VISOKA ŠKOLA ZA INFORMACIJSKE TEHNOLOGIJE
STRUČNI STUDIJ INFORMACIJSKE TEHNOLOGIJE
ZAGREB

ZAVRŠNI RAD

STL DIO STANDARDNE C++ BIBLIOTEKE

Dejan Barušić

Zagreb, rujan 2017. godine

SADRŽAJ

POPIS SLIKA

POPIS TABLICA

1 UVOD

Programski jezik C++ podržava različite načine programiranja ili programske paradigme. Danas je objektno orijentirano programiranje zasluženo najzastupljeniji stil dizajniranja programskog koda, ali nije jedini stil kojeg programer može primijeniti u rješavanju specifičnog problema, a ponekad nije niti najbolji. C++ programski jezik podržava programiranje na nekoliko razina apstrakcije, počevši od najniže gdje programer piše program koji upravlja podacima na razini bitova i bajtova pa do najviše razine, one generičkog programiranja, gdje apstrakcija nadilazi čak i tipove podataka i objekata.

Između ove dvije krajnosti postoji razina strukturiranog programiranja, poznata iz programskog jezika C, kod koje se program sastoji od struktura podataka, petlja za upravljanje tokom izvođenja i poziva pod-funkcija iz *main* funkcije. Ovakav stil programiranja još je uvijek u upotrebi, nakon više od 40 godina od prvog pojavljivanja jezika C. Čak se i dijelovi modernih operacijskih sustava pišu u ovom stilu zajedno sa visoko zahtjevnim programima na osobnim računalima, ali i ugrađenim sustavima poput onih u avio i auto industriji.

Idući korak je razina podatkovne apstrakcije kod koje se i podaci i kod koji njima manipulira pakiraju u zajedničke strukture, umjesto zasebnih podatkovnih struktura i zasebnih funkcija koje rade sa tim strukturama. Zbog podrške ovakvom stilu programiranja u C++ jeziku, moguće je definirati vlastite tipove koji su potpuno ravnopravni ugrađenim tipovima. Razina podatkovne apstrakcije radi sa konkretnim tipovima, dok iduća razina, ona objektno orijentiranog programiranja, koristi virtualne funkcije i polimorfizam. U objektno-orijentiranom stilu programiranja programi su organizirani u skupove objekata koji međusobno surađuju, a svaki objekt predstavlja instancu neke klase. Klase su organizirane u hijerarhije klasa sa mehanizmom nasljeđivanja. Ovakav pristup daje programeru široki raspon mogućnosti u dizajniranju i implementaciji programa.

Standard Template Library (STL) je biblioteka generičkih algoritama i podatkovnih struktura koju je za C++ programski jezik razvio Alexander Stepanov sa suradnicima. Osnovno načelo STL-a je generičko programiranje koje omogućava pisanje algoritama neovisnih o tipu podataka koje obrađuju. Takav pristup ima za prednost učinkoviti polimorfizam koji se izvodi za vrijeme kompajliranja, za razliku od klasičnog polimorfizma koji se izvodi za vrijeme izvođenja programa. Ova biblioteka je imala značajan utjecaj na standardnu C++ biblioteku u koju je u donekle izmijenjenom obliku i integrirana.

Predmet proučavanja ovog rada je STL dio standardne C++ biblioteke koji se sastoji od ove tri

glavne komponente:

- podatkovne strukture ili spremnici (*STL containers*),
- iteratori (*STL iterators*) i
- algoritmi (*STL algorithms*).

Naredna poglavlja obrađuju pojam algoritma, kako se mjeri učinkovitost algoritama i što je generičko programiranje. Zatim se detaljnije obrađuju spremnici podataka i iteratori, dok su algoritmi glavna tema ovog rada i njima je posvećeno najviše prostora.
Spremnici se dijele na sljedeće vrste:

- *array*,
- *vector*,
- *deque*,
- *list*,
- *set* i *multiset* i
- *map* i *multimap*.

Iteratori su obrađeni po ovim cjelinama:

- ulazni iteratori (*input iterators*),
- izlazni iteratori (*output iterators*),
- jednosmjerni iteratori (*forward iterators*),
- dvosmjerni iteratori (*bidirectional iterators*) i
- iteratori sa nasumičnim pristupom (*random-access iterators*).

Algoritmi su obrađeni prema ovim funkcionalnim cjelinama:

- algoritmi koji ne mijenjaju sadržaj kojeg obrađuju (*nonmodifying*),
- algoritmi koji mijenjaju sadržaj kojeg obrađuju (*modifying*),
- algoritmi koji uklanjaju podatke iz ulaznog seta podataka (*removing*),
- algoritmi koji mijenjaju poredak u ulaznom setu podataka (*mutating*),
- algoritmi koji sortiraju ulazni set podataka (*sorting*),
- algoritmi koji zahtijevaju da ulazni set podataka bude sortiran (*sorted-range*) i
- algoritmi za obradu numeričkih vrijednosti (*numeric*).

Svaka cjelina je opisana uz detaljan pregled nekoliko algoritama koji je najbolje predstavljaju.

2 DEFINICIJA ALGORITMA I GENERIČKO PROGRAMIRANJE

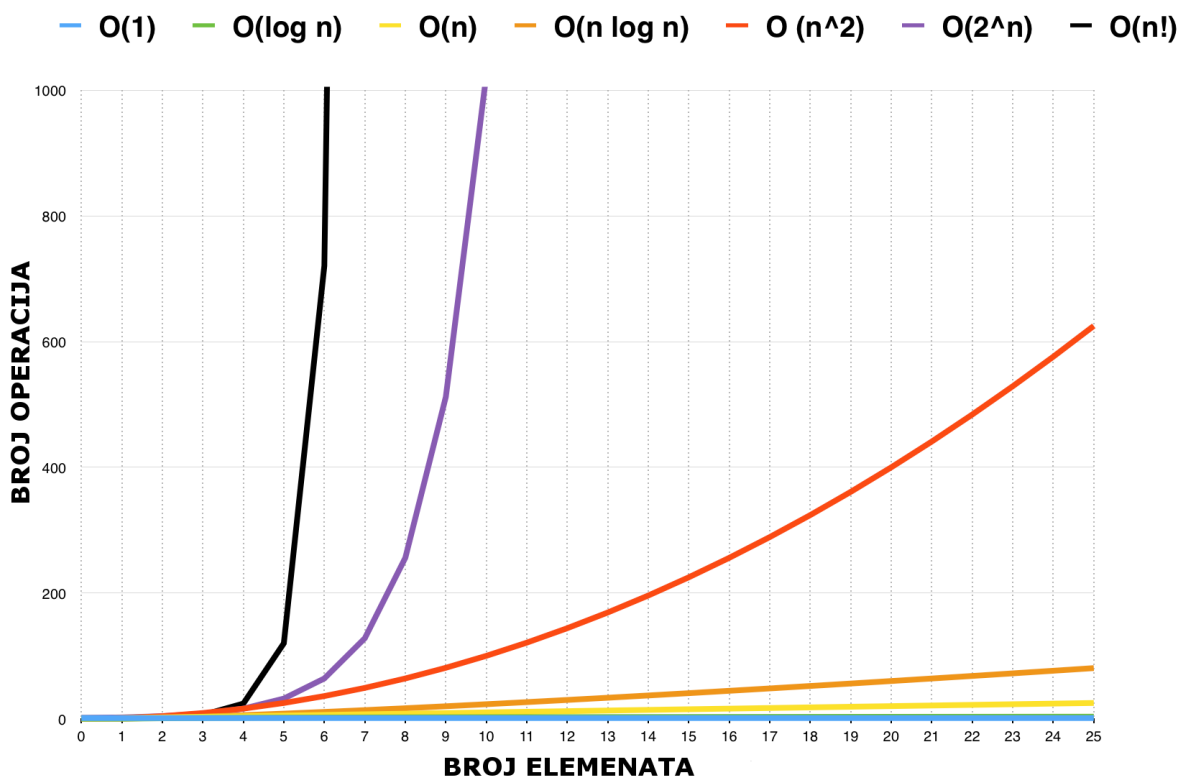
2.1 Definicija algoritma i *Big O* notacija

Algoritam u matematici i računarstvu označava slijed operacija koje izvršavanjem daju određeni rezultat. Algoritam se definira formalnim jezikom i treba biti ostvariv u konačnom broju koraka, u konačnom vremenu i u konačnom memorijskom prostoru. Neformalno, algoritam su pravila koja precizno određuju slijed operacija. Takva definicija uključuje sve računalne programe, čak i one koji ne obavljaju matematičke operacije. Treba napomenuti da program koji se izvršava beskonačno ne zadovoljava definiciju algoritma.

Treba razlučiti algoritam od funkcije koje se izvršava algoritmom jer za bilo koju funkciju može postojati više odgovarajućih algoritama. Dobar primjer su algoritmi za sortiranje. Postoji mnoštvo takvih algoritama koji za isti set ulaznih podataka daju identičan rezultat, ali to čine na mnoštvo različitih načina. To nas dovodi do problema učinkovitosti algoritama. U računarstvu učinkovitost ima dva aspekta. Jedan je broj koraka koje algoritam treba izvršiti kako bi proizveo rezultat, što je zapravo mjera vremena. Na računalima jednakih performansi neki algoritmi će se izvršavati brže od drugih bez obzira što će rezultat biti identičan. Drugi aspekt je prostorni, to jest, količina memorije koja je algoritmu potrebna za izračunavanje rezultata. Izabrati odgovarajući algoritam nije jednoznačno u svim situacijama nego zavisno od uvjeta u kojima će se taj algoritam izvršavati. Na računalima sa vrlo ograničenim memorijskim resursima izvršavanje je ograničeno na algoritme čija prostorna složenost ne prelazi raspoložive resurse bez obzira na moguću veliku vremensku složenost.

