

UNIVERZITET MB
Poslovni i pravni fakultet
Beograd

I SEMINARSKI RAD

TESTIRANJE SOFTVERA

Predmet: Razvoj softvera

Profesor:

Dr Borivoje Milošević

Student:

TESTIRANJE SOFTVERA

Osnovni koncepti i metodologije

Da bi se razumeo način izrade dobrog softvera, procenjivanje rizika kao i mogućnosti koje softver unosi u naš svakodnevni život, neophodno je čvrsto teorijsko i praktično razumevanje softverskog inženjerstva. Izrada dobrog softvera predstavlja "umetnost" koja se ogleda u razumevanju načina apstrahovanja i modelovanja suštinskih elemenata problema, a zatim korišćenju takvih apstrakcija za projektovanje rešenja.

Dobar softver-inženjerska praksa teži da obezbedi softver koji će dati pozitivan doprinos načinu na koji živimo, jer je u današnje vreme rad softvera prisutan u svim aspektima našeg života.

Da bi se razumelo kako se softversko inženjerstvo uklapa u svet računarske nauke treba objasniti da se računarstvo usredsređuje na računare i programske jezike, a softversko inženjerstvo iste koristi kao alate u projektovanju i primeni rešenja nekog problema. Umesto da istražuje dizajn hardvera, softver inženjer se usredsređuje na računar kao sredstvo za rešavanje problema.

Svaki haker može da napiše programski kod koji nešto radi, ali su potrebni umeće i znanje profesionalnog softverskog inženjera da bi se proizveo stabilan i razumljiv kod koji se lako održava i koji efikasno i efektivno radi ono zbog čega je napravljen. Suština softverskog inženjerstva je u projektovanju i razvoju visoko kvalitetnog softvera. Softversko inženjerstvo je stvaralački postupan proces, koji okuplja veliki broj ljudi na poslovima izrade različitih vrsta proizvoda. Prilikom pružanja usluga ili izrade proizvoda uvek se sledi niz koraka kako bi se izvršio neki skup zadataka. Zadaci se obično izvršavaju u istom redosledu.

Testiranje softvera je proces evaluacije i provere da li softverski proizvod ili aplikacija radi ono što bi trebao raditi. Prednosti testiranja uključuju sprečavanje bagova, smanjenje troškova razvoja i poboljšanje performansi.

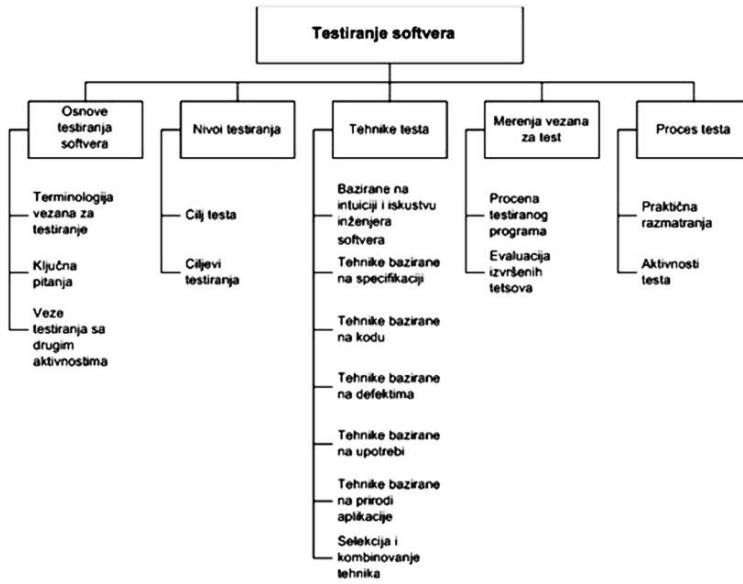
Malo ko može osporiti potrebu za kontrolom kvaliteta prilikom razvoja softvera. Kasna isporuka ili nedostaci softvera mogu naštetići ugledu robne marke - što dovodi do frustriranih i izgubljenih kupaca. U ekstremnim slučajevima, bag ili kvar mogu degradirati međusobno povezane sisteme ili uzrokovati ozbiljne kvarove.

Uzmimo u obzir da Nissan mora povući više od milion automobila zbog softverskog kvara na detektorima senzora vazdušnih jastuka.

Ili, softverska greška koja je uzrokovala neuspeh lansiranja vojnog satelita vrednog 1,2 milijarde USD. Brojke govore same za sebe. Softverski kvarovi u SAD-u koštali su privedu 1,1 bilion dolara u 2016. Štaviše, uticali su na 4,4 milijarde korisnika.

Testiranje softvera je integralni deo procesa razvoja softvera, sa ciljem osiguravanja kvaliteta finalnog proizvoda. U ljudskoj prirodi je da pravimo greške. Neke greške mogu biti sitne i nebitne, dok druge mogu biti veoma ozbiljne i skupe, pa čak i dovesti do ljudskih žrtava.

Podela tema za oblast znanja testiranja softvera je prikazana na slici:



Ne mogu se svi tipovi testiranja primeniti na svaki softverski proizvod, niti je svaki mogući tip naveden. Cilj testiranja i objekat testiranja zajedno utvrđuju kako se identificuje skup testiranja, obazirući se na konzistenciju – koliko testiranja je dovoljno za postizanje željenog cilja – i svoju kompoziciju – koje slučajevе testa bi trebalo selektovati za postizanje željenog cilja.

Kriterijumi za prvo pitanje se nazivaju kriterijumi adekvatnosti testa, dok se oni koji se odnose na drugo pitanje nazivaju kriterijumi selekcije testa. Više tehnika za testiranje (test techniques) je razvijeno proteklih nekoliko decenija, i dalje se predlažu nove.

U dokumentu Gelperina i Hetzela iz 1988. godine pominje se podela istorije testiranja na sledeće faze:

Perod od	Period do	Faza	Opis
-	1956	<i>Debagovanje</i>	Programe je pisao, a zatim i testirao, programer, do trenutka uklanjanja svih bugova. Testovi su kreirani ad hoc, na osnovu iskustva programera i fokusirani na operativnost sistema. Ne postoji jasna granica između testiranja i razvoja.
1957	1978	<i>Demonstracija</i>	Debagovanje se i dalje izvodilo kako bi sistem proradio, ali je dodat još jedan nivo testiranja - da se demonstrira da softver ne samo radi, već da radi ono što treba.
1979	1983	<i>Destrukcija</i>	Fokus testiranja se pomera sa obezbeđenja da softver radi ono što treba, na otkrivanje grešaka u sistemu.
1983	1987	<i>Evaluacija</i>	Testiranje postaje aktivnost na kraju svakog nivoa u razvoju životnog ciklusa, jer se prihvata stanovište da što ranije otkrivanje grešaka u sistemu izaziva manje troškove.
1988	-	<i>Prevencija</i>	Ideja je da se spreče greške u bilo kom stadijumu životnog ciklusa proizvoda testiranjem svakog nivoa. Testiranje se fokusira na ispravljanje grešaka.

Čuveni primeri softverskih otkaza:

Značaj testiranja softvera je možda najlakše prikazati na konkretnim primerima softverskih otkaza, kada testiranje softvera nije izvršeno na adekvatan način i kada greške nisu otkrivene na vreme. U slučaju industrijskog softvera, ove greške često dovode do velikih finansijskih gubitaka, a ponekad, na žalost, i do ljudskih žrtava. Čak i u najpovoljnijem slučaju, greške koje su prisutne u finalnom isporučenom softveru dovode do smanjivanja zadovoljstva krajnjih korisnika, što kao posledicu može imati gubitak poverenja u dati sistem i okretanje alternativnim rešenjima.

1. U januaru 1990. godine, nijedan korisnik američke telekomunikacione kompanije AT&T nije mogao da uspostavi pozive na velikim rastojanjima, zbog softverske greške na relejnim svičevima.
2. NASA - Mars Climate Orbiter misija je doživela ogroman neuspeh 1999. godine. U okviru svoje misije na Marsu, zbog greške u softveru letelica je nepovratno izgubljena u svemiru. Nakon duže istrage koja je usledila, otkriveno je da je napravljena greška pri konverziji imperijalnih jedinica u metrički sistem.
3. Evropska Svemirska Agencija je, poput NASA, takođe je imala svoje trenutke, od kojih se izdvaja misija Ariane 5 let 501. Samo 40 sekundi nakon lansiranja 1996. godine, letelica vredna 370 miliona \$ se raspala zbog softverske greške. Problem je izazvalo prekoračenje u konverziji float -> int. Inercijalni referentni sistem je radio sa 64-bitnim podacima tipa float, koje je konvertovao u 16-bitni integer, pri čemu je izazvan aritmetički overflow – šteta 10 milijardi \$.

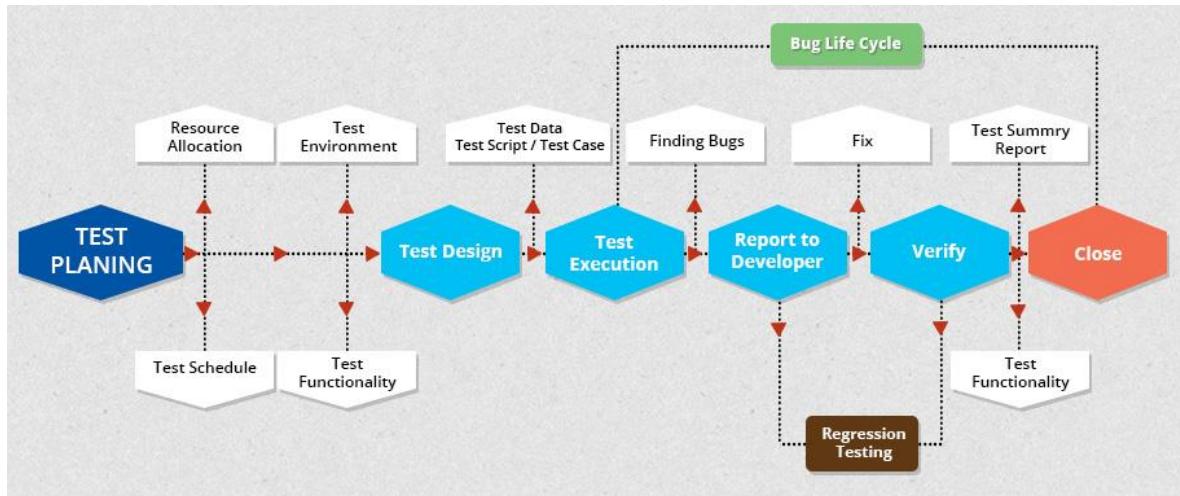
Tipovi testiranja

Inspekcije, recenzije i pregledi su specifcne tehnike fokusirane na ocenjivanje artefakata (pogodno za dokumentaciju, programski kod) i efikasne su metode za poboljšanje kvaliteta i razvojnog procesa proizvoda. Sprovode se na sastancima, gde jedan od učesnika ima ulogu vodećeg, a ostali uloge zapisičar a(beleži pitanja, predloge, probleme i sl.). Ove tehnike opisuju se u okviru IEEE standarda na sledeći nacin:

Recenzija (review) – formalni stastanak na kome se artefakt ili skup artefakata predstavlja korisniku ili drugim stranama koje učestvuju u procesu odobravanja ili komentarisanja.

Inspekcija (inspection) – formalna tehnika procene u kojoj se artefakti detaljno pregledaju. Pregled vrše osobe koje nisu autori, da bi se lakše uočile greške i drugi problemi.

Pregledi (walkthrough) – Tehnika pregleda u kojoj autor upoznaje ostale članove tima sa elementima artefakta koji je izgradio. Ostali članovi učestvuju aktivno u raspravi.



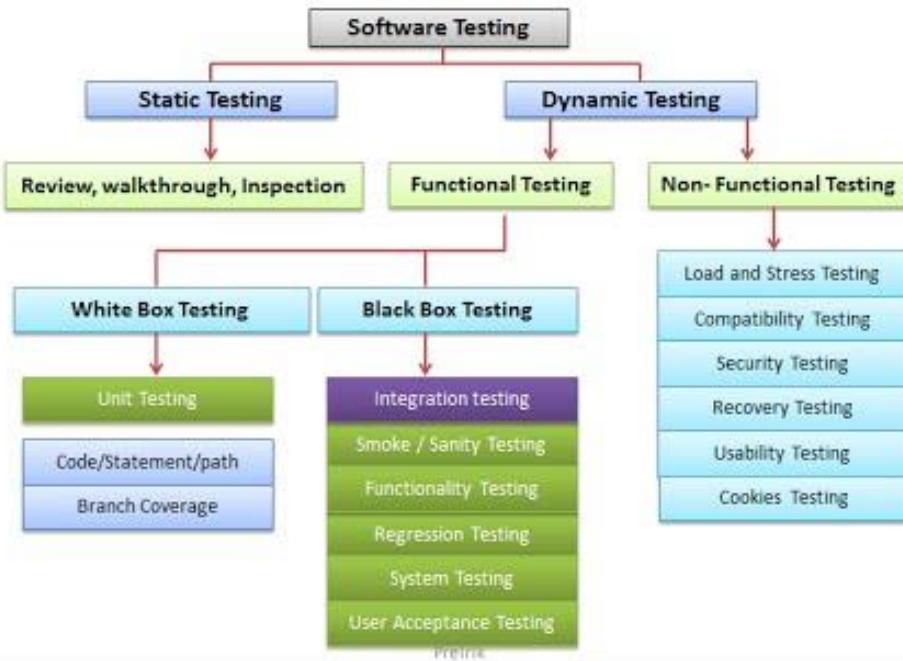
Tehnike testiranja

1. **Jedinično testiranje (unit testing)** – primenjuje se na pojedine klase, module ili komponente programskog koda. Ova tehnika deli se na tehnike bele i crne kutije.
2. **Integraciono testiranje** – primenjuje se na softverski sistem kao celinu.
3. **U testiranja višeg reda spadaju:** testiranje sigurnosti (security testing), testiranje kolicine podataka, testiranje upotrebljivosti, testiranje integriteta (integrity testing), test u stresnim uslovima(stress testing), etalonski test, test zagušenja, test opterecenja, test konfiguracije (configuration testing), test instalacije (installation testing)
4. **Regresiono testiranje** – na osnovu jednom razvijenog testa više puta se sprovodi testiranje softvera (tipično nakon neke izmene u softveru da bi se utvrdilo da nisu pokvarene funkcionalnosti softvera).

