Projekat iz Verifikacije softvera

Jelena Mitrović, 1019/2024
Matematicki fakultet, Univerzitet u Beogradu

December 2024

Sadržaj

1 Uvod

U okviru predmeta Verifikacije softvera, analizirala sam projekat pod nazivom *ECE350Final*, koji je dostupan na adresi `https://github.com/loganizer405/ECE350Final`. Ovaj projekat implementira igru *Durak* u programskom jeziku C++. Igra *Durak* je popularna kartaška igra koja se igra sa špilom od 36 karata, gde cilj igrača nije da postane ,,durak"(najgori igrač), odnosno da ostane poslednji sa kartama u ruci. Igrači se smenjuju u napadu i odbrani, pri čemu napadač stavlja kartu na stolu, a odbrambeni igrač mora odgovoriti odgovarajućom kartom. Ukoliko odbrambeni igrač ne može da odbrani, karta prelazi u ruke napadača. Igra se nastavlja dok neki igrač ne ostane bez karata. Za ovaj program, napisala sam jedinicne testove, primenila alate poput *Valgrind*-a i *Cppcheck*-a, kako bih analizirala performanse programa i osigurala njegovu ispravnost. Korišćenjem ovih alata, analizirala sam memorijsku efikasnost i pronalazila moguće greške u kodu, čime sam doprinela optimizaciji i kvalitetu aplikacije.

2 Jedinični testovi

Jedinični testovi predstavljaju osnovnu tehniku testiranja softvera koja se fokusira na proveru ispravnosti pojedinačnih delova koda, obično funkcija ili metoda. Cilj je izolovati najmanje funkcionalne jedinice sistema i potvrditi da svaka od njih funkcioniše ispravno u različitim scenarijima. Ovakvi testovi omogućavaju rano otkrivanje grešaka, olakšavaju održavanje koda i povećavaju poverenje u stabilnost softverskog sistema.

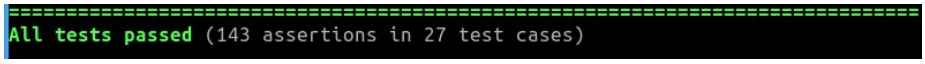
2.1 Catch 2

Catch2 (C++ Automated Test Cases in Headers) je moderna biblioteka za jedinično testiranje u programskom jeziku C++. Namenjena je pisanju čitljivih i konciznih testova sa minimalnim konfiguracijama. Catch2 podržava jednostavno definisanje testova pomoću makroa i nudi intuitivne funkcionalnosti, kao što su grupisanje testova, opisivanje scenarija i lako ispisivanje rezultata testiranja. Zbog svoje jednostavnosti i fleksibilnosti, Catch2 je široko prihvaćena biblioteka u C++ zajednici za jedinično testiranje, omogućavajući programerima brzo uključivanje testova u postojeće projekte. Kombinacija upotrebe jediničnih testova i alata kao što je Catch2 osigurava kvalitet koda i čini razvojni proces pouzdanijim i efikasnijim.

2.2 Analiza rezultata

Testovi su pokrenuti korišćenjem bash skripte *run_tests.sh*, koja se nalazi u direktorijumu *unit_tests*.Za praćenje pokrivenosti koda korišćen je alat ***lcov***, koji omogućava vizuelizaciju i analizu pokrivenosti testovima, pružajući uvid u delove koda koji nisu obuhvaćeni testiranjem. Testirane su klase ***Deck*** i ***Card***, čija se implementacija nalazi u fajlovima ***src/Deck.cpp*** i ***src/Card.cpp***, koje obezbeđuju sve neophodne funkcionalnosti za pravilno funkcionisanje igre.

Zahvaljujući modularnosti i jasno izdvojenim funkcionalnostima, fajlovi ***src/Card.cpp*** i ***src/Deck.cpp*** predstavljaju idealan izbor za pisanje jedinčnih testova, jer omogućavaju jednostavno i efikasno testiranje. Ukupno je napisano 27 test slučajeva, koji obuhvataju proveru 143 različita uslova.Uz manje izmene koda, koje su naglašene u delu 2.3, testovi su uspešno prošli, što ukazuje na stabilnost i ispravnost implementacije. Uspešnost svih testova možemo pogledati na slici 1.



Slika 1: Uspešnost napisanih testova

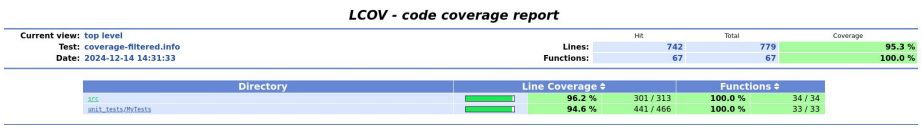
Primenom alata ***lcov*** generisan je sledeći izveštaj:

- ***Pokrivenost linija koda***: 742 od ukupno 779 linija, što čini 95.3%.
- ***Pokrivenost funkcija***: Sve funkcije (67 od 67) su pokrivene testovima, što predstavlja 100.0%.

Ovi rezultati ukazuju na visok nivo testiranja i dobru pokrivenost koda, posebno u pogledu funkcija koje su u potpunosti pokrivene. Međutim, preostalih 4.7% nepokrivenih linija može biti predmet dodatne analize i proširenja testova kako bi se obezbedila još veća pouzdanost implementacije.
Rezultati po direktorijumima:

- **Direktorijum `src`:**
  - ***Pokrivenost linija:*** 301 od ukupno 313 linija, što čini 96.2%.
  - ***Pokrivenost funkcija:*** Sve funkcije (34 od 34) su pokrivene testovima, što predstavlja 100.0%.
- **Direktorijum `unit_tests`:**
  - ***Pokrivenost linija***: 441 od ukupno 446 linija, što čini 94.6%.
  - ***Pokrivenost funkcija***: Sve funkcije (33 od 33) su pokrivene testovima, što predstavlja 100.0%.

