

Matematički fakultet, Univerzitet u Beogradu

SEMINARSKI RAD

Analiza projekta otvorenog koda *Airline Reservation
system*

Natalija Filipović 1013/2024

Avgust 2025

Sadržaj

1	Uvod	2
1.1	O seminarskom radu	2
1.2	Napomena	2
2	Testovi jedinica koda i pokrivenost koda	2
2.1	O testiranju	2
2.2	O QtTest-u	3
2.3	Analiza dobijenih rezultata	3
3	Memcheck	6
3.1	O Memcheck alatu	6
3.2	Analiza dobijenih rezultata	7
4	Cppcheck	9
4.1	O Cppcheck alatu	9
4.2	Analiza dobijenih rezultata	9
5	Clang-format	11
5.1	O Clang-u	11
5.2	Analiza dobijenih rezultata	12
6	Zaključak	13

1 Uvod

1.1 O seminarskom radu

U okviru ovog seminarskog rada izvršena je analiza projekta otvorenog koda, pod nazivom *Airline Reservation System*. Projekat predstavlja sistem za pronalaženje letova, napisan je u programskog jeziku C++, a izvršava se putem terminala.

Rad obuhvata primenu alata za verifikaciju softvera na odabranom projektu i analizu dobijenih rezultata. Korišćeni alati proveravaju ispravnost korišćenja memorije, pomažu u detektovanju i lociranju grešaka, testiraju pokrivenost linija koda prilikom izvršavanja programa sa različitim ulaznim vrednostima i sugerišu stilske izmene kako bi kod bio uniformno formatiran.

U okviru svakog direktorijuma, napravljena je po jedna skripta za pokretanje konkretnog alata i reprodukciju dobijenih rezultata.

1.2 Napomena

Da bi se program kompajlirao i pokrenuo, potrebno je na samom početku izvršiti dve promene u source fajlovima.

U *List.h* fajlu potrebno je izmeniti liniju 3:

- **include "Vector"** zameniti sa **include "Vector.h"**

U fajlu *Graph.h* potrebno je izmeniti funkcije *getCost* i *getCostArrayIndex* na isti način:

- na kraj funkcije dodati naredbu **return -1;**

U fajlu *Stack.h* izmeniti funkciju *pop*:

- na kraj funkcije dodati naredbu **return T();**

Ove greške otkrivene su prilikom prevođenja koda za jedinične testove, pa ih je potrebno izmeniti kako bi bilo kakvo testiranje moglo da se izvrši. Promene ne utiču na uspešan ili neuspešan završetak testova, jer se bez njih uopšte i ne mogu pokrenuti.

2 Testovi jedinica koda i pokrivenost koda

2.1 O testiranju

Testiranje je jedna od najčešće korišćenih tehnika pri dinamičkoj verifikaciji softvera, odnosno pri ispitivanju ispravnosti softvera u toku njegovog

izvršavanja. Izvršavanje testova je proces primene konkretnih test slučajeva i proveru da li su dobijeni rezultati u skladu sa očekivanih vrednostima. Prema nivoima testiranja, razlikujemo testiranje:

- pojedinačnih modula - *testiranje jedinica koda*
- grupe modula - *testiranje komponenti/integraciono testiranje*
- celog sistema - *sistemske testove*

Cilj jediničnih testova je utvrđivanje da li nezavisni delovi koda imaju predviđenu funkcionalnost. To mogu biti podprogrami, klase, metode... Jedinični testovi treba da budu kreirani za sve javne metode klase, treba da pokriju i trivijalne i granične slučajeve, kao i nevalidne vrednosti kako bi se testiralo ponašanje programa u slučaju greške.

Svaki test slučaj prolazi kroz neku putanju programa, a kako broj mogućih putanja može biti prevelik, teži se kreiranju praktično prihvatljivog broja testova sa što većom pokrivenošću koda. Zbog toga se uz testiranje (nakon što svi testovi prođu) generiše i izveštaj o pokrivenosti.

2.2 O QTest-u

QtCreator kao razvojno okruženje pruža mogućnost pisanja i izvršavanje jediničnih testova kroz modul QTest. Da bismo testirali već postojeći projekat, potrebno je da kreiramo novi projekat tipa Qt Unit Test i da u projekat dodamo lokaciju projekta koji se testira. Okruženje će nam generisati test klasu koja nasleđuje QObject u okviru koje ćemo pisati naše test funkcije. QTest pruža niz makroa poput QVERIFY, QCOMPARE i QFAIL, a najznačajniji za ovaj rad biće QVERIFY koji proverava da li je povratna vrednost izraza *true*.