Kako bi se mogao izabrati odgovarajući algoritam za određenu funkciju mora postojati odgovarajuća metoda kojom ih možemo međusobno usporediti. U računarstvu se koristi takozvana *Big O* notacija koja klasificira algoritme prema načinu na koji njihova vremenska ili prostorna složenost ovisi o rastu broja ulaznih elemenata podataka. U takvom zapisu n predstavlja broj ulaznih elemenata pa $O(n)$ znači da složenost raste linearno sa povećanjem broja ulaznih elemenata. $O(1)$ znači da složenost ne ovisi o broju ulaznih elemenata dok $O(n^2)$ znači da složenost raste s kvadratom broja ulaznih elemenata. Slika 1 prikazuje graf sa raznim tipovima složenosti gdje se vidi kako povećanje broja elemenata prikazano na x osi grafa ima utjecaj na broj operacija koje algoritam treba izvršiti, a koje su prikazane na y osi. Kvalitetni algoritmi mogu imati drastičan utjecaj na performanse programa i zbog toga su u standardnoj biblioteci uključeni najmanje složeni algoritmi u odnosu na slične algoritme koji obavljaju istu funkciju. Zbog toga se preporuča korištenje standardne biblioteke umjesto pisanja vlastitih

rješenja.



Slika 1: Ovisnost broja operacija o broju ulaznih elemenata

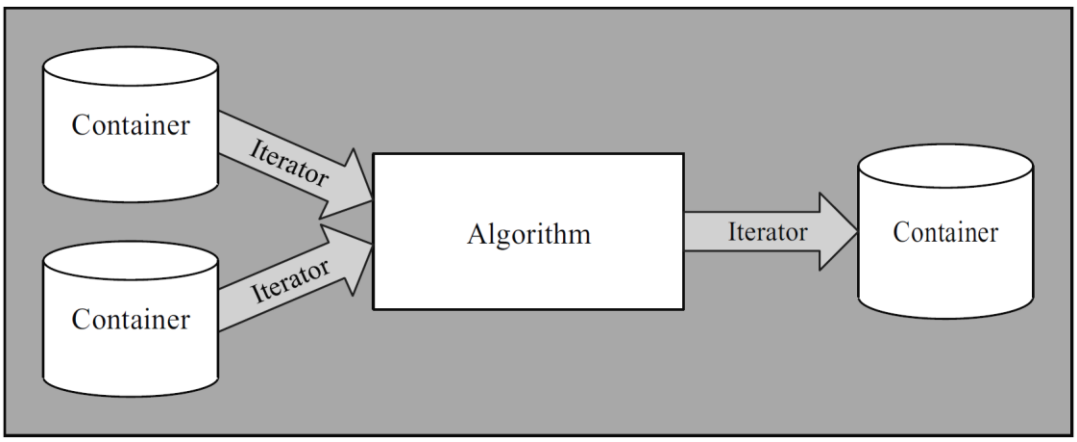
2.2 Generičko programiranje

Generičko programiranje je stil programiranja u kojem se tipovi podataka koje obrađuju algoritmi ne specificiraju za vrijeme pisanja algoritma već u trenutku upotrebe algoritma prilikom pisanja programa. Na primjer, za danih *n* tipova spremnika kao što su liste, vektori i slično i *m* algoritama koji rade sa tim tipovima podataka trebamo imati *n x m* konkretnih implementacija. Generičko programiranje razbija takav blok konkretnih implementacija na apstraktnu razinu gdje je algoritam odvojen od podatkovnih tipova koje obrađuje. U takvom slučaju je potrebna samo *n + m* kombinacija algoritama i spremnika.

Većina modernih programskih jezika pruža ugrađene mogućnosti za generičko programiranje iako se nazivi ponekad razlikuju. U C++ jeziku generičko programiranje se ostvaruje putem pojma *template*, dok u drugim jezicima kao što su C#, Java, Objective-C, Rust i Swift nosi naziv *generics*.

STL dio standardne biblioteke u C++ jeziku se zasniva na odvajanju podataka od operacija. Podacima upravljaju spremnici, a operacije vrše algoritmi dok ih iteratori povezuju kako je

prikazano na slici 2.



Slika 2: Veza između STL komponenata

Ovaj pristup programiranju je različit od objektno orijentiranog pristupa gdje se podaci i operacije sjedinjuju u objekte, no gore opisanim pristupom dobili smo vrlo kompaktnu i fleksibilnu biblioteku.

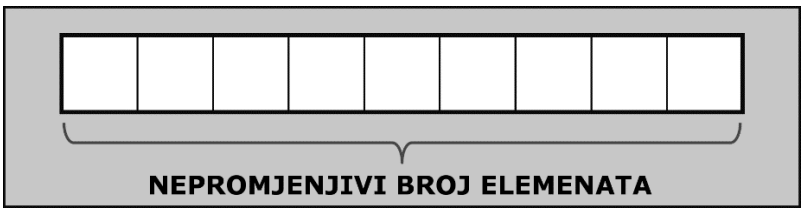
3 STRUKTURE PODATAKA, ITERATORI I ALGORITMI U STL DIJELU C++ STANDARDNE BIBLIOTEKE

3.1 Strukture podataka ili spremnici

Spremnici (*containers*) služe za upravljanje kolekcijama određenih vrsta objekata. Svaka vrsta spremnika ima svoje prednosti i mane te stoga i izbor ovisi o zahtjevima koji trebaju biti zadovoljeni u raznim programima. Spremnici mogu biti implementirani kao uzastopna polja podataka u memoriji (*array*), kao vezane liste gdje svaki element u strukturi pokazuje na idući koji se nalazi na drugoj lokaciji u memoriji, a mogu biti izvedeni po principu da svaki element ima pridružen neki ključ koji ga identificira.
U ovom poglavlju su opisane strukture podataka koje postoje u STL biblioteci i njihove karakteristike.

Array

Array je polje ili niz elemenata nepromjenjive veličine, kako je prikazano na slici 3. U slučaju kad je potrebno promijeniti kapacitet niza, moguće je samo napraviti novi statički niz nepromjenjivog kapaciteta i u njega kopirati sve postojeće elemente.

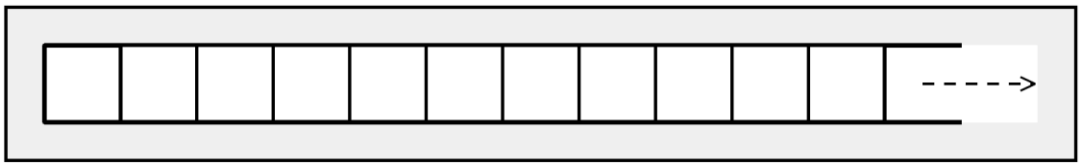


Slika 3: *Array* tip podatkovne strukture

STL *array* pruža standardno sučelje prema iteratoru koje je omotnica oko statičkog *array*-a, kako je definiran u programskom jeziku C. Zbog toga što je to uređena struktura podataka, što znači da se točno zna lokacija svakog elementa strukture u memoriji, svakom elementu se može pristupiti direktno, pod uvjetom da se zna njegova pozicija u polju. Iteratori koji mogu raditi s *array*-em su iteratori sa nasumičnim pristupom (*random-access iterators*). Kako su nepromjenjivog kapaciteta, mora se već kod kreiranja definirati njihov kapacitet, što znači da se ne može koristiti prazan konstruktor.

Vector

Vector predstavlja dinamički niz, što znači da je njegov kapacitet promjenjiv, kao što je prikazano na slici 4.

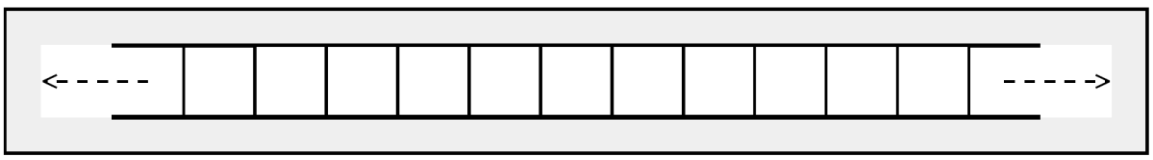


Slika 4: *Vector* tip podatkovne strukture

Ovo je, kao i *array,* uređeni niz sa nasumičnim pristupom čiji iterator je iterator sa nasumičnim pristupom. Pruža dobre performanse, ako se elementi dodaju ili uklanjaju sa kraja niza. U protivnom, kako bi se napravilo mjesta za novi element, sve elemente nakon dodanog treba pomaknuti za jednu poziciju dalje. Svaki *vector* prilikom nastajanja ima određeni kapacitet, a kad se taj početni kapacitet napuni alocira se novi blok memorije i svi elementi se kopiraju na novu lokaciju. Kapacitet se ne povećava za svaki pojedini novi element nego se povećava za neki faktor koji obično iznosi 1.5 do 2 puta kapaciteta prije povećanja. Ovakvom implementacijom se ostvaruje amortizirana konstantna složenost kod dodavanja elemenata u *vector* jer se povećavanje kapaciteta obavi svega nekoliko puta za veliki broj dodanih elemenata.

Deque

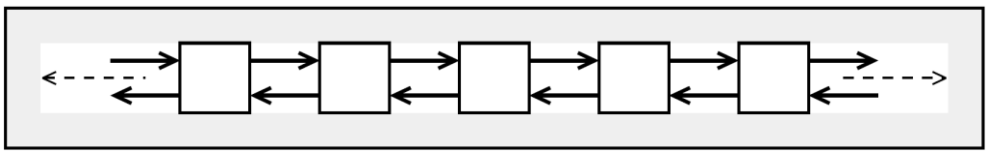
Deque je spremnik sličan *vector*-u. Sadrži elemente u dinamičkom nizu, pruža gotovo identično sučelje kao *vector* i omogućava nasumičan pristup elementima. Razlika je što *deque* može dodavati i uklanjati elemente sa kraja niza, ali i sa početka, kako je prikazano na slici 5.