Ovi rezultati ukazuju na visok nivo testiranja u oba direktorijuma. Direktorijum `src` ima nešto višu pokrivenost linija (96.2%) u poređenju sa direktorijumom `unit_tests` (94.6%). Ovaj podatak je posebno važan jer direktorijum `src` sadrži osnovni implementacijski kod projekta, dok `unit_tests` sadrži testove koji proveravaju ispravnost implementacije. Izvestaj iz LCOV-a mozete pogledati na slici 2.



Slika 2: Izlaz iz alata LCOV

Detaljniji rezultati i izvestaji testiranja nalaze se u direktorijumu *unit_tests/coverage-report*. Napisani testovi se nalaze u direktorijumu *unit_tests/MyTests*.

2.3 Izmene u kodu i uočene greške

Da bi testiranje odredjenih funkcionalnosti bilo moguće, bilo je potrebno implementirati neke dodatne funkcionalosti. Neke od dodatih funkcionalosti:

- Implementirana je funkcija koja proverava da li je karta prazna. Ova funkcija mi je bila potrebna da testiram odredjenje scenarije, poput uzimanje karte iz praznog spila. Potpis funcije:

```
bool Card::isEmpty();
```

- Nad klasom *Card* implementirani su operatori `==`, `!=` i `<`, koji su mi takođe bili potrebni u svrhu testiranja.

```
bool operator==(const Card& other) const;
bool operator!=(const Card& other) const;
bool operator<(const Card& other) const;
```

- Nad klasom Deck je takodje implementiran operator poredjenja `==` koji proverava jednakost dva špila.

```
  bool operator==(const Deck &other) const {
    if (deck.size() != other.deck.size()) {
        return false;
    }
  std::set<Card> thisDeckSet(deck.begin(), deck.end());
  std::set<Card> otherDeckSet(other.deck.begin(),other.
     deck.end());
 return thisDeckSet == otherDeckSet;
}
```

Prilikom pisanja testova, uoceno je da u kodu postoje sitne greske u implementaciji koje su ispravljene.

- Veličina špila se ne postavlja dobro

```
Deck::Deck(){
  size = 36;
  char suits[4] = {'H', 'D', 'S', 'C'};
  Card c;
  for(int i = 0; i < 4; i++){
    c.setSuit(suits[i]);
    for(int j = 6; j <= 14; j++){
      c.setVal(j);
      this->addCard(c);
    }
```

```
11    }
12  }
13  void Deck::addCard(Card c){
14    deck.push_back(c);
15    size++;
16  }
```

Problem: promenljiva *size* se na pocetku inizijalizuje na 36, ali se i 36 inkrementira pozivom funkcije addCard().Ovo dovodi do problema da je promenljiva *size* umesto vrednosti 36, ima vrednost 72.
Resenje: u konstrukoru postavljamo promenljivu *size* na 0.

- Uocena je greska u funkciji setVal():

```
1  bool Card::setVal(char value){
2    val = value;
3    return true;
4  }
```

Ova funkcija je promenjena kako bi se ograničila validnost unosa. Prethodno je bilo dozvoljeno da se unesu bilo koje vrednosti tipa `char`, dok sada unos mora biti u skladu sa specifičnim skupom vrednosti: brojevi od **6 do 14** , slova **J, Q, K, A**,i specijalna vrednost **-1**, koja sugerise da je karta prazna.Ovime je unapređena preciznost i tačnost unosa.

- U igri su dozvoljene karte sa vrednostima od 6 do 14, dok je u prethodnoj implementaciji bilo omogućeno postavljanje vrednosti od 1 do 14. Ovu grešku sam ispravila tako da sada vrednosti budu u opsegu od 6 do 14.

Klasa `main.cpp` nije testirana jer nije dovoljno modularna, s obzirom na to da je sav kod smešten u jednu funkciju. Dodatnu komplikaciju predstavlja postojanje jedne globalne promenljive koja otežava refaktorisanje i testiranje. Pokušala sam da je refaktorisem, ali nisam uspela da je podelim u nezavisne celine koje bi mogle da se testiraju. Iako klasa nije testirana, ona u najvećoj meri poziva testirane funkcije iz klasa `Card` i `Deck`. Prostor za dalja unapređenja predstavlja refaktorisanje `main` funkcije kako bi testiranje postalo moguće.

3 Valgrid

Valgrind je moćan alat za analizu i profilisanje programa koji se koristi za detekciju grešaka u vezi sa memorijom, performansama. Razvijen je kao open-source alat i najčešće se koristi u razvoju softvera, naročito za programe napisane u C i C++ jezicima. Glavni cilj Valgrinda je da pomogne programerima da identifikuju greške koje su obično teže uočljive, kao što su curenje memorije, neinicijalizovane promenljive i problemi sa pristupom memoriji.