2.3 Analiza dobijenih rezultata

Pri generisanju test primera, ideja je bila da se napravi po jedan primer za svaku funkcionalnost sistema. Sistem nudi osnovnu i naprednu pretragu letova, a zatim u okviru osnovne pretrage direktan ili let sa presedanjem. Napredna pretraga omogućava pretragu direktnog leta sa najmanjom cenom, najmanjim vremenom i let sa presedanjem. To nam ukupno daje 5 mogućih

testova. Testirano je i ponašanje programa u slučaju da let sa zahtevanim parametrima ne postoji, što nam daje ukupno 10 testova.

Svi test slučajevi su prošli ispitivanje. Obzirom da je cela aplikacija zasnovana na interakciji programa i korisnika kroz poruke na standardni ulaz i izlaz, oni su preusmereni u bafere. Zatim je provereno da li program vraća očekivani tekst, odnosno da li se u bafere u koji smo preusmerili izlaz nalazi očekivana poruka. Takvu proveru omogućio nam je makro `QVERIFY` koji testira da li funkcija za pronalaženje podstringa vraća vrednost `true`:

`QVERIFY(programOutputString.contains("Očekivana poruka sistema."))`

Ukoliko funkcija vrati vrednost `true`, izlaz programa sadrži očekivanu poruku i test prolazi.

```
***** Start testing of test *****
Config: Using QTest library 5.15.3, Qt 5.15.3 (x86_64-little_endian-lp64 shared (dynamic) release build; by GCC 11.3.0), ubuntu 22.04
PASS : test::initTestCase()
PASS : test::testBasicDirectFlight_Found()
PASS : test::testBasicDirectFlight_NotFound()
PASS : test::testBasicConnectedFlight_Found()
PASS : test::testBasicConnectedFlight_NotFound()
PASS : test::testAdvancedMinimumCostFlight_Found()
PASS : test::testAdvancedMinimumCostFlight_NotFound()
PASS : test::testAdvancedMinimumTimeFlight_Found()
PASS : test::testAdvancedMinimumTimeFlight_FoundLater()
PASS : test::testAdvancedConnectedFlight_Found()
PASS : test::testAdvancedConnectedFlight_NotFound()
PASS : test::cleanupTestCase()
Totals: 12 passed, 0 failed, 0 skipped, 0 blacklisted, 8ms
```

Slika 1: Rezultat testiranja

Nakon toga, ispitana je pokrivenost koda ovim testovima, pomoću alata *lcov*. Rezultati su sledeći:

Ukupno:

- **Pokrivenost linija koda: 92.9%**, odnosno 975 linija od ukupno 1049
- **Pokrivenost funkcija: 98.3%**, odnosno 58 funkcija od ukupno 59

Direktorijum *src*:

- **Pokrivenost linija koda: 91.7%**, odnosno 818 linija od ukupno 892
- **Pokrivenost funkcija: 100.0%**, odnosno 43 funkcija od ukupno 43

Direktorijum *unit_tests*:

- **Pokrivenost linija koda: 100.0%**, odnosno 157 linija od ukupno 157

LCOV - code coverage report					
Current view: top level		Hit		Total	Coverage
Test: coverage_filtered.info		Lines:	975	1049	92.9 %
Date: 2025-08-17 18:14:45		Functions:	58	59	98.3 %
Directory	Line Coverage ↕	Functions ↕			
/home/nata/Desktop/2024_Analysis_Airline-Reservation-	<div><div></div></div>	100.0 %	157 / 157	93.8 %	15 / 16
System/unit_tests/testis	<div><div></div></div>	91.7 %	818 / 892	100.0 %	43 / 43
src	<div><div></div></div>				

Generated by: LCOV version 1.14

Slika 2: Izveštaj o ukupnoj pokrivenosti

- **Pokrivenost funkcija: 93.8%**, odnosno 15 funkcija od ukupno 16

Detaljnou analizom rezultata za direktorijum *src*, koji sadrži izvorni kod, dolazi se do nekoliko zaključaka. Sve funkcije glavnog programa su pozvane barem jednom. Fajl *ListUi.h* koji sadrži jednu funkciju za ispis liste svih letova u bazi nije analiziran. Razlog je to što se funkciju koja se u njemu nalazi nikada ne poziva. Problem nije u test slučajevima, već se zaključuje da se ona nikada ne poziva od strane samog programa.