Slika 5: *Deque* tip podatkovne strukture

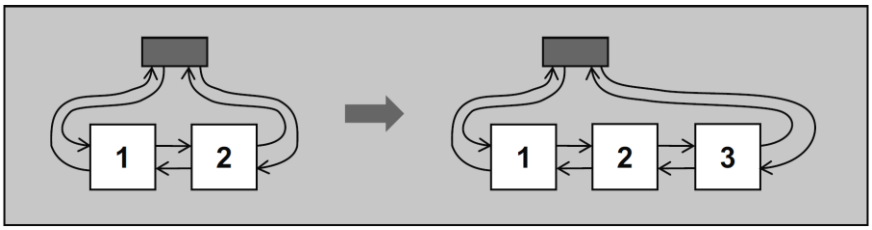
List

Ovaj tip spremnika upravlja elementima preko dvostruko vezanih lista, kako je prikazano na slici 6.



Slika 6: *List* tip podatkovne strukture

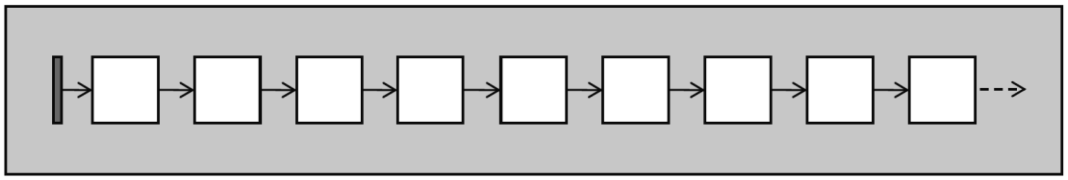
Objekt *list* sadrži dva pokazivača koji pokazuju na prvi i zadnji element u nizu. Svaki element sadrži pokazivače na prethodni i idući element. Kod dodavanja novih elemenata nije potrebno kopirati elemente kao kod *array*-a, već samo promijeniti ove pokazivače kako je prikazano na slici 7.



Slika 7: Dodavanje elementa u spremnik tipa *list*

Spremnik *list* nema mogućnosti nasumičnog pristupa elementima. Da bi se pronašao traženi element potrebno je proći niz od prvog elementa prema zadnjem ili obratno dok se ne pronađe traženi element. Nije moguće koristiti iteratore sa nasumičnim pristupom, već samo bidirekcionalne.

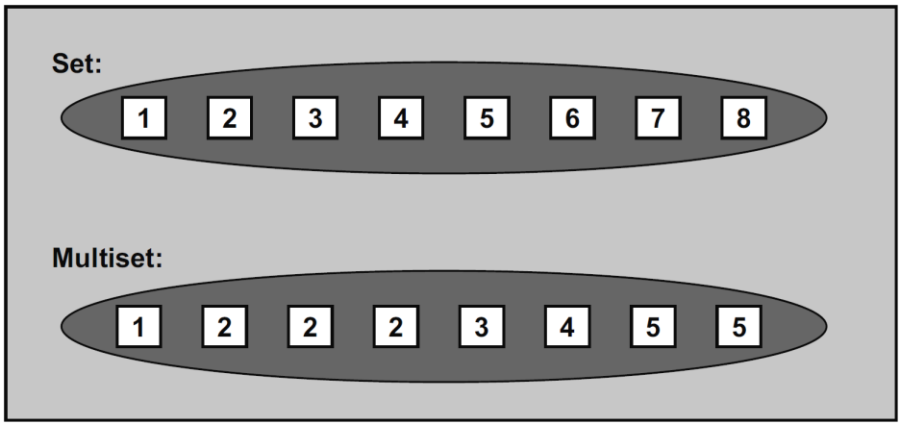
Sa C++11 standardom uvedena je jednosmjerna lista (*forward_list*) koja je ograničena na pomicanje iteratora od prvog elementa prema zadnjem, ali ne i obratno, kako je prikazano na slici 8.



Slika 8: *Forward list* tip spremnika

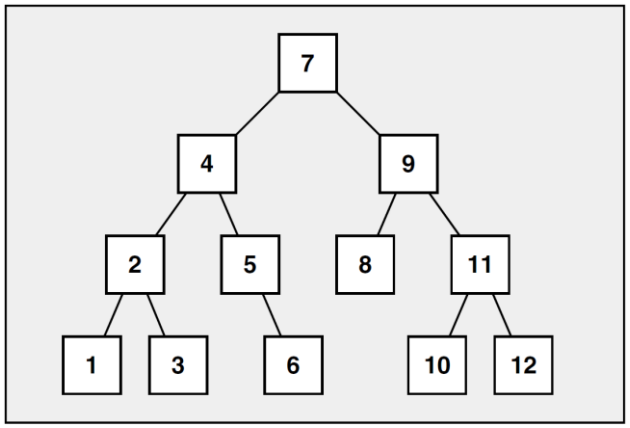
Set i multiset

Ovi spremnici automatski sortiraju elemente prema nekom zadanom kriteriju. Razlika među njima je ta što *set* ne dopušta duplikate među elementima, dok *multiset* dopušta, kako je prikazano na slici 9.



Slika 9: *Set* i *multiset* tipovi spremnika

Kako bi elementi ovih struktura mogli biti sortirani, treba postojati mogućnost njihove usporedbe. Set i multiset su obično implementirani kao balansirana binarna stabla, kako je prikazano na slici 10. Takva implementacija osigurava brzu pretragu koja je reda O(log n). Problem nastaje kod promjene vrijednosti elementa. Kako su elementi sortirani, kad se promijeni vrijednost pojedinog elementa, cijelo stablo treba biti ponovo sortirano.

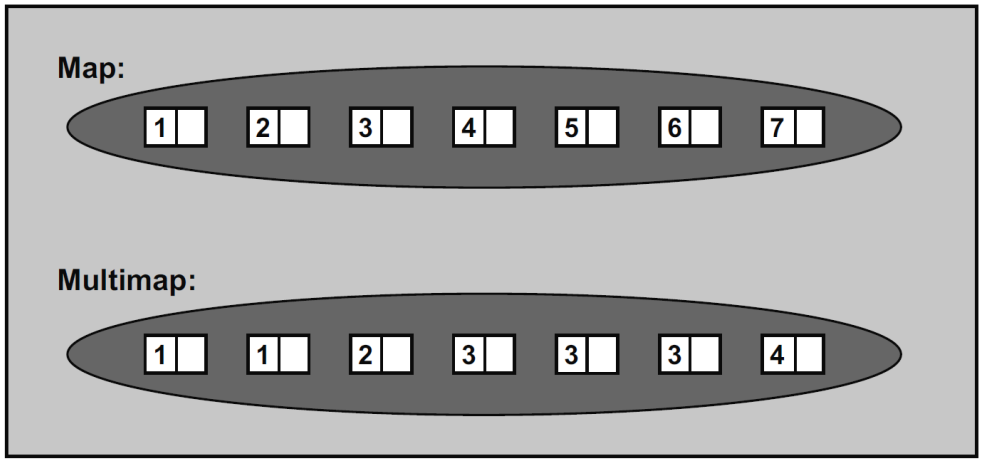


Slika 10: Unutarnji izgled *set* i *multiset* spremnika

Set i *multiset* ne podržavaju iteratore sa nasumičnim pristupom. Nadalje, za iteratore su svi elementi konstante, kako se ne bi poremetila struktura stabla pa se na njima ne mogu koristiti algoritmi koji mijenjaju vrijednost elemenata. Za promjenu vrijednosti mogu se koristiti samo metode koje pružaju sami spremnici.

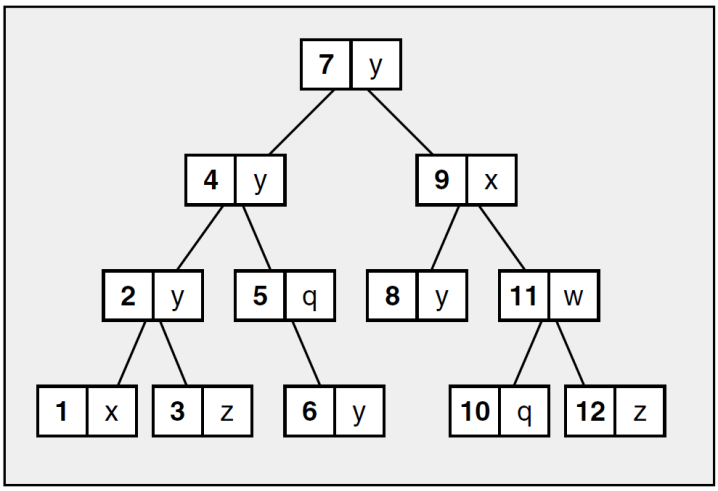
Map i multimap

Ovi spremnici upravljaju elementima po principu ključ/vrijednost. Vrijednosti se automatski sortiraju prema kriteriju korištenom za ključ. Razlika između *map* i *multimap* je ta što *map* ne dopušta duplikate u ključevima dok *multimap* dopušta, kako je prikazano na slici 11.



Slika 11: *Map* i *multimap* tipovi spremnika

I map i multimap su obično implementirani kao balansirana binarna stabla sa ključem kao kriterijem za sortiranje, kako je prikazano na slici 12. Takva implementacija osigurava brzu pretragu po ključu dok je pretraga po vrijednosti spora. Problem nastaje kod promjene ključa. Stari ključ je potrebno ukloniti iz stabla i zatim ubaciti novi. Promjena vrijednosti nije problematična.