Glavni Valgrind alati

1.Memcheck analizira upotrebu memorije u programu i detektuje različite greške kao što su curenje memorije, pristup memoriji nakon što je ona oslobodjena (use-after-free), čitanje neinicijalizovanih promenljivih i druge slične greške.
2.Callgrind – Alat za profilisanje koji se koristi za analizu performansi programa. Callgrind prati kako program koristi kes i funkcionalnosti procesora, pružajući detaljne informacije o tome kako se program izvršava i koliko vremena provodi u svakoj funkciji.
3.Massif – Alat za analizu korišćenja memorije u vremenu, koji daje detalje o tome koliko memorije je korišćeno tokom izvršavanja programa.
4.Cachegrind – Slično kao Callgrind, Cachegrind je alat za analizu upotrebe keš memorije. Koristi se za pronalaženje problema sa performansama koji su povezani sa načinom na koji program koristi procesorski keš. Pomaže u optimizaciji programa za bolje iskorišćenje resursa.
5.Helgrind – Alat za detekciju grešaka u višenitnom okruženju.

Prednosti Valgrinda

1.Detekcija grešaka: Valgrind je posebno koristan u detekciji grešaka koje mogu biti teško uočljive tokom uobičajenog testiranja.
2.Jednostavnost upotrebe: Alat je jednostavan za upotrebu i zahteva minimalnu konfiguraciju. Da bi se koristio, program se samo pokreće kroz Valgrind, koji tada analizira njegov rad.

Ograničenja Valgrinda

Usmerenost na C i C++:Iako se Valgrind koristi i za druge jezike, njegova najveća snaga je u C i C++ programima, gde se često javlja potreba za analizom memorije.
Problem sa skaliranjem: Valgrind može biti spor, posebno pri analiziranju velikih aplikacija. Pruža detaljne informacije, što može dovesti do značajnog smanjenja performansi pri testiranju.Program koji se analizira Valgind alatom može da se izvršava čak i do 100 puta sporije.
Nije pogodan za sve vrste grešaka: Iako je izuzetno koristan za otkrivanje problema sa memorijom, Valgrind nije alat za pronalaženje svih vrsta grešaka, kao što su logičke greške ili greške u implementaciji.
U cilju analize perfomansi programa pokrenuti su sledeci Valgrind alati: ***Memcheck***,***Callgrind*** i ***Massif***, a rezultate mozete pogledati u nastavku rada.

3.1 Memcheck

Memcheck je glavni alat u okviru Valgrinda. Memcheck radi tako što prati sve alokacije i dealokacije memorije tokom izvršavanja programa, upozoravajući programere na potencijalne probleme. Takođe, pomaže u identifikaciji grešaka koje mogu dovesti do nepredvidivog ponašanja, što je naročito korisno u složenim

softverskim projektima gde je precizno upravljanje memorijom ključno za stabilnost i performanse programa.

3.1.1 Analiza rezultata

Alat je pokrenut koriscenjem bash skripte *run_memcheck.sh* , koja se nalazi u direktorijumu *valgrind/memcheck*.
Jedan od izlaza iz Memcheck-a:

```
==3627== HEAP SUMMARY:
==3627==     in use at exit: 0 bytes in 0 blocks
==3627==   total heap usage: 148 allocs, 148 frees,
   75,842 bytes allocated
==3627==
==3627== All heap blocks were freed -- no leaks are
   possible
==3627==
==3627== ERROR SUMMARY: 0 errors from 0 contexts (
   suppressed: 0 from 0)
```

Objašnjenje:
1.,,*In use at exit: 0 bytes in 0 blocks"*: Ovaj deo pokazuje da na kraju izvršavanja programa nije bilo alocirane memorije koja nije oslobođena. To znači da nijedan memorijski blok nije ostao neoslobodjena, što znači da programu ne curi memoriju. Svaka alokacija memorije je pravilno oslobođena.
2.,,*Total heap usage: 148 allocs, 148 frees, 75,842 bytes allocated"*: Ovde vidimo ukupan broj alokacija i oslobađanja memorije tokom izvršavanja programa. Program je alocirao 148 puta i isto toliko puta je oslobođena memorija. Ukupno je alocirano 75,842 bajta, što može biti korisno da vidimo koliko memorije program koristi.
3.,,*All heap blocks were freed – no leaks are possible:"* Ovo je najvažniji deo izlaza. Ova poruka znači da su sve alocirane memorijske blokove ispravno oslobođene i da nema curenja memorije, što je znak se pravilno upravlja dinamičkom memorijom u ovom programu.
4.*"ERROR SUMMARY: 0 errors from 0 contexts:"* Ovo znači da Valgrind nije detektovao nikakve greške u memoriji tokom izvršavanja programa. Nema grešaka kao što su pokušaji pristupa nealokiranoj memoriji, dvostruko oslobađanje memorije ili korišćenje već oslobođene memorije.
Zaključak:
Program izgleda vrlo dobro u pogledu upravljanja memorijom. Memcheck je potvrdio da nema curenja memorije, nevalidnog pristupa ili drugih problema sa memorijom. S obzirom na ovo, možemo biti siguran da nema značajnih problema sa memorijom.Izlazi iz Memchecka mogu se pronaći u fajlovima *valgrind_memcheck_output_pid.txt*.Izveštaj koji je u ovom poglavlju analiziran nalazi se u fajlu *valgrind_memcheck_output.txt*. Svi izveštaji nalaze se u direktorijumu *valgrind/memcheck*.

