Programiranje 2, letnji semestar 2023/4

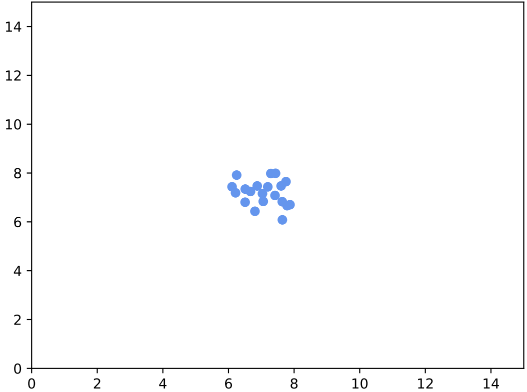
Potprogrami

Stefan Nikolić

Prirodno-matematički fakultet, Novi Sad

08.04.2024.

Pretpostavimo da imamo sledeći problem



Kako da nađemo približan centar grupe tačaka?

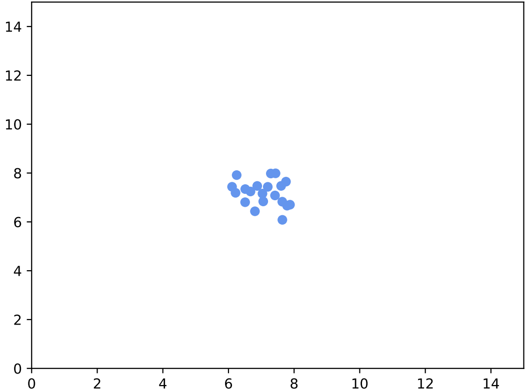
Prvo pitanje

Kako da predstavimo tačke koristeći samo ono znanje koje smo do sada stekli na predavanjima?

Odvojeni nizovi koordinata

```
float xs[NUM_PTS] = {...
float ys[NUM_PTS] = {...
```

Da se vratimo na početni problem

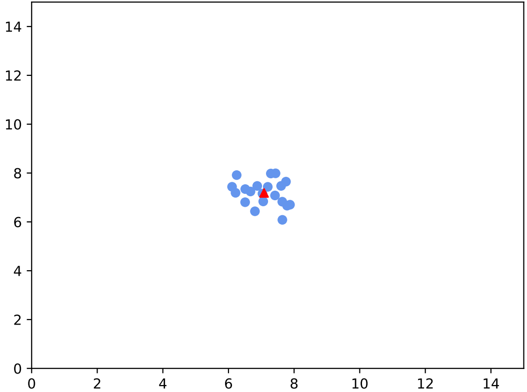


Kako da nađemo približan centar grupe tačaka?

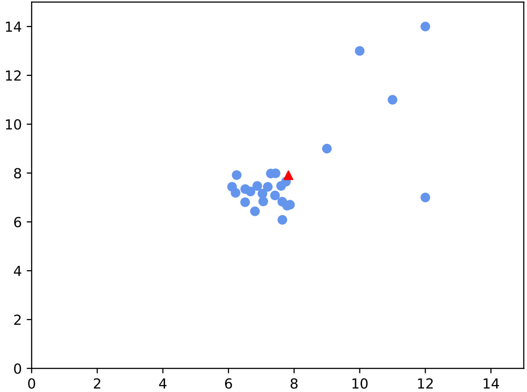
Prva ideja

centar = (prosek x koordinata svih tačaka, prosek y koordinata svih tačaka)

To je razumna aproksimacija

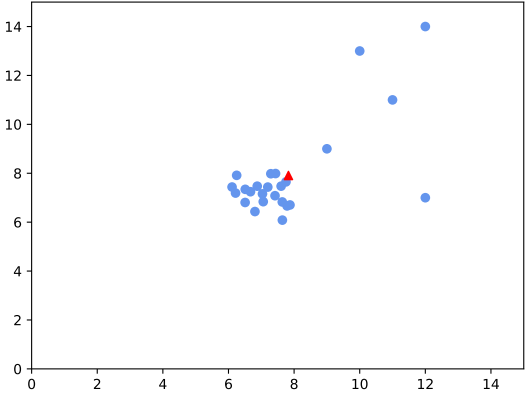


To je razumna aproksimacija



Kada nema udaljenih tačaka koje nisu deo grupe (eng. *outliers*)

Da li možemo da smislimo nešto robusnije?

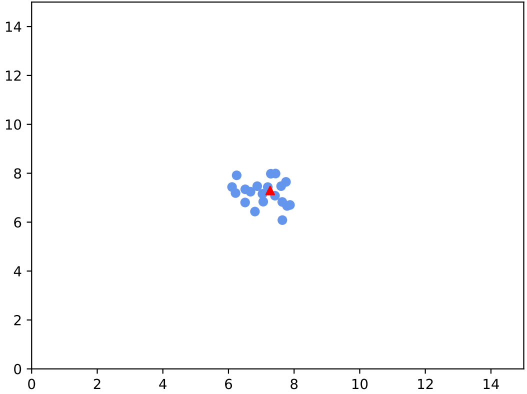


Tako da smanjimo uticaj udaljenih tačaka

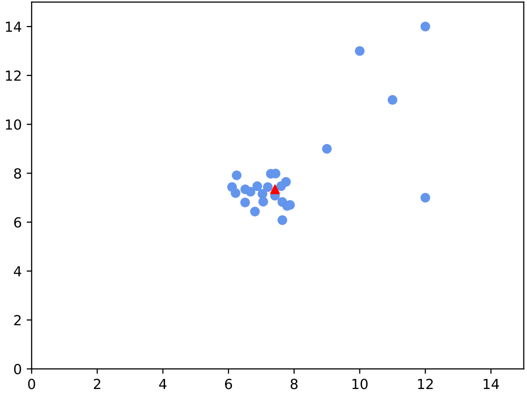
Druga ideja

centar = (medijana x koordinata svih tačaka, medijana y koordinata svih tačaka)

I to je razumna aproksimacija



I to je razumna aproksimacija



otpornija na prisustvo udaljenih tačaka

Kako računamo medijanu?

float xs[5] = {1.2, 2.7, 0.3, 4.2, 9.1}

Prvi korak: sortiramo niz

$\{0.3, 1.2, 2.7, 4.2, 9.1\}$

Drugi korak: biramo element u sredini

$\{0.3, 1.2, 2.7, 4.2, 9.1\}$

Šta radimo ako imamo paran broj elemenata?

Šta radimo ako imamo paran broj elemenata?

$\{0.3, 1.2, 2.7, 3.1, 4.2, 9.1\}$

medijana = $\frac{2{,}7+3{,}1}{2} = 2.9$

Kako to možemo implementirati u C++-u, koristeći dosadašnja znanja?

Najpre inicijalizujemo nizove koordinata

```cpp
#include <iostream>
#define NUM_PTS 25

using namespace std;

int main()
{
        float xs[NUM_PTS] = {7.7857203028720035, 7.642458246115663, 6.215313359871936, 7.059634724385643, 6.670815698546658, 6.876282852217749, 7.036072823998538, 7.290710191791188, 7.639716393843683, 7.752535310413255, 6.1089490156447095, 7.6043411248489505, 7.418263501562334, 6.2496483487612196, 6.8065125974698395, 7.434551692906639, 6.51122811120167, 7.198011826278233, 7.874699069289809, 6.507268194942792, 12, 9, 12, 11, 10};

        float ys[NUM_PTS] = {6.663959610602355, 6.0833932514505, 7.1901041284124805, 6.837614857113309, 7.24503886439218, 7.47176421269738, 7.157717200452314, 7.980448542746833, 6.826401869632361, 7.647518865141043, 7.437274473128388, 7.472813287622878, 7.081873708605056, 7.915294590939337, 6.43390232064712, 7.988414876843875, 7.3426188638785606, 7.434664291111027, 6.703619537079404, 6.804945018073546, 14, 9, 7, 11, 13};

```

Zatim koristimo insertion sort sa prošlog časa da sortiramo x koordinate

```
//Koristimo insertion sort sa proslog casa da sortiramo x koordinate
for(unsigned i = 1; i < NUM_PTS; ++i)
{
        float val = xs[i];
        unsigned j = i;
        while((j > 0) && (val < xs[j - 1]))
        {
                xs[j] = xs[j - 1];
                --j;
        }
        xs[j] = val;
}
```

Pa izračunamo medijanu

```
float median_x = 0;
if(NUM_PTS % 2)
	median_x = xs[NUM_PTS / 2];
else
	median_x = 0.5 * (xs[NUM_PTS / 2] + xs[NUM_PTS / 2 + 1]);
```

Ponovimo sve to za y koordinate

```
	//Koristimo insertion sort sa proslog casa da sortiramo y koordinate
	for(unsigned i = 1; i < NUM_PTS; ++i)
	{
		float val = ys[i];
		unsigned j = i;
		while((j > 0) && (val < ys[j - 1]))
		{
			ys[j] = ys[j - 1];
			--j;
		}
		ys[j] = val;
	}

	float median_y = 0;
	if(NUM_PTS % 2)
		median_y = ys[NUM_PTS / 2];
	else
		median_y = 0.5 * (ys[NUM_PTS / 2] + ys[NUM_PTS / 2 + 1]);


	cout << "centar = (" << median_x << ", " << median_y << ")\n";

	return 0;
}
```

I u samo 55 linija koda dobijamo rezultat

```
centar = (7.41826, 7.34262)
```

Da li možemo da pojednostavimo program?

Izdvojimo računanje medijane u potprogram

```
#include <iostream>
#define NUM_PTS 25

using namespace std;

float median(float a[], unsigned len)
{
	//Koristimo insertion sort sa proslog casa da sortiramo niz
	for(unsigned i = 1; i < len; ++i)
	{
		float val = a[i];
		unsigned j = i;
		while((j > 0) && (val < a[j - 1]))
		{
			a[j] = a[j - 1];
			--j;
		}
		a[j] = val;
	}

	if(len % 2)
		return a[len / 2];

	return 0.5 * (a[len / 2] + a[len / 2 + 1]);
}
```

Zatim umesto dve kopije istog koda, pozivamo dva puta istu funkciju

```
int main()
{
        float xs[NUM_PTS] = {7.7857203028720035, 7.642458246115663, 6.215313359871936, 7.059634724385643, 6.670815698546658, 6.876282852217749, 7.036072823998538, 7.290710191791188, 7.639716393843683, 7.752535310413255, 6.1089490156447095, 7.6043411248489505, 7.418263501562334, 6.2496483487612196, 6.8065125974698395, 7.434551692906639, 6.51122811120167, 7.198011826278233, 7.874699069289809, 6.507268194942792, 12, 9, 12, 11, 10};

        float ys[NUM_PTS] = {6.663959610602355, 6.0833932514505, 7.1901041284124805, 6.837614857113309, 7.24503886439218, 7.47176421269738, 7.157717200452314, 7.980448542746833, 6.826401869632361, 7.647518865141043, 7.437274473128388, 7.472813287622878, 7.081873708605056, 7.915294590939337, 6.43390232064712, 7.988414876843875, 7.3426188638785606, 7.434664291111027, 6.703619537079404, 6.804945018073546, 14, 9, 7, 11, 13};


        cout << "centar = (" << median(xs, NUM_PTS) << ", " << median(ys, NUM_PTS) << ")\n";

        return 0;
}
```

U 37 linija koda dobijamo isti rezultat

```
centar = (7.41826, 7.34262)
```

Nešto o terminologiji

Potprogram (eng. *subprogram*) je za nas skup naredbi sa jasno definisanim ulazom i izlazom, koje je moguće ponovo koristiti, nad proizvoljnim podacima odgovarajućeg tipa.