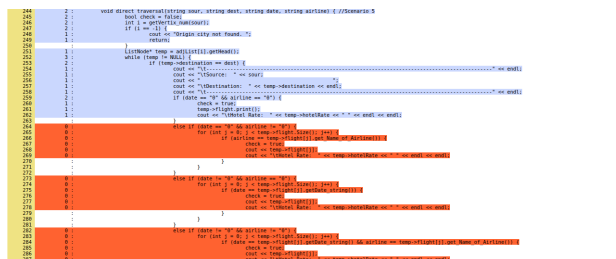
LCOV - code coverage report					
Current view: top level - src		Hit		Total	Coverage
Test: coverage_filtered.info		Lines:	818	892	91.7 %
Date: 2025-08-18 20:41:37		Functions:	43	43	100.0 %
Filename	Line Coverage ↕	Functions ↕			
DateAndTime.h	<div><div></div></div>	100.0 %	26 / 26	100.0 %	1 / 1
Flight.h	<div><div></div></div>	93.6 %	73 / 78	100.0 %	2 / 2
FlightUi.h	<div><div></div></div>	84.2 %	32 / 38	100.0 %	1 / 1
Graph.h	<div><div></div></div>	88.4 %	413 / 467	100.0 %	20 / 20
List.h	<div><div></div></div>	100.0 %	38 / 38	100.0 %	4 / 4
Stack.h	<div><div></div></div>	92.6 %	25 / 27	100.0 %	6 / 6
UserInterface.h	<div><div></div></div>	100.0 %	90 / 90	100.0 %	1 / 1
Vector.h	<div><div></div></div>	76.9 %	20 / 26	100.0 %	7 / 7
makeGraph.h	<div><div></div></div>	99.0 %	101 / 102	100.0 %	1 / 1

Generated by: LCOV version 1.14

Slika 3: Detaljan izveštaj o pokrivenosti direktorijuma *src*

Takođe u fajlovima *FlightUi.h* i *Graph.h*, pokrivenost linija koda je oko 85% jer nisu pokriveni svi slučajevi u kontekstu različitih parametara koje je moguće proslediti. Dodatni test primeri sa različitim ulaznim vrednostima za recimo datume i avio kompanije, popravili bi pokrivenost. Na primer, naš test za testiranje osnovne pretrage direktnih letova ne specifikuje ni datum ni avio kompaniju, ali ako bismo prosledili neke konkretne vrednosti, pokrili bismo dodatne linije koda. Koje tačno linije u fajlu *Graph.h* nisu pokrivene, mogu se videti na slici 4.

Pokrivenost linija je 100% u *unit_tests* direktorijumu, a sve test funkcije su pozvane bar jednom. Pokazatelj 15/16 za pokrivenost funkcija najverovatnije



Slika 4: Primer linija koda koje nisu pokrivene

potiče od funkcije koju je sistem(kompajler ili alat) automatski generisao, a ne od funkcije definisane u samim testovima.

Dobijeni rezultati ukazuju na visok nivo pokrivenosti sa samo 10 test slučajeva. Detaljan izveštaj sačuvan je u direktorijumu *coverage_report*.

3 Memcheck

3.1 O Memcheck alatu

Profajliranje je vrsta dinamičke analize programa (program se analizira tokom izvršavanja) koja obuhvata merenje vremenske i memorijske efikasnosti programa. Rezultat je skup podataka dobijen izvršavanjem programa sa određenim ulaznim podacima, npr. koliko se često neka funkcija ili kod izvršava, koliko je vremena provedeno u nekom bloku, podatke o alokaciji memorije itd. Programi koji vrše profajliranje se zovu profajleri, a Valgrind je jedan od njih. Valgrind je zapravo platforma za pravljenje alata za dinamičku analizu mašinskog koda. Alat za dinamičku analizu koda se kreira kao dodatak, pisan u programskom jeziku C, na jezgro Valgrind-a. Pošto se vrši analiza mašinskog koda, može se koristiti za programe pisane u bilo kom programskom jeziku, ali značajno usporavaju njegov rad. Svaki alat pokreće se na sledeći način:

valgrind --tool=alat [argumenti alata] ./a.out [argumenti za a.out]

Valgrind distribucija sadrži naredne alate:

- **Memcheck** - detektor memorijskih grešaka
- **Massif** - praćenje rada dinamičke memorije, odnosno profajler hip memorije

- **Cachegrind** - profajler keš memorije
- **Callgrind** - profajler funkcija
- **Helgrind** - detektor grešaka niti

Memcheck je najpoznatiji alat Valgrinda. Detektuje memorijske greške korisničkog programa kao što su pristupanje memoriji van opsega steka i hipa, curenje memorije, pristup već oslobođenoj memoriji, nepravilno oslobađanje memorije usled nepravilnog uparenih funkcija, korišćenje neinicijalizovanih vrednosti. Prijavljuje greške čim ih pronađe navodeći broj linije i kojim putem se stiglo do tog mesta (*stack trace*).

3.2 Analiza dobijenih rezultata

U ovom radu memcheck je pokrenut sa sledećim argumentima:

- **-leak-check=full** - detaljna provera curenja memorije, za svaki slučaj curenja posebno, ne daje samo sumirane rezultate
- **-show-leak-kinds=all** - prikazuju se svi tipovi curenja memorije: definitivno izgubljena, indirektno izgubljena, moguće izgubljena i memorija koja je jos uvek dostupna
- **-track-origins=yes** - ispisuje se tačna linija svakog nevalidnog čitanja ili pisanja (*da bi ovo radilo, potrebno je da program bude kompajliran sa debug opcijom -g*)

Izveštaj dobijen pokretanjem memcheck-a ukazuje na nekoliko važnih problema pri upotrebi memorije. Pre svega, program pokrenut pod memcheck-om nije moguće izvršiti jer se javlja **segmentation fault** i izvršavanje se prekida.

```
nata@nata-Inspiron-3521:~/Desktop/2024_Analysis_Airline-Reservation-System/memcheck$ ./run_memcheck.sh
Starting compilation...
Starting memcheck analysis...
./run_memcheck.sh: line 13: 13225 Segmentation fault      (core dumped) valgrind --tool=memcheck --leak
-file="$OUTPUT_FILE" --track-origins=yes $SOURCE_DIR/airline_reservation_system
```