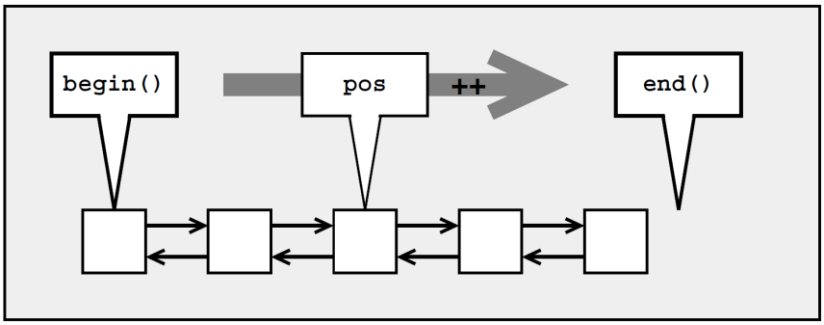


Slika 12: Unutarnji izgled *map* i *multimap* spremnika

Map i *multimap* ne podržavaju iteratore sa nasumičnim pristupom. Za iteratore su ključevi konstante, kao i kod *set* spremnika, kako se ne bi poremetila struktura stabla pa se ne mogu koristiti algoritmi koji mijenjaju vrijednost ključa ili uklanjaju element iz stabla. Za promjenu vrijednosti ključa mogu se koristiti samo javne metode koje pružaju same strukture podataka.

3.2 Iteratori

Iteratori služe za pristupanje pojedinim elementima u spremnicima. Svaki tip spremnika je vezan na svoj tip iteratora koji zna na koji način pristupati njegovim elementima. Iteratori imaju definirano jednostavno sučelje prema korisniku koje je gotovo identično za različite vrste iteratora. Na taj način je omogućeno da isti algoritam radi sa različitim tipovima spremnika. Iterator brine za pristup elementima dok algoritam komunicira samo sa iteratorom preko standardnog sučelja. Slika 13 prikazuje korištenje iteratora za pristup elementima *list* spremnika.



Slika 13: Dohvat svih elemenata *list* spremnika pomoću iteratora

Za pristup rasponu elemenata koriste se dva iteratora i petlja. Iterator ***pos*** se postavlja na početak i petlja se vrti sve dok ***pos*** ne dođe na poziciju nakon zadnjeg elementa. *List* kao i drugi tipovi spremnika sadrži metode ***begin()*** i ***end()*** koje vraćaju iteratore pozicionirane na prvi element i na element nakon zadnjeg. Ovo je primjer C++ koda koji ispisuje vrijednosti svih elemenata spremnika na standardni izlaz:

```
for (auto pos = list.begin(); pos != list.end(); ++pos) {
    std::cout << *pos << std::endl;
}
```

U STL biblioteci postoje pet tipova iteratora a to su *input*, *output*, *forward*, *bidirectional* i *random-access*.

Input

Input iteratori mogu ići samo prema naprijed element po element i čitati vrijednosti. Vrijednost pojedinog elementa mogu pročitati samo jednom. Tipičan primjer je čitanje sa tipkovnice. Nakon pročitane jedne riječi iduće čitanje dohvaća novu riječ.

Output

Output iteratori mogu ići samo prema naprijed element po element sa mogućnošću zapisivanja. Nove vrijednosti zapisuju element po element i mogu proći neki raspon elemenata samo jednom. Tipičan primjer je ispis na ekran. Ako jedan iterator ispiše riječ na ekran te zatim drugi iterator ispiše drugu riječ, ona prva nije prebrisana već je dodana nakon prve.

Forward

Forward iteratori čitaju prema naprijed element po element kao i *input* iteratori, ali uz to pružaju dodatne mogućnosti, kao što je garancija da će dva iteratora koji pokazuju na isti element i za koje operator jednakosti daje logičku istinu nakon pomicanja oba iteratora na idući element oba pokazivati na istu vrijednost.

Bidirectional

Identični su *forward* iteratorima uz dodatnu mogućnost prolaza prema natrag. Mogu čitati i pisati vrijednosti, ako zadovoljavaju i zahtjevima koje ispunjavaju *output* iteratori.

Random-access

Ovi iteratori imaju sve mogućnosti *bidirectional* iteratora uz dodatnu mogućnost nasumičnog pristupa elementima.

3.3 Algoritmi

Algoritmi u STL biblioteci služe za obradu elemenata koje iz spremnika dohvaćaju iteratori. Pošto je sučelje svih iteratora identično, jednom napisani algoritam može obrađivati podatke iz bilo koje vrste spremnika. Svaki od algoritama uzima barem dva iteratora koji određuju početak i kraj niza elemenata u spremniku. Početni iterator dohvaća prvi element i predaje ga algoritmu na obradu, zatim se pomiče na idući element sve dok početni iterator ne pokaže na isti element kao i završni iterator. Ovaj proces se odvija obično u *for*, petlji ali može se koristiti i *for_each* algoritam koji poziva neku operaciju nad svakim elementom spremnika. Osnovnim algoritmima se mogu pridružiti dodatne funkcije bez promjena u kodu algoritma. Na primjer, algoritmu za pretragu može se pridružiti funkcija koja provjerava ispunjava li element neki poseban kriterij.

Često je potrebno prebrojavati sadržaj cijelog spremnika pa se algoritmu prosljeđuju iteratori koji pokazuju na prvi i zadnji element u spremniku. Svi standardni spremnici imaju metode *begin* i *end* koje dohvaćaju upravo ovakve iteratore. Na ovaj način je pojednostavljeno pisanje

programa pa je za prebrojavanje *čitavog* spremnika dovoljno pisati:

```
count(spremnik.begin(), spremnik.end(), vrijednost);
```

Algoritmi se dakle mogu promatrati kao petlje u smislu da ponavljaju neku operaciju određeni broj puta i daju neki rezultat. Postavlja se pitanje zašto koristiti algoritme iz STL biblioteke umjesto da sami pišemo specifičnu petlju za određeni problem? Takve petlje mogu biti razumljivije za čitanje, pogotovo samom autoru programa, ali STL algoritmi imaju važne prednosti kao što je efikasnost. Autori algoritama su vodili računa da pozivi metoda poput *begin* i *end* budu učinkoviti tako da nema nepotrebnih izračuna kod provjera uvjeta petlje. Navedeni autori su također vrlo dobro upoznati sa implementacijama STL spremnika i iteratora. Iz tog razloga su u puno boljoj poziciji optimizirati algoritme i iteratore za rad sa specifičnim implementacijama pojedinih spremnika. Još jedan važan faktor je taj što se u STL biblioteci nalaze vrlo efikasni algoritmi koji zbog svoje kompleksnosti prosječan programer ne bi sam implementirao, već bi izabrao jednostavniji i razumljiviji algoritam sa lošijim performansama. Kod pisanja vlastitih implementacija uvijek postoji mogućnost pogrešaka koje utječu kako na razvoj programa tako i na posao oko njegovog održavanja. STL biblioteka olakšava i održavanje programa jer programeri upoznati sa bibliotekom već po imenu algoritma znaju o čemu se radi, dok je kod čitanja koda drugih programera ponekad teško odrediti što se željelo postići.

Algoritmi u STL biblioteci su podijeljeni na sedam kategorija, prema načinu korištenja. Pojedini algoritmi mogu zadovoljavati uvjete i više kategorija kao što je *for_each* koji može samo čitati ili čitati i pisati pa spada u *modifying* i *nonmodifying* kategorije.

3.3.1 Algoritmi koji ne mijenjaju vrijednost elemenata (*nonmodifying*)

U ovu kategoriju spadaju svi algoritmi koji pretražuju spremnike, prebrojavaju i uspoređuju elemente ili skupove elemenata. Korisni su za rad sa stringovima, na primjer traženje riječi ili znaka u tekstu, brojanje znakova, ali i rad sa brojevima kao što je traženje najvećeg ili najmanjeg broja. Rade sa *input* i *forward* vrstama iteratora i mogu se koristiti sa svim standardnim spremnicima. Tablica 1 daje popis algoritama ove kategorije sa kratkim opisom njihovih funkcija.

Tablica 1: Pregled *nonmodifying* algoritama

Naziv	***Opis***
`for_each()`	Odrađuje neku operaciju na svakom elementu zadanog niza

`count()`	Vraća broj elemenata
`count_if()`	Vraća broj elemenata koji zadovoljavaju postavljeni kriterij
`min_element()`	Vraća iterator koji pokazuje na element koji sadrži najmanju vrijednost
`max_element()`	Vraća iterator koji pokazuje na element koji sadrži najveću vrijednost
`minmax_element()`	Vraća par iteratora koji pokazuju na elemente koji sadrže najmanju i najveću vrijednost
`find()`	Traži prvi element sa zadanom vrijednošću
`find_if()`	Traži prvi element koji zadovoljava postavljeni kriterij
`find_if_not()`	Traži prvi element koji ne zadovoljava postavljeni kriterij
`search_n()`	Traži prvih n uzastopnih elemenata sa odgovarajućim svojstvima
`search()`	Traži prvo pojavljivanje zadanog niza elemenata u nekom nizu
`find_end()`	Traži zadnje pojavljivanje zadanog niza elemenata u nekom nizu
`find_first_of()`	Traži prvo pojavljivanje jednog od više zadanih elemenata
`adjacent_find()`	Traži prva dva uzastopna elementa koja su po nekom zadanom kriteriju jednaka
`equal()`	Provjerava jesu li dva niza elemenata jednaka
`is_permutation()`	Provjerava da li dva neuređena skupa sadrže jednake elemente
`mismatch()`	Vraća prvi element od dva niza koji u oba nije jednak
`lexicographical_compare()`	Provjerava je li jedan niz leksikografski manji od drugog
`is_sorted()`	Provjerava jesu li elementi niza sortirani
`is_sorted_until()`	Vraća prvi element djelomično sortiranog niza koji nije sortiran
`is_partitioned()`	Provjerava jesu li elementi niza odvojeni u dvije grupe prema zadanom kriteriju
`partition_point()`	Vraća element koji niz razdvaja na one elemente koji zadovoljavaju kriterij i one koji ne zadovoljavaju
`is_heap()`	Provjerava je li niz sortiran kao *heap*
`is_heap_until()`	Vraća prvi element niza djelomično sortiranog kao *heap* koji nije sortiran
`all_of()`	Provjerava da li svi elementi zadovoljavaju kriterij
`any_of()`	Provjerava da li barem jedan element zadovoljava kriterij
`none_of()`	Provjerava da li nijedan element ne zadovoljava kriterij