Česta je i podela testiranja na:

1. black-box, white-box i gray-box testiranje,
2. manuelno i automatsko testiranje,
3. staticko i dinamicko testiranje.

Types of Software Testing:



Osnovna klasifikacija testiranja:

Prema pristupu, testiranje se deli na

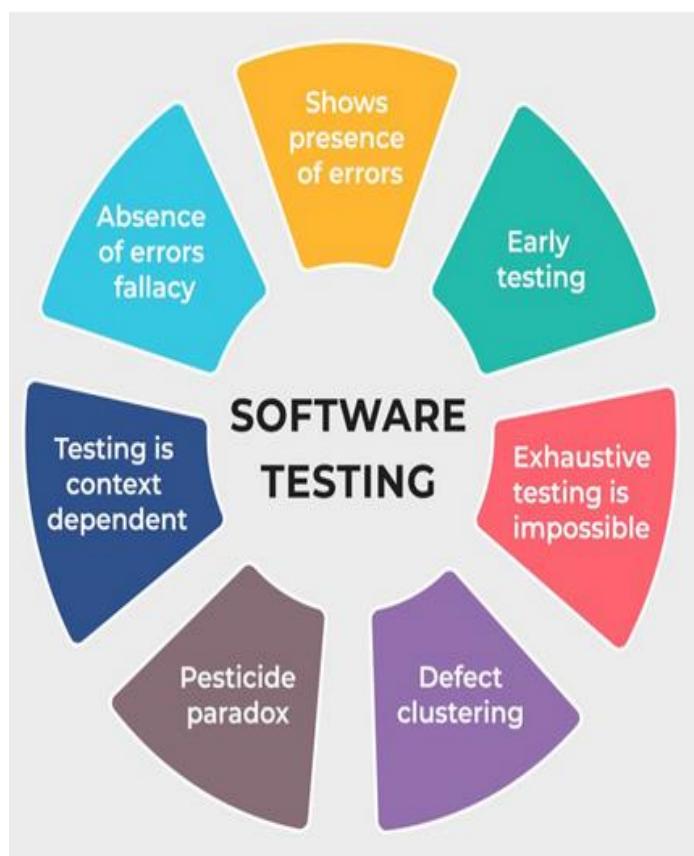
Funkcionalno – testiranje bazirano na specifikaciji	Funkcionalno testiranje posmatra program kao crnu kutiju, i implementacija u ovom slučaju nije poznata. Naziva se još metode crne kutije. Softver se prosto posmatra kao funkcija koja mapira vrednosti sa ulaza na izlaz sistema. Testovi se određuju isključivo na osnovu specifikacije softvera.
Strukturno – testiranje bazirano na samom kodu	Strukturno testiranje se fokusira na samoj implementaciji programa i raspoloživom dostupnom kodu. Naziva se još metode bele kutije . Fokus je na izvršavanju svih programske strukture i struktura podataka u softveru koji se testira, i na osnovu toga se određuju testovi.

Testiranje se prema nivou deli na:

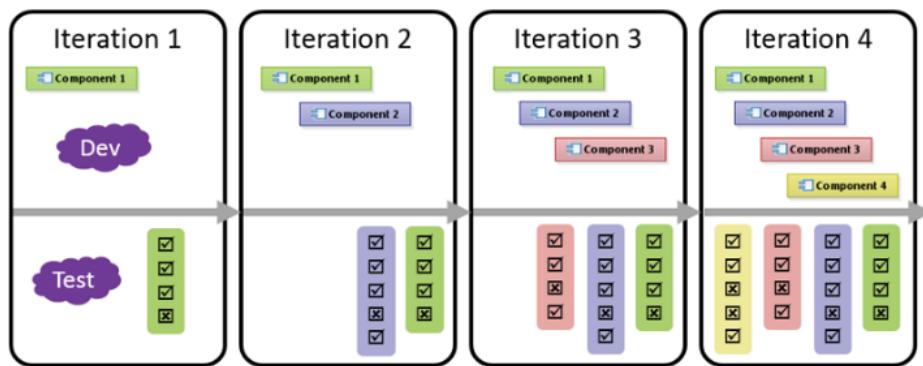
Jedinično	Sprovode programeri (engl. unit testing) i odnosi se na testiranje pojedinačnih jedinica izvornog koda ili delova klase. Najmanja funkcionalna jedinica izvornog koda je najčešće jedna metoda unutar klase.
Integraciono	Glavni fokus je na verifikaciji funkcionalnosti i interfejsa između integrisanih modula.
Sistemsko	Proverava ponašanje tog sistema kao celine u odnosu na specifikaciju sistema.

Postoje sedam osnovnih principa testiranja:

1. Testiranje pokazuje prisustvo defekata – testiranje može da pokaže da su defekti prisutni, ali ne može da dokaže da u sistemu nema nijednog defekta.
2. Iscrpno testiranje nije moguće – testiranje svih mogućih kombinacija ulaza i preduslova u praksi nije moguće za bilo koji netrivijalni sistem. Radi se analiza rizika.
3. Rano testiranje – kako bi se što pre otkrili defekti, testiranje treba da počne što je pre moguće.
4. Grupisanje defekata (engl. *defect clustering*) – testiranje treba fokusirati proporcionalno broju očekivanih i kasnije pronađenih defekata (gustini defekata) po modulima.
5. Paradoks pesticida – ukoliko se isti testovi ponavljaju u svakoj iteraciji testiranja, na kraju taj isti skup testova više neće moći da pronađe nijedan novi defekt.
6. Testiranje zavisi od konteksta. Testiranje se različito izvršava u različitim kontekstima.
7. Odsustvo grešaka ne garantuje da sistem radi kako treba – pronalaženje i ispravljanje defekata ne pomaže u slučaju da je sistem neupotrebljiv, ili da ne ispunjava želje i očekivanja korisnika.



Testiranje efikasnosti je kritičan aspekt za praćenje bržeg tempa razvoja životnih ciklusa. Optimizira se pokretanjem najmanjeg broja testova koji pronalaze najveći broj problema. Čak i timovi radeći u tradicionalnijim razvojnim životnim ciklusima, gde se ispitivanje sprovodi u jednoj fazi, utvrdili su da oni ne mogu pratiti regresijsko testiranje svaki put kad dobiju novi zadatak — sa popravcima nedostataka, promenama postojećih karakteristika, pa čak nova funkcionalnost sve u paketu u novu verziju. Slika pokazuje da se nakon samo nekoliko iteracija kvantitativno značajno povećao broj novih postavki, a time i broj testova.

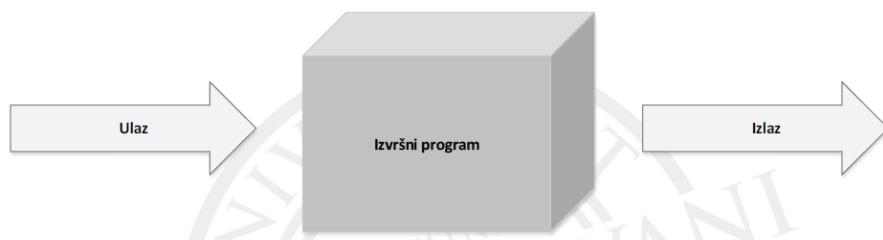


TESTIRANJE METODAMA CRNE KUTIJE:

Funkcionalno testiranje, poznato takođe pod imenom testiranje metodama crne kutije, jeste oblik testiranja softvera gde unutrašnja struktura, dizajn i implementacija softvera ili softverske jedinice koja se testira nisu poznati testeru. Softver se posmatra kao funkcija koja mapira vrednosti iz ulaznog domena sistema u izlazni. Kod ove metode vidljivi su samo ulazi i izlazi, a funkcionalnost je određena promatranjem dobijenih izlaznih podataka na temelju odgovarajućih poznatih ulaza. Prilikom testiranja, na osnovu različitih ulaznih podataka dobijeni izlazni podaci se upoređuju s unapred očekivanim te se na taj način vrši vrednovanje ispravnosti programa.

Pristup crne kutije je metoda testiranja u kojoj se dolazi do testnih podataka na temelju specificiranih funkcijskih zahteva, ne uzimajući u obzir konačnu strukturu programa. Ovo testiranje naziva se još i testiranje upravljanje podacima (engl. data driven), ulazno/izlazno testiranje (engl. input/output driven) ili testiranje zasnovano na zahtevima (engl. requirements based).

Pošto implementacija i struktura programa nisu poznati, program se posmatra kao crna kutija (odатле потиче и име). Jedini izvor informacija koja se koristi za određivanje i dizajn testova jeste specifikacija zahteva programa.



Podela ulaznih podataka na klase ekvivalencije je jedna od osnovnih metoda crne kutije. Može se primeniti na bilo kom nivou testiranja, i najčešće je dobar izbor odakle početi testiranje. Glavna ideja kod ove tehnike je da se ulazni domen podataka podeli na takav način da se program ponaša na isti način za sve ulazne vrednosti koje pripadaju istoj klasi ekvivalencije. Drugim rečima, sistem na isti način obrađuje sve vrednosti koje pripadaju istoj klasi ekvivalencije. Za svaki zakon se razmatraju dve grupe klasa prema zadovoljenosti zakona.

TESTIRANJE METODAMA BELE KUTIJE:

Strukturno testiranje (testiranje vođeno logikom (engl. logic driven testing) ili testiranje zasnovano na dizajnu (engl. design-based testing)), još poznato i kao testiranje metodama bele ili staklene kutije. To je tehnika testiranja u kojoj je stvarna implementacija softvera poznata testerima. Dok se u tehnikama crne kutije tester fokusira na to šta sistem radi, u metodama bele kutije fokus je na tome kako sistem radi.

Tehnike testiranja bele kutije su:

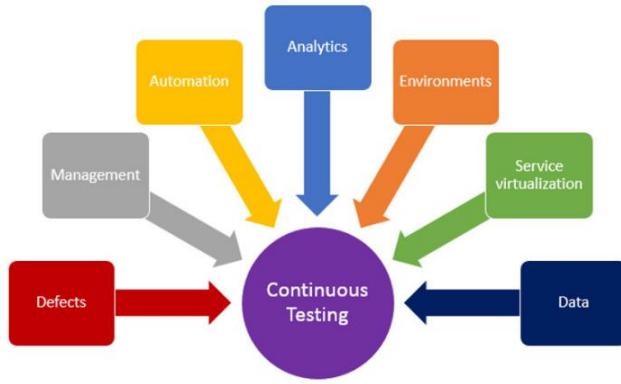
1. Pokrivanje iskaza (Statement coverage)
2. Pokrivanje odluka (Decision coverage)
3. Pokrivanje zakona
4. Metoda graničnog testiranja

Cilj testiranja je da se izvrše sve programske strukture, kao i strukture podataka u programu. Specifikacija se ne proverava prilikom primene metoda bele kutije, već se posmatra samo kod. Strukturno testiranje se može primeniti na svim nivoima testiranja (jedinično, integraciono, sistemsko), i najčešće se primenjuje nakon metoda crne kutije, koje su bazirane na specifikaciji i verifikuju funkciju sistema.

Sami programeri često koriste strukturno testiranje na jediničnom i integracionom nivou, pošto se očekuje da su komponente koje su razvili temeljno testirane pre nego što se obavi dalja integracija tih komponenti u sistem i isporuči testerima na testiranje. Metode bele kutije su značajne i zbog toga što pomažu u određivanju mere temeljnosti testiranja, kroz procenu pokrivenosti programskog koda.

Elementi kontinualnog testiranja:

Kultura izgradnja kontinualnog testiranja zahteva ljude, vežbu, sredstva, alate i vreme. Pronalaženje prave ravnoteže odluka u svim praksama testiranja ključno je za postizanje kontinualnog testiranja. Ali pre nego počnemo implementirati kod za testiranje, jedna aktivnost koja se često zanemaruje je pregled koda. Sve što će biti uključeno u izgradnju i korišćenje tokom implementacije u okruženje (testiranje ili proizvodnja) treba pregledati tim stručnjaka okupljenih u tu svrhu. Pregledi koda moraju se efikasno sprovoditi i smatrati ih kvalitetnim; inače, tim ih može smatrati gubljenjem vremena. Recenzija procesa treba da potvrди da novi i modifikovani softver sledi standarde organizacije i pridržava se njihove najbolje prakse.