3.2 Callgrind

Callgrind je alat iz Valgrinda koji se koristi za profilisanje performansi programa, fokusirajući se na praćenje poziva funkcija i analizu potrošnje CPU resursa. On omogućava uvid u to koje funkcije se najviše pozivaju, koliko vremena provode u izvršavanju i kako se program ponaša tokom rada.Callgrind prati i snima tok izvršavanja programa, beležeći broj poziva funkcija i njihovo trajanje. Ovi podaci pomažu u identifikaciji uskih grla u performansama, što je ključno za optimizaciju koda. Posebno je koristan za aplikacije sa visokim zahtevima za resursima, jer omogućava precizno lociranje problematičnih delova koda.

Rezultati analize mogu se vizualizovati pomoću alata kao što je **KCachegrind**, što omogućava lakše razumevanje podataka kroz grafički prikaz poziva funkcija, njihovih trajanja i međusobnih odnosa.

3.2.1 Analiza rezultata

Alat je pokrenut koriscenjem bash skripte *run_callgrind.sh*, koja se nalazi u direktorijumu *valgrind/callgrind*.

3.2.2 Najveća potrošnja instrukcija

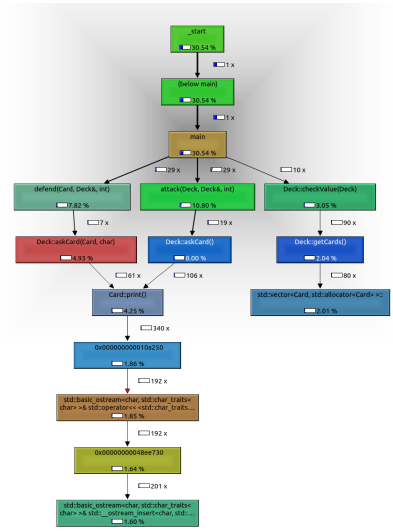
Najveću potrošnju resursa izaziva funkcija `_dl_lookup_symbol_x` iz sistemskog koda, što ukazuje na učestalu upotrebu dinamičkog povezivanja. Ovo može biti uzrokovano učestalim pozivima funkcija iz spoljnog koda, koji zahtevaju dinamičko povezivanje prilikom učitavanja biblioteka.

3.2.3 Najčešće pozivane funkcije u igri

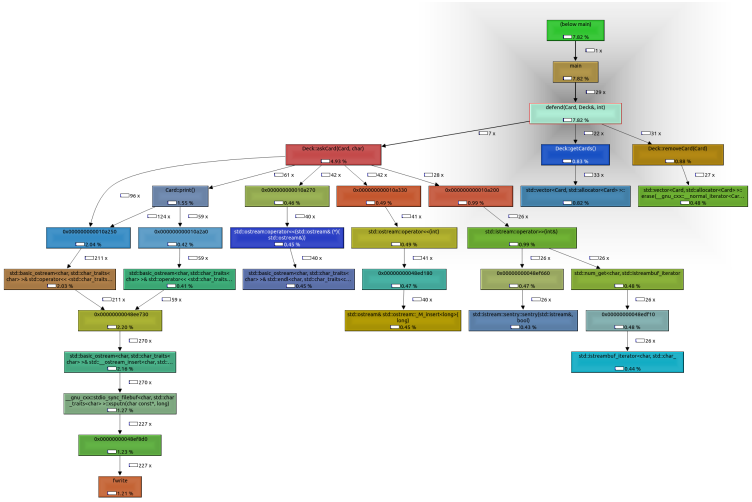
Funkcije se izdvajaju kao najčešće pozivane tokom izvođenja igre:

- `Card::print()`
- `defend(Card, Deck&, int)`
- `attact(Card, Deck&, int)`
- `Deck::askCard(Card, char)`
- `Card::getNumValue()`
- `Deck::getCards()`
- `Deck::removeCard(Card)`

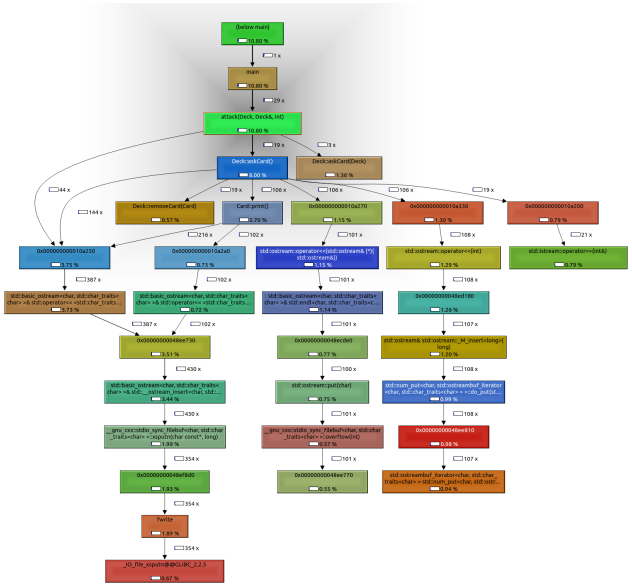
Ove funkcije igraju ključnu ulogu u logici igre, što sugeriše potrebu za daljom optimizacijom. Nakon dobijenih rezultata, radi vizuelizacije pokrenut je alat KCatcheGrind. Na graficima 3 4 5, mozemo videti pozive funkcija tokom izvrsavanja programa.



Slika 3: Najčešće pozvane funkcije



Slika 4: Najčešće pozvane funkcije



Slika 5: Najčešće pozvane funkcije

Grafici se nalaze u direktorijumu *valgirnd/callgrind*.