Nešto o terminologiji

Osim termina potprogram, za takav skup naredbi se još koriste i nazivi funkcija (eng. *function*), metoda (eng. *method*), rutina (eng. *routine*), podrutina (eng. *subroutine*), procedura (eng. *procedure*)...

Nešto o terminologiji

Neki autori insistiraju na tome da ovi termini nisu ekvivalentni

(na primer, da je procedura isključivo potrprogram koji ne vraća vrednost, ili da je metoda funkcija unutar klase), no oko toga često ne postoji konsenzus.

Nešto o terminologiji

U svakom slučaju, to su sve detalji koji za nas nisu bitni. Ono što je bitno je razumeti **SUŠTINU** pojma potprograma.

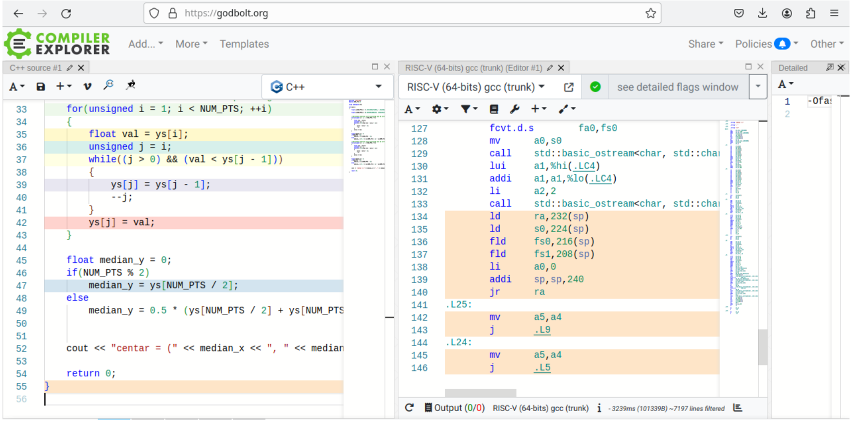
Zašto koristimo potprograme?

Dobijamo kraći izvorni kod

```
                }
                a[j] = val;
        }

        if(len % 2)
                return a[len / 2];

        return 0.5 * (a[len / 2] + a[len / 2 + 1]);
}

int main()
{
        float xs[NUM_PTS] = {7.7857203028720035, 7.642458246115663, 6.215313359871936
, 7.059634724385643, 6.670815698546658, 6.876282852217749, 7.036072823998538, 7.29071
0191791188, 7.639716393843683, 7.752535310413255, 6.1089490156447095, 7.6043411248489
505, 7.418263501562334, 6.2496483487612196, 6.8065125974698395, 7.434551692906639, 6.
5112281120167, 7.198011826278233, 7.874699069289809, 6.507268194942792, 12, 9, 12, 1
1, 10};

        float ys[NUM_PTS] = {6.663959610602355, 6.0833932514505, 7.1901041284124805,
6.837614857113309, 7.24503886439218, 7.47176421269738, 7.157717200452314, 7.980448542
746833, 6.826401869632361, 7.647518865141043, 7.437274473128388, 7.472813287622878, 7
.081873708605056, 7.915294590939337, 6.43390232064712, 7.988414876843875, 7.342618863
8785606, 7.434664291111027, 6.703619537079404, 6.804945018073546, 14, 9, 7, 11, 13};


        cout << "centar = (" << median(xs, NUM_PTS) << ", " << median(ys, NUM_PTS) <<
 ")\n";

        return 0;
}

/32% kraci kod
center_with_function.cpp [+][RO]                               39,1          Bot
```

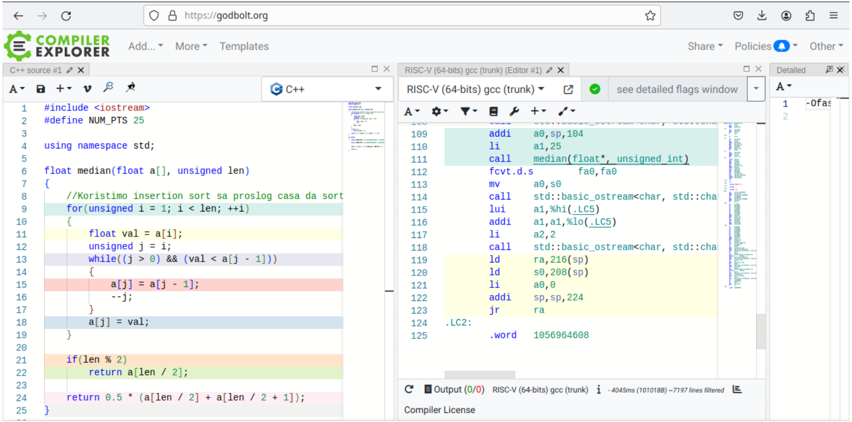
```
                xs[j] = val;
        }

        float median_x = 0;
        if(NUM_PTS % 2)
                median_x = xs[NUM_PTS / 2];
        else
                median_x = 0.5 * (xs[NUM_PTS / 2] + xs[NUM_PTS / 2 + 1]);

        //Koristimo insertion sort sa proslog casa da sortiramo y koordinate
        for(unsigned i = 1; i < NUM_PTS; ++i)
        {
                float val = ys[i];
                unsigned j = i;
                while((j > 0) && (val < ys[j - 1]))
                {
                        ys[j] = ys[j - 1];
                        --j;
                }
                ys[j] = val;
        }

        float median_y = 0;
        if(NUM_PTS % 2)
                median_y = ys[NUM_PTS / 2];
        else
                median_y = 0.5 * (ys[NUM_PTS / 2] + ys[NUM_PTS / 2 + 1]);


        cout << "centar = (" << median_x << ", " << median_y << ")\n";

        return 0;
}
center.cpp [RO]                                                55,1          Bot
```

Ali i kraći mašinski kod



Asembli bez upotrebe funkcije za izračunavanje medijane

Ali i kraći mašinski kod



Asembli sa upotrebom funkcije za izračunavanje medijane

O tome kako su potprogrami zapravo implementirani ćemo više reći uskoro

Održavanje je često bitnije od razvoja

Analogija: nije problem samo rezbariti drvo, već i brisati prašinu

Potprogrami nam omogućuju modularni razvoj softvera

Menjamo implementaciju funkcije
ali njen interfejs ostaje isti

```c
void sort(float a[], unsigned len)
{
	//Koristimo insertion sort sa proslog casa da sortiramo niz
	for(unsigned i = 1; i < len; ++i)
	{
		float val = a[i];
		unsigned j = i;
		while((j > 0) && (val < a[j - 1]))
		{
			a[j] = a[j - 1];
			--j;
		}
		a[j] = val;
	}
}

float median(float a[], unsigned len)
{
	sort(a, len);

	if(len % 2)
		return a[len / 2];

	return 0.5 * (a[len / 2] + a[len / 2 + 1]);
}

int main()
{
```

```c
void sort(float a[], unsigned len)
{
	//Koristimo Shell sort sa proslog casa da sortiramo niz
	unsigned gap = 1;
	do
	{
		gap *= 3;
		++gap;
	} while(gap <= len);

	do
	{
		gap /= 3;
		for(unsigned i = gap; i < len; ++i)
		{
			float val = a[i];
			unsigned j = i;
			while(val < a[j - gap])
			{
				a[j] = a[j - gap];
				j -= gap;
				if(j < gap)
					break;
			}
			a[j] = val;
		}
	} while(gap > 1);
}

float median(float a[], unsigned len)
{
	sort(a, len);
```

Samim tim ne moramo
menjati ni logiku preostalog
dela programa

Neke od prednosti modularnosti

- Prvo možemo napisati najjednostavniju moguću implementaciju (na primer selection sort)
- Ako se ispostavi da nije dovoljno dobra za naše potrebe, onda možemo utrošiti više vremena na pisanje bolje ali složenije implementacije (na primer Shell sort)

Neke od prednosti modularnosti

Modularnost je veoma korisna za timski razvoj: dok jedna osoba piše jedan potprogram, druga može da piše neki drugi

Neke od prednosti modularnosti

Vrlo često, ni ne moramo pisati nove funkcije, već možemo koristiti one koje su drugi napisali, ili koje smo mi ranije napisali za neki drugi program

Nije dobro izmišljati toplu vodu, ali

Alan Mishchenko

ABC: The Way It Should Have Been Designed

Research Scientist
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California, USA

Thursday, 28 September 2017 at 11:00 in room BC 420

Abstract:

Twelve years ago, in September 2005, the first public version of ABC was released. It featured technology-independent synthesis by DAG-aware rewriting, technology mapping for standard cells and lookup tables, and simple combinational equivalence checking, all based on the And-Inverter Graphs (AIG) data-structure used to unify the computation flow. In the coming years ABC has been adopted as an optimization engine and a research environment by a number of academic and industrial users. The use that followed exposed a number of shortcomings in the original design of ABC. This talk focuses on what is present and, more importantly, what is missing in ABC, and how ABC could be redesigned to make it more versatile and user-friendly. The motivation for this talk is to help academic researchers maximize the usefulness of their tools and set a new standard for future versions of ABC.

ABC: The Way It Should Have Been Designed

Alan Mishchenko

Department of EECS, UC Berkeley

Jedna od glavnih poruka: ne treba se suzdržavati od pisanja ključnih delova programa iznova i iznova

Nešto o paradigmama

Šta smo radili do sada?

