď po spustení programu klikneme na tlačidlo *Otvor súbor* a vyberieme vstupný súbor *Vstupne_data_100*. Štruktúrované dáta načítané zo súboru sa nám zobrazujú v hlavnom dialógovom okne v ovládacom prvku *Control List*. (vlastné spracovanie)



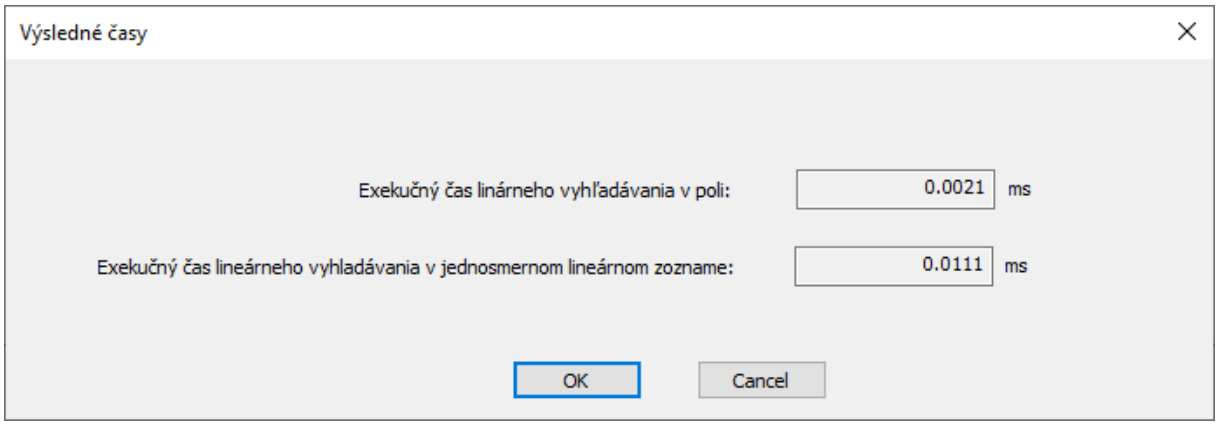
Pre vyhľadávania v 100 záznamoch si vyberieme vyhľadávanie v prvom atribúte *meno* a ako hľadaný výraz zadáme meno *Tamara*. Po kliknutí na tlačidlo *Vyhľadať* vyplníme formulár s určeným atribútom a hľadaným výrazom.



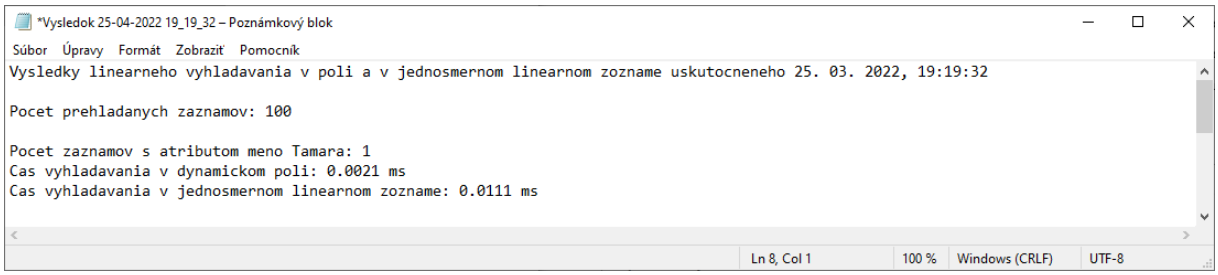
Po vyplnení a kliknutí na tlačidlo *OK* spustíme vyhľadávanie a zobrazíme dialógové okno s vyhľadanými záznamami.



Ako môžeme vidieť na obrázku, medzi 100 záznamami sa nachádzal len jeden s menom Tamara. Tlačidlom zobraziť výsledky zobrazíme exekučné časy vyhľadávaní.