Brojanje elemenata pomoću *count* i *count_if* algoritama

Count i *count_if* broje pojavljivanje nekog elementa u rasponu elemenata određenim s dva iteratora. Rade na način da iterator koji kreće od prvog elementa raspona putuje element po element dok ne dođe do pozicije na koju pokazuje iterator kraja raspona. *Count* algoritam uspoređuje svaki dereferencirani element iz raspona pomoću operatora == sa zadanim elementom. Element na koji pokazuje iterator kraja raspona nije uključen u obradu, što vrijedi za sve algoritme STL biblioteke. U slučaju da su elementi jednaki, interni brojač se uvećava za 1. Na kraju se sadržaj internog brojača prosljeđuje kao povratna vrijednost u cjelobrojnom tipu s predznakom. Dok je *count* algoritam ograničen na usporedbu elemenata operatorom ==, algoritam *count_if* je daleko fleksibilniji. Kod *count_if* algoritma argument je, umjesto elementa za usporedbu, funkcija koja prima jedan element tipa koji se prebrojava, a povratna vrijednost joj je tipa *bool* – funkcija treba vraćati *true* ako se element treba brojati, odnosno *false* ako se ne treba. Na korisniku je da odredi svu logiku koja određuje povratnu vrijednost koja tako može ovisiti o internom stanju elementa, usporedbi s nekim drugim vrijednostima ili uopće ne mora ovisiti o stanju elementa. Na taj način je postignuta još jedna razina apstrakcije jer algoritam više ne treba znati ništa o elementima koje obrađuje zbog toga što je sva logika upakirana u vanjsku funkciju. Takva funkcija se naziva *unary predicate* i može biti napisana kao standardna funkcija s odgovarajućim potpisom ili kao *lambda* funkcija što ima prednost jednostavnijeg zapisa, ali i činjenice da se vjerojatno neće koristiti nigdje drugdje u kodu.

Složenost algoritama za brojanje je linearna, to jest O(n). Algoritam mora obaviti obradu onoliko elemenata koliko ih ima u zadanom rasponu, što znači da broj operacija direktno ovisi o broju elemenata.

Traženje elemenata pomoću *find* i *find_if* algoritama

Find i *find_if* traže prvo pojavljivanje nekog elementa u rasponu između dva iteratora. Kod obje varijante je povratna vrijednost iterator koji pokazuje na pronađeni element, odnosno na kraj niza ako elementa nema. Ovi algoritmi rade na sličan način kao algoritmi za brojanje. Kod *find* se provjerava jednakost elemenata u rasponu sa zadanim elementom i vraća iterator koji pokazuje na prvi jednaki element. Kod *find_if* se elementi u rasponu prosljeđuju funkciji koja svojom internom logikom definiranom od korisnika određuje je li proslijeđeni element taj koji se traži. Takva funkcija prima jedan element tipa traženog i vraća vrijednost tipa *bool* - *true* ako je element taj koji se traži, odnosno *false* ako nije. Algoritam u ovisnosti o povratnoj vrijednosti funkcije završava i vraća iterator na pronađeni element ili nastavlja pretragu. Složenost

algoritama za traženje je O(n) jer kao i kod algoritama za brojanje treba obaviti usporedbu na svakom elementu raspona. Kod asocijativnih i neuređenih podatkovnih struktura je bolje koristiti funkcije za traženje koje pružaju same strukture zbog njihove manje složenosti. Za asocijativne strukture ona iznosi O(log n), a za neuređene čak O(1). Kod sortiranih raspona elemenata je bolje koristiti *lower_bound*, *upper_bound*, *equal_range* ili *binary_search* algoritme koji su efikasniji na sortiranim elementima.

3.3.2 Algoritmi koji mijenjaju vrijednost elemenata (*modifying*)

Ovi algoritmi mijenjaju vrijednosti elemenata, direktno ili prilikom njihovog kopiranja u drugi raspon pri čemu elementi iz odredišnog raspona ostaju nepromijenjeni. Korisni su za obavljanje operacija na cijelom rasponu elemenata, popunjavanju vrijednosti elemenata u rasponu, kopiranje ili premještanje elemenata ili nizova elemenata. Ne mogu se koristiti sa asocijativnim i neuređenim spremnicima jer su njihovi elementi nepromjenjivi.
Tablica 2 daje popis algoritama ove kategorije sa kratkim opisom njihovih funkcija.

Tablica 2: Pregled *modifying* algoritama

Naziv	***Opis***
`for_each()`	Odrađuje neku operaciju na svakom elementu zadanog niza
`copy()`	Kopira niz elemenata
`copy_if()`	Kopira elemente koji zadovoljavaju kriterij
`copy_n()`	Kopira n elemenata
`copy_backward()`	Kopira niz elemenata počevši od zadnjeg
`move()`	Premješta niz elemenata
`move_backward()`	Premješta niz elemenata počevši od zadnjeg
`transform()`	Modificira i kopira elemente, kombinira elemente dva niza
`merge()`	Spaja dva niza u jedan
`swap_ranges()`	Zamjenjuje mjesta elementima iz dva niza
`fill()`	Zamjenjuje elemente sa zadanom vrijednosti
`fill_n()`	Zamjenjuje n elemenata sa zadanom vrijednosti
`generate()`	Zamjenjuje elemente sa rezultatom neke operacije
`generate_n()`	Zamjenjuje n elemenata sa rezultatom neke operacije
`iota()`	Zamjenjuje svaki element sa sekvencom rastućih vrijednosti
`replace()`	Zamjenjuje elemente koji imaju određenu vrijednost sa drugom

	vrijednosti
`replace_if()`	Zamjenjuje elemente koji zadovoljavaju kriterij sa drugom vrijednosti
`replace_copy()`	Zamjenjuje elemente koji imaju određenu vrijednost sa drugom vrijednosti prilikom kopiranja
`replace_copy_if()`	Zamjenjuje elemente koji zadovoljavaju kriterij sa drugom vrijednosti prilikom kopiranja

Obrada raspona elemenata pomoću *for_each* i *transform* algoritama

For_each na svakom elementu raspona primjenjuje operacije definirane u funkciji ili funkcijskom objektu kojeg prima kao argument. Funkcijski objekt je objekt koji ima definiran operator **( )** koji prima jedan element tipa koji se obrađuje. Takav objekt može primijeniti operacije na izvornom objektu i mijenjati svoje stanje. Funkcija može biti napisana i kao *lambda* funkcija. *Transform* obavlja isti posao uz razliku da elemente niza uzima kao kopiju dok ih *for_each* prima kao referencu. *Transform* tako ima prednost da modificirane elemente iz izvornog raspona može spremiti natrag na isto mjesto ili ih može spremiti u neki drugi niz. *For_each* kao argumente prima iteratore na početak i kraj raspona elemenata i funkciju dok *transform* treba još početni iterator na niz gdje će spremati rezultate. Prednost *for_each* algoritma je brzina jer nema kopiranja elemenata kao kod *transform* algoritma.
Povratna vrijednost *transform* algoritma je iterator s pozicijom iza zadnjeg elementa na kojem je primjenjena funkcija. Povratna vrijednost f*or_each* algoritma je funkcijski objekt koji je obavljao operacije na elementima.

Kopiranje raspona elemenata pomoću *copy* i *copy_if* algoritama

Ovi algoritmi kopiraju raspon elemenata između dva *input* iteratora u raspon čiji je početak definiran trećim *output* iteratorom. Ponašanje algoritma je nedefinirano ako izlazni iterator pokazuje na element unutar raspona između dva ulazna iteratora. Razlika je što *copy* kopira sve elemente u rasponu dok *copy_if* kopira samo one za koje funkcija tipa *unary predicate* vrati *true*. Povratna vrijednost je iterator koji pokazuje na prvi element iza posljednjeg kopiranog u odredišni raspon. Na korisniku je da osigura dovoljno prostora za elemente u odredišnom nizu ili da koristi iterator koji dodaje element, poput onoga kojeg vraća funkcija *back_inserter*. Moguće je korištenje standardnih *stream* iteratora pa se tako može izvesti kopiranje unosa sa tipkovnice ili datoteke u spremnik i kopiranje sadržaja spremnika na ekran ili u datoteku.

3.3.3 Algoritmi koji uklanjaju elemente iz niza (r*emoving*)

Ovo je posebna vrsta *modifying* algoritama. Oni ne mogu promijeniti broj elemenata u spremniku već pomiču elemente koji se neće uklanjati na mjesto elementa koji je uklonjen. Na kraju vraćaju iterator koji pokazuje na prvi element iza zadnjeg elementa novog raspona. Za brisanje elemenata iz spremnika potrebno je pozvati metodu spremnika *erase* sa ovim parametrima:

- iterator kojeg je vratio neki od *removing* algoritama, a koji označava početak raspona za brisanje i
- iterator koji označava kraj spremnika

Ne mogu se koristiti sa asocijativnim i neuređenim podatkovnim strukturama čiji elementi se smatraju konstantama.
Tablica 3 daje popis algoritama ove kategorije sa kratkim opisom njihovih funkcija.