Način programiranja koji smo do sada proučavali nazivamo *proceduralnim programiranjem*

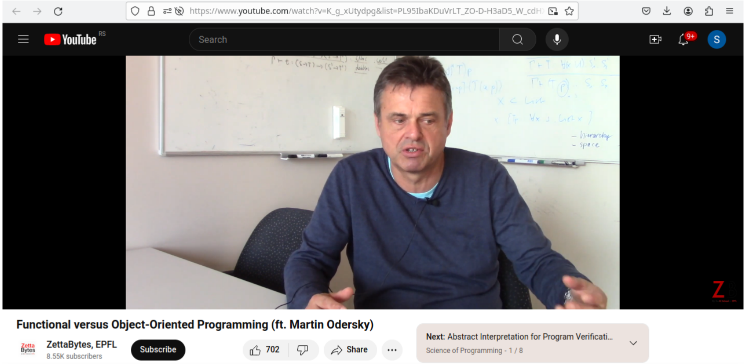
Šta smo radili do sada?

Proceduralno programiranje se zasniva na podeli problema na potprograme (funkcije, procedure)

Šta ćemo raditi od sledeće nedelje?

Počevši od sledeće nedelje, bavićemo se objektno-orijentisanim programiranjem, u kom potprograme pridružujemo podacima nad kojima operišu, stvarajući posebnu vrstu modula koju nazivamo *objektima*

Postoji i funkcionalno programiranje



https://www.youtube.com/watch?v=K_g_xUtydpg&list=PL95IbaKDuVrLT_ZO-D-H3aD5_W_cdHXEq

I proceduralno i funkcionalno programiranje sežu u 1930-te

Alan Turing
OBE FRS

Turing in 1936

Born	Alan Mathison Turing 23 June 1912 Maida Vale, London, England
Died	7 June 1954 (aged 41) Wilmslow, Cheshire, England

Proceduralno programiranje

Alonzo Church

Born	June 14, 1903 Washington, D.C., U.S.
Died	August 11, 1995 (aged 92) Hudson, Ohio, U.S.

Funkcionalno programiranje

Nagradno pitanje

Šta su Tjuring i Čerč bili po obrazovanju?

Alan Mathison Turing

Biography MathSciNet

Ph.D. Princeton University 1938 

Dissertation: *Systems of Logic Based on Ordinals*

Advisor: Alonzo Church

Siméon Denis Poisson

Biography MathSciNet

Ph.D. École Polytechnique 1800

Dissertation:

Advisor 1: Joseph Louis Lagrange
Advisor 2: Pierre-Simon Laplace

↓

Michel Chasles

Biography MathSciNet

Ph.D. École Polytechnique 1814

Dissertation:

Advisor: Siméon Denis Poisson

→

H. A. (Hubert Anson) Newton

B.A. Yale University 1850

Dissertation:

Advisor: Michel Chasles

↓

E. H. (Eliakim Hastings) Moore

Biography MathSciNet

Ph.D. Yale University 1885

Dissertation: *Extensions of Certain Theorems of Clifford and Cayley in the Geometry of n Dimensions*

Advisor: H. A. (Hubert Anson) Newton

→

Oswald Veblen

Biography MathSciNet

Ph.D. The University of Chicago 1903

Dissertation: *A System of Axioms for Geometry*

Advisor: E. H. (Eliakim Hastings) Moore

↓

Alonzo Church

Biography MathSciNet

Ph.D. Princeton University 1927

Dissertation: *Alternatives to Zermelo's Assumption*

Mathematics Subject Classification: 06—Order, lattices, ordered algebraic structures

Advisor: Oswald Veblen

→

Alan Mathison Turing

Biography MathSciNet

Ph.D. Princeton University 1938

Dissertation: *Systems of Logic Based on Ordinals*

Advisor: Alonzo Church

Siméon Denis Poisson

Biography MathSciNet

Ph.D. École Polytechnique 1800

Dissertation:

Advisor 1: Joseph Louis Lagrange
Advisor 2: Pierre-Simon Laplace

Michel Chasles

Biography MathSciNet

Ph.D. École Polytechnique 1814

Dissertation:

Advisor: Siméon Denis Poisson

Gaston Darboux

Biography MathSciNet

Ph.D. École Normale Supérieure Paris 1866

Dissertation: *Sur les surfaces orthogonales*

Advisor: Michel Chasles

C. Émile (Charles) Picard

Biography MathSciNet

Ph.D. École Normale Supérieure Paris 1877

Dissertation: *Applications des complexes lineaires a l'etude des surfaces et des courbes gauches*

Mathematics Subject Classification: 14—Algebraic geometry

Advisor: Gaston Darboux

Mihailo Petrović

Ph.D. Université Paris IV-Sorbonne 1894

Dissertation: *Sur le zeros et les infinis des integrales des equations differentielles algebriques*

Advisor 1: Charles Hermite
Advisor 2: C. Émile (Charles) Picard
Advisor 3: Henri Poincaré

Jovan Karamata

MathSciNet

Ph.D. University of Belgrade 1926

Dissertation: *O jednoj vrsti granica sličnih određenim integralima*

Advisor: Mihailo Petrović

Bogoljub Stanković

MathSciNet

Ph.D. Serbian Academy of Sciences 1954

Dissertation: *Sur une classe d'equations integrales singulieres*

Mathematics Subject Classification: 45—Integral equations

Advisor 1: Vojislav Gregor Avakumović
Advisor 2: Jovan Karamata

Stevan Pilipović

MathSciNet

Ph.D. University of Novi Sad 1979

Dissertation: *On Some Spaces of Generalized Functions from the Point of View of Sequential Theory*

Mathematics Subject Classification: 40—Sequences, series, summability

Advisor: Bogoljub Stanković

Vratimo se na potprograme

Šta nam oni još omogućuju?

Verifikacija

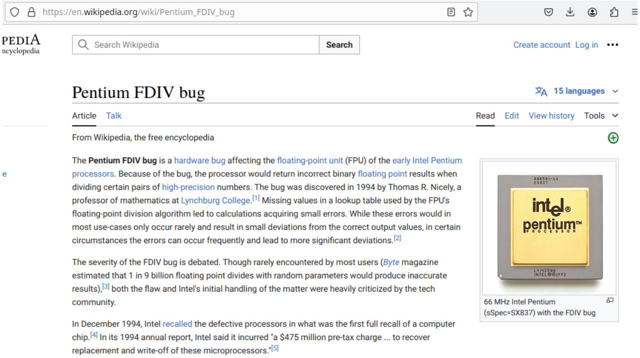
Neophodno je da osiguramo da naš program ispravno radi

Dva osnovna pristupa verifikaciji

1. Funkcionalna verifikacija (testiranje): za niz ulaznih podataka PROVERAVAMO da li je izlaz očekivan

2. Formalna verifikacija (često je nazivamo samo „verifikacija”): DOKAZUJEMO da je naš program ispravan za SVE ulazne podatke

Funkcionalna verifikacija (testiranje)



https://en.wikipedia.org/wiki/Pentium_FDIV_bug

Search Wikipedia | Search | Create account | Log in

Pentium FDIV bug

15 languages

Article | Talk | Read | Edit | View history | Tools

From Wikipedia, the free encyclopedia

The **Pentium FDIV bug** is a hardware bug affecting the floating-point unit (FPU) of the early Intel Pentium processors. Because of the bug, the processor would return incorrect binary floating point results when dividing certain pairs of high-precision numbers. The bug was discovered in 1994 by Thomas R. Nicely, a professor of mathematics at Lynchburg College.[1] Missing values in a lookup table used by the FPU's floating-point division algorithm led to calculations acquiring small errors. While these errors would in most use-cases only occur rarely and result in small deviations from the correct output values, in certain circumstances the errors can occur frequently and lead to more significant deviations.[2]

The severity of the FDIV bug is debated. Though rarely encountered by most users (*Byte* magazine estimated that 1 in 9 billion floating point divides with random parameters would produce inaccurate results),[3] both the flaw and Intel's initial handling of the matter were heavily criticized by the tech community.

In December 1994, Intel recalled the defective processors in what was the first full recall of a computer chip.[4] In its 1994 annual report, Intel said it incurred "a $475 million pre-tax charge ... to recover replacement and write-off of these microprocessors."[5]

66 MHz Intel Pentium (sSpec=SX837) with the FDIV bug

Testiranje nam može pomoći da otkrijemo da naš program ne radi, ali ako ne prođemo sve kombinacije ulaznih podataka, testiranjem ne možemo dokazati da ispravno radi

Podela programa na potprograme smanjuje broj kombinacija ulaza u svakom od njih

Tada ograničen broj testova pokriva veći broj mogućih kombinacija i povećava verovatnoću da smo otkrili sve greške

Kako izgleda testiranje?

Funkcija koju testiramo

ulazi

a = {2, 5, 7, 1, 4}

a = {1, 1, 8, 6, 1}

a = {13, 7}

```
float median(float a[], unsigned len)
{
        sort(a, len);

        if(len % 2)
                return a[len / 2];

        return 0.5 * (a[len / 2] + a[len / 2 + 1]);
}
```

proizvedeni rezultat $\stackrel{?}{=}$

očekivani rezultati

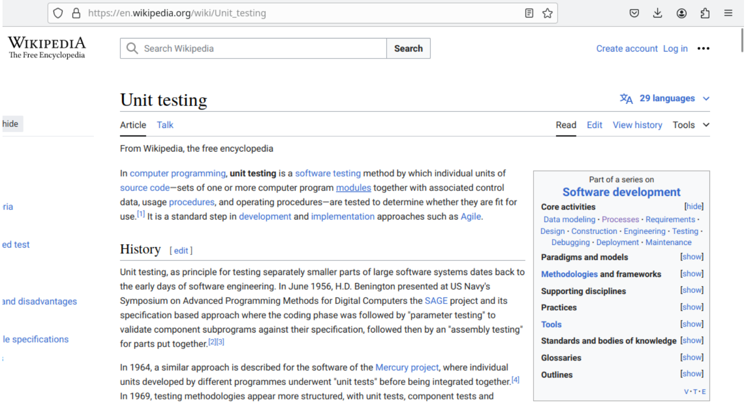
4

1

10

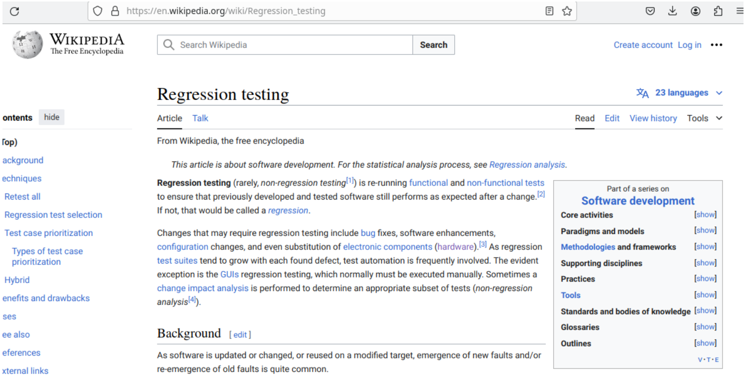
Ako uočimo da se rezultat razlikuje od očekivanog, detektovali smo grešku (ali je još nismo locirali)

Lociranje grešaka je lakše ako najpre testiramo svaku funkciju posebno



Logika: ako funkcija nije ispravna, neće biti ni druga koja je koristi

Testove bi trebalo automatski pokretati pri svakoj izmeni


https://en.wikipedia.org/wiki/Regression_testing

WIKIPEDIA
The Free Encyclopedia

Search Wikipedia | Search

Create account Log in

Regression testing

23 languages

Article Talk

Read Edit View history Tools

From Wikipedia, the free encyclopedia

This article is about software development. For the statistical analysis process, see Regression analysis.