ALATI ZA TESTIRANJE SOFTVERA:

Pored raznih tehnika i metoda testiranja softvera razvijeni su i mnogi alati koji olakšavaju ili potpuno automatizuju proces testiranja. Gotovo da je nemoguće učiniti proces testiranja jasno definisanim i ponovljivim bez prednosti koje alati za testiranje nude.

Osnovna podela testiranja podrazumeva manuelno i automatsko testiranje. Naravno, manuelno testiranje se ne odigrava u potpunosti ručno. Oba oblika testiranja koriste alate koji omogućavaju efikasnije testiranje.

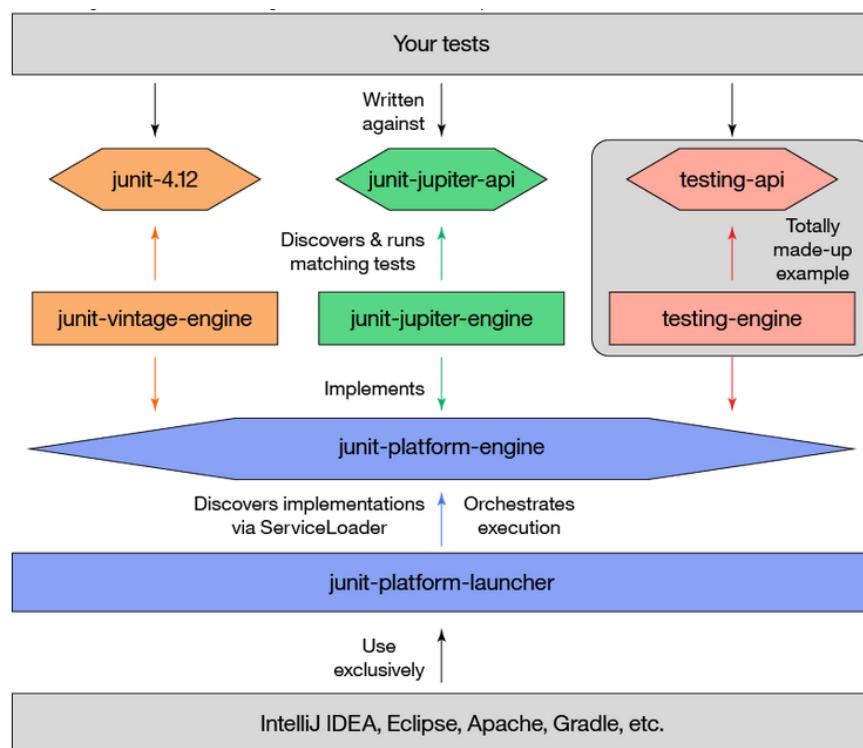
Razvoj automatskih alata za testiranje je reakcija na napredovanje web baziranih, klijent/server i alata za razvoj softvera koji su omogućili ubrzani razvoj aplikacija sa znatno boljim funkcionalnostima. Sektori za testiranje dolaze u dodir sa softverom koji je jako unapređen, ali i kompleksan. Novi alati za testiranje se razvijaju kao podrška procesu obezbeđenja kvaliteta.

Firma **RationalRose** nudi vrlo razradjeno rešenje automatizacije procesa testiranja u vidu niza alata različitih namena, koji sarađuju međusobno i sa ostalim razvojnim alatima za druge faze razvojnog ciklusa softvera:

NAZIV	DELOVANJE
Rational Administrator	- služi za administraciju repozitorijuma svih artefakata koji se odnose na projekat i aktivnost testiranja, pravljenje, ažuriranje projekata, dodela prava pristupa pojedinim članovima tima itd.
Rational TestManager	- prvenstveno se koristi za organizaciju i upravljanje procesom testiranja, pravljenje test planova, dizajn testova i evaluaciju rezultata testiranja.
Rational Robot	- implementacija testa kroz automatizovano funkcionalno i performansno testiranje zasnovano na test skriptovima, sa funkcijom snimanja korisnikovih akcija nad aplikacijom i autogenerisanja skripta.
Rational TestFactory	- analiziranje strukture aplikacija (prvenstveno GUI, Java, C++, VB) radi podrške implementacije testa kroz automatizovano generisanje test skriptova.
Rational QualityArchitect	- specijalizovan za testiranje komponentnih tehnologija srednjeg sloja, EJB i COM.
Rational ManualTest	- alat za implementaciju manuelnog testa.
Rational SiteCheck	- alat za ispitivanje i kontrolisanje strukture i funkcije web aplikacija.
Rational Purify	- detektuje greške u izvršavanju i curenje (leakage) memorije.
Rational Quantify	- sakupljanje i analiza podataka o performansama
Rational PureCoverage	- podaci o pokrivenosti koda (white box testiranje).

JUNIT FRAMEWORK ALAT:

- JUnit je mali, ali moćan Java *framework* za kreiranje i izvršavanje automatskih jediničnih testova u programskom jeziku Java. Jedinično testiranje podrazumeva da se testira mali deo programa – metoda, klasa ili nekoliko manjih klasa koje čine jednu komponentu i slično, u izolaciji od drugih delova koda. Na ovaj način se brzo može uočiti ukoliko nešto nije u redu sa kodom.
- JUnit se često koristi u klasičnom testiranju, kao sredstvo za automatizaciju testiranja. Sa druge strane, JUnit ima ključnu ulogu u *test-driven razvoju* programa, i promoviše ideju *prvo testiraj pa onda kodiraj*, gde se podrazumeva da programer paralelno sa pisanjem koda piše i jedinične testove.
- Ovaj pristup se zasniva na ideji da se pre programiranja određene jedinice napiše jedinični test koji će verifikovati da li je ta jedinica ispravno programirana. Na ovaj način se programiranje svodi na naizmenične cikluse pisanja jediničnih testova i programiranja samih jedinica, što kao rezultat obično ima povećanje produktivnosti programera i stabilnosti programskega koda, i smanjenje vremena koje se kasnije troši na dibagovanje programa.



Cross-browser testing

Tehnologija se neprestano razvija i poboljšava. Uz poboljšanja brzine i alata za Javascript, isporuka Web aplikacija eksponencijalno se ubrzala. Tržište brauzera i uredjaja takodje raste iz dana u dan. Prošli su dani kada su programeri morali brinuti o Web stranicama samo na Internet Exploreru. Sada moramo zadovoljiti veliki ekosistem u smislu broja uredjaja i brauzera na tržištu. To znači da programeri moraju osigurati da njihova aplikacija besprekorno radi na svim brauzerima na tržištu. Kompatibilnost brauzera takodje gradi poverenje kod vaših korisnika jer vaša Web stranica izgleda dobro i dobro radi na svim uredjajima. U startu se postavljaju pitanja:

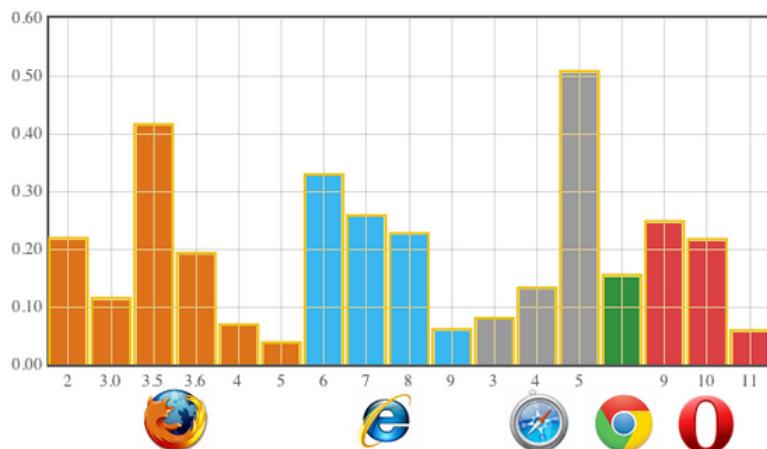
Zašto testiranje na više brauzera? Testiranje naše aplikacije na više brauzera na različitim uredjajima pomoći će nam da:

1. Izvršimo validaciju koda: prijavljuju li neki brauzeri pogreške koda?
2. Utvrdimo da li je web stranica spora ili čak uzrokuje padove?
3. Responzivan dizajn: je li dizajn dosledno responzivan?
4. Utvrdimo nedoslednosti korisničkog sinterfejsa: postoje li još neki nedostaci u dizajnu?
5. Pronadjemo probleme specifične za brauzer
6. Učinimo svoju web aplikaciju kompatibilnom sa brauzerima na različitim uredjajima
7. Pronadjemo uticaje brauzera na korisničko iskustvo i popravimo ih.

Ko sprovodi testiranje na više brauzera?

Testiranje na više brauzera uglavnom provodi inženjer za osiguranje kvaliteta (QA), kada mu je posao dodeljen za testiranje. Takodje se testira nakon što se proizvod podvrgne testiranju prihvatanja od strane korisnika (UAT), kada vlasnici proizvoda ili poslovni analitičari pregledavaju Web mesto.

Za svaku pogrešku bila bi potrebna stvarna istraga da se otkrije šta ju je uzrokovalo, a imamo ih na hiljade. Ali naš kod i kod naših korisnika nezavisni su od brauzera koji se testira (što je cela poenta i Sauce OnDemand i Seleniuma), pa ako pogledamo samo relativne stope pogrešaka prema brauzeru, možemo videti koji su brauzeri najmanje pouzdani.



Stope pogrešaka (procenat) po brauzeru i verziji

DIBAGERI:

Dibageri daju napredne funkcije kao što su pokretanje programa korak po korak (single-stepping), zaustavljanje ili pauziranje izvođenja programa na takozvanom *breakpointu* tokom određenih zbivanja, a neki čak i mogućnost menjanja programa dok se izvodi. Iste funkcije koje čine debugger korisnim za rješavanje bugova čine ga i pomagalom pri razbijanju softverske zaštite, tj. krakiranju program koji nije besplatan u svrhe da se može koristiti neograničeno bez ikakvog plaćanja ili kupovanja. Korisni su i za testiranje performansi programa. Pojedini dibageri rade samo sa specifičnim programskim jezikom dok drugi mogu raditi sa više njih.

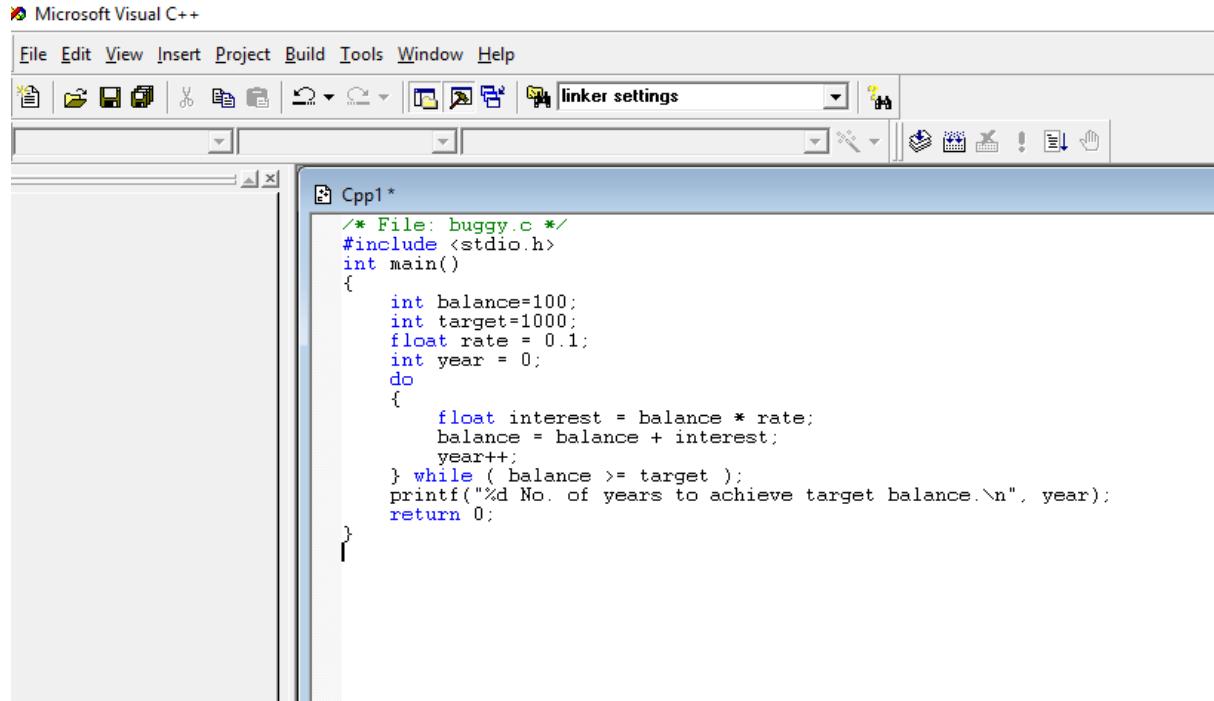
Većina popularnih dibagera daje samo jednostavni komandnolinjski interfejs (command-line interface - CLI), često iz razloga da maksimiziraju portabilnost i minimaliziraju trošenje sistemskih resursa računara. Ipak, popravljanje grešaka u programu preko grafičkog interfejsa (GUI) dibagera se često smatra jednostavnijim i produktivnijim.