Tablica 3: Pregled *removing* algoritama

Naziv	***Opis***
`remove()`	Uklanja element sa zadanom vrijednosti
`remove_if()`	Uklanja element koji zadovoljava zadani kriterij
`remove_copy()`	Kopira i uklanja elemente koji ne odgovaraju zadanoj vrijednosti
`remove_copy_if()`	Kopira i uklanja elemente koji ne zadovoljavaju zadani kriterij
`unique()`	Uklanja uzastopne duplikate
`unique_copy()`	Kopira i uklanja uzastopne duplikate

Uklanjanje elemenata pomoću *remove* i *remove_if* algoritama

Remove uklanja svaki element u rasponu između dva iteratora koji je jednak zadanoj vrijednosti. Jednakost se provjerava operatorom `==`. Parametri su mu iteratori koji pokazuju na početak i kraj raspona i *const* referenca na traženi element. *Remove_if* umjesto traženog elementa, kao parametar prima funkciju tipa *unary predicate* i uklanja one elemente za koje ta funkcija vrati *true*. Redoslijed elemenata koji nisu uklonjeni ostaje stabilan. *Remove_if* obično radi kopiju funkcijskog objekta tijekom izvođenja algoritma. To znači da rezultat operacije funkcijskog objekta ne smije ovisiti, na primjer, o tome koliko puta je taj funkcijski objekt pozvan jer je moguće da *remove_if* tijekom izvođenja napravi novu kopiju funkcijskog objekta kojem je brojač poziva postavljen na inicijalno stanje. U takvom slučaju rezultat rada *remove_if*

algoritma može biti neispravan. Preporuka je da funkcijski objekt za rad sa *remove_if* algoritmom ne pamti interna stanja. Složenost ovih algoritama je linearna.

3.3.4 Algoritmi za mijenjanje redoslijeda elemenata *(mutating)*

Ovi algoritmi služe za mijenjanje redoslijeda elemenata, ali ne i njihove vrijednosti. Ne mogu se koristiti sa asocijativnim spremnicima jer oni automatski uređuju svoj raspored elemenata ni sa neuređenim spremnicima jer oni nemaju uređen redoslijed elemenata.
Tablica 4 daje popis algoritama ove kategorije sa kratkim opisom njihovih funkcija.

Tablica 4: Pregled *mutating* algoritama

Naziv	***Opis***
`reverse()`	Okreće redoslijed elemenata
`reverse_copy()`	Kopira elemente prilikom okretanja redoslijeda
`rotate()`	Rotira redoslijed elemenata tako pomiče elemente na jednu stranu određeni broj pozicija pri čemu one koji su pomaknuti izvan niza na jednoj strani stavlja na natrag u niz na drugoj strani
`rotate_copy()`	Kopira elemente prilikom rotiranja redoslijeda
`next_permutation()`	Radi permutaciju redoslijeda elemenata
`prev_permutation()`	Radi permutaciju redoslijeda elemenata
`shuffle()`	Miješa redoslijed elemenata u slučajni poredak
`random_shuffle()`	Miješa redoslijed elemenata u slučajni poredak
`partition()`	Mijenja redoslijed elemenata tako da one koji zadovoljavaju zadani kriterij stavlja na početak
`stable_partition()`	Isto kao `partition()` ali uz zadržavanje relativnog odnosa između elemenata
`partition_copy()`	Kopira elemente prilikom kopiranja tako da su oni koji ispunjavaju kriterij naprijed

Razdvajanje skupa elemenata na dva podskupa algoritmom *partition*

Ovaj algoritam pomiče na početak raspona one elemente za koje funkcija tipa *unary predicate*, koju prima kao argument, vrati *true*. Dolazi u dvije varijante, *partition* i *stable_partition*. Razlika je u tome što *partition* ne čuva relativan raspored između elemenata koji zadovoljavaju kriterij dok ga *stable_partition* čuva. Povratna vrijednost je iterator koji pokazuje na prvi element koji nije zadovoljio kriterij. Složenost je za *partition* linearna, a za *stable_partition* ovisi ima li dovoljno alocirane slobodne memorije. Ukoliko ima složenost je linearna, a u

protivnom iznosi O(*n log n)*.

3.3.5 Algoritmi za sortiranje (*sorting)*

Algoritmi za sortiranje su posebna vrsta *mutating* algoritama jer mijenjaju raspored elemenata. Smješteni su u posebnu kategoriju jer su kompleksniji od jednostavnih *mutating* algoritama. Ovi algoritmi zahtijevaju da podatkovna struktura podržava iterator sa nasumičnim pristupom pa se ne mogu koristiti sa neuređenim i asocijativnim spremnicima.
Tablica 5 daje popis algoritama ove kategorije sa kratkim opisom njihovih funkcija.

Tablica 5: Pregled *sorting* algoritama

Naziv	***Opis***
`sort()`	Sortira elemente
`stable_sort()`	Sortira uz čuvanje redoslijeda jednakih elemenata
`partial_sort()`	Sortira do prvih n elemenata
`partial_sort_copy()`	Kopira elemente u sortirani niz
`nth_element()`	Postavlja vrijednost elementa na poziciji n na onu vrijednost koju bi imao da je niz sortiran
`partition()`	Mijenja redoslijed elemenata tako da one koji zadovoljavaju zadani kriterij stavlja na početak
`stable_partition()`	Isto kao `partition()` ali uz zadržavanje relativnog odnosa između elemenata koji ispunjavaju kriterij i onih koji ne ispunjavaju
`partition_copy()`	Kopira elemente prilikom kopiranja tako da su oni koji ispunjavaju kriterij naprijed
`make_heap()`	Pretvara niz u hrpu (heap)
`push_heap()`	Dodaje element na hrpu
`pop_heap()`	Uklanja element sa hrpe
`sort_heap()`	Sortira hrpu koja nakon toga prestaje biti hrpa

Sortiranje elemenata *sort* i *stable_sort* algoritmima

Ovi algoritmi sortiraju elemente raspona definiranog s dva iteratora. Moguće je sortiranje pomoću operatora < definiranog za tip elementa koji se sortira ili pomoću funkcije tipa *binary predicate*. Takva funkcija prima dva elementa tipa koji se sortira, a kao povratnu vrijednost vraća *bool* - *true* ako prvi argument treba biti ispred drugog. Ova varijanta je korisna ako elementi trebaju biti sortirani po drugačijem kriteriju od onog definiranog operatorom <. Da bi

se neka funkcija mogla koristiti kao *binary predicate* mora zadovoljiti *strict weak ordering*, što znači da mora zadovoljiti ova četiri kriterija:

1. mora biti asimetrična (ako je $a < b$, onda je $b < a$),
2. mora biti tranzitivna (ako je $a < b$ i $b < c$ onda je i $a < c$),
3. mora biti irefleksivna ($a < a$ uvijek mora biti *false*) i
4. mora zadovoljiti tranzitivnost jednakosti (ako je $a == b$ i $b == c$ tada je i $a == c$).

Razlika između *sort* i *stable_sort* je u tome što *stable_sort* garantira da će relativan raspored jednakih elemenata ostati nepromijenjen dok *sort* isto ne garantira. Složenost u prosjeku iznosi O(n log n) za *sort* dok kod *stable_sort* ovisi ima li dovoljno slobodne alocirane memorije. U slučaju da nema, složenost raste na $O(n \log^2 n)$.

3.3.6 Algoritmi za korištenje na sortiranim nizovima *(sorted-range)*

Ovo su algoritmi koji zahtijevaju da su elementi na kojem će raditi sortirani po njihovom vlastitom kriteriju. Algoritmi za sortirane elemente imaju logaritamsku složenost, dok slični algoritmi koji rade s nesortiranim elementima imaju linearnu složenost. Prema standardu, korištenje ovih algoritama na nesortiranim elementima rezultira nedefiniranim ponašanjem, ali većina implementacija podržava i takav način rada. U pravilu se koriste sa iteratorima sa nasumičnim pristupom. Mogu se koristiti i sa drugačijim tipovima iteratora, ali tada njihova složenost raste i postaje linearna. Asocijativni i neuređeni spremnici imaju vlastite implementacije nekih algoritama u ovoj skupini tako da je bolje koristiti njihovu implementaciju zbog boljih performansi.
Tablica 6 daje popis algoritama ove kategorije sa kratkim opisom njihovih funkcija.

Tablica 6: Pregled *sorted-range* algoritama

Naziv	***Opis***
`binary_search()`	Provjerava sadržava li niz zadani element
`includes()`	Provjerava da li je svaki element niza također i element drugog niza
`lower_bound()`	Traži prvi element koji je veći ili jednak zadanoj vrijednosti
`upper_bound()`	Traži prvi element koji je veći od zadane vrijednosti
`equal_range()`	Vraća niz elemenata jednakih zadanoj vrijednosti
`merge()`	Spaja elemente iz dva niza u jedan
`set_union()`	Stvara sortiranu uniju dva niza

`set_intersection()`	Stvara sortirani presjek dva niza
`set_difference()`	Stvara sortirani niz elementa iz elemenata prvog niza a koji nisu prisutni u drugom nizu
`set_symmetric_difference()`	Stvara sortirani niz elementa koji sadrži sve elemente koji se nalaze u točno jednom od dva niza
`inplace_merge()`	Spaja dva uzastopna sortirana niza u jedan
`partition_point()`	Vraća iterator koji niz razdvaja na one elemente koji zadovoljavaju kriterij i one koji ne zadovoljavaju

Traženje prve ili zadnje moguće pozicije elementa pomoću *lower_bound* i *upper_bound* algoritama

Lower_bound traži prvi element, u rasponu elemenata definiranim s dva iteratora, koji je veći ili jednak zadanom elementu dok *upper_bound* traži prvi element koji je veći od zadanog. Drugim riječima, *lower_bound* traži prvu poziciju u rasponu sortiranih elemenata gdje se zadani element može ubaciti bez da se poremeti sortiranost raspona, a *upper_bound* traži zadnju takvu poziciju. Povratna vrijednost im je iterator koji pokazuje na odgovarajuću poziciju u rasponu ili koji pokazuje na kraj raspona, ako takva pozicija ne postoji. Onaj tko poziva ove algoritme mora osigurati da je raspon elemenata adekvatno sortiran. Kriterij traženja može biti element za usporedbu ili funkcija tipa *binary predicate*. Složenost je logaritamska za iteratore sa nasumičnim pristupom a linearna za druge vrste iteratora.