Regression testing (rarely, *non-regression testing*[1]) is re-running functional and non-functional tests to ensure that previously developed and tested software still performs as expected after a change.[2] If not, that would be called a *regression*.

Changes that may require regression testing include bug fixes, software enhancements, configuration changes, and even substitution of electronic components (hardware).[3] As regression test suites tend to grow with each found defect, test automation is frequently involved. The evident exception is the GUIs regression testing, which normally must be executed manually. Sometimes a change impact analysis is performed to determine an appropriate subset of tests (*non-regression analysis*[4]).

Background [edit]

As software is updated or changed, or reused on a modified target, emergence of new faults and/or re-emergence of old faults is quite common.

Part of a series on
Software development

Core activities	[show]
Paradigms and models	[show]
Methodologies and frameworks	[show]
Supporting disciplines	[show]
Practices	[show]
Tools	[show]
Standards and bodies of knowledge	[show]
Glossaries	[show]
Outlines	[show]

V・T・E

Da bismo bili sigurni da nova izmena nije pokvarila nešto što smo nakon prethodnog testiranja pretpostavili da je ispravno

Kako nalazimo očekivane rezultate?

ulazi

a = {2, 5, 7, 1, 4}

a = {1, 1, 8, 6, 1}

a = {13, 7}

Funkcija koju testiramo

```
float median(float a[], unsigned len)
{
        sort(a, len);

        if(len % 2)
                return a[len / 2];

        return 0.5 * (a[len / 2] + a[len / 2 + 1]);
}
```

proizvedeni rezultat $\stackrel{?}{=}$

očekivani rezultati

4

1

10

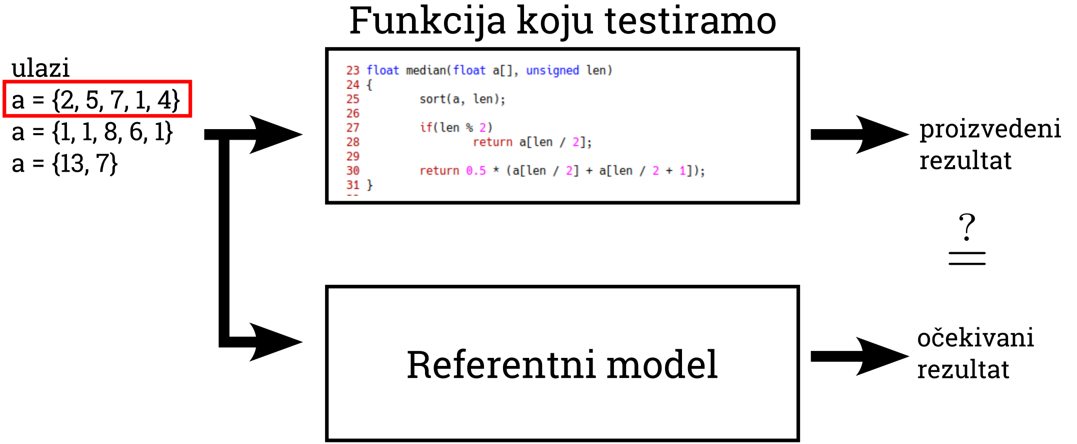
Kako nalazimo očekivane rezultate?

Jedna mogućnost je da ih ručno izračunamo

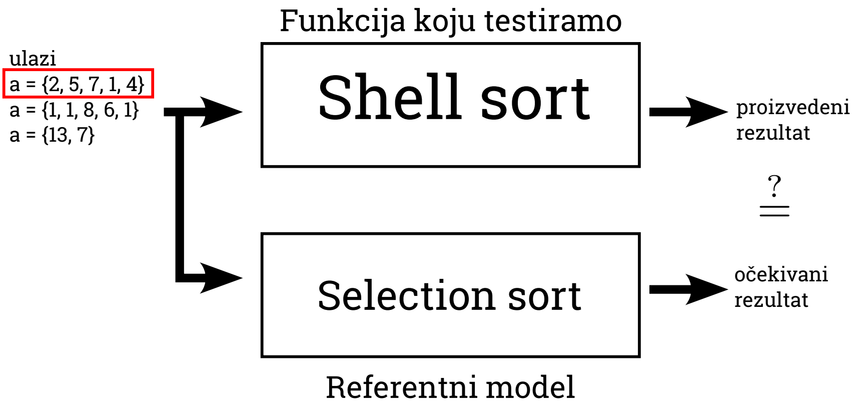
Kako nalazimo očekivane rezultate?

Druga je da imamo referentni model u čiju ispravnost verujemo i sa kojim poredimo ponašanje testirane funkcije

Šta može biti referentni model?

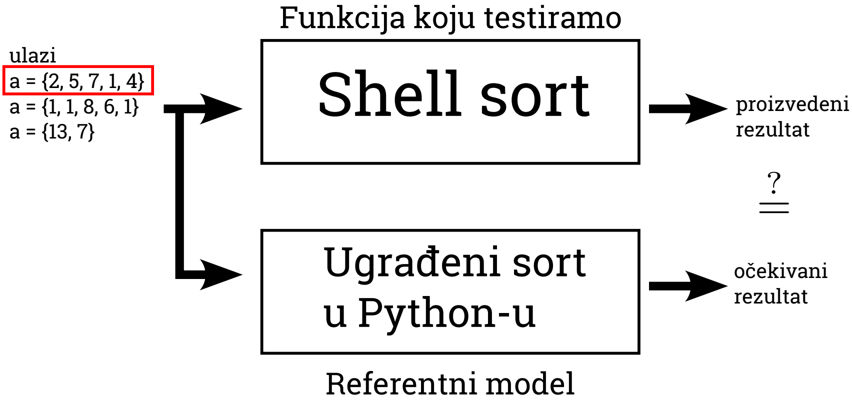


Šta može biti referentni model?



Referentni model je najčešće sporija ali znatno jednostavnija implementacija iste funkcije

Šta može biti referentni model?

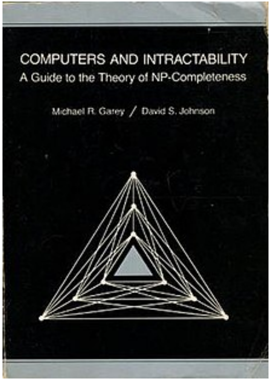


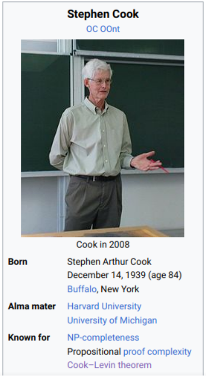
Referentni model je najčešće sporija ali znatno jednostavnija implementacija iste funkcije

Testiranje upotrebom referentnog modela

Ulazne podatke možemo generisati i nasumično, a očekivane rezultate možemo sačuvati u fajl, da ne bismo mnogo puta tokom razvoja čekali na evaluaciju spore implementacije korišćene u modelu

Opšti princip: upotreba testiranja za procenu optimalnosti





Nekada su problemi koje rešavamo toliko složeni da ne znamo ni jedan algoritam pomoću kog možemo da nađemo tačno rešenje za proizvoljnu instancu problema u razumnom vremenu

Jedan konkretan primer

Problem: Imamo graf i potrebno je da svakom čvoru dodelimo koordinate u 2D ravni, tako da se nijedna dva čvora ne preklapaju a da zbir Euklidovih rastojanja između svaka dva čvora među kojima postoji grana bude minimalan.

U opštem slučaju, ne znamo kako da nađemo optimalno rešenje.

Ali možemo da osmislimo algoritme koji konstruišu prihvatljiva rešenja

`https://github.com/verilog-to-routing/vtr-verilog-to-routing`

Legalno naspram optimalnog rešenja

Postoje uslovi koje rešenje mora da zadovolji da bi bilo **LEGALNO**

Na primer, u prethodnom problemu, svako rešenje u kom su svim čvorovima dodeljene koordinate i nijedna dva čvora se ne preklapaju je legalno

Legalno naspram optimalnog rešenja

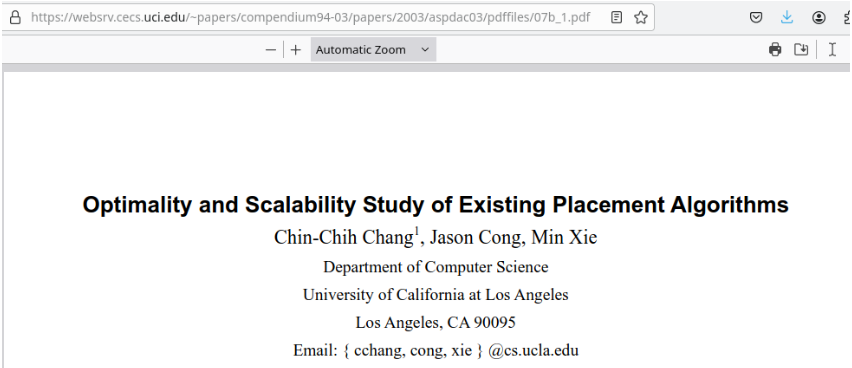
Optimalno rešenje je ono legalno rešenje koje je najbolje spram postavljenog kriterijuma

U prethodnom primeru, to je rešenje sa minimalnom zbirnom dužinom grana

Optimalno rešenje ne mora biti jedinstveno

Kako da proverimo koliko je naš algoritam daleko od optimuma?