Neki od poznatih debuggera su:

- OllyDbg
- SoftICE
- Java Platform Debugger Architecture
- CodeView
- WinDbg
- Eclipse debugger
- Microsoft Visual Studio Debugger

Analiza grešaka u C++ primeru koda

Ovaj odjeljak koristi primer TextFindera za ilustraciju otklanjanja grešaka u Qt C++ aplikacijama u Debug modu. TextFinder čita tekstualnu datoteku u QString i zatim je prikazuje s QTextEdit. Da bi pogledali klasu TextFinder i videli zapamćenjene podatke, postavite prelomnu tačku u buggy.c, kako sledi:



The screenshot shows the Microsoft Visual Studio IDE interface. The title bar says "Microsoft Visual C++". The menu bar includes File, Edit, View, Insert, Project, Build, Tools, Window, Help. The toolbar has various icons for file operations. A tab labeled "linker settings" is visible. The main code editor window is titled "Cpp1*" and contains the following C code:

```
/* File: buggy.c */
#include <stdio.h>
int main()
{
    int balance=100;
    int target=1000;
    float rate = 0.1;
    int year = 0;
    do
    {
        float interest = balance * rate;
        balance = balance + interest;
        year++;
    } while ( balance >= target );
    printf("%d No. of years to achieve target balance.\n", year);
    return 0;
}
```

Izvršimo program i zapamtimo ga na odgovarajućem folderu. U izveštajnom prozoru dobijamo posle kompilacije:

```
-----Configuration: buggy - Win32 Debug-----
Compiling...buggy.c
c:\temp\buggy.c(7) : warning C4305: 'initializing' : truncation from 'const double' to 'float'
c:\temp\buggy.c(12) : warning C4244: '=' : conversion from 'float' to 'int', possible loss of
data
buggy.obj - 0 error(s), 2 warning(s)
```

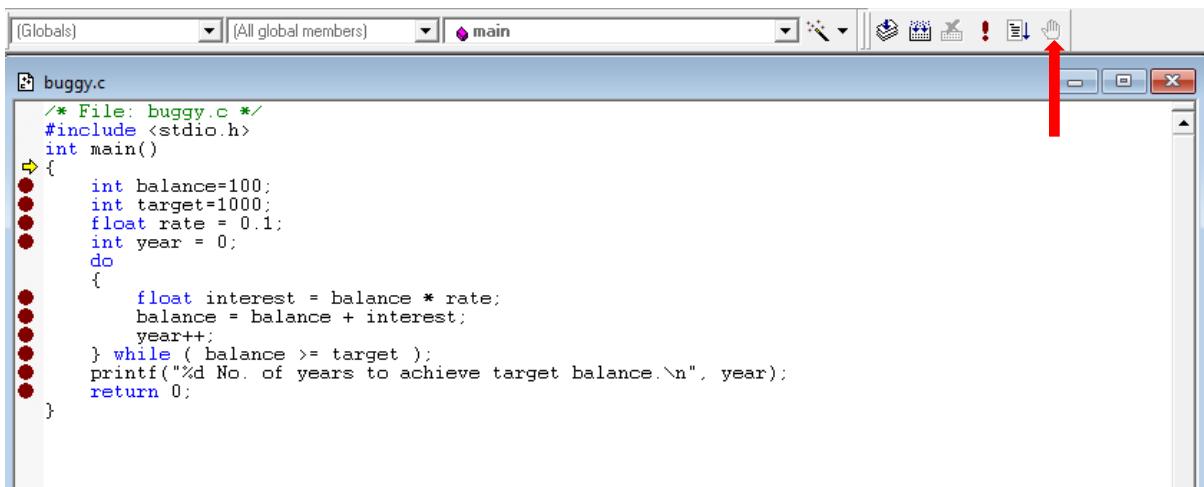
U izveštajnom prozoru dobijamo posle build – izvršenja programa:

```
-----Configuration: buggy - Win32 Debug-----
Linking...
buggy.exe - 0 error(s), 0 warning(s)
```

I na kraju u posebnom prozoru izveštaj izvršenja programa koji je bez grašaka i upozorenja:

```
"C:\Temp\Debug\buggy.exe"
1 No. of years to achieve target balance.
Press any key to continue
```

Da bi pogledali klasu TextFinder i videli zapamćenjene podatke, postavite prelomnu tačku u buggy.c, kako sledi. Kliknite u meniju na opciju insert/delete breakpoint i lociramo se na na redove gde menjamo položaj pokazivača kako bi postavili prelomne tačke.



Odaberite Debug > Start Debugging > Start Debugging of Startup Project ili pritisnite F5. Za prikaz informacija o breakpoint idite na prikaz Debug view. Dobijamo:

```
Loaded 'C:\Windows\SysWOW64\KernelBase.dll', no matching symbolic information found.
The program 'C:\Temp\Debug\buggy.exe' has exited with code 0 (0x0).
Loaded 'APP01.EXE', no matching symbolic information found.
Loaded 'wow64_image_section', no matching symbolic information found.
Loaded 'C:\Windows\SysWOW64\kernel32.dll', no matching symbolic information found.
Loaded 'C:\Windows\SysWOW64\KernelBase.dll', no matching symbolic information found.
```

Ako izaberemo nove tačke – breakpoint na linijama while i printf i ponovo izvršimo Debug/Go opciju dobijamo:

The screenshot shows a debugger interface with the following details:

- File:** buggy.c
- Breakpoints:** Two breakpoints are set: one at the start of the `while` loop and another at the end of the `main` function.
- Code:**

```
/* File: buggy.c */
#include <stdio.h>
int main()
{
    int balance=100;
    int target=1000;
    float rate = 0.1;
    int year = 0;
    do
    {
        float interest = balance * rate;
        balance = balance + interest;
        year++;
    } while ( balance >= target );
    printf("%d No. of years to achieve target balance.\n", year);
    return 0;
}
```
- Watch Window:** Shows the current values of variables:

Name	Value
balance	110
target	1000
year	1

- Status Bar:** Displays messages about loaded DLLs:
 - Loaded 'wow64_image_section', no matching symbolic information found.
 - Loaded 'C:\Windows\SysWOW64\kernel32.dll', no matching symbolic information found.
 - Loaded 'C:\Windows\SysWOW64\KernelBase.dll', no matching symbolic information found.

Program će dati izveštaj bez greške.

Šta je Debugger - GDB?

GDB ili GNU Debugger je GNU projekat koji pomaže u otklanjanju pogrešaka softverskih aplikacija i analizi onoga što se dogadja tokom izvodjenja programa. Pomaže u:

- istražiti nekorektno ponašanje našeg programa.
- pronaći uzrok logičke pogreške koju je teško pronaći samo gledanjem izvornog koda.
- analizira pad programa koji se dogodio u našoj aplikaciji.

Kako instalirati GDB?

U ovom slučaju koristićemo postojeću onlajn aplikaciju koja se nalazi na linku:

https://www.onlinegdb.com/online_c++_debugger

Šta nudi GDB help?

GDB nam daje kontrolu nad izvodjenjem programa. Možemo:

- zaustaviti/pauzirati izvršavanje programa odredjivanjem usova
- kada je naš program pauziran, možemo analizirati stanje programa, proveriti vrednosti varijabli i registara
- možemo promeniti vrednosti varijabli/registra kako bi videli uticaj te promene na ponašanje programa i to možemo učiniti bez menjanja izvornog koda
- Dakle, GDB nije neka čarolija koja će automatski debagovati vaš program, već je njegovo oružje koje možete koristiti da otkrijete uzrok greške.

Podržani programski jezici

GDB podržava C, C++, D, Go, Objective-C, Fortran, Java, OpenCL C, Pascal, Assembly, Modula-2, i Ada jezike. Now you would have brief idea about what is GDB and how it can be your savior when something goes wrong in your program. Lets goto [next](#) section which describes [how to install GDB](#).

Šta možemo videti kada je program pauziran?

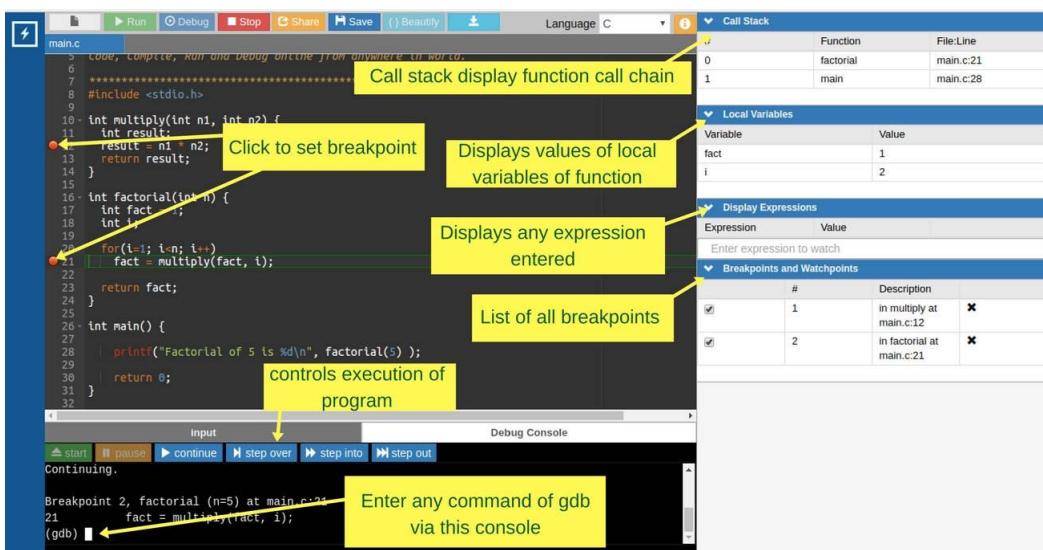
Možemo videti call stack a i vrednosti lokalnih varijabli trenutne funkcije. Call stack prikazuje lanac poziva funkcija. Kao što se vidi na donjoj slici, "glavna" funkcija poziva funkciju "factorial", a " factorial " poziva funkciju "multiplay".

#	Function	File:Line
0	multiply	main.c:12
1	factorial	main.c:21
2	main	main.c:28

Prozor lokalnih varijabli prikazuje vam vrednosti lokalnih varijabli trenutne funkcije. Kao što je prikazano na slici ispod, vrijednost "fact" je 1, "i" je 2.

Variable	Value
fact	1
i	2

Ovdje je jedna slika koja prikazuje korištenje OnlineGDB debuggera.

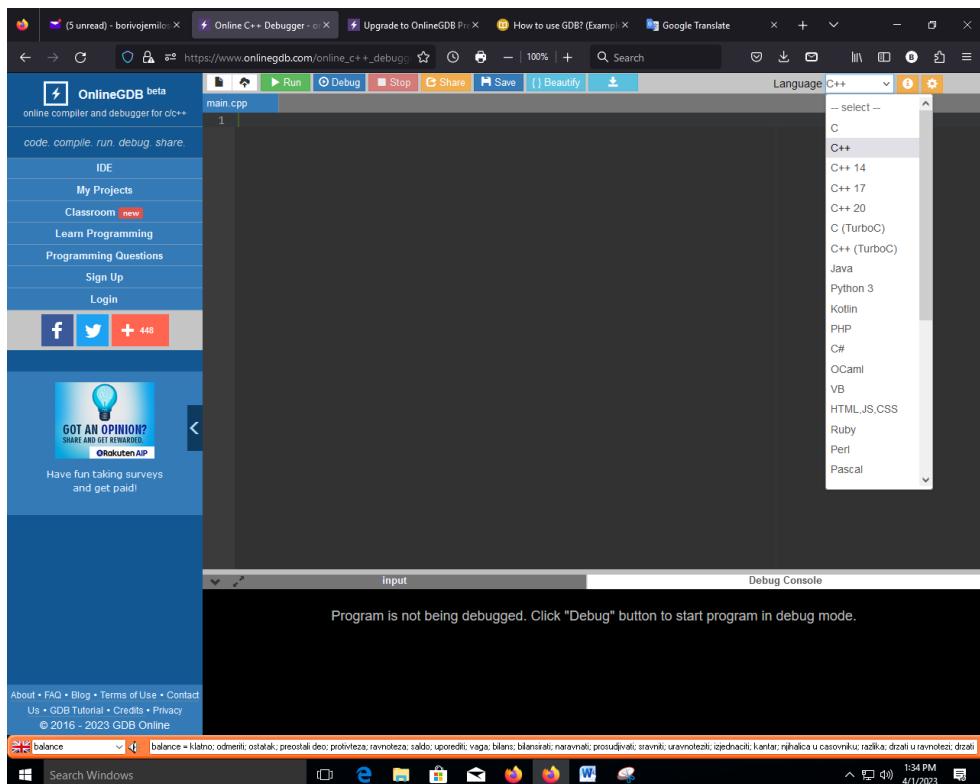


Sada imamo kontrolu i nadzor nad izvođenjem programa, tako da možemo pregledati tok izvođenja programa. Prelomne tačke, koračne naredbe i prikazi varijabli osnovne su osobine debuggera.