Slika 5: Pokretanje skripte za memcheck

Na 4 mesta u programu, pristupa se nedodeljenoj memoriji. Pokušava se čitanje sa 2 i pisanje na 2 nedodeljene adrese. Proces se završava sa **segmentation fault (SIGSEGV)** greškom. Sve 4 greške javljaju se prilikom poziva funkcije *readFromFile* u klasi *makeGraph*, u *liniji 34*. Deo poruke koji kaže da se pristupa adresi 0x0 znači da se dereferencira null pointer. Na osnovu ostalih detalja poruke moguće greške su dodeljivanje vrednosti string pokazivaču koji nije inicijalizovan, kao i dodeljivanje vrednosti stringu koji je manji od stvarne dužine vrednosti.

```

==13225== Invalid read of size 8
==13225== at 0x4989FA8: std::__cxx11::basic_stringchar, std::char_traitschar>, std::allocatorchar> >::M_assign(std::__cxx11::basic_stringchar, std::char_traitschar>, std::allocatorchar> > const& (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.30)
==13225== by 0x498A4AC: std::__cxx11::basic_stringchar, std::char_traitschar>, std::allocatorchar> >::operator=(std::__cxx11::basic_stringchar, std::char_traitschar>, std::allocatorchar> > const& (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.30)
==13225== by 0x1002CD: makeGraph::readFromFile(std::__cxx11::basic_stringchar, std::char_traitschar>, std::allocatorchar> >, std::__cxx11::basic_stringchar, std::char_traitschar>, std::allocatorchar> >, int&, Vector<List&> (makeGraph.h:26)
==13225== by 0x11098B: Graph::Graph(std::__cxx11::basic_stringchar, std::char_traitschar>, std::allocatorchar> >, std::__cxx11::basic_stringchar, std::char_traitschar>, std::allocatorchar> >) (Graph.h:241)
==13225== by 0x10C07F: main (Main.cpp:8)
==13225== Address 0x4de1380 is 16 bytes after a block of size 8 alloc'd
==13225== at 0x484A2F3: operator new[](unsigned long) (in /usr/libexec/valgrind/vgpreload_memcheck-and64-linux.so)
==13225== by 0x1001F2: makeGraph::readFromFile(std::__cxx11::basic_stringchar, std::char_traitschar>, std::allocatorchar> >, std::__cxx11::basic_stringchar, std::char_traitschar>, std::allocatorchar> >, int&, Vector<List&> (makeGraph.h:26)
==13225== by 0x11098B: Graph::Graph(std::__cxx11::basic_stringchar, std::char_traitschar>, std::allocatorchar> >, std::__cxx11::basic_stringchar, std::char_traitschar>, std::allocatorchar> >) (Graph.h:241)
==13225== by 0x10C07F: main (Main.cpp:8)
==13225== Invalid write of size 8
==13225== at 0x4989FA8: std::__cxx11::basic_stringchar, std::char_traitschar>, std::allocatorchar> >::M_assign(std::__cxx11::basic_stringchar, std::char_traitschar>, std::allocatorchar> > const& (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.30)
==13225== by 0x498A4AC: std::__cxx11::basic_stringchar, std::char_traitschar>, std::allocatorchar> >::operator=(std::__cxx11::basic_stringchar, std::char_traitschar>, std::allocatorchar> > const& (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.30)
==13225== by 0x1002CD: makeGraph::readFromFile(std::__cxx11::basic_stringchar, std::char_traitschar>, std::allocatorchar> >, std::__cxx11::basic_stringchar, std::char_traitschar>, std::allocatorchar> >, int&, Vector<List&> (makeGraph.h:26)
==13225== by 0x11098B: Graph::Graph(std::__cxx11::basic_stringchar, std::char_traitschar>, std::allocatorchar> >, std::__cxx11::basic_stringchar, std::char_traitschar>, std::allocatorchar> >) (Graph.h:241)
==13225== by 0x10C07F: main (Main.cpp:8)
==13225== Address 0x4de1380 is 8 bytes after a block of size 8 alloc'd
==13225== at 0x484A2F3: operator new[](unsigned long) (in /usr/libexec/valgrind/vgpreload_memcheck-and64-linux.so)
==13225== by 0x1001F2: makeGraph::readFromFile(std::__cxx11::basic_stringchar, std::char_traitschar>, std::allocatorchar> >, std::__cxx11::basic_stringchar, std::char_traitschar>, std::allocatorchar> >, int&, Vector<List&> (makeGraph.h:26)
==13225== by 0x11098B: Graph::Graph(std::__cxx11::basic_stringchar, std::char_traitschar>, std::allocatorchar> >, std::__cxx11::basic_stringchar, std::char_traitschar>, std::allocatorchar> >) (Graph.h:241)
==13225== by 0x10C07F: main (Main.cpp:8)

```

Slika 6: Poruke o čitanju i pisanju na nevalidnu adresu

Postoje blokovi memorije koji su alocirani, ali nisu oslobođeni do završetka programa. Od alociranih 8 blokova, samo 2 su oslobođena.

```

==13225== HEAP SUMMARY:
==13225==      in use at exit: 73,728 bytes in 6 blocks
==13225==    total heap usage: 8 allocs, 2 frees, 74,672 bytes allocated
==13225==

```

Slika 7: Stanje hipa prilikom izvršavanja

Ako pogledamo rezultate o curenju memorije, zaključujemo da program nema značajnih gubitaka memorije: nema definitivno, indirektno ili potencijalno izgubljenih blokova. Problem blokova koji su i dalje slobodni na kraju programa rešava se dodavanjem destruktora za klase *Graph* i *makeGraph* i funkcije *free* za dva niza alocirana u *readFromFile*: *linija 26* i *linija 27* funkciji koja nigde u kodu nisu oslobođena.

Primenom memcheck alata otkrivene su nedopustive greške u radu sa memorijom. Poruke koje su dobijene ukazuju da je potrebno detaljno analizirati

```

==13225== LEAK SUMMARY:
==13225==    definitely lost: 0 bytes in 0 blocks
==13225==    indirectly lost: 0 bytes in 0 blocks
==13225==    possibly lost: 0 bytes in 0 blocks
==13225==    still reachable: 73,728 bytes in 6 blocks
==13225==                of which reachable via heuristic:
==13225==                    newarray      : 968 bytes in 1 blocks
==13225==    suppressed: 0 bytes in 0 blocks
==13225==
==13225== For lists of detected and suppressed errors, rerun with: -s
==13225== ERROR SUMMARY: 4 errors from 4 contexts (suppressed: 0 from 0)

```

Slika 8: Rezime curenja memorije i detektovanih grešaka

fajlove *Graph.h* i *makeGraph.h*, konkretno funkcije koje učitavaju podatke o letovima i inicijalizuju liste, odnosno vektore i nizove. Prikazane podatke, informacije o linijama koda i pozivima funkcija grešaka nalaze se u fajlu *2025-08-19_23:45:15.memcheck.out* direktorijuma *memcheck*.