3.2.4 Sistemske i bibliotečke funkcije

Veliki deo troška resursa dolazi iz upotrebe standardnih biblioteka, naročito funkcija povezanim sa ispisivanjem podataka, kao što je `std::ostream`. Ovaj trošak je rezultat intenzivnog ispisivanja informacija tokom igre. Takođe, operacije sa `std::vector` (kao što su `begin`, `end`, i `erase`) troše značajne resurse, što ukazuje na čestu modifikaciju vektora, što može dovesti do prekomernih alokacija memorije i negativno uticati na performanse.

3.2.5 Preporuke za optimizaciju

Da bi se poboljšale performanse i smanjio trošak resursa, preporučuje se sledeće:

- **Optimizacija operacija sa `std::vector`**: Preporučuje se unapred rezervisanje memorije koristeći `reserve`, kako bi se smanjio broj alokacija i prekomernih manipulacija vektorima.
- **Implementacija move semantike**: Korišćenjem move semantike, broj operacija kopiranja i premeštanja objekata može biti smanjen, čime se postiže veća efikasnost.Move semantika smanjuje nepotrebno kopiranje velikih objekata. Umesto da kopira podatke, ona samo "preseli" resurse iz

jednog objekta u drugi. To znači da je brža i efikasnija, jer ne troši vreme i memoriju na pravljenje duplikata.

Primena ovih preporuka može značajno unaprediti efikasnost igre, smanjiti potrošnju resursa, kao i poboljšati stabilnost i sigurnost koda.

Detaljan izlaz iz Callgrinda nalazi se u fajlovima *callgrind_output_pid.txt* i *callgrind_annotated_output_pid.txt*.Svi izvestaji se nalaze u direktorijumu *valgrind/callgrind/*.

3.3 Massif

Massif je alat za profilisanje u kontekstu memorijske potrošnje, koji je deo GNU alata za analizu performansi (perf tools). Koristi se za praćenje upotrebe heap memorije tokom vremena, a posebno je koristan u aplikacijama koje imaju problema sa prekomernom potrošnjom memorije ili curenjem memorije.

Massif radi tako što koristi valgrind za simulaciju rada programa i sakupljanje podataka o memoriji,a zatim generiše izveštaje koji prikazuju koji delovi programa troše najviše memorije, kako se memorijska potrošnja menja u vremenu, kao i gde se memorija alocira i oslobađa.

3.3.1 Analiza rezultata

Alat je pokrenut koriscenjem bash skripte *run_massif.sh*, koja se nalazi u direktorijumu *valgrind/massif/*.

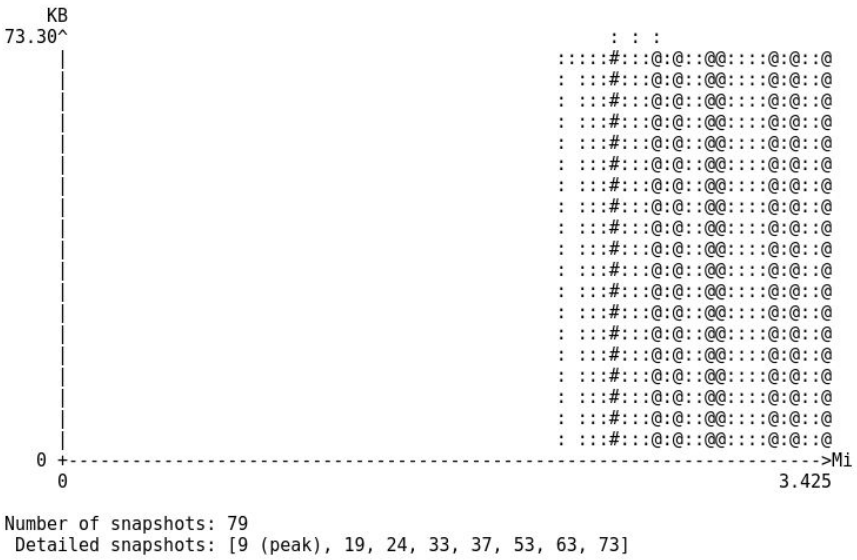
3.3.2 Glavni uvidi

- Rezultati ukazuju na to da je skoro celokupna memorija (72,862 bajta) alocirana na heap-u.
- Većina memorije (96.91%) alocira se iz standardnih C++ biblioteka tokom inicijalizacije (`_dl_init`). Ovo je očekivano ponašanje i ne zahteva dodatnu optimizaciju.
- Direktne alokacije u kodu su povezane sa funkcijama kao što su `Deck::askCard` (za ulazne operacije) i `main` (za izlazne operacije).

3.3.3 Preporuke za optimizaciju

- **Smanjite IO operacije:** Umesto da podatke unosite i ispisujete jedan po jedan koristeći *std::cin* i *std::cout*, prikupite ih u grupu (bafer) i obradite ih odjednom. Na taj način, program će raditi brže i efikasnije.
- **Optimizujte ključne funkcije**
- **Refaktorisanje upravljanja memorijom:** Smanjite učestalost alokacije i dealokacije objekata tako što ćete ponovo koristiti već alocirane dinamičke strukture umesto da ih stalno stvarate i oslobađate.

Na slikama 6 i 7 se moze videti izlaz iz alata u vidu grafa, gde je na x osi predstavljeno vreme (izrazeno brojem izvrsenih instrukcija), a na y osi je predstavljena ukupna kolicina memorije na hipu koju program zauzima u datom vremenskom trenutnku.