Primer kako koristiti Online GDB debugger:

GDB je kratica za "Gnu Debugger". Ovo je moćan paket za otklanjanje grešaka na nivou izvora koji nam omogućuje da vidimo šta se dogadja unutar našeg programa. Debugger je pomoći program koji pokreće ciljni program u kontrolisanom okruženju gde možemo kontrolisati izvodjenje programa i videti stanje programa kada je program pauziran. GDB je takav debugger koji se koristi za debug naprimer našeg C/C++ programa. OnlineGDB pruža interfejs za korišćenje GDB-a u Cloud - oblaku iz našeg brouzera. Ovaj odeljak će pokazati kako koristiti GDB naredbe prolazeći kroz primer. Ovde je prikazan program koji ima greške u skriptu. Ovaj program je napisan za izračunavanje broja godina potrebnih za postizanje ciljanog salda pod uslovom trenutnog salda i godišnje kamatne stope. Ali on daje 1 godinu kao rezultat za bilo koju vrednost tekućeg balansa i interesne stope.

Pokretanjem linka: https://www.onlinegdb.com/online_c++_debugger dobijamo editorski prozor debugger_a i u njega pastujemo program koji ''možda'' sadrži greške:



U padajućem meniju, iz mnoštva programskega jezika na raspolaganje za debbuging, izabratemo programski jezik C++

Koristimo sledeće komande za najčešće gdb operacije.

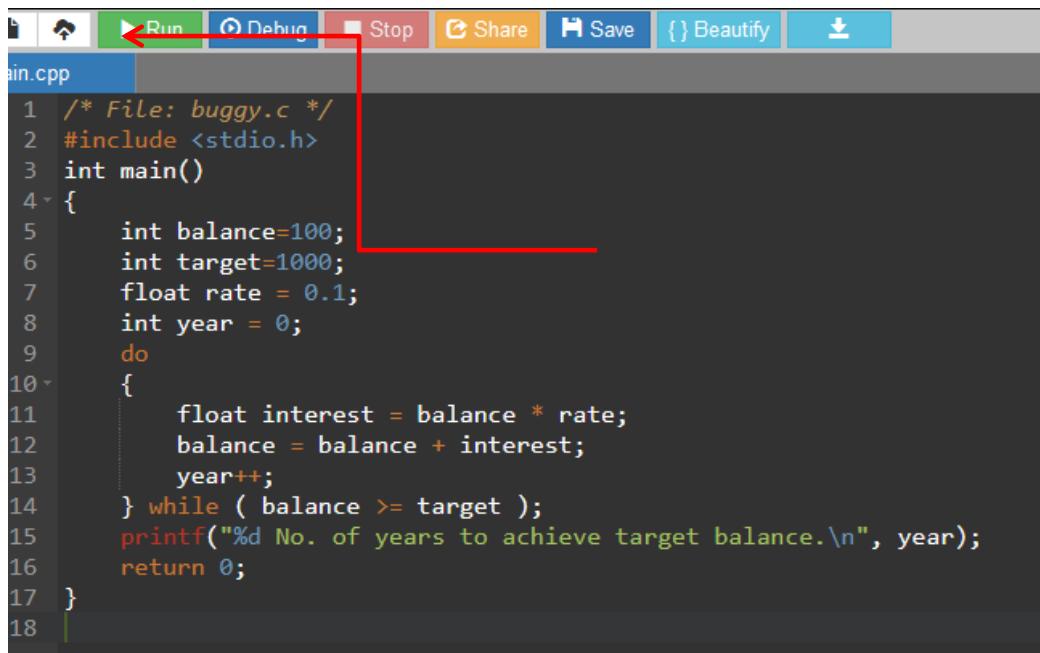
- l – list
- p – print
- c – continue
- s – step
- ENTER: Izvršava prethodno izvršenu komandu ponovo

Ovaj odjeljak će pokazati kako koristiti GDB naredbe prolazeći kroz primer. Ovde je prikazan program koji ima greške u skriptu. Ovaj program je napisan za izračunavanje broja godina potrebnih za postizanje ciljanog salda pod uslovom trenutnog salda i godišnje kamatne stope. Ali on daje 1 godinu kao rezultat za bilo koju vrednost tekućeg balansa i interesne stope.

```
/* File: buggy.c */
#include <stdio.h>
int main()
{
    int balance=100;
    int target=1000;
    float rate = 0.1;
    int year = 0;
    do
    {
        float interest = balance * rate;
        balance = balance + interest;
        year++;
    } while ( balance >= target );
    printf("%d No. of years to achieve target balance.\n", year);
    return 0;
}
```

Slede koraci za debug ovog programa:

Step 2: Run program sa GDB\$



The screenshot shows a debugger interface with a toolbar at the top containing icons for Run, Debug, Stop, Share, Save, and Beautify. A red arrow points to the 'Run' icon. Below the toolbar is a code editor window displaying the 'main.cpp' file. The code contains several syntax errors, notably misspelled words like 'whire' and 'balance' instead of 'while' and 'balance'. The code logic is intended to calculate the number of years required to reach a target balance starting from \$100 at a 10% annual interest rate.

```
1 /* File: buggy.c */
2 #include <stdio.h>
3 int main()
4 {
5     int balance=100;
6     int target=1000;
7     float rate = 0.1;
8     int year = 0;
9     do
10    {
11        float interest = balance * rate;
12        balance = balance + interest;
13        year++;
14    } whire ( balance >= target );
15    printf("%d No. of years to achieve target balance.\n", year);
16    return 0;
17 }
18 
```

U okviru konzole dobijamo informaciju da je izlaz negativan !!!

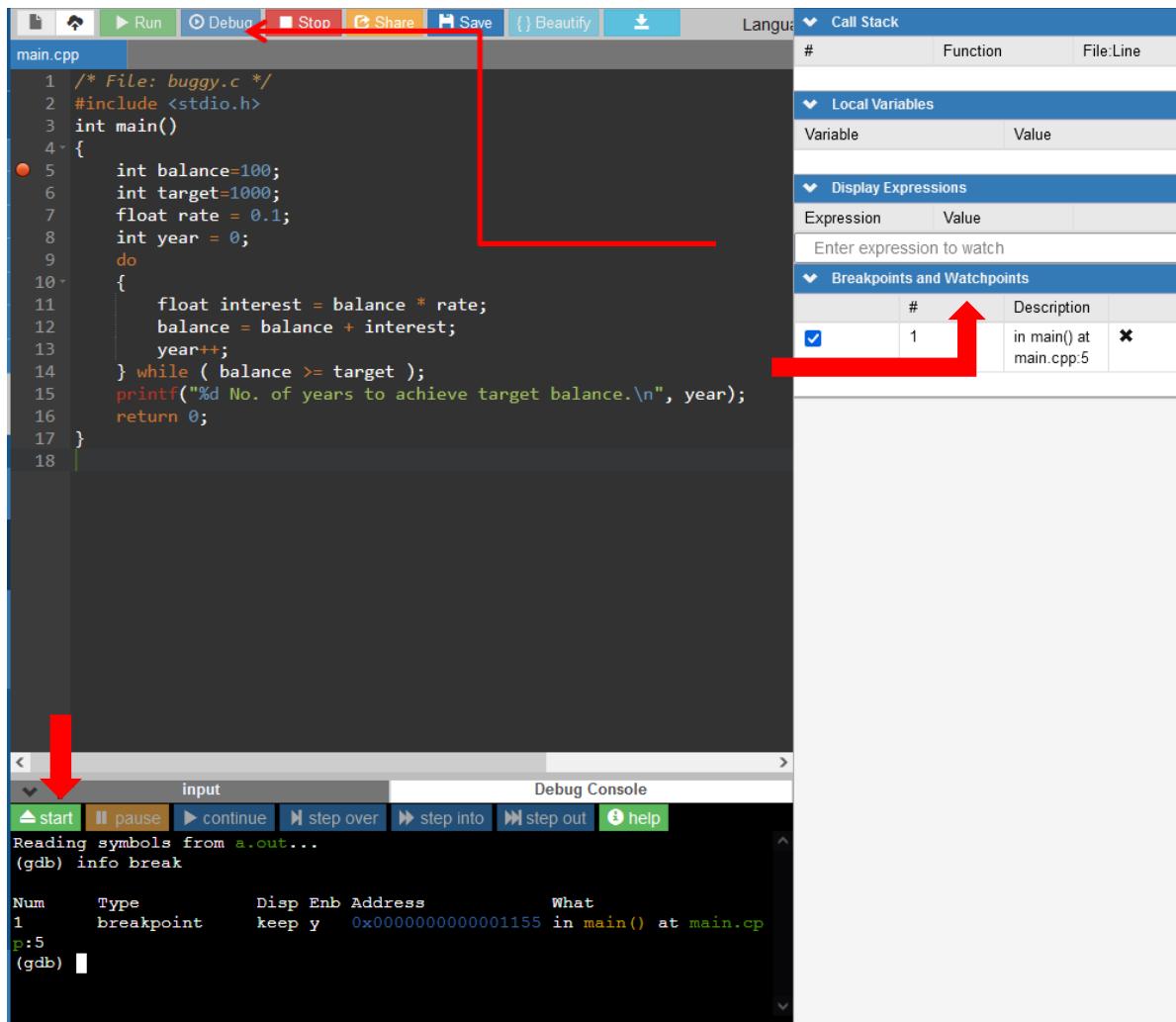
```
1 No. of years to achieve target balance.

...Program finished with exit code 0
Press ENTER to exit console.
```

Step 3: Postavljamo breakpoint (kliknemo na prazno mesto iza broja linije i pojavljuje se crveni kružić) na glavnoj funkciji, linija 3.

```
main.cpp
1 /* File: buggy.c */
2 #include <stdio.h>
3 int main()
4 {
5     int balance=100;
6     int target=1000;
7     float rate = 0.1;
8     int year = 0;
9     do
10    {
11         float interest = balance * rate;
12         balance = balance + interest;
13         year++;
14     } while ( balance >= target );
15     printf("%d No. of years to achieve target balance.\n", year);
16     return 0;
17 }
```

Step 4: Startovati program



Breakpoint 1, main () linija.c:5
5 int balance=100;

Kako je prekid postavljen na glavnu funkciju, program će se zaustaviti na glavnoj funkciji i čekati (gdb) naredbu.

Step 5: Ići korak po korak opcijom Continue dok ne dodjemo do linije 13 preko opcije (gdb) s

The screenshot shows the CLion IDE interface with the following components visible:

- Editor:** Shows the code file `main.cpp` with several breakpoints marked by orange circles. The code initializes variables `balance`, `target`, `rate`, and `year`, and then enters a loop to calculate interest and increment the year until the balance reaches or exceeds the target.
- Call Stack:** A table showing the current call stack.
- Local Variables:** A table showing local variable values.
- Display Expressions:** A table for entering expressions to watch.
- Breakpoints and Watchpoints:** A table listing seven breakpoints, all of which are currently active (checked).
- Debug Console:** A terminal window titled "input" containing GDB commands and output. The output shows symbols being read from `a.out` and a list of breakpoints. One breakpoint is highlighted with a yellow background and labeled "P:5".

U okviru konzole možemo primetiti vrednosti promenljivih i njihove pojave:

```
6 int target=1000;
(gdb) s
7 float rate = 0.1;
(gdb) s
8 int year = 0;
(gdb) s
11 float interest = balance * rate;
(gdb) s
12 balance = balance + interest;
(gdb) s
13 year++;
```

To ovako izgleda u editoru i konzoli:

The screenshot shows a debugger interface with the following details:

- Top Bar:** Run, Debug, Stop, Share, Save, Beautify, Language (set to C/C++), Call Stack.
- Code Editor (main.cpp):** The code is a C program that calculates the number of years required to achieve a target balance starting from a given balance. It includes several breakpoints (marked with red circles) and a cursor at line 6. The code is as follows:

```
1 /* File: buggy.c */
2 #include <stdio.h>
3 int main()
4 {
5     int balance=100;
6     int target=1000;
7     float rate = 0.1;
8     int year = 0;
9     do
10    {
11         float interest = balance * rate;
12         balance = balance + interest;
13         year++;
14     } while ( balance >= target );
15     printf("%d No. of years to achieve target balance.\n", year);
16     return 0;
17 }
```

- Call Stack:** Shows a single entry: main at main.cpp:6.
- Local Variables:** Shows the values of local variables: balance (100), target (0), rate (0), and year (0).
- Display Expressions:** An input field for entering an expression to watch, currently empty.
- Breakpoints and Watchpoints:** A list of 7 breakpoints, all of which are active (checked). Each breakpoint is associated with a line number and a description: 1 (main.cpp:13), 2 (main.cpp:12), 3 (main.cpp:11), 4 (main.cpp:8), 5 (main.cpp:7), 6 (main.cpp:6), and 7 (main.cpp:5).
- Bottom Navigation:** Buttons for start, pause, continue, step over, step out, and help.
- Debug Console:** Displays GDB session logs for breakpoints 7 and 6, showing the continuation of the program.