Konstruišemo instance problema za koje unapred znamo optimalno rešenje



Optimality and Scalability Study of Existing Placement Algorithms

Chin-Chih Chang[1], Jason Cong, Min Xie

Department of Computer Science

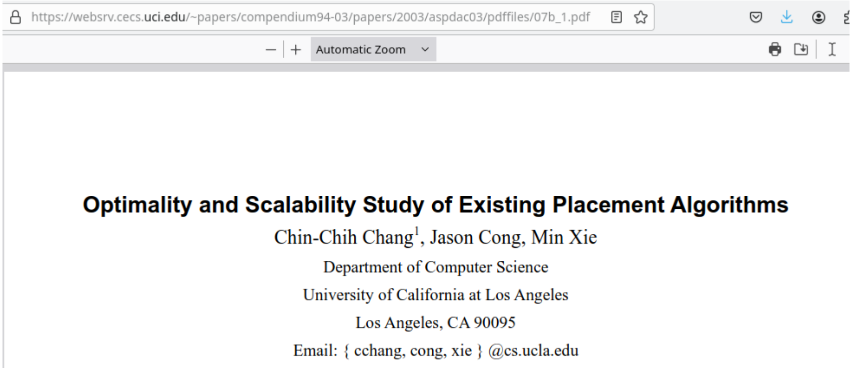
University of California at Los Angeles

Los Angeles, CA 90095

Email: { cchang, cong, xie } @cs.ucla.edu

Slično kao kod funkcionalne verifikacije, ni ovde ne možemo da garantujemo optimalnost algoritma u opštem slučaju tako što potvrdimo da daje optimalan odgovor na sve generisane primere

Konstruišemo instance problema za koje unapred znamo optimalno rešenje



Optimality and Scalability Study of Existing Placement Algorithms

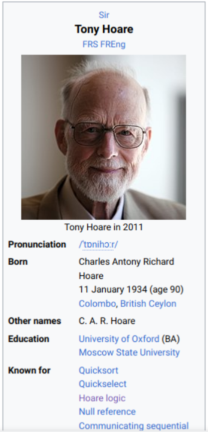
Chin-Chih Chang[1], Jason Cong, Min Xie
Department of Computer Science
University of California at Los Angeles
Los Angeles, CA 90095
Email: { cchang, cong, xie } @cs.ucla.edu

Međutim, ako su primeri koje konstruišemo slični onim koji se javljaju u praksi, ovakvim pristupom možemo da detektujemo da je naš algoritam daleko od optimuma, pa možemo da pokušamo da ga unapredimo

Formalna verifikacija

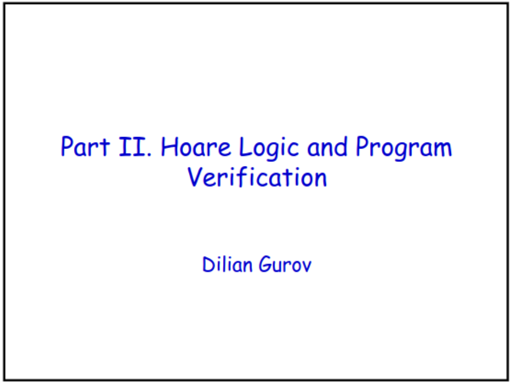
Formalna verifikacija omogućuje dokazivanje da naš program ispravno radi za **SVE** kombinacije ulaza

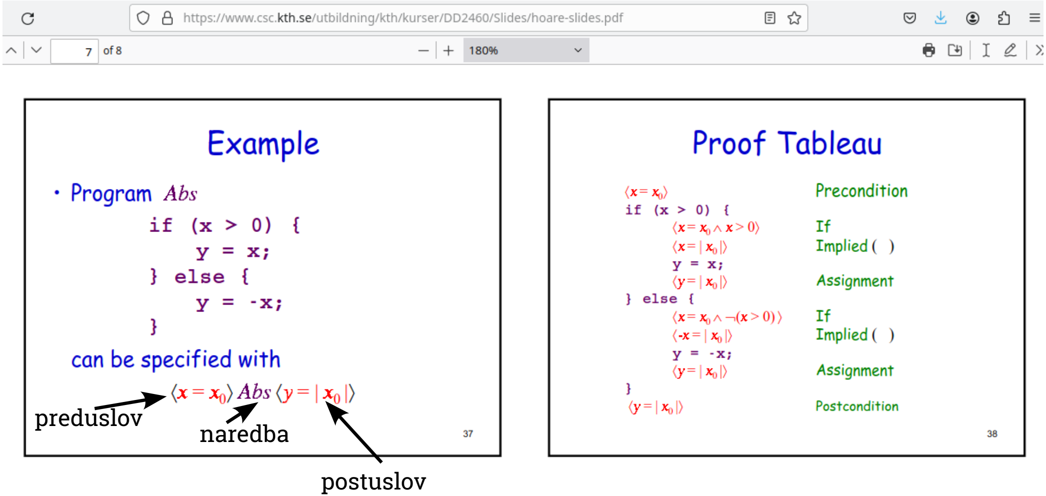
Da bismo znali šta dokazujemo, potrebna nam je formalna specifikacija programa



Za to često upotrebljavamo Horovu logiku

Za kratku ilustraciju ćemo pozajmiti jedan primer iz Švedske





Ako se program neposredno pre izvršenja naredbe nalazi u stanju u kom je preduslov zadovoljen, nakon što izvršimo naredbu, mora preći u stanje u kom je zadovoljen postuslov

$\langle x = x_0 \rangle$ preduslov za čitav program; pretpostavimo da važi

```
if (x > 0) {
```

$\langle x = x_0 \land x > 0 \rangle$

$\langle x = |x_0| \rangle$

```
    y = x;
```

$\langle y = |x_0| \rangle$

```
} else {
```

$\langle x = x_0 \land \neg(x > 0) \rangle$

$\langle -x = |x_0| \rangle$

```
    y = -x;
```

$\langle y = |x_0| \rangle$

```
}
```

$\langle y = |x_0| \rangle$ postuslov za čitav program

$\langle x = x_0 \rangle$ preduslov za čitav program; pretpostavimo da važi

```
if (x > 0) {
```

$\boxed{\langle x = x_0 \wedge x > 0 \rangle}$ Ako smo ušli u granu if(x > 0), x mora biti veće of nule

$\langle x = |x_0| \rangle$

```
        y = x;
```

$\langle y = |x_0| \rangle$

```
} else {
```

$\langle x = x_0 \wedge \neg(x > 0) \rangle$

$\langle -x = |x_0| \rangle$

```
        y = -x;
```

$\langle y = |x_0| \rangle$

```
}
```

$\langle y = |x_0| \rangle$ postuslov za čitav program

$\langle x = x_0 \rangle$ preduslov za čitav program; pretpostavimo da važi

```
if (x > 0) {
```

$\langle x = x_0 \wedge x > 0 \rangle$

$\boxed{\langle x = | x_0 | \rangle}$ Iz prethodnog iskaza, ovo važi po definiciji

```
    y = x;
```

$\langle y = | x_0 | \rangle$

```
} else {
```

$\langle x = x_0 \wedge \neg(x > 0) \rangle$

$\langle -x = | x_0 | \rangle$

```
    y = -x;
```

$\langle y = | x_0 | \rangle$

```
}
```

$\langle y = | x_0 | \rangle$ postuslov za čitav program

$\langle x = x_0 \rangle$ preduslov za čitav program; pretpostavimo da važi

```
if (x > 0) {
```

$\langle x = x_0 \wedge x > 0 \rangle$

$\langle x = |x_0| \rangle$

```
    y = x;
```

$\boxed{\langle y = |x_0| \rangle}$ Prosta posledica dodele

```
} else {
```

$\langle x = x_0 \wedge \neg(x > 0) \rangle$

$\langle -x = |x_0| \rangle$

```
    y = -x;
```

$\langle y = |x_0| \rangle$

```
}
```

$\langle y = |x_0| \rangle$ postuslov za čitav program

$\langle x = x_0 \rangle$ preduslov za čitav program; pretpostavimo da važi

```
if (x > 0) {
```

$\langle x = x_0 \wedge x > 0 \rangle$

$\langle x = | x_0 | \rangle$

```
    y = x;
```

$\langle y = | x_0 | \rangle$

```
} else {
```

$\boxed{\langle x = x_0 \wedge \neg(x > 0) \rangle}$ Ako smo u else grani, x je <= 0

$\langle -x = | x_0 | \rangle$

```
    y = -x;
```

$\langle y = | x_0 | \rangle$

```
}
```

$\langle y = | x_0 | \rangle$ postuslov za čitav program

$\langle x = x_0 \rangle$ preduslov za čitav program; pretpostavimo da važi

```
if (x > 0) {
```

$\langle x = x_0 \wedge x > 0 \rangle$

$\langle x = |x_0| \rangle$

```
    y = x;
```

$\langle y = |x_0| \rangle$

```
} else {
```

$\langle x = x_0 \wedge \neg(x > 0) \rangle$

$\boxed{\langle -x = |x_0| \rangle}$ Tada ovo važi po definiciji apsolutne vrednosti

```
    y = -x;
```

$\langle y = |x_0| \rangle$

```
}
```

$\langle y = |x_0| \rangle$ postuslov za čitav program

$\langle x = x_0 \rangle$ preduslov za čitav program; pretpostavimo da važi

```
if (x > 0) {
```

$\langle x = x_0 \wedge x > 0 \rangle$

$\langle x = | x_0 | \rangle$

```
    y = x;
```

$\langle y = | x_0 | \rangle$

```
} else {
```

$\langle x = x_0 \wedge \neg(x > 0) \rangle$

$\langle -x = | x_0 | \rangle$

```
    y = -x;
```

$\boxed{\langle y = | x_0 | \rangle}$ A ovo je posledica dodele

```
}
```

$\langle y = | x_0 | \rangle$ postuslov za čitav program

$\langle x = x_0 \rangle$ preduslov za čitav program; pretpostavimo da važi

```
if (x > 0) {
```

$\langle x = x_0 \wedge x > 0 \rangle$

$\langle x = | x_0 | \rangle$

```
    y = x;
```

$\boxed{\langle y = | x_0 | \rangle}$

```
} else {
```

$\langle x = x_0 \wedge \neg(x > 0) \rangle$

$\langle -x = | x_0 | \rangle$

```
    y = -x;
```

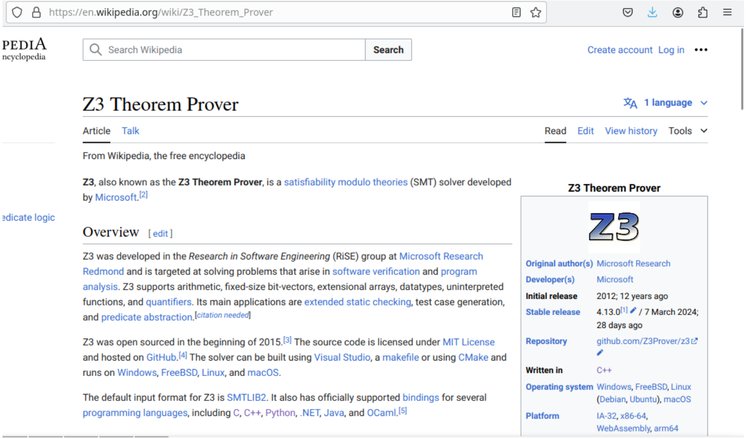
$\boxed{\langle y = | x_0 | \rangle}$

```
}
```