Nájsť záznamy s požadovaným kritériom trvalo programu v poli 0,0021 milisekundy, zatiaľ čo v lineárnom zozname 0,0111 milisekundy. Tlačidlom *Uložiť výsledky* nachádzajúcom sa v dialógovom okne s vyhľadanými záznamami tieto výsledky uložíme do výstupného dátového súboru. Vo výstupnom súbore máme zaznamenané tieto informácie: čas vyhľadávania, počet prehľadaných záznamov, ktorých bolo 100, hľadaný výraz bol Tamara, program vyhľadával v atribúte meno, počet záznamov so zadaným hľadaným výrazom, ktoré program našiel bol a v neposlednom rade najdôležitejší údaj o exekučných časoch vyhľadávaní.



Vyhľadanie s týmto kritériom a vybraným atribútom vykonáme ešte 9 krát a výsledky uložíme do výstupných dátových súboroch, z ktorých potom zistíme exekučné časy. Tieto časy spriemerujeme a zapíšeme do tabuľky výsledkov. (vlastné spracovanie)

Rovnako budeme postupovať aj pri vyhľadávaní väčších objemoch dát, pričom si vždy do programu načítame už pripravené vstupné súbory so vstupnými dátami rôznych objemov. Pri rôznych objemoch dát otestujeme však aj vyhľadávanie podľa rôznych iných atribútov ako meno a iných kritérií. So zväčšovaním množstva záznamov je pravdepodobné, že množstvo vyhľadaných záznamov bude rásť bez ohľadu na zvolený atribút. Lineárny vyhľadávací algoritmus bude mať naďalej rovnaké podmienky pri vyhľadávaní v poli aj v jednosmernom lineárnom zozname (rovnaký atribút aj hľadaný výraz), čím sa zabráni možnému znevýhodneniu vyhľadávania v jednej z týchto dvoch dátových štruktúr. (vlastné spracovanie)

Prehľad zvolených atribútov a kritérií pre rôzne objemy dát je spísaný v tabuľke č. 3.

Počet záznamov	**Zvolený atribút**	**Hľadaný výraz**
100	meno	Tamara
1000	priezvisko	Kings
5000	dátum narodenia	27.02.1963
8000	pracovná pozícia	Marketing Manager
10000	mzda	2210

Tabuľka č. 3 – Priemerné časy vyhľadávaní (vlastné spracovanie)

3.7.1 Záver meraní

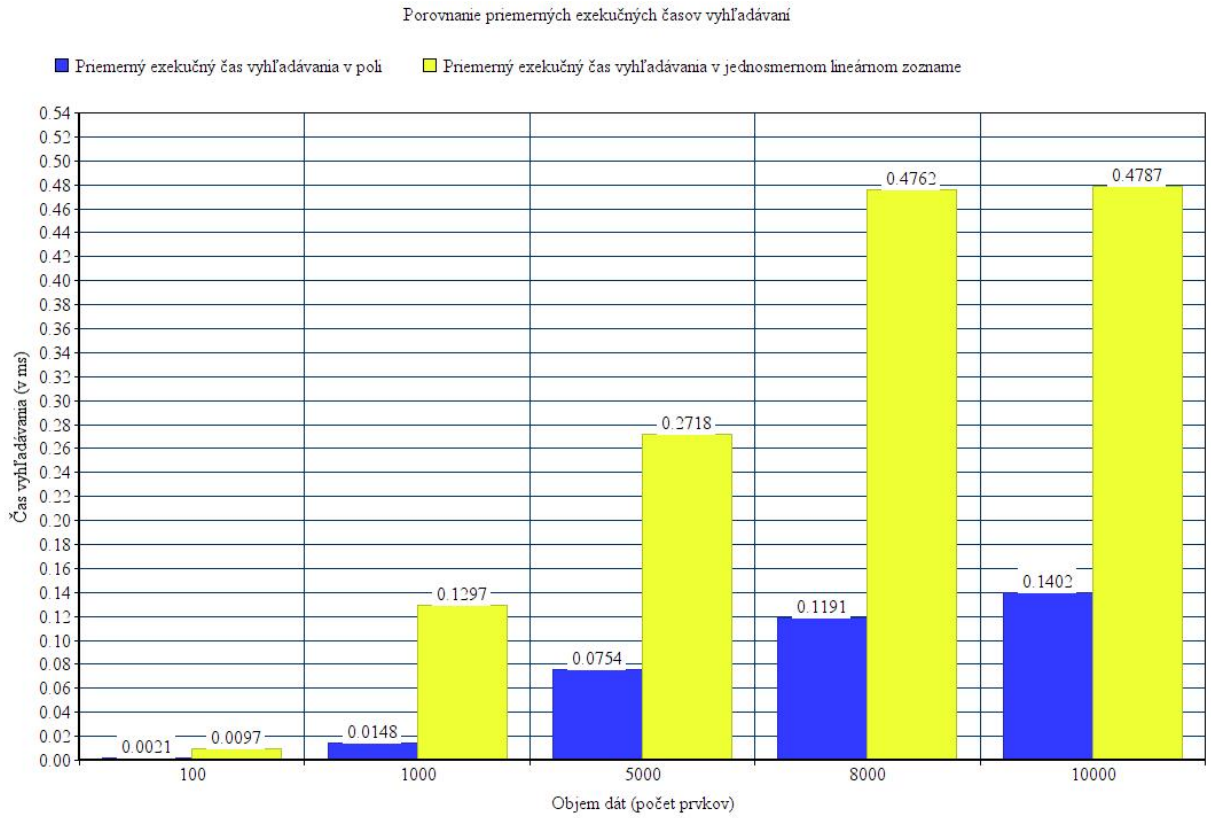
Po vykonaní všetkých meraní sme spriemerované výsledky zapísali do tabuľky č. 4.