4 Cppcheck

4.1 O Cppcheck alatu

Cppcheck je statički analizator koda za C/C++ jezik. Fokusira se na otkrivanje nedefinisanog ponašanja i opasnih konstrukcija koda kao što su dereferenciranje null pokazivača, curenje memorije, neinicijalizovane ili neiskorišćene promenljive i drugi potencijalni sigurnosni problemi. Analiza se vrši nad izvornim fajlovima, pa nije potrebno kompajliranje programa. Prednost cppcheck-a je što proizvodi relativno malo lažnih rezultata u poređenju sa drugim statičkim analizatorima, ali zbog toga može da propusti određene greške. Podrazumevano, cppcheck proverava greške, upozorenja, nedostatke u performansama, stilu i prenosivosti koda.

4.2 Analiza dobijenih rezultata

Cppcheck je prvo pokrenut sa opcijama:

- ***enable=all*** - aktivira sve moguće provere
- ***suppress=missingInclude*** - ignorišu se upozorenja vezana za nedostajuće include fajlove zbog čistijeg izveštaja

tako da prijavljuje samo pouzdane greške i upozorenja, a zatim je dodata i opcija

- ***inconclusive*** - prijavljuju se i potencijalne greške i problemi

Dve otkrivene greške odnose se na funkcije koje nemaju *void* povratnu vrednosti, ali postoji putanja kroz njih koja nema *return value*; naredbu. Ove greške su već otkrivene prilikom pokretanja jediničnih testova. Upozorenja se uglavnom odnose na neinicijalizovana polja u konstruktoru. To je značajno jer mogu dobiti nasumične vrednosti i samim tim kasnije u kodu, kada se referišu, mogu izazvati neočekivane greške i ponašanja. Još jedno važno upozorenje odnosi se na operator dodele u klasi *Vector*, koji ne proverava slučaj samododele, što može dovesti i do pada programa.

Cppcheck report - Airline Reservation System

error warning portability performance style information | cppcheck clang-tidy | File: Filter:

Line	Id	CWE	Severity	Message
15	uninitMemberVar	398	warning	Member variable 'Flight.FlyingTime' is not initialized in the constructor.
15	uninitMemberVar	398	warning	Member variable 'Flight.LandingTime' is not initialized in the constructor.
23	uninitMemberVar	398	warning	Member variable 'costInfo.time' is not initialized in the constructor.
125	missingReturn	758	error	Found a exit path from function with non-void return type that has missing return statement
52	missingReturn	758	error	Found a exit path from function with non-void return type that has missing return statement
64	operatorEqToSelf	398	warning	'operator=' should check for assignment to self to avoid problems with dynamic memory.

Defect summary

Toggle all

Show # Defect ID

- 36 passedByValue
- 8 shadowVariable
- 3 constParameter
- 3 uninitMemberVar
- 2 missingReturn
- 1 noConstructor
- 1 operatorEqToSelf
- 1 unreadVariable
- 1 unusedPrivateFunction
- 1 unusedVariable
- 1 useInitializationList
- 1 variableScope

59 total

Slika 9: Greške i upozorenja cppcheck-a

Poruke vezane za stil i performanse uglavnom su preporuke da se argumenti prosleđuju putem reference umesto vrednosti, da se određene funkcije deklarišu kao *const*, kao i upozorenja da su u istom opsegu definisane dve promenljive sa istim imenom, i da unutrašnja sakriva spoljašnju.