Step 6: Sada na konzoli ispisujemo vrednosti balance, target, rate, interest naredbom p *****

```
(gdb) p balance  
$1 = 110  
(gdb) p rate  
$2 = 0.1000000  
(gdb) p interest  
$3 = 10
```

Step 7: Ići korak po korak dok ne dodjemo do linije 15

```
(gdb)s % (start)  
14 } while ( balance >= target );  
(gdb) s % (start)  
15 printf("%d No. of years to achieve target balance..\n", year);
```

The screenshot shows a debugger interface with the following details:

- Code Editor:** The file `main.cpp` is open, displaying a C++ program that calculates the number of years required to reach a target balance starting from 100. The code includes a do-while loop and a printf statement.
- Breakpoints:** There are nine breakpoints set across the code, numbered 1 through 9, each with a description of being in the `main()` function at a specific line.
- Call Stack:** Shows the main function at `main.cpp:14`.
- Local Variables:** Shows variables `balance`, `target`, `rate`, and `year` with their current values: 110, 1000, 0.1, and 1 respectively.
- Display Expressions:** An empty input field for entering expressions to watch.
- Breakpoints and Watchpoints:** A table listing the nine breakpoints with their numbers, descriptions, and status.
- Debug Console:** The console shows the command `(gdb) continue` entered, with a red arrow pointing to the input field. Previous commands show breakpoints being set and continued execution.

Step 8: Ispisati vrednosti target, year, balance

```
(gdb) p target
```

```
$5 = 1000
```

```
(gdb) p year
```

```
$6 = 1
```

```
(gdb) p balance
```

```
$7 = 110
```

Vrednost `balance` je 110, a `target` je 1000. To se čini OK. Kodiranje u petlji while, tj. `balance >= target`, bit će lažan i petlja će se prekinuti. Dakle, uslov je ovdje pogrešan. Trebao bi biti `balance < target`, svaka čast za debugging ! Otklonili smo pogreške u svom prvom programu sa greškama i to automatski bez intervencije u editoru.

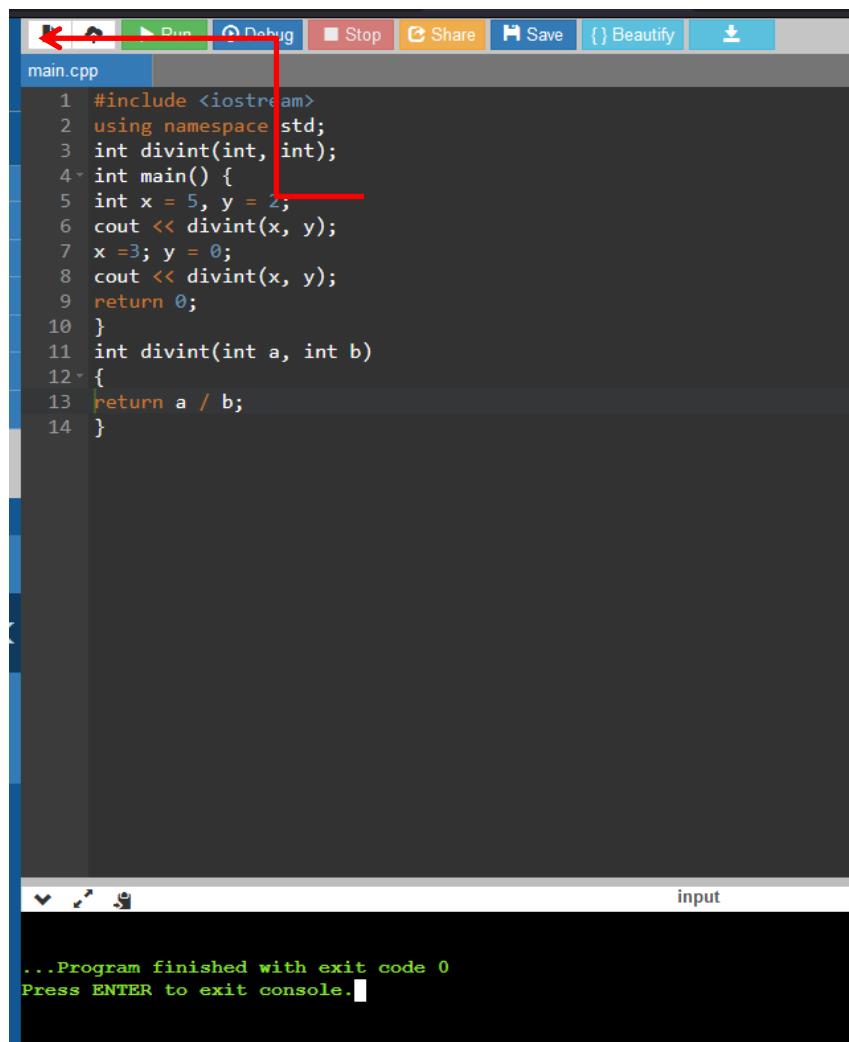
Primer analize grešaka 2

Ovaj primer pokazuje kako bi našli grešku koja se dogadja zbog deljenja sa nulom. Napišimo program za generisanje core dump.

Novi program:

```
#include <iostream>
using namespace std;
int divint(int, int);
int main()
{int x = 5, y = 2;
cout << divint(x, y);
x =3; y = 0;
cout << divint(x, y);
return 0;}
int divint(int a, int b)
{return a / b;}
```

Pokrenite editor kao što je opisano i unesite program. Pokrećemo program Run:



The screenshot shows a code editor window with a dark theme. The file 'main.cpp' is open, containing the provided C++ code. A red arrow points from the top-left of the image to the 'Run' button in the toolbar. Another red arrow points from the bottom-left of the image to the line 'x =3; y = 0;' in the code, highlighting the error. The code editor has tabs for Run, Debug, Stop, Share, Save, Beautify, and a download icon. The status bar at the bottom shows 'input' and the message '...Program finished with exit code 0 Press ENTER to exit console.'

Pokrećemo debugger i obeležavamo breakpoint na liniji 5:

The screenshot shows a debugger interface with the following components:

- Top Bar:** Includes buttons for Run, Debug, Stop, Save, Beautify, and Language selection.
- Code Editor:** Displays the C++ code for `main.cpp`. A red arrow points to the line number 5, where a breakpoint is set (indicated by a red dot).
- Call Stack:** Shows the call stack with one entry: `main` at `main.cpp:5`.
- Local Variables:** Shows variables `x` and `y` both with value 0.
- Display Expressions:** An input field for entering expressions to watch.
- Breakpoints and Watchpoints:** A table showing a single breakpoint:

	#	Description
<input checked="" type="checkbox"/>	1	in main() at main.cpp:5
- Debug Console:** Shows the GDB session output:

```
(gdb) run
Starting program: /home/a.out
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Breakpoint 1, main () at main.cpp:5
5 int x = 5, y = 2;
(gdb) 
```

A red arrow points to the "start" button in the GDB toolbar.

Nastavljamo korak po korak sa opcijom Continue:

main.cpp

```

1 #include <iostream>
2 using namespace std;
3 int divint(int, int);
4 int main() {
5     int x = 5, y = 2;
6     cout << divint(x, y);
7     x = 3; y = 0;
8     cout << divint(x, y);
9     return 0;
10 }
11 int divint(int a, int b) {
12 {
13     return a / b;
14 }

```

Call Stack

#	Function	File:Line
0	divint	main.cpp:13
1	main	main.cpp:8

Local Variables

Variable	Value
----------	-------

Display Expressions

Expression	Value
------------	-------

Breakpoints and Watchpoints

#	Description
<input checked="" type="checkbox"/> 1	in main() at main.cpp:5

Program received signal
SIGFPE, Arithmetic exception.
0x08048681 in divint(int, int)
(a=3, b=0) at crash.cc:13
13 return a / b;

input

Debug Console

start pause continue step over step into step out help

Breakpoint 1, main () at main.cpp:5
5 int x = 5, y = 2;
(gdb) continue
Continuing.

Program received signal SIGFPE, Arithmetic exception.
0x00005555555520a in divint (a=3, b=0) at main.cpp:13
13 return a / b;
(gdb)

U liniji 13 javlja se greška deljenja sa nulom !!!

Prekidamo debugger opcijom STOP, vršimo promenu promenljive na vrednost y=1. Sa ovako postavljenom izmenom u programu ponovo startujemo izvršavanje programa RUN, pokrećemo Debugger DEBAG i startujemo ga START, već postavljeni breakpoint ostaje:

main.cpp

```

1 #include <iostream>
2 using namespace std;
3 int divint(int, int);
4 int main() {
5     int x = 5, y = 2;
6     cout << divint(x, y);
7     x = 3; y = 1;
8     cout << divint(x, y);
9     return 0;
10 }
11 int divint(int a, int b) {
12 {
13     return a / b;
14 }

```

Call Stack

#	Function	File:Line
0	main	main.cpp:5

Local Variables

Variable	Value
x	0
y	0

Display Expressions

Expression	Value
------------	-------

Breakpoints and Watchpoints

#	Description
<input checked="" type="checkbox"/> 1	in main() at main.cpp:5

input

Debug Console

start pause continue step over step into step out help

(gdb) run
Starting program: /home/a.out
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at main.cpp:5
5 int x = 5, y = 2;
(gdb)

Na osnovu ovoga sledi izveštaj:

The screenshot shows a debugger interface with the following components:

- Editor Area:** Displays the code file `main.cpp`. The code defines a function `divint` and its usage in `main`. A red dot at line 5 indicates a breakpoint.
- Call Stack:** Shows the current call stack.
- Local Variables:** Shows the values of local variables.
- Display Expressions:** Allows entering expressions to watch.
- Breakpoints and Watchpoints:** Shows a list of breakpoints, with one entry at line 5 of `main.cpp`.
- Debug Console:** Shows the GDB command-line interface with the following session:

```
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".  
Breakpoint 1, main () at main.cpp:5  
5      int x = 5, y = 2;  
(gdb) continue  
Continuing.  
23[Inferior 1 (process 242) exited normally]  
(gdb) █
```

Izvršavanje programa kroz debugger pokazalo je da sada nema grešaka pa se program normalno izvršava !!!

Veštačka inteligencija i Paraplexity AI

Nauka i tehnologija su širom otvorile vrata novim sistemima koji se koriste u svakodnevnom životu građana, privrednih subjekata i funkcionisanju organa vlasti. Kao jedna od najznačajnijih novih digitalnih pojava izdvaja se veštačka inteligencija. Ukratko, veštačka inteligencija predstavlja opšti naziv za napredne računarske sisteme koji teže da simuliraju funkcionisanje ljudske inteligencije na takav način da su mašine sposobne da zamene ulogu i rad čoveka u raznim aktivnostima, od prostih do složenih. Njihov rad se zasniva na prethodno unetim parametrima, odnosno informacijama koje predstavljaju osnovne elemente na kojima se bazira i razvoj celokupne nauke i tehnologije. Kako su računarski sistemi sposobni da obrađuju velike količine informacija i podataka, potencijali veštačke inteligencije su ogromni, budući da se mnogiposlovi mogu brže, efikasnije i uspešnije obaviti njenim korišćenjem. To se odnosi na različite matematičke i ekonomski proračune, pretraživanja baza podataka, usluge prevoza i prodaje, pružanje medicinske pomoći itd. Kod veštačke inteligencije reč je o tome da se računarski sistemi razviju do tog nivoa da mogu samostalno obavljati funkcije koje tradicionalno obavljaju isključivo ljudi. Reč je o razvoju novih informacionih sistema koji su sposobni da samostalno uče i da preduzimaju različite aktivnosti, onako kako to čini inteligentno ljudsko biće. Pri tome, za jedan sistem možemo reći da je intelligentan onda kada je sposoban da nezavisno i efikasno reši probleme koji se pred njega stavljuju.

Veštačka inteligencija je simulacija procesa ljudske inteligencije od strane mašina, posebno računarskih sistema. Specifične primene veštačke inteligencije uključuju ekspertne sisteme, obradu prirodnog jezika, prepoznavanje govora itd. Kako se interesovanje oko veštačke inteligencije ubrzavalo, dobavljači su se trudili promovisati kako se njihovi proizvodi i usluge koriste. AI zahteva temelje specijaliziranog hardvera i softvera za pisanje i obuku algoritama mašinskog učenja. Nijedan programski jezik nije sinonim za AI, ali Python, R, Java, C++ i Julia imaju postavke popularne među AI programerima. Sistemi veštačke inteligencije funkcionišu tako da unose velike količine označenih podataka o obuci, analiziraju podatke radi korelacija i uzoraka i koriste te obrasce za predviđanje budućih stanja. Na ovaj način chatbot koji se hrani primerima teksta može naučiti generisati realistične razmene sa ljudima ili alat za prepoznavanje slika može naučiti identifikovati i opisati objekte na slikama pregledom miliona primera. Nove generativne AI tehnike koje se brzo poboljšavaju mogu stvoriti realističan tekst, slike, glazbu i druge medije. Programiranje veštačke inteligencije je na kognitivna veštine koja uključuju sljedeće:

1. *Učenje.* Ovaj aspekt programiranja veštačke inteligencije usredsredjen je na prikupljanje podataka i stvaranje pravila kako ih pretvoriti u korisne informacije.
2. *Pravila,* koja se nazivaju algoritmi, daju računardkom uredajima upute korak po korak za izvršenje odredjenog zadatka.
3. *Rasudivanje.* Ovaj aspekt programiranja veštačke inteligencije fokusiran je na izbor pravog algoritma za postizanje željenog rezultata.
4. *Samoispravljanje.* Ovaj aspekt AI programiranja osmišljen je za kontinuirano fino podešavanje algoritama i osiguravanje da daju najtačnije moguće rezultate.
5. *Kreativnost.* Ovaj aspekt veštačke inteligencije koristi neuronske mreže, sisteme utemeljene na pravilima, statističke metode i druge tehnike veštačke inteligencije za generisanje novih slika, novog teksta, nove muzike i novih ideja.