Počet prehľadaných záznamov	**Počet vyhľadaných záznamov**	**Priemerný čas vyhľadávania v poli (v ms)**	**Priemerný čas vyhľadávania v jednosmernom lineárnom zozname (v ms)**
100	1	0,0021	0,0097
1000	1	0,0148	0,1297
5000	2	0,0754	0,2718
8000	65	0,1191	0,4762
10000	4	0,1402	0,4787

Tabuľka č. 4 – Priemerné časy vyhľadávaní (vlastné spracovanie)

Z výsledných časov v tabuľke vidíme, že pri každom objeme dát dokázal program vyhľadávať rýchlejšie ako v jednosmernom lineárnom zozname. Zistili sme, že nami vybrané objemy dát nemajú žiadny vplyv nato, či program vyhľadáva rýchlejšie v poli alebo v jednosmernom lineárnom zozname. Objem dát má vplyv na celkovú dĺžku vyhľadávania bez ohľadu nato, v akej dátovej štruktúre sú uložené, pretože lineárne vyhľadávanie má lineárnu časovú zložitosť. Čím viac záznamov potrebujeme prehľadať, tým dlhšie to bude programu trvať. Z tabuľky môžeme vidieť, že prehľadať 100 záznamov trvalo programu v poli 0,0021 milisekundy a v jednosmernom lineárnom zozname 0,0097 milisekundy, zatiaľ čo prehľadať 10000 záznamov mu trvalo v poli 0,1402 milisekundy (približne 66 krát dlhšie) a v jednosmernom lineárnom zozname 0,4787 milisekundy (približne 49 krát dlhšie). (vlastné spracovanie)

Na obrázku č. 30 je graf, ktorý porovnáva priemerné exekučné časy vyhľadávaní v poli a v jednosmernom lineárnom zozname pri rôznych objemoch dát.



Obrázok č. 30 – Porovnanie exekučných časov vyhľadávaní (vlastné spracovanie)

Na grafe je vidieť, že vo všetkých meraniach bolo vyhľadávanie v poli niekoľkonásobne rýchlejšie. Tiež si môžeme všimnúť, že časy všetkých vyhľadávaní s narastajúcim objemom dát lineárne rastú s výnimkou hodnôt pri objeme dát 8000 záznamov, kde je nárast času vyšší, pretože pri vyhľadávaní v 8000 záznamoch bolo nájdených až 65 vyhovujúcich prvkov, zatiaľ čo pri 5000 záznamoch kritériu vyhovovali len dva prvky a pri 10000 záznamoch vyhovovali len štyri prvky. To však nemení nič na tom, že aj pri týchto vyhľadávaniach bolo vyhľadávanie v poli rýchlejšie. (vlastné spracovanie)