Cppcheck report - Airline Reservation System				
<div> error warning portability performance style information cppcheck clang-tidy File: Filter: </div>				
Defect summary				
<input checked="" type="checkbox"/> Toggle all Show # Defect ID				
<input checked="" type="checkbox"/> 36 passedByValue	298	passedByValue	398	performance
<input checked="" type="checkbox"/> 27 functionConst	298	passedByValue	398	performance
<input checked="" type="checkbox"/> 8 functionStatic	338	shadowVariable	398	style
<input checked="" type="checkbox"/> 8 shadowVariable	376	passedByValue	398	performance
<input checked="" type="checkbox"/> 3 constParameter	376	passedByValue	398	performance
<input checked="" type="checkbox"/> 3 uninitializedMemberVar	376	passedByValue	398	performance
<input checked="" type="checkbox"/> 2 missingReturn	418	shadowVariable	398	style
<input checked="" type="checkbox"/> 1 noConstructor	468	passedByValue	398	performance
<input checked="" type="checkbox"/> 1 operatorEqToSelf	468	passedByValue	398	performance
<input checked="" type="checkbox"/> 1 unreadVariable	501	shadowVariable	398	style
<input checked="" type="checkbox"/> 1 unusedPrivateFunction	537	passedByValue	398	performance
<input checked="" type="checkbox"/> 1 unusedVariable	541	unreadVariable	563	style
<input checked="" type="checkbox"/> 1 useInitializationList	662	passedByValue	398	performance
<input checked="" type="checkbox"/> 1 variableScope	./Airline-Reservation-System/src/List.h			
94 total	27	useInitializationList	398	performance
Statistics	75	functionConst	398	style, inconcl.
	102	functionConst	398	style, inconcl.
	./Airline-Reservation-System/src/ListUI.h			
	8	functionStatic	398	performance, inconcl.
	./Airline-Reservation-System/src/Stack.h			
	23	functionConst	398	style, inconcl.
	23	functionConst	398	style, inconcl.
				Function parameter 'dest' should be passed by const reference.
				Function parameter 'date' should be passed by const reference.
				Local variable 'k' shadows outer variable
				Function parameter 'sour' should be passed by const reference.
				Function parameter 'trans' should be passed by const reference.
				Function parameter 'dest' should be passed by const reference.
				Function parameter 'date' should be passed by const reference.
				Local variable 'k' shadows outer variable
				Function parameter 'src' should be passed by const reference.
				Function parameter 'airline' should be passed by const reference.
				Local variable 'y' shadows outer variable
				Function parameter 'src' should be passed by const reference.
				Variable 'tempDate' is assigned a value that is never used.
				Function parameter 'airline' should be passed by const reference.
				Variable 'source' is assigned in constructor body. Consider performing initialization in initialization list.
				Technically the member function 'List::isEmpty' can be const.
				Technically the member function 'List::getCursor' can be const.
				Technically the member function 'ListUI::displayList' can be static (but you may consider moving to unnamed namespace).
				Technically the member function 'Stack < Flight >::isEmpty' can be const.
				Technically the member function 'Stack < string >::isEmpty' can be const.

Slika 10: Preporuke vezane za stil i performanse

5 Clang-format

5.1 O Clang-u

Clang je open-source kompilator za C familiju jezika. *Clang Tools* predstavljaju samostalne alate komandne linije, dostupne u okviru kompilatora, koji C++ programerima pružaju funkcionalnosti kao što su:

- ***clang-check*** - provera sintaksne i semantičke ispravnosti koda
- ***clang-tidy*** - statička analiza za pronalaženje bagova
- ***clang-format*** - formatiranje koda

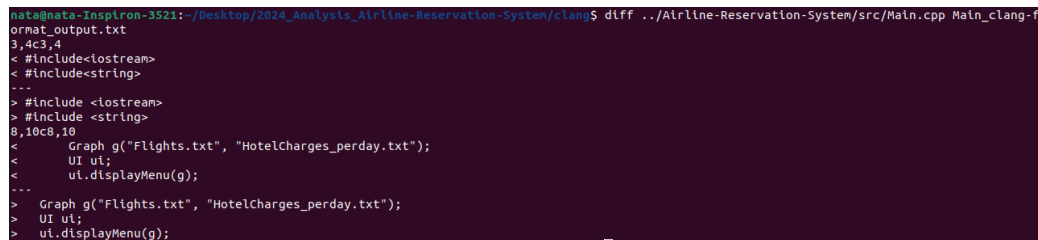
Clang-format je alat za formatiranje koda - formatiraju se razmaci, uvlačenja, raspored zagrada, dužina linija i razni drugi stilski aspekti. Njegovom primenom eliminiše se potreba za ručnim formatiranjem, čime se čuva vreme i trud programera. Obezbeđuje uniforman izgled koda u celom projektu, što poboljšava čitljivost, olakšava održavanje i smanjuje moguće konflikte prilikom spajanja. Formatiranje može biti automatsko na osnovu predefinisanih stilova, a pruža i mogućnost definisanja sopstvenog stila u *.clang-format* fajlu. Takođe, moguće je primeniti izmene direktno u originalne fajlove ili generisati nove sa predloženim izmenama.