$\langle y = | x_0 | \rangle$ postuslov za čitav program

Pošto smo dokazali da postuslov proističe iz preduslova za sve moguće putanje izvršenja programa, dokazali smo da je program tačan

Dokaze ne moramo sprovoditi ručno



Međutim, automatski dokazivači imaju ograničenu moć

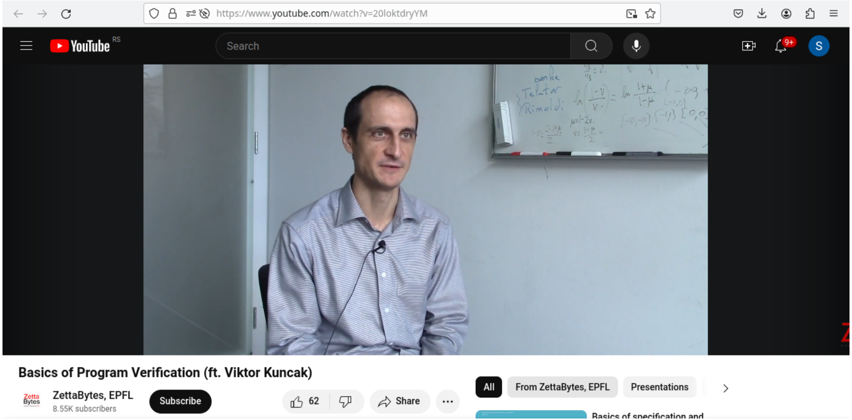
Prednost modularnosti

Što je veći broj stanja u programu i putanja između njih, automatski dokazivač mora da pretraži veći prostor, pa je manje verovatno da će naći dokaz u razumnom vremenu

Prednost modularnosti

Potprogrami imaju manji prostor stanja od celog programa, pa je lakše verifikovati neke od njih (ciljamo na one kod kojih greške imaju najopasnije posledice)

Više o verifikaciji možemo čuti od Viktora Kunčaka



https://www.youtube.com/watch?v=20loktdryYM lara.epfl.ch/~kuncak/cv.pdf

Automatic Zoom

Early Achievements

- Best Student of University of Novi Sad in Class of 2000.
- Aleksandar Popović Award for Best Science Project (*Modular Interpreters in Haskell*, Advisor: Prof. Mirjana Ivanović), University of Novi Sad, 2000.
- Student of the Year of Faculty of Science, University of Novi Sad, 2000.
- Mileva Marić-Einstein Award for accomplishments in Computer Science, University of Novi Sad, 1999
- Awards of Excellence for Student Projects (*Early Deadlock Prevention*, Advisor: Prof. Zoran Budimac), University of Novi Sad, 1999; (*Herbrand's Theorem and the Resolution Method*, Advisor: Prof. Gradimir Vojvodić), University of Novi Sad, 1998
- Fellowship of the Serbian Foundation for Scientific Youth Development, 1995-1998
- University of Novi Sad Fellowship, 1998-2000
- Honorable Mention, 27th Int. Physics Olympiad, Oslo, Norway, 1996, First prizes on National Physics Competition of FR Yugoslavia in 1992, 1994 and 1996.

Mala digresija

Sir
Tony Hoare
FRS FREng

Tony Hoare in 2011

Pronunciation /ˈtɒnihɔːr/

Leonid Anatolievich Levin

Leonid Levin in 2010

Born	November 2, 1948 (age 75) Dnipropetrovsk, Ukrainian SSR, Soviet Union
Alma mater	Moscow University Massachusetts Institute of

Šta su zajedničko imali Ljevin i Hor?

Mala digresija

Mathematics Genealogy Project

C. A. R. (Tony) Hoare

Ph.D.

Dissertation:

Advisor 1: Andrei Nikolayevich Kolmogorov
Advisor 2: Leslie Fox

Mathematics Genealogy Project

Leonid Anatolievich Levin

MathSciNet

Ph.D. Lomonosov Moscow State University 1972

Dissertation: *Some Theorems on the Algorithmic Approach to Probability Theory and Information Theory (unsuccessful defence for political reasons)*

Advisor 1: Andrei Nikolayevich Kolmogorov

Ph.D. Massachusetts Institute of Technology 1979

Dissertation: *A General Notion of Independence of Mathematical Objects: Its Applications to Some Problems of Probability Theory, Mathematical Logic and Algorithm Theory*

Mathematics Subject Classification: 03—Mathematical logic and foundations

Advisor 1: Albert Ronald da Silva Meyer

Андреј Колмогоров

Лични подаци	
Датум рођења	25. април 1903.
Место рођења	Тамбов, Руска Империја
Датум смрти	26. октобар 1987. (84 год.)
Место смрти	Москва, СССР
Држављанство	СССР

Mala digresija

Mathematical analysis of quicksort shows that, on average, the algorithm takes $O(n \log n)$ comparisons to sort n items. In the worst case, it makes $O(n^2)$ comparisons.

History [edit]

The quicksort algorithm was developed in 1959 by Tony Hoare while he was a visiting student at Moscow State University. At that time, Hoare was working on a machine translation project for the National Physical Laboratory. As a part of the translation process, he needed to sort the words in Russian sentences before looking them up in a Russian-English dictionary, which was in alphabetical order on magnetic tape.[5] After recognizing that his first idea, insertion sort, would be slow, he came up with a new idea. He wrote the partition part in Mercury Autocode but had trouble dealing with the list of unsorted segments. On return to England, he was asked to write code for Shellsort. Hoare mentioned to his boss that he knew of a faster algorithm and his boss bet a sixpence that he did not. His boss ultimately accepted that he had lost the bet. Hoare published a paper about his algorithm in The Computer Journal Volume 5, Issue 1, 1962, Pages 10–16. Later, Hoare learned about ALGOL and its ability to do recursion that enabled him to publish an improved version of the algorithm in ALGOL in *Communications of the Association for Computing Machinery*, the premier computer science journal of the time.[2][6] The ALGOL code is published in Communications of the ACM (CACM), Volume 4, Issue 7 July 1961, pp 321 Algorithm 63: partition and Algorithm 64: Quicksort.

Quicksort gained widespread adoption, appearing, for example, in Unix as the default library sort subroutine. Hence, it lent its name to the C standard library subroutine `qsort`[7] and in the reference implementation of Java.

Robert Sedgewick's PhD thesis in 1975 is considered a milestone in the study of Quicksort where he resolved many open problems related to the analysis of various pivot selection schemes including Samplesort, adaptive partitioning by Van Emden[8] as well as derivation of expected number of comparisons and swaps.[7] Jon Bentley and Doug McIlroy in 1993 incorporated various improvements for use in programming libraries,

Best-case performance	$O(n \log n)$ (simple partition) or $O(n)$ (three-way partition and equal keys)
Average performance	$O(n \log n)$
Worst-case space complexity	$O(n)$ auxiliary (naive) $O(\log n)$ auxiliary (Hoare 1962)
Optimal	No

Kad smo već radili sortiranje

A govorili smo i o Dancigu i rešavanju otvorenih problema za domaći

Андрей Николаевич Колмогоров

Имя при рождении	фр. *Andreï Nikolaïevitch Kolmogorov*
Дата рождения	25 апреля 1903[1][2]
Место рождения	Тамбов, Российская империя[5]
Дата смерти	20 октября 1987[3][4][...] (84 года)

Карацуба Анатолий Алексеевич

Дата рождения	31 января 1937
Место рождения	Грозный
Дата смерти	28 сентября 2008 (71 год)
Место смерти	Москва, Россия
Страна	СССР, Россия
Научная сфера	математика
Место работы	МИАН, МГУ
Альма-матер	МГУ (мехмат)
Учёная степень	доктор физико-

https://en.wikipedia.org/wiki/Karatsuba_algorithm

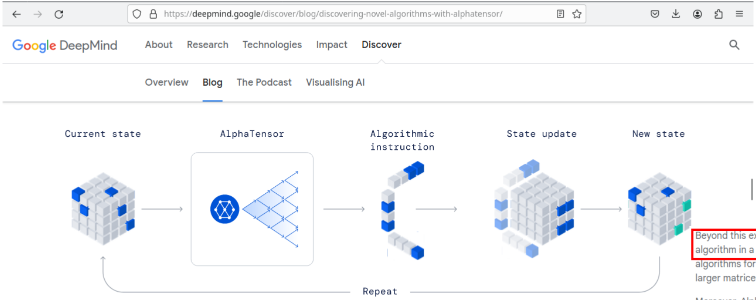
History [edit]

The standard procedure for multiplication of two *n*-digit numbers requires a number of elementary operations proportional to n^2, or $O(n^2)$ in big-O notation. Andrey Kolmogorov conjectured that the traditional algorithm was *asymptotically optimal*, meaning that any algorithm for that task would require $\Omega(n^2)$ elementary operations.

In 1960, Kolmogorov organized a seminar on mathematical problems in cybernetics at the Moscow State University, where he stated the $\Omega(n^2)$ conjecture and other problems in the complexity of computation. Within a week, Karatsuba, then a 23-year-old student, found an algorithm that multiplies two *n*-digit numbers in $O(n^{\log_2 3})$ elementary steps, thus disproving the conjecture. Kolmogorov was very excited about the discovery; he

Priča o Karacubinom množenju:
nekad se isplati ići i protiv autoriteta

Veštačka inteligencija nas polako sustiže



Single-player game played by AlphaTensor, where the goal is to find a correct matrix multiplication algorithm. The state of the game is a cubic array of numbers (shown as grey for 0, blue for 1, and green for -1), representing the remaining work to be done.

Volker Strassen

Volker Strassen giving the Knuth Prize lecture at SODA 2009

Born April 29, 1936 (age 87) Düsseldorf-Gerresheim,

Beyond this example, AlphaTensor's algorithm improves on Strassen's two-level algorithm in a finite field for the first time since its discovery 50 years ago. These algorithms for multiplying small matrices can be used as primitives to multiply much larger matrices of arbitrary size.

Moreover, AlphaTensor also discovers a diverse set of algorithms with state-of-the-art complexity – up to thousands of matrix multiplication algorithms for each size, showing that the space of matrix multiplication algorithms is richer than previously thought.