Bez ohľadu na množstvo dát je vyhľadávanie v poli stále rýchlejšie. Ako sme skôr v práci spomínali, prvky poľa sú v pamäti uložené súvisle za sebou. Súvislé uloženie prvkov je kľúčové k ľahkému a rýchlemu prístupu k jednotlivým prvkom. Prvky jednosmerného lineárneho zoznamu sú v pamäti rozmiestnené nesúvisle a prepojené sú len ukazovateľom uloženom v každom prvku, ktorý ukazuje na nasledujúci prvok v zozname. Toto uloženie prvkov v pamäti si môžeme skontrolovať aj v našom programe. V tabuľke č. 5 sú vypísané adresy prvých piatich prvkov z poľa a z jednosmerného lineárneho zoznamu z jedného z našich meraní. (vlastné spracovanie)

Číslo prvku	Adresa prvku poľa		Adresa prvku jednosmerného lineárneho zoznamu	
	Hexadecimálny tvar	Desiatkový tvar	Hexadecimálny tvar	Desiatkový tvar
0	00000157DF1B0AC0	1476916873920	00000157E56DDAA8	1477022964392
1	00000157DF1B0AF0	1476916873968	00000157E56D64C8	1477022934216
2	00000157DF1B0B20	1476916874016	00000157E56D6648	1477022934600
3	00000157DF1B0B50	1476916874064	00000157E56DDD68	1477022965096
4	00000157DF1B0B80	1476916874112	00000157E56D6468	1477022934120

Tabuľka č. 5 – Priemerné časy vyhľadávaní (vlastné spracovanie)

Pre lepšie znázornenie sme hexadecimálny tvar čísla, v ktorom sa adresy bežne udávajú previedli do desiatkovej sústavy, ktorá je ľuďom bližšia. Pri prvkoch v poli môžeme vidieť, že adresy idú za sebou v pravidelných intervaloch 48 bajtov. Prvý prvok je od druhého vzdialený 48 bajtov, druhý od tretieho 48 bajtov atď. To znamená, že program potrebuje na uloženie jedného objektu zamestnanec 48 bajtov pamäte. Vďaka tomu môžeme aj pomocou matematických operácií pristupovať k jednotlivým prvkom. Ak by sme napríklad chceli, aby ukazovateľ na prvý prvok ukazoval na druhý prvok, stačí k nemu prirátať hodnotu potrebnú na uloženie jedného prvku (v našom prípade 48 bajtov). To, že program vie, že prvky poľa musia byť v pamäti vždy pri sebe a aj v konštantnej vzdialenosti od seba mu umožňuje rýchly prístup k prvkom a rýchle prechádzanie prvkami poľa. (vlastné spracovanie)

Ako môžeme vidieť v tabuľke č. 5, prvky jednosmerného lineárneho zoznamu nie sú v pamäti blízko seba a nie sú od seba ani v konštantnej vzdialenosti. To, že sú prvky medzi sebou poprepájané len ukazovateľmi ich robí pre program ťažšie prístupné a náročnejšie pre prehľadávanie. (vlastné spracovanie)

Práve toto usporiadanie prvkov v pamäti je jeden z dôvodov, prečo je vyhľadávanie v poli rýchlejšie. Z týchto skutočností a našich výsledkov meraní vyplýva, že polia sú vhodnejšou dátovou štruktúrou pre vyhľadávanie štruktúrovaných dát v jazyku C++. (vlastné spracovanie)

Záver

Počas celej práce sme zisťovali informácie o poliach a jednosmernom lineárnom zozname, ktoré nás viedli k tomu, že polia sú zrejme vhodnejšou dátovou štruktúrou pre vyhľadávanie štruktúrovaných dát v jazyku C++. Tieto fakty sme overili a potvrdili vytvorením programu, pomocou ktorého sme exekučné časy vyhľadávaní v týchto dátových štruktúrach. Nezávisle od objemu dát bolo vždy vyhľadávanie v poli časovo efektívnejšie. Ak sa jedná o vyhľadávanie dát je zrejme lepšie uložiť tieto dáta v poli.