Slika 6: Izlaz iz Massifa

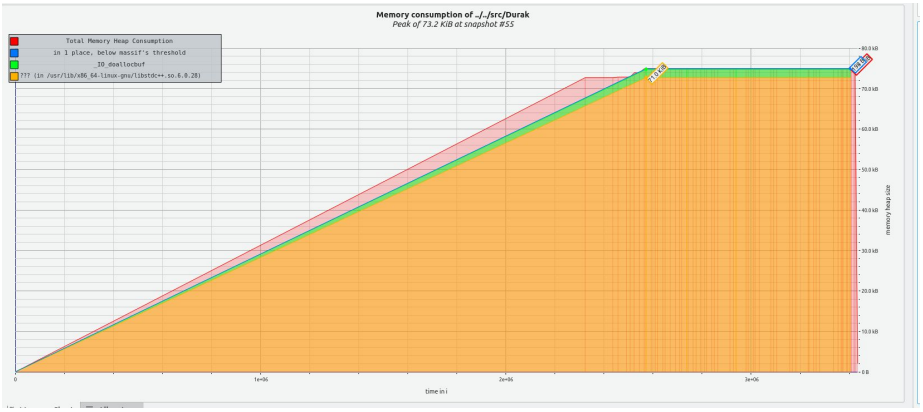
n	time(i)	total(B)	useful-heap(B)	extra-heap(B)	stacks(B)
0	0	0	0	0	0
1	2,323,410	72,712	72,704	8	0
2	2,436,175	72,736	72,706	30	0
3	2,457,223	72,848	72,832	16	0
4	2,492,123	72,872	72,834	38	0
5	2,506,177	72,896	72,850	46	0
6	2,522,790	73,928	73,888	40	0
7	2,537,191	73,928	73,888	40	0
8	2,557,153	74,960	74,912	48	0
9	2,570,612	75,008	74,926	82	0

Slika 7: Izlaz iz Massifa

Na osnovu ovoga, može se zaključiti da program pokazuje stabilnu i efikasnu potrošnju memorije tokom vremena, sa ukupno 79 snimaka memorije. Maksimalna potrošnja memorije (peak) dostignuta je na snimku 9 sa ukupno 75,008 B, od čega korisna memorija zauzima 74,926 B, dok dodatna memorija (overhead) iznosi samo 82 B. Rast memorijske potrošnje je postepen, od početnih 72,712 B do vrhunca na snimku 9, nakon čega ostaje stabilna bez značajnih oscilacija.

Važno je naglasiti da dodatna memorija (extra-heap) raste u malim granicama, što ukazuje na efikasno upravljanje memorijom, dok nema naznaka značajnih curenja memorije.

Na slici 8, možemo videti rezultate koje smo dobili vizualizacijom podataka iz Massifa.Grafikon prikazuje konstantan rast memorijske potrošnje tokom celokupnog trajanja programa, što sugeriše da se memorija alocira, ali se nikada ne oslobađa, verovatno zbog velikog broja vektora koji se ne uklanjaju do kraja programa. Ovaj trend je jasno vidljiv na grafikonu jer memorija ne opada, već se stabilizuje na višem nivou do završetka programa.



Slika 8: Vizuelizacija jednog od izlaza iz Massifa

Zaključno, projekat pokazuje stabilnu potrošnju memorije, ali dodatna optimizacija u kritičnim tačkama može poboljšati efikasnost i skalabilnost.
Koriscenjem opcije *ms_print*, dobijaju se citljivije informacije iz izvestaja, koje su zapisane u fajlovima ***ms_print.pid.txt***.

4 Clang-tidy

Clang-Tidy je statički analizator koda koji je deo Clang ekosistema i koristi se za automatsko otkrivanje grešaka, poboljšanje kvaliteta koda i primenu stilskih smernica u C, C++ i drugim jezicima koji podržavaju Clang kompilator. Alat se uglavnom koristi za analizu koda pre kompajliranja, kako bi se identifikovali potencijalni problemi u kodu, neskladnosti sa stilom kodiranja, pa čak i mogućnosti za optimizaciju. Osnovne funkcionalnosti Clang-Tidy alata:

Statička analiza koda: Clang-Tidy analizira kod bez potrebe za njegovim izvršavanjem, otkrivajući greške, slabosti i potencijalne greške koje bi mogle proći nezapaženo tokom uobičajenog testiranja ili tokom kompilacije.

Stilske provere: Alat može proveriti usklađenost koda sa definisanim stilskim pravilima, kao što su pravilno uvlačenje, korišćenje razmaka umesto tabova, imenovanje promenljivih i funkcija itd.

Automatske ispravke: Clang-Tidy omogućava automatsko ispravljanje određenih tipova problema, kao što su dodavanje nedostajućih zaglavlja, preimenovanje promenljivih ili korišćenje boljih funkcionalnosti iz standardnih biblioteka.