3.3.7 Numerički algoritmi *(numeric)*

Numerički algoritmi su namijenjeni za obradu numeričkih elemenata, ali se mogu koristi i za druge tipove podataka.

Tablica 7 daje popis algoritama ove kategorije sa kratkim opisom njihovih funkcija.

Tablica 7: Pregled *numeric* algoritama

Naziv	***Opis***
`accumulate()`	Kombinira sve vrijednosti elemenata (računa sumu, produkt i slično)
`inner_product()`	Kombinira sve elemente dva niza
`adjacent_difference()`	Kombinira svaki element sa njemu prethodnim
`partial_sum()`	Kombinira svaki element sa svima prethodnima

Računanje rezultata obrade na cijelom rasponu elemenata pomoću *accumulate* algoritma

Accumulate obavlja neku operaciju nad svakim elementom definiranog raspona i pri tome sprema međurezultat. Početna vrijednost međurezultata je definirana parametrom algoritma. Algoritam ima dvije varijante. Prva prima raspon elemenata i početno stanje međurezultata, a vraća zbroj međurezultata sa svim vrijednostima elemenata u rasponu. Druga varijanta prima i funkciju koja će se koristiti umjesto zbrajanja kao kod prve varijante. Takva funkcija kao argumente prima međurezultat i jedan element iz raspona, obavlja neke operacije definirane od korisnika i kao povratnu vrijednost vraća novo stanje međurezultata. Ovo se ponavlja za svaki element raspona gdje se svaki element obrađuje sa rezultatom prethodnog. Iz toga se može zaključiti da je ovaj algoritam zapravo rekurzivan. Druga varijanta je daleko fleksibilnija jer pruža samo kostur pozivanja funkcije na elementima dok samu implementaciju ostavlja na volju korisnika.

4 PRAKTIČNI RAD-PROGRAM ZA FILTRIRANJE I ISPIS PODATAKA DRŽAVA

Praktični rad je aplikacija čiji je zadatak filtriranje i prikaz podataka zemalja svijeta kao što su ime, broj stanovnika, površina i slično. Podaci su spremljeni u CSV datoteci a program ih čita i sprema u strukture *Country* za svaku zemlju. Prvi red datoteke sadrži nazive podataka koje program sprema u spremnik *vector<string> keys* i koju kasnije koristi prilikom ispisa podataka. Struktura *Country* se sastoji od imena zemlje tipa *string*, reference na spremnik *keys* i niza podataka spremljenih u *vector< Field> values.* Struktura *Field* sadrži vrijednost tipa *double* i odgovarajuće operatore kako bi omogućila pravilno izvođenje algoritama u slučaju da CSV datoteka ne sadrži sve podatke o nekoj zemlji. Podaci o zemljama u strukturama *Country* su spremljeni u spremnik *vector<Country> countries*. Za filtriranje i ispis podataka zadužen je objekt *DataProcessor* koji kao argumente konstruktora prima referencu na spremnik *keys*, referencu na spremnik *countries* i referencu na *output stream* koji će koristiti za ispis podataka. *DataProcessor* koristi STL algoritme za filtriranje i sortiranje podataka sadržane u funkcijama opisanim u nastavku ovog poglavlja.

Funkcija *getIndexes* koristi algoritam *find* za traženje ključeva tipa *string* pohranjenih u spremniku *keys* tipa *vector*. Navedena funkcija kao argument prima *vector string*-ova sa ključevima koje treba pronaći, a vraća *vector int*-ova sa indeksima pronađenih ključeva u spremniku *keys*. Objekt tipa *DataProcessor* sadrži referencu na spremnik *keys* kao privatni član. Algoritam *find* ovdje kao argumente prima iteratore koji pokazuju početak i kraj spremnika sa traženim ključevima i ključ tipa *string* koji se nalazi na poziciji brojača petlje u spremniku *keys*. Ako je algoritam vratio iterator koji ne pokazuje na kraj spremnika, to znači da je traženi ključ pronađen i njegov indeks se dodaje u *vector*, koji će se vratiti kao rezultat funkcije.

```
vector<int> DataProcessor::getIndexes(const vector<string> &selectedKeys) {
    vector<int> indexes;  vector<string> validKeys;
    for(auto i = selectedKeys.cbegin(); i != selectedKeys.cend(); ++i) {
        string validKey = getValidKey(*i);
        if(!validKey.empty())
            validKeys.push_back(validKey);
    }
    for(unsigned i = 1; i < keys.size(); ++i) {
        if(find(validKeys.cbegin(), validKeys.cend(), keys.at(i)) !=
        validKeys.cend())
            indexes.push_back(i - 1);
    }
```

```
    return indexes;
}
```

Funkcija *getValidKey* prima dio ključa kao tip *string*, provjerava postoji li neki ključ u spremniku *keys* kojem je taj dio ključa podskup i ako postoji, vraća taj ključ kao povratnu vrijednost funkcije. Ova funkcija koristi dva standardna algoritma, *transform* i *find_if*. Algoritam *transform* je korišten kako bi svaki znak *string*-a zamijenio sa malim slovom. Radi tako da primi dva iteratora na početak i na kraj *string*-a i treći koji označava početak spremnika gdje će se spremati rezultati. U ovom slučaju treći iterator je jednak prvom koji pokazuje na početak *string*-a, što znači da će se znakovi prebrisati sa novom vrijednosti. Zadnji argument je funkcija *tolower* koja prima jedan znak i vraća njegovu nižu vrijednost. Algoritam *find_if* je korišten kako bi pronašao *string* koji sadrži odgovarajući *substring*. Za to je napisana *lambda* funkcija koja prima traženi *substring* kao argument i poziva funkciju *string.find* koja vraća poziciju traženog *substring*-a u *string*-u ili *npos*, ako isti nije pronađen.

```
string DataProcessor::getValidKey(const string &partialKey) {
    if (partialKey.empty())
        return "";
    string lowerKey(partialKey);
    transform(lowerKey.begin(), lowerKey.end(), lowerKey.begin(), ::tolower);
    auto result = find_if(keys.cbegin(), keys.cend(),
        [&lowerKey](string validKey) {
        transform(validKey.begin(), validKey.end(), validKey.begin(), ::tolower);
        return validKey.find(lowerKey) != string::npos;
    });
    return result != keys.end() ? *result : "";
}
```

Funkcija *average* izračunava prosječnu vrijednost jednog atributa niza objekata tipa *Country*, ali samo za objekte koji sadrže definiranu vrijednost (npr. prosjek broja stanovnika određenih zemalja). Objekt tipa *DataProcessor* čuva referencu na navedeni spremnik kao privatni član zajedno sa dva iteratora koja omeđuju raspon podataka koje obrađuju algoritmi nazvana *selectionFirst* i *selectionLast*. Algoritam *count_if* služi za prebrojavanje samo onih objekata tipa *Country* koje za zadani atribut sadrže definiranu vrijednost. Funkcija tipa *unary predicate*, koja određuje koji objekti će se brojati, je napisana kao *lambda* funkcija koja za svaki objekt tipa *Country* provjerava je li podatak na zadanom indeksu valjan pomoću funkcije *isUndefined*, koja je član objekta *Field*, koji sadrži podatak. Algoritam *accumulate* ovdje služi za zbrajanje svih vrijednosti niza objekata tipa *Country* koje se nalaze na zadanom indeksu. Operator zbrajanja za tip *Field* je definiran na način da je zbroj definirane vrijednosti sa nedefiniranom

uvijek samo ona definirana vrijednost. Povratna vrijednost funkcije *average* je zbroj definiranih vrijednosti koje je vratio algoritam *accumulate*, podijeljen sa brojem objekata sa definiranim vrijednostima koje je vratio algoritam *count_if*.

```
double DataProcessor::average(const string &column) {
    int index;
    try {
        index = getIndex(column);
    }
    catch (const invalid_argument &e) {
        output << e.what() << endl;
        return numeric_limits<double>::quiet_NaN();
    }
    int countriesWithValues = count_if(selectionFirst, selectionLast,
        [index](const vsite::Country &country) {
        return !country.values.at(index).isUndefined();
        });
    if (countriesWithValues == 0) {
        output << "Selection contains no values." << endl;
        return numeric_limits<double>::quiet_NaN();
    }
    vsite::Field sum = accumulate(selectionFirst, selectionLast,
                          vsite::Field(0.0),
        [index](vsite::Field left, vsite::Country right) {
            return left + right.values.at(index);
        });
    return sum / countriesWithValues;
}
```

Funkcija *byColumn* koristi algoritam *partition* za razdvajanje objekata tipa *Country* između iteratora *first* i *last* na one čija je vrijednost, zadana argumentom indeks, veća ili jednaka vrijednosti *min* i manja ili jednaka vrijednosti *max* i na one čija vrijednost izlazi iz navedenog raspona. Algoritam *partition* razdvaja objekte i vraća poziciju prvog objekta koji ne zadovoljava uvjet i zatim ta pozicija postaje novi kraj raspona, to jest, između iteratora *first* i *last* se nalaze samo objekti koji zadovoljavaju uvjet.

```
void DataProcessor::byColumn(
       const int &index, const double &min, const double &max,
       vector<vsite::Country>::iterator &first, vector<vsite::Country>::iterator
&last) {
    last = partition(first, last,
        [&index, &min, &max](const vsite::Country &country) {
        return country.values.at(index) >= min && country.values.at(index) <= max;
    }); }
```