Dovoljno smo odlutali; vratimo se na potprograme

Sintaksa potprograma

Deklaracija funkcije

povratni_tip identifikator_funkcije(niz_parametara);

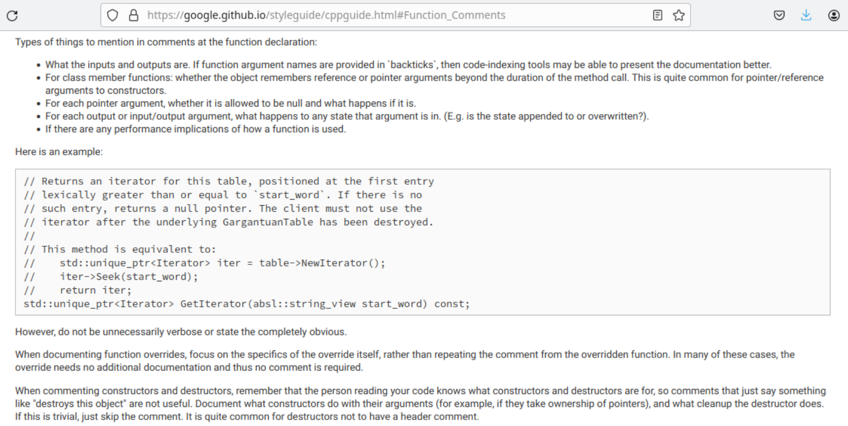
Deklaracija specificira interfejs funkcije sa ostalim funkcijama, ali ne i njenu implementaciju

(govori nam kako da koristimo funkciju, ali ne i kako ona radi)

Primer

```
int abs(int x);
```

Preporuka


https://google.github.io/styleguide/cppguide.html#Function_Comments

Types of things to mention in comments at the function declaration:

- What the inputs and outputs are. If function argument names are provided in `backticks`, then code-indexing tools may be able to present the documentation better.
- For class member functions: whether the object remembers reference or pointer arguments beyond the duration of the method call. This is quite common for pointer/reference arguments to constructors.
- For each pointer argument, whether it is allowed to be null and what happens if it is.
- For each output or input/output argument, what happens to any state that argument is in. (E.g. is the state appended to or overwritten?).
- If there are any performance implications of how a function is used.

Here is an example:

```
// Returns an iterator for this table, positioned at the first entry
// lexically greater than or equal to `start_word`. If there is no
// such entry, returns a null pointer. The client must not use the
// iterator after the underlying GargantuanTable has been destroyed.
//
// This method is equivalent to:
//    std::unique_ptr<Iterator> iter = table->NewIterator();
//    iter->Seek(start_word);
//    return iter;
std::unique_ptr<Iterator> GetIterator(absl::string_view start_word) const;
```

However, do not be unnecessarily verbose or state the completely obvious.

When documenting function overrides, focus on the specifics of the override itself, rather than repeating the comment from the overridden function. In many of these cases, the override needs no additional documentation and thus no comment is required.

When commenting constructors and destructors, remember that the person reading your code knows what constructors and destructors are for, so comments that just say something like "destroys this object" are not useful. Document what constructors do with their arguments (for example, if they take ownership of pointers), and what cleanup the destructor does. If this is trivial, just skip the comment. It is quite common for destructors not to have a header comment.

Preporuka je ostaviti komentar sa opisom šta deklarisana funkcija radi, kako je treba pozvati...

Definicija funkcije

```
povratni_tip identifikator_funkcije(niz_parametara)
{
	telo_funkcije;
}
```

Definicija funkcije specificira njenu implementaciju (govori nam i kako funkcija radi)

Primer

```c
/* Copyright (C) 1991-2024 Free Software Foundation, Inc.
   This file is part of the GNU C Library.

   The GNU C Library is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.

   The GNU C Library is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Lesser General Public License for more details.

   You should have received a copy of the GNU Lesser General Public
   License along with the GNU C Library; if not, see
   <https://www.gnu.org/licenses/>.  */

#include <stdlib.h>

#undef	abs

/* Return the absolute value of I.  */
int
abs (int i)
{
  return i < 0 ? -i : i;
}

```

Definicija funkcije

```
povratni_tip identifikator_funkcije(niz_parametara)
{
    telo_funkcije;
}
```

Primetimo da je po formi početak definicije identičan deklaraciji

Zašto su nam onda uopšte potrebne deklaracije?

Posredna rekurzija

```cpp
#include <iostream>
#define MAX_STEP 20
using namespace std;

void state_read(unsigned step)
{
        if(step == MAX_STEP)
                return;

        cout << "Unesite broj izmedju 10 i 20: ";
        int a;
        cin >> a;

        if(a < 10 || a > 20)
        {
                cout << "Uneti broj nije u opsegu [10, 20].\n";
                state_read(step + 1);
        }
        else
                state_compute(step, a);
}

void state_compute(unsigned step, unsigned a)
{
        if(step == MAX_STEP)
                return;

        cout << "a ^ 2 = " << a * a << endl;

        state_read(step + 1);
}

int main()
{
        state_read(0);

        return 0;
```

Posredna rekurzija

```
circular_dependency.cpp: In function 'void state_read(unsigned int)':
circular_dependency.cpp:20:17: error: 'state_compute' was not declared in this scope
   20 |                 state_compute(step, a);
      |                 ^~~~~~~~~~~~~

shell returned 1
```

Posredna rekurzija

```
#include <iostream>
#define MAX_STEP 20
using namespace std;

void state_compute(unsigned, unsigned);

void state_read(unsigned step)
{
        if(step == MAX_STEP)
                return;

        cout << "Unesite broj izmedju 10 i 20: ";
        int a;
        cin >> a;

        if(a < 10 || a > 20)
        {
                cout << "Uneti broj nije u opsegu [10, 20].\n";
                state_read(step + 1);
        }
        else
                state_compute(step, a);
}

void state_compute(unsigned step, unsigned a)
{
        if(step == MAX_STEP)
                return;

        cout << "a ^ 2 = " << a * a << endl;

        state_read(step + 1);
}

int main()
{
        state_read(0);
```

Posredna rekurzija

```
Unesite broj izmedju 10 i 20: 7
Uneti broj nije u opsegu [10, 20].
Unesite broj izmedju 10 i 20: 13
a ^ 2 = 169
Unesite broj izmedju 10 i 20: 12
a ^ 2 = 144
Unesite broj izmedju 10 i 20: 22
Uneti broj nije u opsegu [10, 20].
Unesite broj izmedju 10 i 20: 19
a ^ 2 = 361
Unesite broj izmedju 10 i 20:
```

Posredna rekurzija

Ukoliko funkcija nije eksplicitno deklarisana, njena definicija će poslužiti i kao deklaracija

Pre prvog poziva bilo koje funkcije, ona mora biti deklarisana, da bi kompajler mogao da izvrši prevođenje u jednom prolazu

Graf poziva

Funkcije možemo predstaviti čvorovima u grafu, a činjenica da jedna poziva drugu usmerenim granama između odgovarajućih čvorova

Ako je takav *graf poziva* (eng. *call graph*) acikličan, onda možemo da složimo definicije u topološkom poretku i izbegnemo upotrebu eksplicitnih deklaracija

U suprotnom, to nije moguće

Ipak, čak i kada nisu potrebne, deklaracije su preporučljive

https://www.gnu.org/software/make/manual/html_node/How-Make-Works.html

2.3 How `make` Processes a Makefile

By default, `make` starts with the first target (not targets whose names start with '.' unless they also contain one or more '/'). This is called the *default goal*. (*Goals* are the targets that `make` strives ultimately to update. You can override this behavior using the command line (see Arguments to Specify the Goals) or with the `.DEFAULT_GOAL` special variable (see Other Special Variables).

In the simple example of the previous section, the default goal is to update the executable program `edit`; therefore, we put that rule first.

Thus, when you give the command:

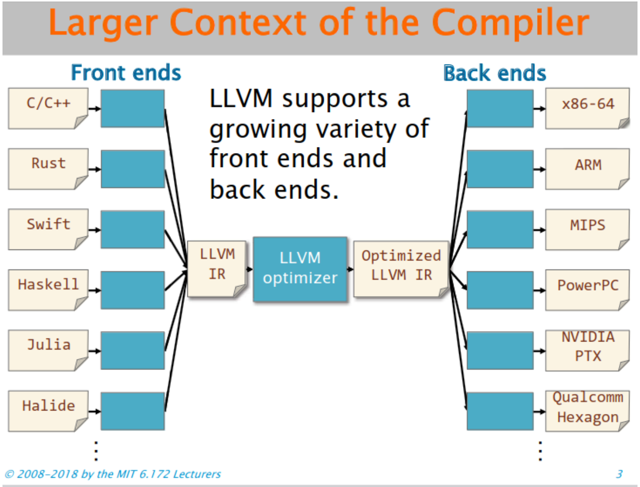
```
make
```

`make` reads the makefile in the current directory and begins by processing the first rule. In the example, this rule is for relinking `edit`; but before `make` can fully process this rule, it must process the rules for the files that `edit` depends on, which in this case are the object files. Each of these files is processed according to its own rule. These rules say to update each '`.o`' file by compiling its source file. The recompilation must be done if the source file, or any of the header files named as prerequisites, is more recent than the object file, or if the object file does not exist.

The other rules are processed because their targets appear as prerequisites of the goal. If some other rule is not depended on by the goal (or anything it depends on, etc.), that rule is not

Podela programa na više fajlova omogućuje jednostavnu inkrementalnu kompilaciju

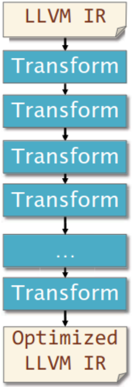
Kako izgleda proces kompilacije?



https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018

Kako izgleda proces kompilacije?

Simple Model of the Compiler



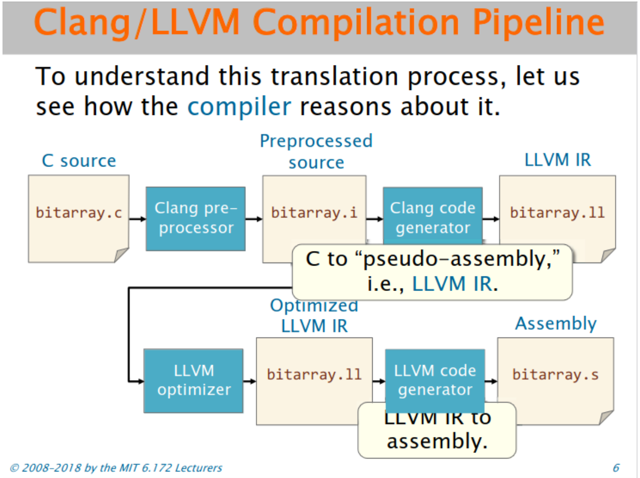
An optimizing compiler performs a sequence of **transformation passes** on the code.

- Each transformation pass analyzes and edits the code to try to optimize the code's performance.
- A transformation pass might run multiple times.
- Passes run in a predetermined order that seems to work well most of the time.