5.2 Analiza dobijenih rezultata

U okviru *clang* direktorijuma nalaze se generisani fajlovi sa predloženim izmenama. Korišćen je podrazumevani LLVM stil, koji koristi uvlačenje od dva razmaka (ne podržava tabulatore), maksimalna dužina linije je 80 karaktera, a tela uslova se uvek stavljaju u vitičaste zagrade, čak i kada su jednolinijski izrazi. Pomoću komande *diff* možemo dobiti izveštaj koje linije koda su izmenjene u fajlu i uporedni ispis starog i novog koda.

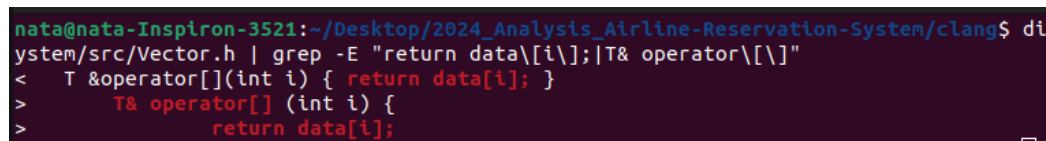
Kada se uporedi sadržaj originalnog i izmenjenog koda, uočava se nizak nivo brige o stilskim pravilima. Ne postoji ujednačeno uvlačenje linija, u velikom broju linija nema potrebnih razmaka, ni usklađenog rasporeda zagrada. Ovi propusti otežavaju razumevanje i održavanje sistema, kao i lociranje prijavljenih grešaka. Sve predložene izmene mogu se pogledati u direktorijumu *clang* pokretanjem komande:

diff imeŽeljenogFajla_clangformat_output.txt ../AirlineReservationSystem/src/imeŽeljenogFajla
za svaki izvorni fajl pojedinačno. Na slikama 11 i 12 mogu se videti rezultati za fajlove *Main.cpp* i *Vector.h*



```
nata@nata-Inspiron-3521: ~/Desktop/2024_Analysis_Airline-Reservation-System/clang$ diff ../Airline-Reservation-System/src/Main.cpp Main_clang-format_output.txt
3,4c3,4
< #include<iostream>
< #include<string>
...
> #include <iostream>
> #include <string>
8,10c8,10
<     Graph g("Flights.txt", "HotelCharges_perday.txt");
<     UI ui;
<     ui.displayMenu(g);
...
> Graph g("Flights.txt", "HotelCharges_perday.txt");
> UI ui;
> ui.displayMenu(g);
```

Slika 11: Primer upotrebe diff komande i njenog rezultata



```
nata@nata-Inspiron-3521: ~/Desktop/2024_Analysis_Airline-Reservation-System/clang$ diff system/src/Vector.h | grep -E "return data\[i\];|T& operator\[\"
< T &operator[](int i) { return data[i]; }
> T& operator[] (int i) {
>     return data[i];
```

Slika 12: Razlika u formatiranju zagrada kod jednolinijskih funkcija

6 Zaključak

Primenom alata otkrivene su brojne greške i nedopustivi propusti. Najznačajnije su greške u radu sa memorijom i inicijalizacije promenljivih i polja. Nekoliko njih izazvalo je iznenadni prekid izvršavanja, a kod pojedinih alata problem je bio i samo kompajliranje. Stilski projekat ne prati nijedno pravilo, što otežava čitljivost i održavanje koda. Takođe, sve funkcije su deklarisanе u header fajlovima, pa je preporuka da se izvrši refaktorisanje fajlova kako bi se odvojila deklaracija od definicije.

Sve ovo ukazuje na potrebu za detaljnim pregledom projekta i izmenama u skladu sa predstavljenom analizom, kako bi se obezbedila njegova stabilnost i efikasnost. Ovim seminarskim radom pokazana je važnost i korisnost alata za verifikaciju softvera, čak i kada program naizgled funkcioniše ispravno.

Literatura

- [1] Milena Vujošević Jančić, *Verifikacija softvera*
- [2] Materijali sa vežbi
- [3] *Clang: a C language family frontend for LLVM*, dostupno na: <https://clang.llvm.org/>
- [4] *Cppcheck*, dostupno na: <https://docs.platformio.org/en/latest/advanced/static-code-analysis/tools/cppcheck.html>
- [5] *Memcheck*, dostupno na: <https://valgrind.org/docs/manual/mc-manual.html>