Pri analyzovaní rôznych dátových štruktúr sme však zistili, že aj jednosmerný lineárny program má oproti poliam určité výhody, ako je rýchlejšie vkladanie a odstraňovanie prvkov. V úvode práce sme spomínali, že je úlohou programátora, aby zvolil taký nástroj, pomocou ktorého čo najlepšie optimalizuje program. Ak teda programátor tvorí program, v ktorom sa prvky v dátovej štruktúre často presúvajú, odstraňujú alebo pridávajú do stredu alebo na začiatok dátovej štruktúry, je na ňom, aby zvážil, či je výhodnejšie radšej optimalizovať tieto operácie prednostnejšie ako napríklad vyhľadávanie alebo triedenie prvkov. Pre optimalizáciu týchto operácií je potom lepšie zvoliť jednosmerný lineárny zoznam. Ak však programátor dopredu vie, že prvky sa budú pridávať a odstraňovať najmä na konci radu prvkov a väčšinu času sa s nimi nebude hýbať, pre operácie s týmito prvkami je zrejme vhodnejšie použiť na ukladanie štruktúrovaných dát pole.

Zoznam použitej literatúry

Access functions and encapsulation. In: learncpp.com [online]. 2022 [cit. 2022-04-12]. Dostupné na: https://www.learncpp.com/cpp-tutorial/access-functions-and-encapsulation/

Arrays (Part I). In: learncpp.com [online]. 2022 [cit. 2022-04-12]. Dostupné na: https://www.learncpp.com/cpp-tutorial/arrays-part-i/

Back To Basics: C++ Containers. In: youtube.com [online]. 2021. [cit. 2022-04-12]. Dostupné na: https://www.youtube.com/watch?v=6OoSgY6NVVk&ab_channel=javidx9

BARNETT, Granville -TONGO, Luca. *Data Structures and Algorithms: Annotated Reference with Examples* [ekniha]. 1. vyd. 2008. 112 s. [cit. 2022-03-25]. Dostupné na: https://apps2.mdp.ac.id/perpustakaan/ebook/Karya%20Umum/Dsa.pdf

Dynamic memory allocation with new and delete. In: learncpp.com [online]. 2022 [cit. 2022-04-05]. Dostupné na: https://www.learncpp.com/cpp-tutorial/dynamic-memory-allocation-with-new-and-delete/

Introduction to compound data types. In: learncpp.com [online]. 2022 [cit. 2022-04-10]. Dostupné na: https://www.learncpp.com/cpp-tutorial/introduction-to-compound-data-types/

Introduction to fundamental data types. In: learncpp.com [online]. 2021 [cit. 2022-04-10]. Dostupné na: https://www.learncpp.com/cpp-tutorial/introduction-to-fundamental-data-types/

Introduction to objects and variables. In: learncpp.com [online]. 2021 [cit. 2022-04-10]. Dostupné na: https://www.learncpp.com/cpp-tutorial/introduction-to-objects-and-variables/

Introduction to structs, members, and member selection. In: learncpp.com [online]. 2022 [cit. 2022-04-11]. Dostupné na: https://www.learncpp.com/cpp-tutorial/introduction-to-structs-members-and-member-selection/

KIRCH-PRINZ, Ulla – PRINZ, Peter. *A Complete Guide to Programming in C++*. 1. vyd. Sudbury: Jones and Bartlett Publishers, 2001. 825 s. ISBN: 0-7637-1817-3

MALIK, Davender. *Data structures using C++*. 2. vyd. Boston: Course Technology, 2009. 912 s. ISBN: 978-0-324-78201-1

NAEEM, T. *Understanding Structured, Semi-Structured, and Unstructured Data*. In: astera.com [online]. Westlake Village [cit. 2022-03-22] Dostupné na: https://www.astera.com/type/blog/structured-semi-structured-and-unstructured-data/

PRATA, Stephen. *Mistrovství v C++ 4. aktualizované vydání*. 4. vyd. Brno: Computer Press, 2013. 1176 s. ISBN 978-80-251-3828-1

REESE, Richard. *Understanding and Using C Pointers*. 1. vyd. Sebastopol: O'Reilly Media, Inc., 2013. 226 s. ISBN: 978-1-449-34418-4

WEISS, Mark. *Data Structures and Algorithm Analysis in C++*. 4. vyd. Londýn: Pearson Education, 2013. 635 s. ISBN: 978-0-13-284737-7

Welcome to object-oriented programming. In: learncpp.com [online]. 2022 [cit. 2022-04-11]. Dostupné na: https://www.learncpp.com/cpp-tutorial/welcome-to-object-oriented-programming/