4.1 Analiza rezultata

Alat `clang-tidy` primenjen je u svrhu analize programa i otkrivanja grešaka za koje je namenjen. Alat je pokrenut korišćenjem skripte *run_clang.sh*, koja se nalazi u direktorijumu *clang*. Otkriveno je jako puno upozorenja, koje možemo klasifikovati u nekoliko kategorija:

1. *Nepoštovanje pravila o formatiranju kodova*:

   Redosled #include direktiva nije pravilan Header fajlovi nisu sortirani u odgovarajućem redosledu. Kod:

```
#include <iostream>
#include "Card.h"
```

   Bi trebalo zameniti sa:

```
#include "Card.h"
#include <iostream>
```

   kako bi bilo u skaladu sa *LLVM include order pravilom.*

   Nedostatak zagrada kod *if/else* blokova Sve naredbe u if/else blokovima treba da budu unutar viticastih zagrada radi bolje čitljivosti i izbegavanja grešaka.Kod oblika:

```
   if(val == '0')
    cout << "10";
     else
    cout << val;
```

   bi trebalo zameniti sa:

```
   if(val == '0'){
    cout << "10"
    }else {
    cout << val;
    }
```

2. *Problemi sa modernim C++ stilom*

   Nekorišćenje trailing return tipova : Funkcije treba da koriste auto i trailing return tip.Potpise oblika:

```
Card attack(Deck table, Deck &hand, int player);
```

zameniti sa:

```
auto attack(Deck table, Deck &hand, int player) ->
    Card;
```

Ovo poboljšava čitljivost i omogućava dosledniji stil u modernom C++ kodu.

const funkcije Metode poput *getNumValue()*, *print()* i jos mnogo drugih, mogu biti označene kao const jer ne menjaju stanje objekta.

C-stil nizovi Deklaracija niza *char suits[4]* koristi običan C-stil nizova, što može biti problematično jer ne pruža informacije o veličini niza, ne omogućava lako upravljanje elementima i ne koristi prednosti koje pruža C++ standardna biblioteka. Takvi nizovi su skloni greškama, poput prekoračenja granica niza, jer ne pružaju mehanizme za bezbednu manipulaciju. **Rešenje:** Umesto C-stil niza, preporučuje se korišćenje modernih struktura podataka iz C++ standardne biblioteke, poput *std::array* ili *std::vector*. One nude bolju sigurnost, čitljivost, i fleksibilnost. Na primer:

- `std::array<char, 4>` je statički alociran niz koji poznaje svoju veličinu u trenutku kompajliranja.
- `std::vector<char>` je dinamički alociran niz koji omogućava promenu veličine tokom izvršavanja programa.

Petlje sa brojačima Klasične *for* petlje sa brojačima često se koriste za iteraciju kroz nizove ili kolekcije. Međutim, one zahtevaju dodatnu pažnju pri upravljanju brojačem (npr. i), što može dovesti do grešaka, kao što su prekoračenje granica niza, nepravilna ažuriranja brojača ili neodgovarajući uslovi zaustavljanja. Takođe, ovakav pristup može smanjiti čitljivost koda, posebno kada je logika unutar petlje složenija.

Tradicionalna *for* petlja izgleda ovako:

```
for (int i = 0; i < 4; i++) {
    std::cout << suits[i] << std::endl;
}
```

Ovaj pristup zahteva ručno praćenje granica niza (i ¡ 4), što je rizično i nije intuitivno, naročito u složenijim aplikacijama.

Rešenje: Umesto klasičnih petlji sa brojačem, preporučuje se korišćenje petlji zasnovanih na rasponima, koje su uvedene u C++11 standardu. Ove petlje omogućavaju elegantniji i čitljiviji način iteracije kroz elemente kolekcije, bez potrebe za eksplicitnim upravljanjem brojačem.

```
for (char suit : suits) {
    std::cout << suit << std::endl;
}
```

Provera veličine sa `size() == 0`: Ručna provera da li je kontejner prazan pomoću izraza *size() == 0* često se koristi, ali nije idealna jer može biti manje čitljiva i potencijalno manje efikasna za određene tipove kontejnera. Iako ovaj pristup funkcioniše, *empty()* metoda je specifično dizajnirana za ovu svrhu i bolje izražava nameru programera.

Rešenje: Umesto ručne provere sa *size() == 0*, preporučuje se korišćenje metode *empty()*, koja jasno i direktno proverava da li je kontejner prazan. Ovaj pristup je čitljiviji i može biti efikasniji za neke kontejnere, jer ne mora da računa veličinu.

```
if (tableCards.empty()) {
}
```

Listing 1: Provera sa empty()

Metoda `empty()` povećava čitljivost i čini kod u skladu sa modernim C++ standardima.

3. *Problemi sa inicijalizacijom promenljivih:*
Alat clang-tidy je primetio da na nekim mestima postoje promenljive koje nisu inicijalizovane, sto može dovesti do nepredvidjenog ponašanja.Svaka promenljiva treba biti inicijalizovana prilikom deklaracije.

4. *Upotreba ,,magic numbers"*:
Konstantni brojevi poput *14, 13, 12, 48, 6, i 9* koriste se direktno u kodu. Preporučuje se zamena ovih vrednosti imenovanim konstantama radi čitljivosti i održavanja.

5. *Problemi sa konverzijama i kastovanjem*:

C-stil kastovanje Kastovanje zapisano kao *(tip)objekat*, može biti problematično jer ne ukazuje jasno na tip kastovanja koji se primenjuje. Ovakvo kastovanje smanjuje čitljivost koda i može dovesti do nebezbednih operacija.

```
int a = 10;
double b = (double)a;
```

Rešenje: Umesto toga, koristi se specifični operator *static_cast* za statički bezbedne konverzije, što čini nameru programera jasnijom.

```
int a = 10;
double b = static_cast<double>(a);
```

Narrowing conversions Operacije poput

```
val = value + 48
```

su označene kao potencijalno problematične jer može doći do neočekivanih konverzija. Konkretno ova operacija se koristi za pretvaranje celobrojne vrednosti (*int*) u njen odgovarajući karakter (*char*) u ASCII tabeli.