Veštačka inteligencija postala je ključna za mnoge današnje najveće i najuspešnije kompanije, uključujući Alphabet, Apple, Microsoft i Metu, gdje se AI tehnologije koriste za poboljšanje operacija i nadmašivanje konkurencije. Na primer, u podružnici Alphabet-a, Googleu, veštačka inteligencija je ključna za njegovu mašinu pretrage.

Slede neke od prednosti veštačke inteligencije.

1. Dobar u poslovima usmerenim na detalje. Veštačka inteligencija se pokazala jednako dobrom ili boljom od lekara u dijagnostkovaniju određenih vrsta raka, uključujući rak dojke i melanom.
2. Smanjeno vreme za zadatke koji opterećuju podatke. AI se naširoko koristi u industrijama sa velikim brojem podataka - Big Data, uključujući bankarstvo i vrednosne papire, farmaciju i osiguranje, kako bi se smanjilo vreme potrebno za analizu velikih skupova podataka.
3. Finansijske usluge, naprimjer, rutinski koriste AI za obradu zaheva za kredit i otkrivanje prevara. Šteti rad i povećava produktivnost. Ovde je primer upotrebe automatizacije skladišta, koja je porasla tokom pandemije i očekuje se da će se povećati sa integracijom veštačke inteligencije i mašinskog učenja.
4. Daje dosledne rezultate. Najbolji AI alati za prevodjenje pružaju visok nivo doslednosti, nudeći čak i malim kompanijama mogućnost da dopru do kupaca na njihovom materinjem jeziku.
5. Može poboljšati zadovoljstvo korisnika kroz personalizaciju. AI može personalizovati sadržaj, poruke, oglase, preporuke i web stranice pojedinačnim korisnicima. Virtualni agenti pokretani veštačkom inteligencijom uvek su dostupni. AI programi ne moraju spavati niti uzimati pauze, pružajući uslugu 24/7.

Slede neki od nedostataka veštačke inteligencije.

1. Skup.
2. Zahteva duboku tehničku stručnost.
3. Ograničena ponuda kvalifikovanih radnika za izradu AI alata. Odražava pristranosti svojih podataka o obuci, na nivou.
4. Nedostatak sposobnosti generalizovanja sa jednog zadatka na drugi.
5. Ukida ljudska radna mesta, povećavajući stope nezaposlenosti.

Koje su 4 vrste veštačke inteligencije?

Arend Hintze, asistent profesora integrativne biologije i računarskih nauka i inženjerstva na Državnom fakultetu Michigan, objasnio je da se veštačka inteligencija može kategorizirati u četiri vrste, počevši od intelligentnih sistema za specifične zadatke koji su danas u širokoj upotrebni i napreduju do osjetljivih sistema, koji ipak ne postoje. Kategorije su sledeće.

Tip 1: *Reaktivni uredjaji*. Ovi AI sistemi nemaju memoriju i specifični su za zadatak. Primer je Deep Blue, IBM-ov šahovski program koji je 1990-ih pobedio Garryja Kasparova. Deep Blue može identifikovati figure na šahovskoj tabli i predvidjati, ali budući da nema pamćenja, ne može koristiti prošla iskustva da bi informisao buduća.

Tip 2: *Ograničena memorija*. Ovi AI sistemi imaju memoriju, tako da mogu koristiti prošla iskustva za donošenje budućih odluka. Neke od funkcija donošenja odluka u samovozećim automobilima osmišljene su na ovaj način.

Tip 3: Teorija uma. Teorija uma je psihološki pojam. Kada se primeni na AI, to znači da bi sistem imao socijalnu inteligenciju za razumevanje emocija. Ova vrsta veštačke inteligencije moći će zaključiti ljudske namere i predvidjeti ponašanje, što je neophodna veština za sistem veštačke inteligencije da postanu integralni članovi ljudskih timova.

Tip 4: Samosvest. U ovoj kategoriji, AI sistemi imaju osjećaj sebe, što im daje svest. Maštine sa samosveću razumeju svoje trenutno stanje. Ova vrsta veštačke inteligencije još ne postoji.

Koji su primeri AI tehnologije i kako se ona danas koristi? AI je ugrađen u niz različitih vrsta tehnologije. Evo sedam primera.

Automatizacija. U kombinaciji sa AI tehnologijama, alati za automatizaciju mogu proširiti opseg i vrste zadataka koji se obavljaju. Primer je robotska automatizacija procesa (RPA), vrsta softvera koja automatizuje ponavljanje zadatke obrade podataka utemeljene na pravilima koje tradicionalno obavljaju ljudi. U kombinaciji sa mašinskim učenjem i novim alatima veštačke inteligencije, RPA može automatizovati veće delove poslovnih poslova, omogućujući RPA-ovim taktičkim botovima da prosledjuju podatke iz veštačke inteligencije i reaguju na promene procesa.

Mašinsko učenje. Ovo je nauka o pokretanju računara bez programiranja. Duboko učenje je podskup mašinskog učenja koji se, vrlo jednostavno rečeno, može smatrati automatizacijom prediktivne analitike. Postoje tri vrste algoritama mašinskog učenja:

Učenje pod nadzorom. Skupovi podataka označeni su tako da se uzorci mogu otkriti i koristiti za označavanje novih skupova podataka.

Učenje bez nadzora. Skupovi podataka nisu označeni i razvrstani su prema sličnostima ili razlikama.

Učenje as potkrepljivanjem. Skupovi podataka nisu označeni, ali nakon izvodjenja radnje ili nekoliko radnji, AI sistem dobija povratnu informaciju.

Mašinski vid. Ova tehnologija daje mašini mogućnost da vidi. Mašinski vid hvata i analizira vizualne informacije pomoću kamere, analogno-digitalne translacije i digitalne obrade signala. Često se uporedjuje sa ljudskim vidom, ali mašinski vid nije vezan biologijom i može se programirati da, na primer, vidi kroz zidove. Koristi se u nizu aplikacija od identifikacije potpisa do analize medicinske slike.

Računarski vid, koji je fokusiran na mašinsku obradu slike, često se povezuje sa mašinskim vidom. Obrada prirodnog jezika (NLP). Ovo je obrada ljudskog jezika računarskim programom. Jedan od starijih i najpoznatijih primera NLP-a je detekcija neželjene pošte, koja gleda predmet i tekst e-pošte i odlučuje je li bezvredna. Trenutni pristupi NLP-u utemeljeni su na mašinskom učenju. NLP zadaci uključuju prevod teksta, analizu osećaja i prepoznavanje govora.

Robotika. Ovo polje inženjeringu usredsredjeno je na dizajn i proizvodnju robota. Roboti se često koriste za obavljanje zadataka koje ljudi teško izvode ili obavljaju dosledno. Na primer, roboti se koriste u proizvodnim trakama za proizvodnju automobila ili ih koristi NASA za pomeranje velikih objekata u svemiru. Istraživači takođe koriste mašinsko učenje za izradu robota koji mogu komunicirati u društvenim okruženjima.

Samovozeći automobili. Autonomna vozila koriste kombinaciju računarskog vida, prepoznavanja slike i dubinskog učenja za izgradnju automatiziranih veština za upravljanje vozilom dok ostaju u zadanoj traci i izbegavaju neočekivane prepreke, poput pešaka.

Generisanje teksta, slike i zvuka. Generativne AI tehnike, koje stvaraju različite vrste medija iz tekstualnih upita, intenzivno se primenjuju u svim kompanijama kako bi se stvorio naizgled neograničen raspon vrsta sadržaja od fotorealistične umetnosti do odgovora e-poštom i scenarija.

Perplexity AI

Perplexity AI je alat za chat i interfejs za pretraživanje koji koristi napredne tehnologije veštačke inteligencije, kao što je OpenAI GPT-3 model, za pružanje trenutnih odgovora na složena pitanja. Izgrađen je korišćenjem istog modela kao Chat GPT i dizajniran je za otključavanje baze znanja otkrivanjem i deljenjem informacija. Perplexity AI je novi AI chat alat koji deluje kao neverovatno moćna pretraga. Kada korisnik unese pitanje, model pretražuje internet kako bi dao odgovor. A ono što je sjajno kod ovog alata je njegova sposobnost da prikaže izvor informacija koje pruža.

Web stranica ima slične postavke kao Google. Sadrži glavnu traku za pretraživanje u središtu, gde u osnovi postavljate AI bilo koje pitanje. Međutim, Perplexity AI ima veliku prednost. Čini se da može pružiti informacije u stvarnom vremenu.

Naprimjer, na pitanje kog dana je Samsung Galaxy S23 pušten u prodaju, AI je uspeo dati tačan odgovor. Ovo je prilično izvanredno sa obzirom na to da je pametni telefon objavljen tek ove nedelje.

Još jedan popularan AI chat bot, Chat GPT, malo zaostaje u ovom području. AI model ima vrlo ograničeno znanje o dogadjajima koji su se dogodili posle 2021. i nema mogućnost citiranja svojih izvora u meri u kojoj to može Perplexity AI. Zanimljivo je da je ovaj AI bot mašine za pretragu napravljen pomoću OpenAI GPT-3 modela. Dakle, sada znamo što može učiniti i kako bismo ga trebali koristiti?

Korišćenje Perplexity AI

Evo nekoliko primera kako možemo koristiti AI mašinu za pretragu, iako ovo nije kompletan popis.

- Odgovaranje na FAQ
- Pronalaženje instrukcija korak po korak
- Definisanje pojmove
- Nalaženje radnog vremena
- Nalaženje vremenske prognoze

Šta je Chat GPT?

To je AI chatbot autogenerisani sistem koji je stvorio OpenAI za online brigu o korisnicima. To je unapred obučeni generativni chat koji koristi Neuro Lingvističko Programiranje (NLP - Natural Language Processing) obradu prirodnog jezika. Izvor njegovih podataka su udžbenici, web stranice i razni članci, koje koristi za modeliranje vlastitog jezika za reagiranje odgovora na ljudsku interakciju. Ovaj chatbot sistem pruža informacije i odgovore na upite putem veštačke inteligencije. Popularna verzija Chat GPT-a je model GPT-3.

Za šta se koristi Chat GPT?

Glavna osobina Chat GPT-a je generisanje odgovora kakve bi ljudi dali, u tekstualnom okviru. Stoga je prikladan za chatbotove, razgovore AI sistema i virtualne asistente. Medjutim, takodje može dati prirodne odgovore na pitanja u tonu razgovora i može generisati priče, pesme i još mnogo toga. Šta više, može:

- Napisati kod
- Napisati članak ili post na blogu
- Prevesti
- Debugging bilo kog programskog koda u bilo kom jeziku
- Napisati priču/pjesmu
- Preporučiti akorde i tekst

Kako bi AI izvršio jedan od ovih zahteva, sve što trebamo učiniti je upisati naredbu u chatbot. U nedavnim vestima, OpenAI je objavio novu osobinu Chat GPT dodatka za poboljšanje trenutnih mogućnosti chatbota. ChatGPT će biti veći i bolji nego ikad, sa mogućnošću korišćenja znanja i osobinu mnogih aplikacija trećih strana. Početni set dodataka već je pokrenut, a uključuje popularne platforme kao što su Expedia, Klarna, Slack, Wolfram, OpenTable, Shopify i druge. U biti, ovo novo ažuriranje znači da ChatGPT može doneti alate sa ovih platformi. Na primjer, nacrtajte grafikone koristeći Wolfram ili nam pošaljite rezervacije restorana iz OpenTable.

Na čemu se trenira Chat GPT?

Oslanja se na NLP (Natural Language Processing). To je izvrstan alat za istraživače i programere koji rade na raznim NLP projektima, a ima mnogo specifičnih zadataka, domena i aplikacija dostupnih za rad. Dobro je uvežban na jadrostranim i nepristrasnim podacima, u obliku teksta iz knjiga, članaka i web stranica. Chat GPT može reprodukovati izlazne podatke i pouzdanost – ključno za mnoge osetljive aplikacije i druge vredne AI sisteme. Medjutim, još uvek je sklon greškama i pristranostima i zavisi od podataka i obuci – dostavljenim 2021. Kao ljudi, sve smo više u interakciji sa urednjima koje pokreće AI, a Chat GPT je revolucija u području Al-a. To je robustan model i posebno napredan zahvaljujući svojim mogućnostima dubokog učenja i NLP-u. Ultimativno, može generisati odgovore slične ljudskim i lako je razumljiv korisnicima.

U čemu je kodiran Chat GPT?