Funkcija *sortByName* koristi algoritam *sort* za sortiranje objekata tipa *Country* prema kriteriju imena koje je sadržano u članu *name* tipa *string*. Algoritam za usporedbu koristi funkciju tipa *binary predicate* koja je napisana kao *lambda* funkcija. Navedena funkcija koristi operator usporedbe definiran u tipu *string* te se tako zadani objekti tipa *Country* sortiraju abecedno prema imenu.

```
void DataProcessor::sortByName(vector<vsite::Country>::iterator &first,
                               vector<vsite::Country>::iterator &last) {
    sort(first, last,
        [](const vsite::Country &left, const vsite::Country &right) {
        return left.name < right.name;
    });
}
```

Funkcija *inputString* služi za čitanje niza *string*-ova sa standardnog ulaza. Prvo je pomoću funkcije *getline* dohvaćen cijeli niz znakova sa konzole sve do pritiska tipke *enter*. Zatim je kreiran objekt tipa *istringstream* koji će raditi nad dohvaćenim znakovima i kojem su pomoću objekta tipa *csvCtype* kao delimiteri definirani znakovi zareza (,) i točke-zareza (;). Ostatak posla obavlja algoritam *copy* na način da kao argumente primi *istream_iterator* koji pokazuje na početak *string*-a, *istream_iterator* sa praznim konstruktorom koji označava kraj *string*-a i funkciju koja dodaje nove *string*-ove na kraj spremnika tipa *vector string*. Algoritam *copy* polazi od početka ulaznog *string*-a i kopira znakove sve dok ne dođe do znaka koji je definiran kao delimiter, u ovom slučaju to je zarez ili točka-zarez. Nakon toga algoritam *copy* taj novi *string* predaje funkciji koja je proslijeđena kao argument, što je u ovom slučaju funkcija *back_inserter*. Algoritam *copy* zatim nastavlja ponavljati navedene operacije sve dok iterator ne stigne do kraja ulaznog *string*-a.

```
vector<string> inputStrings() {
    string line;
    getline(cin, line);
    istringstream iss(line);
    locale L(locale::classic(), new csvCtype);
    iss.imbue(L);
    vector<string> strings;
    copy(istream_iterator<string>(iss),
             istream_iterator<string>(),
             back_inserter(strings));
    return strings;
}
```

5 ZAKLJUČAK

Najviša razina apstrakcije koju programski jezik C++ omogućava je razina generičkog programiranja. Generički stil programiranja zajedno sa STL dijelom standardne biblioteke omogućuje programeru da razmišlja na razini koncepata umjesto konkretnih objekata, ali i da kod koji piše bude kompaktniji i bez suvišnih ponavljanja.

Kroz ovaj rad je predstavljeno generičko programiranje pomoću STL dijela standardne biblioteke programskog jezika C++. Ovakav stil je samo jedan od mogućih koje jezik C++ podržava sa svim svojim prednostima i manama. Generičko programiranje je posebno korisno u sustavima koji se mogu lako opisati pomoću algoritama i tipova podataka, dok je objektno - orijentirano programiranje pogodno za opisivanje problemske domene i njeno rješavanje pomoću objektno-orijentiranih mehanizama. Istodobno korištenje generičkog i objektno - orijentiranog stila je teško izvedivo jer svaki pristup funkcionira na zasebnoj razini apstrakcije. Bolji pristup je stoga primijeniti odgovarajući stil programiranja za rješavanje konkretnog problema. C++ je moćan alat koji programeru pruža mogućnosti za različite stilove pa je time pogodan za rješavanje različitih tipova problema. Na programeru je da nauči koristiti jezik i njegove mogućnosti kako bi mogao prepoznati koje komponente programskog jezika C++ upotrijebiti u svakoj situaciji.

Praktični rad daje uvid u mogućnosti i način korištenja STL algoritama. U samom programu je korišteno tek nekoliko algoritama iz kojih je vidljiva jednostavnost upotrebe i moć STL algoritama. Način upotrebe pojedinih STL algoritama varira tek u detaljima tako da je programeru vrlo jednostavno naviknuti se na njihovu upotrebu. STL biblioteka sadrži širok raspon tipova algoritama koje je moguće dodatno proširiti vlastitim funkcijama a da biblioteka nije previše opširna i nepraktična za upotrebu.

Za filtriranje podataka u programu je korišten *partition* algoritam pomoću kojeg su traženi podaci grupirani na početak spremnika kako bi im se moglo jednostavno pristupati pomoću iteratora. Različiti filteri koriste isti *partition* algoritam a različita logika filtriranja je napisana u pridruženim *predicate* funkcijama. Na ovakav način je pomoću standardnih sučelja algoritama i malo dodatnog koda ostvarena velika razlika u funkcionalnosti a ostvarena je jednostavnost i čitljivost koda.

U drugom dijelu programa je korišten *copy* algoritam prilikom čitanja vrijednosti iz datoteke. *Input string stream* objekt, kreiran iz jedne linije datoteke koja sadrži podatke jedne zemlje, modificiran je tako da se za odvajanje podataka koristi zarez umjesto standardnog razmaka. Na ovaj način, zbog toga što su podaci spremljeni u datoteci sa vrijednostima odvojenim zarezom,

copy algoritam automatski odvaja pojedine vrijednosti iz datoteke, sprema ih u strukturu *Field* pomoću operatora >> definiranim u navedenoj strukturi i zatim navedenu strukturu sprema u spremnik *vector<Field> values* koji se kasnije pridružuje strukturi *Country*. Kod kojim se ostvaruje sve navedeno se sastoji od svega nekoliko linija koda što ponovo pokazuje jednostavnost korištenja STL algoritama i čitljivost napisanog koda.

LITERATURA

[1] Nicolai M. Josuttis: The C++ Standard Library, A Tutorial and Reference, 2nd Edition

[2] Bjarne Stroustrup: The C++ Programming Language, 4th Edition

[3] https://en.wikipedia.org/wiki/Algorithm (pristupljeno 27.06.2017.)

[4] https://en.wikipedia.org/wiki/Big_O_notation (pristupljeno 27.06.2017.)

[5] https://en.wikipedia.org/wiki/Generic_programming (pristupljeno 27.06.2017.)

[6] http://www.cplusplus.com (pristupljeno 03.07.2017.)

[7] http://en.cppreference.com (pristupljeno 03.07.2017.)

[8] http://www.drdobbs.com/stl-algorithms-vs-hand-written-loops/184401446 (pristupljeno 07.07.2017.)

SAŽETAK

Tema proučavanja ovog rada je STL dio standardne biblioteke C++ programskog jezika koji uključuje spremnike podataka, iteratore i algoritme.

U uvodnom dijelu ovog rada napravljen je pregled sastavnih dijelova STL dijela biblioteke koji su detaljnije obrađeni u narednim poglavljima.

Drugo poglavlje daje definiciju algoritma i opisuje pojam generičkog programiranja. Opisano je što se pod pojmom algoritma podrazumijeva u matematici i računarstvu te kako se uspoređuju performanse algoritama pomoću *Big O* notacije. Dalje u poglavlju se objašnjava stil generičkog programiranja i opisuje se ostvarivanje takvog stila programiranja u C++ programskom jeziku uz pomoć STL dijela standardne biblioteke.

Treće poglavlje opisuje standardne spremnike podataka, iteratore i algoritme koji su dio C++ jezika i biblioteke. U prvom dijelu navedenog poglavlja su opisani standardni spremnici podataka sa specifičnim prednostima i manama svakog tipa spremnika. Potom slijedi objašnjenje korištenja iteratora za povezivanje spremnika koji sadrže podatke sa algoritmima koji obavljaju operacije nad tim podacima i pregled standardnih iteratora u C++ jeziku. Zadnji dio ovog poglavlja je posvećen standardnim algoritmima u STL dijelu biblioteke C++ jezika. Opisan je općeniti način rada standardnih algoritama. Zatim je napravljen pregled svih standardnih algoritama po kategorijama uz opis korištenja pojedinih algoritama u specifične svrhe kao što su brojanje, traženje, kopiranje, brisanje, sortiranje i obavljanje operacija na nizovima elemenata.

U četvrtom dijelu se nalaze dijelovi programa koji opisuju mogućnosti STL algoritama u C++ jeziku.

Na kraju rada je zaključak u kojem se stil generičkog programiranja podržan C++ jezikom i STL dijelom standardne biblioteke uspoređuje sa drugim programskim stilovima koje omogućuje C++ programski jezik.

Ključne riječi: C++, STL biblioteka, STL spremnici podataka, STL iteratori, STL algoritmi, generičko programiranje, Big O notacija

SUMMARY

This paper is based on the STL part of the C++ programming language standard library that includes containers, iterators, and algorithms.

The introductory part of this paper consists of an overview of the constituent parts of the STL part of the standard library, which will be further elaborated in the following chapters.

The second chapter gives the definition of an algorithm and describes the concept of generic programming. It describes the concept of an algorithm in mathematics and computing and how similar algorithms are compared based on their performance using the Big O notation. The further part of the chapter explains the style of generic programming and describes it's realization in the C++ programming language with the help of the STL part of the standard library.

Chapter 3 describes standard containers, iterators, and algorithms that are a part of the C++ STL library. The first part of this chapter describes the standard containers, with the specific advantages and disadvantages of each type of container included. Subsequently follows an explanation on using iterators to connect containers containing data with algorithms performing operations and review of standard iterators in the C++ language. The last part of this chapter is dedicated to standard algorithms in the STL section of the C++ language library. First is described a general way of operation of standard algorithms, and then follows an overview of all standard algorithms by category, together with a description of the use of specific algorithms for specific purposes such as counting, searching, copying, deleting, sorting and performing operations on ranges of elements.

In the fourth part there are extracts of the program that demonstrates the features of STL algorithms in C++ language.

At the end of the paper comes a conclusion which compares the style of generic programming supported by C++ language and STL library to other programming styles that C++ programming language supports.

Keywords: C++, STL library, STL containers, STL iterators, STL algorithms, generic programming, Big O notation