6. *Problemi sa grananjem*:

   Pokriće svih mogućih grana u switch naredbi: Nema default slučaja u switch naredbi, što može dovesti do nepredviđenih ponašanja ako vrednost ne odgovara nijednom case.

   Else nakon *return* je nepotreban Korišćenje *else* nakon *return* unutar *if* bloka je suvišno jer se izvršavanje funkcije prekida sa *return*. Na primer:

```
if (condition) {
    return value;
} else {
    return otherValue;
}
```

   Rešenje: Uklanjanjem *else* blok postaje čitljiviji:

```
if (condition) {
    return value;
}
return otherValue;
```

   Korišćenje previše pojedinačnih *if/else* blokova: Umesto toga, potrebno je da se razmotri alternativni pristup, poput korišćenja mapa ili enumeracija za vrednosti karata.

7. *Drugi problemi*:

   Promenljive se ne koriste optimalno Nekoliko promenljivih može se optimizovati za bolju čitljivost.

Detaljan spisak svih upozorenja nalazi se u fajlu *clang-tidy_output.txt*, u direktorijumu *clang/*.

5 Cppcheck

Cppcheck je alat za statičku analizu koda koji je specijalizovan za C i C++ programe. Njegov osnovni cilj je da pomogne u identifikaciji potencijalnih grešaka, nesigurnosti, i neefikasnosti u kodu pre nego što dođe do izvršenja. Cppcheck se fokusira na otkrivanje problema kao što su greške u sintaksi, logici, potencijalni problemi sa memorijom, nedostatak resursa i drugi problemi koji mogu biti teški za uočiti tokom normalnog razvoja. Za razliku od nekih drugih

alata, Cppcheck ne zahteva kompajliranje koda i ne koristi spoljne biblioteke. Može se koristiti direktno iz komandne linije, kao i integrisati u IDE-ove. Alat pruža različite nivoe izveštaja, od osnovnih upozorenja do dubljih analiza, a može se koristiti za proveru koda na različitim platformama. S obzirom na to da je open-source, Cppcheck je često korišćen u industriji i akademiji, posebno kada je potrebno brzo detektovati i ispraviti potencijalne greške u velikim kodnim bazama.

5.1 Analiza rezultata

Alat *Cppcheck* pokrenut je korišćenjem bash skripte *run_cppCheck.sh* koja se nalazi u direktorijumu *cppcheck*.Detaljan izlaz iz alata *Cppcheck* se takodje nalazi u direktorijumu *cppcheck* pod nazivom *output.xml*. Alat je detektovao greške koje možemo klasifikovati u nekoliko kategorija:

5.1.1 Greške vezane za `const` kvalifikator

Poput upozorenja koje je generisao alat *clang-tidy*, ovu grupu gresaka cine funkcije koje mogu biti *const*, jer ne menjaju stanje objekta, a nisu oznacene kao *const*.

5.1.2 Greške vezane za eksplicitne konstruktore

Preporučuje se korišćenje `explicit` ključe reči za konstruktore koji imaju samo jedan argument. To pomaže u izbegavanju grešaka vezanih za implicitne konverzije tipova i čini kod sigurnijim.

- **Greška:** `Deck` klasa ima konstruktor sa jednim argumentom koji nije `explicit`.

5.1.3 Greške vezane za negativne indekse

Ove greške sugerišu da postoje potencijalni problemi u kodu vezani za negativne indekse u nizovima. Preporučuje se proveriti uslove i osigurati da ne dođe do pristupa negativnim indeksima u nizovima.

- **Greška:** Indeks `n <= 0` može biti suvišan ili se koristi negativni indeks `-1`.

Ova upozorenja su se pokazala kao potencijalno relevantna za dalju analizu. Konkretno, upozorenja vezana za negativne indekse pojavljuju se u fajlu *Deck.cpp* i linijama 160, 165, 190, 194, 232 i 236.
Delovi koda u kojima se javljaju ovakve greske:

```
    while(n <= 0 || n > size){
    cout << "Invalid choice!\nChoose a card:\n";
    cin >> n;
}
```

Smatram da ovo upozorenje nije relevantno, jer je potrebna provera da li je $n \leq 0$, s obzirom na to da n predstavlja indeks karte u nizu, a karte se broje od 1. Ova provera je neophodna kako bi se osiguralo da korisnik unese validan broj indeksa.

```
1     Card c = deck[n - 1];
```

Ova linija je označena kao potencijalno problematična, međutim, s obzirom na to da *while* petlja proverava uslov `n <= size`, a *size* je promenljiva koja se inicijalizuje sa `deck.size()`, smatram da ovaj deo koda nije problematičan.

5.1.4 Greške vezane za neiskorišćene funkcije

Greške u vezi sa neiskorišćenim funkcijama sugerišu da određene funkcije možda nisu potrebne ili da nisu pozvane unutar programa, što može ukazivati na nepotreban kod.

- **Greška:** Funkcija `isEmpty` nije iskorišćena.

Iako je alat detektovao ovu funkciju kao neiskorišćenu, ova funkcija je koriscena za potrebe testiranja.

5.1.5 Zakljucak

Alat *cppcheck* nije pronašao značajne greške, što nam omogućava da zaključimo da kod ne sadrži greške koje su specifično ciljane ovim alatom. Iako `cppcheck` efikasno identifikuje potencijalne probleme u kodu, u ovom slučaju, nije detektovan nijedan veci problem koji bi mogao da utiče na stabilnost i ispravnost programa, prema parametrima koje alat pokriva.