Chat GPT izgrađen je na GPT-3 jezičnom modelu koji je razvio OpenAI. Iako izvorni kod za GPT-3 nije objavljen, post na stranici StackExchange sugerise da je GPT-3 napisan u "istom modelu i arhitekturi kao GPT-2". Post dalje navodi da je "izvorni kod GPT-2 100% Python". Sama aplikacija je očito sposobna pružiti kod u Pythos C++ i JavaScriptu, ako se od nje zatraži da ga izradi. Iako izvorni kod za GPT-3 nije objavljen, post na stranici StackExchange sugerira da je GPT-3 napisan u "istom modelu i arhitekturi kao GPT-2". ChatGPT je takodje AI za razgovor. Pitate se što je AI za razgovor? Pa, to je sistem veštačke inteligencije dizajniran da se uključi u razgovore sa ljudskim korisnicima. Ova interakcija može se odvijati putem tekstualnih interfejsa, kao što je slučaj s ChatGPT-om. Cilj konverzacijeske veštačke inteligencije je simulirati ljudski razgovor što je moguće bliže, a da

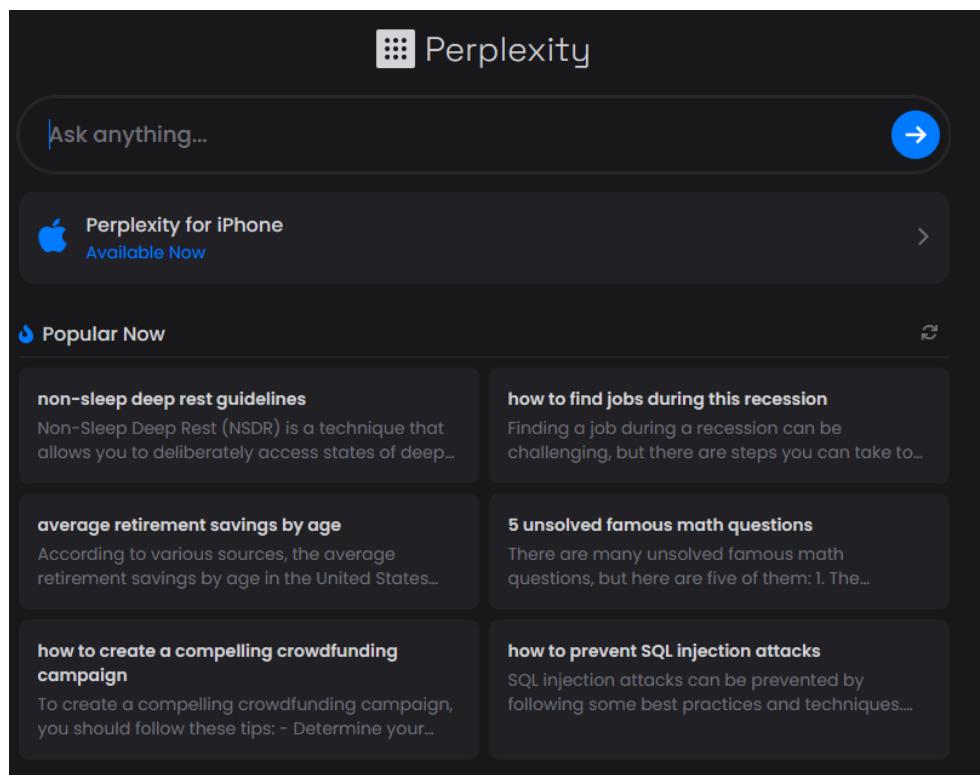
bi se to postiglo, ovi sistemi trebaju biti obučeni na velikim skupovima podataka konverzacijskih podataka. Ovde ChatGPT blista, uvežban je na ogromnoj količini tekstualnih podataka o kojima smo govorili u prethodnom odeljku, Chat GPT ima sposobnost proizvesti odgovore koji su vrlo slični ljudskim odgovorima na širok raspon tekstualnih pitanja, što ga čini vrlo uvjerljiva tehnologija.

Perplexity AI Debugger

Paraplexity AI Debugger, ali i za mnoge druge potrebe, možemo koristiti online i to besplatno na linku:

<https://www.perplexity.ai/>

Otvara se prozor vrlo sličan Google pretraživaču:



Upitaćemo Perplexity AI šta zna o C Debuggeru i evo odgovora:

c debugger

PERPLEXITY

View Detailed

There are several C/C++ debuggers available for developers to use. Some of the popular ones include GDB, LLDB, Visual Studio Code, and online debuggers like Online GDB and C Tutor.

GDB is a popular command-line debugger for C and C++ programs that is available on most Unix-based systems^[1]. It can be used to run a program in debug mode or to analyze a core file. To use GDB, developers can run the command `gdb ./program` to start the debugger and run the program^[1].

LLDB is another popular debugger that is available on macOS and Linux systems. It is designed to be a replacement for GDB and provides a similar set of features^[2].

Visual Studio Code is a popular code editor that provides built-in support for C/C++ debugging. Developers can configure the debugger in Visual Studio Code by creating a `launch.json` file that specifies the path to the executable they want to debug^[3]. Visual Studio Code also supports debugging memory dumps^[2].

Online GDB and C Tutor are online debuggers that allow developers to write, compile, and debug C/C++ programs in a web browser. These tools can be useful for developers who don't have access to a local development environment or who want to quickly test out a code snippet^{[4][5]}.

Overall, the choice of debugger depends on the developer's preference and the specific requirements of the project.

1. Skript napisan u programskom jeziku C, koji želimo da proverimo kroz Perplexity AI debugger izgleda ovako:

```
/* File: buggy.c */
#include <stdio.h>
int main()
{
    int balance=100;
    int target=1000;
    float rate = 0.1;
    int year = 0;
    do
    {
        float interest = balance * rate;
        balance = balance + interest;
        year++;
    } while ( balance >= target );
    printf("%d No. of years to achieve target balance.\n", year);
    return 0;
}
```

Postavićemo sledeće pitanje i uneti script u oblast Ask Anithing

Is script correct?:/* File: buggy.c */ #include <stdio.h> int main() { int balance=100; int target=1000; float rate = 0.1; int year = 0; do { float interest = balance * rate; balance = balance + interest; year++; } while (balance >= target); printf("%d No. of years to achieve target balance.\n", year); return 0; }

Perplexity AI odgovara:

Navedeni skript nije potpun i sadrži sintaksnu grešku. Kodu nedostaju završne vitičaste zagrade za do-while petlju i završne vitičaste zagrade za glavnu funkciju. Ispravljeni kod trebao bi izgledati ovako:

```
/* file: buggy.c */
#include <stdio.h>

int main() {
    int balance=100;
    int target=1000;
    float rate = 0.1;
    int year = 0;
    do {
        float interest = balance * rate;
        balance = balance + interest;
        year++;
    } while (balance < target);
    printf("It takes %d years to reach the target balance.\n", year);
    return 0;
}
```

I sa neverovatnim komentarom:

(*The corrected code calculates the number of years it takes to reach a target balance with a given interest rate. The program starts with an initial balance of 100 and continues to add interest to the balance until it reaches the target balance of 1000. The number of years it takes to reach the target balance is printed to the console using printf() function. To check if the code is correct, you can compile and run it using a C compiler such as gcc. You can also use a debugger such as GDB to step through the code and check for errors.*)

Ispravljeni kod izračunava broj godina koje su potrebne da se postigne ciljni saldo uz zadanu kamatu stopu. Program počinje sa početnim saldom od 100 i nastavlja dodavati kamate na saldo sve dok ne dosegne ciljni saldo od 1000. Broj godina koji je potreban da se postigne ciljni saldo ispisuje se na konzoli pomoću funkcije printf(). Za proveru, ako je kod ispravan, možete ga prevesti i pokrenuti koristeći C kompjajler kao što je gcc. Takođe možete koristiti program za ispravljanje pogrešaka kao što je GDB da prođete kroz kod i proverite greške.

2. Skript napisan u programskom jeziku Python, koji želimo da proverimo kroz Perplexity AI debugger i koji izračunava MRC verovatnoću otkaza za Rayleigh kanal sa fedingom TK signala izgleda ovako:

```

gamma_ratio_dB = np.arange(start=-10,stop=40,step=2)
Ns = [1,2,3,4,10,20] #number of received signal paths

gamma_ratio = 10** (gamma_ratio_dB/10) #Average SNR/SNR threshold in
dB

fig, ax = plt.subplots(1, 1)

for N in Ns:
    n = np.arange(start=0,stop=N,step=1)
    P_outage = 1 - np.exp(-1/gamma_ratio)
    *np.sum(((1/gamma_ratio)**n[:,None])/factorial(n[:,None]),axis=0)
    ax.semilogy(gamma_ratio_dB,P_outage,label='N='+str(N))

ax.legend()
ax.set_xlim(-10,40);ax.set_ylim(0.0001,1.1)
ax.set_title('MRC outage probability (Rayleigh fading channel)')
ax.set_xlabel(r'$10\log_{10}(\Gamma/\gamma_t)$')
ax.set_ylabel('Outage probability');fig.show()

```

Iskopiramo sadržaj skripta u Peroplexity region i pokrenemo sistem:

Find error in Python code:
`gamma_ratio_dB = np.arange(start=-10,stop=40,step=2)`
`Ns = [1,2,3,4,10,20]` #number of received signal paths
`gamma_ratio = 10** (gamma_ratio_dB/10)`
`#Average SNR/SNR threshold in dB`
`fig, ax = plt.subplots(1, 1)` for N in Ns:
`n = np.arange(start=0,stop=N,step=1)`
`P_outage = 1 - np.exp(-`
`1/gamma_ratio)*np.sum(((1/gamma_ratio)**n[:,None])/factorial(n[:,None]),axis=0)`
`ax.semilogy(gamma_ratio_dB,P_outage,label='N='+str(N))` ax.legend()
`ax.set_xlim(-10,40);ax.set_ylim(0.0001,1.1)` ax.set_title('MRC outage probability (Rayleigh fading channel)')
`ax.set_xlabel(r'$10\log_{10}(\Gamma/\gamma_t)$')`
`ax.set_ylabel('Outage probability');`fig.show()

Sistem predstavlja odgovor u vidu:

The Python code has an error in the line `gamma_ratio = 10**(gamma_ratio_db/10)`. The error is that `gamma_ratio_db` is a NumPy array, and the `**` operator does not work with arrays. Instead, we need to use the NumPy function `np.power()` to raise each element of the array to the power of 10^{[1][2]}.

Here's the corrected code:

```
import numpy as np

gamma_ratio_db = np.arange(start=-10, stop=40, step=2)
ns = [1, 2, 3, 4, 10, 20]

gamma_ratio = np.power(10, gamma_ratio_db/10) # corrected line

for n in ns:
    snr_threshold = 2 * np.log2(n) * gamma_ratio
    print(f"n = {n}, snr_threshold = {snr_threshold}")
```

In this code, we first import the NumPy library. We then define `gamma_ratio_db` as a NumPy array using `np.arange()`, which generates an array of values from -10 to 38 (inclusive) with a step of 2. We also define a list `ns` with some values of `n`.

The corrected line calculates `gamma_ratio` as an array of values obtained by raising 10 to the power of each element of `gamma_ratio_db` divided by 10. This is equivalent to converting the values in `gamma_ratio_db` from decibels to linear scale.

The rest of the code uses `gamma_ratio` to calculate `snr_threshold` for each value of `n` in `ns`. The `snr_threshold` is calculated using the formula `2 * log2(n) * gamma_ratio`, which is a vectorized operation that applies to each element of `gamma_ratio`. Finally, the code prints the values of `n` and `snr_threshold` for each iteration of the loop.

References:

- [1] Real Python. "NumPy arange(): How to Use np.arange()." Real Python, 2022, <https://realpython.com/how-to-use-numpy-arange/>.
- [2] NumPy. "numpy.power." NumPy v1.20 Manual, 2022, <https://numpy.org/doc/stable/reference/generated/numpy.power.html>.



PREVOD:

Python kod ima grešku u redu `gamma_ratio = 10**(gamma_ratio_db/10)`. Greška je u tome što je `gamma_ratio_db` polje NumPy, a operator `**` ne radi sa nizovima. Umesto toga, trebamo koristiti NumPy funkciju `np.power()` za podizanje svakog elementa niza na potenciju 10.

U ovom kodu prvo uvozimo biblioteku NumPy. Zatim definišemo `gamma_ratio_db` kao NumPy polje koristeći `np.arange()`, koji generiše niz vrednosti od -10 do 38 (inkluzivno) sa korakom od 2. Takođe definišemo listu `ns` sa nekim vrednostima `n`. Ispravljena linija

izračunava `gamma_ratio` kao niz vrednosti dobijenih podizanjem 10 na potenciju svakog elementa `gamma_ratio_db` podijeljenog sa 10. To je ekvivalentno pretvaranju vrijednosti u `gamma_ratio_db` iz decibela u linearu skalu. Ostatak koda koristi `gamma_ratio` za izračunavanje `snr_threshold` za svaku vrednost `n` u `ns`. `Snr_threshold` izračunava se pomoću formule $2 * \log_2(n) * \text{gamma_ratio}$, koja je vektorizirana operacija koja se primenjuje na svaki element `gamma_ratio`. Na kraju, kod ispisuje vrednosti `n` i `snr_threshold` za svaku iteraciju petlje.

NEVEROVATNO !!!