 5

https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018

Kako izgleda proces kompilacije?



https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018

Linker povezuje više objektnih fajlova u jedan izvršni

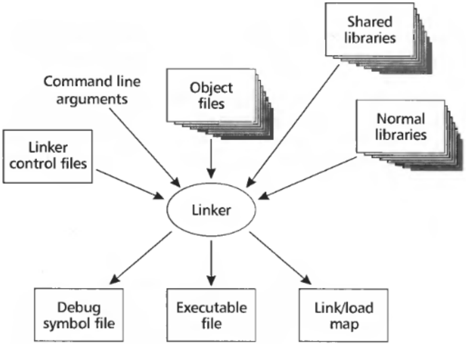


FIGURE 1.1 • The linker process.

Ilustracije iz knjige John R. Levine, ”Linkers and Loaders”

Zaglavlja (eng. *header* fajlovi)

```
struct tacka
{
        int x;
        int y;
};
```

tacka.h 5,1 All

```
#include "tacka.h"
#include "tacka_2.h"
#include <iostream>

using namespace std;

void ispisi_tacku(const struct tacka* t)
{
        cout << "(" << t -> x << ", " << t -> y << ")\n";
}
```

tacka.cpp 10,1

```
#include "tacka.h"

void ispisi_tacku(const struct tacka* t);
```

tacka_2.h 3,41 All

```
#include "tacka.h"
#include "tacka_2.h"

int main()
{
        struct tacka A;
        A.x = -6;
        A.y = 11;

        ispisi_tacku(&A);

        return 0;
```

include_1.cpp 1,1

Zaglavlja (eng. *header* fajlovi)

```
In file included from tacka_2.h:1,
                 from tacka.cpp:2:
tacka.h:1:8: error: redefinition of 'struct tacka'
    1 | struct tacka
      |        ^~~~~
In file included from tacka.cpp:1:
tacka.h:1:8: note: previous definition of 'struct tacka'
    1 | struct tacka
      |        ^~~~~
In file included from tacka_2.h:1,
                 from tacka.cpp:2:
tacka.h:1:8: error: redefinition of 'struct tacka'
    1 | struct tacka
      |        ^~~~~
In file included from tacka.cpp:1:
tacka.h:1:8: note: previous definition of 'struct tacka'
    1 | struct tacka
      |        ^~~~~
```

Zaštita od višestrukog uključenja (eng. *include guards*)

Pri nailasku na direktivu #include, pretprocesor će iskopirati sadržaj fajla iz direktive u trenutni fajl

No, i u fajlu koji uključujemo možemo imati pretprocesorske direktive

Kako bismo osigurali da se neki deo petlje izvrši samo jednom?

Kako bismo osigurali da se neki deo petlje izvrši samo jednom?

```
bool flag = false;
while(uslov)
{
	if(!flag)
	{
		flag = true;
		naredbe;
	}
}
```

Isti princip primenjujemo i ovde

```
#ifndef TACKA_H_
#define TACKA_H_

struct tacka
{
        int x;
        int y;
};

#endif
```

tacka.h 10,6 All

```
#include "tacka.h"

void ispisi_tacku(const struct tacka* t);
```

tacka_2.h 3,41 All

```
#include "tacka.h"
#include "tacka_2.h"
#include <iostream>

using namespace std;

void ispisi_tacku(const struct tacka* t)
{
        cout << "(" << t -> x << ", " << t -> y << ")\n";
}
```

tacka.cpp 10,1

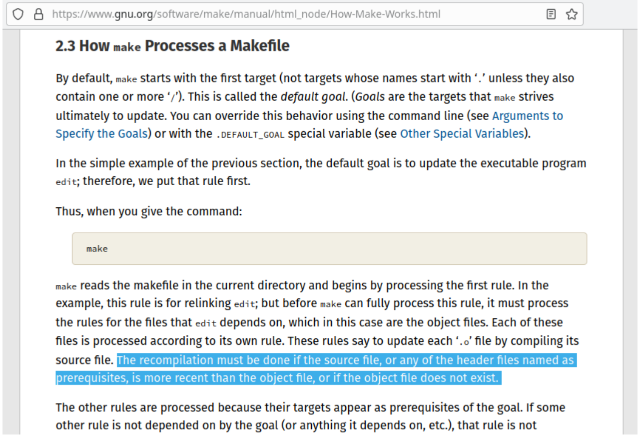
```

int main()
{
        struct tacka A;
        A.x = -6;
        A.y = 11;

        ispisi_tacku(&A);

        return 0;
}
```

include_1.cpp 3,0-1

Isti princip primenjujemo i ovde

```
(-6, 11)
```

Čak i kada nisu potrebne, deklaracije su preporučljive



https://www.gnu.org/software/make/manual/html_node/How-Make-Works.html

2.3 How `make` Processes a Makefile

By default, `make` starts with the first target (not targets whose names start with '`.`' unless they also contain one or more '`/`'). This is called the *default goal*. (*Goals* are the targets that `make` strives ultimately to update. You can override this behavior using the command line (see Arguments to Specify the Goals) or with the `.DEFAULT_GOAL` special variable (see Other Special Variables).

In the simple example of the previous section, the default goal is to update the executable program `edit`; therefore, we put that rule first.

Thus, when you give the command:

```
make
```

`make` reads the makefile in the current directory and begins by processing the first rule. In the example, this rule is for relinking `edit`; but before `make` can fully process this rule, it must process the rules for the files that `edit` depends on, which in this case are the object files. Each of these files is processed according to its own rule. These rules say to update each '`.o`' file by compiling its source file. The recompilation must be done if the source file, or any of the header files named as prerequisites, is more recent than the object file, or if the object file does not exist.

The other rules are processed because their targets appear as prerequisites of the goal. If some other rule is not depended on by the goal (or anything it depends on, etc.), that rule is not

Kada pokrenemo bildovanje, make proverava da li je izvorni fajl menjan od poslednje kompilacije

Čak i kada nisu potrebne, deklaracije su preporučljive

https://www.gnu.org/software/make/manual/html_node/How-Make-Works.html

2.3 How `make` Processes a Makefile

By default, `make` starts with the first target (not targets whose names start with '.' unless they also contain one or more '/'). This is called the *default goal*. (*Goals* are the targets that `make` strives ultimately to update. You can override this behavior using the command line (see Arguments to Specify the Goals) or with the `.DEFAULT_GOAL` special variable (see Other Special Variables).

In the simple example of the previous section, the default goal is to update the executable program `edit`; therefore, we put that rule first.

Thus, when you give the command:

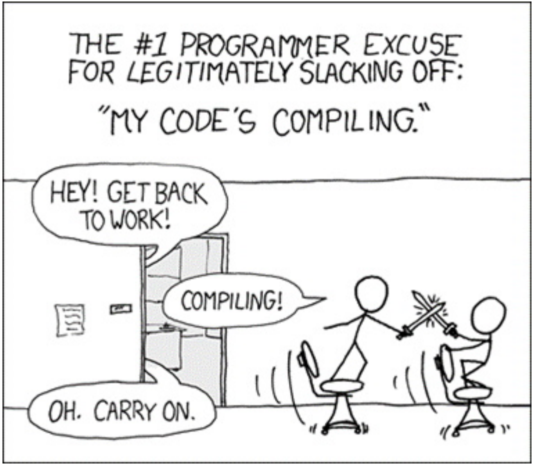
```
make
```

`make` reads the makefile in the current directory and begins by processing the first rule. In the example, this rule is for relinking `edit`; but before `make` can fully process this rule, it must process the rules for the files that `edit` depends on, which in this case are the object files. Each of these files is processed according to its own rule. These rules say to update each '`.o`' file by compiling its source file. The recompilation must be done if the source file, or any of the header files named as prerequisites, is more recent than the object file, or if the object file does not exist.

The other rules are processed because their targets appear as prerequisites of the goal. If some other rule is not depended on by the goal (or anything it depends on, etc.), that rule is not

Ako nije, preskače ga i time štedi značajno vreme

Ta ušteda može biti zaista značajna

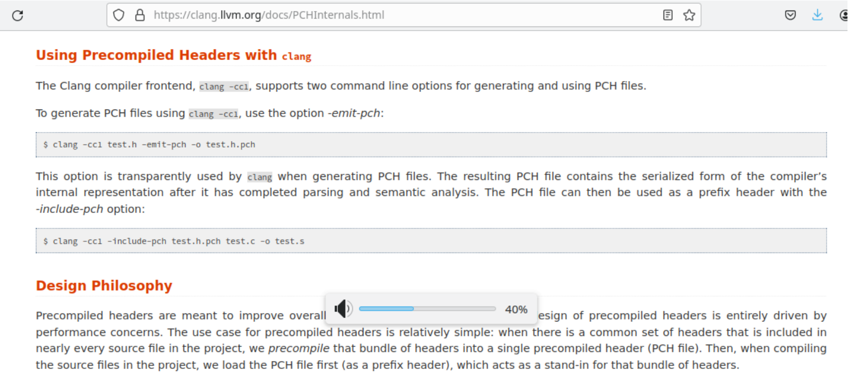


Ne zaboravimo, #include kopira ceo sadržaj fajla

U principu bismo mogli da koristimo i #include ”fajl.cpp”

Ali onda više ne bismo imali mogućnost inkrementalne kompilacije

Ponekad i kompilacija zaglavlja oduzima znatno vreme



Using Precompiled Headers with `clang`

The Clang compiler frontend, `clang -cc1`, supports two command line options for generating and using PCH files.

To generate PCH files using `clang -cc1`, use the option *-emit-pch*:

```
$ clang -cc1 test.h -emit-pch -o test.h.pch
```

This option is transparently used by `clang` when generating PCH files. The resulting PCH file contains the serialized form of the compiler's internal representation after it has completed parsing and semantic analysis. The PCH file can then be used as a prefix header with the *-include-pch* option:

```
$ clang -cc1 -include-pch test.h.pch test.c -o test.s
```

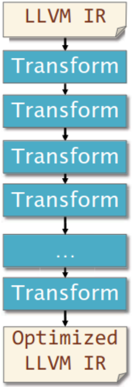
Design Philosophy

Precompiled headers are meant to improve overall esign of precompiled headers is entirely driven by performance concerns. The use case for precompiled headers is relatively simple: when there is a common set of headers that is included in nearly every source file in the project, we *precompile* that bundle of headers into a single precompiled header (PCH file). Then, when compiling the source files in the project, we load the PCH file first (as a prefix header), which acts as a stand-in for that bundle of headers.

Tada i njih možemo unapred kompajlirati

Još neki otvoreni problemi

Simple Model of the Compiler

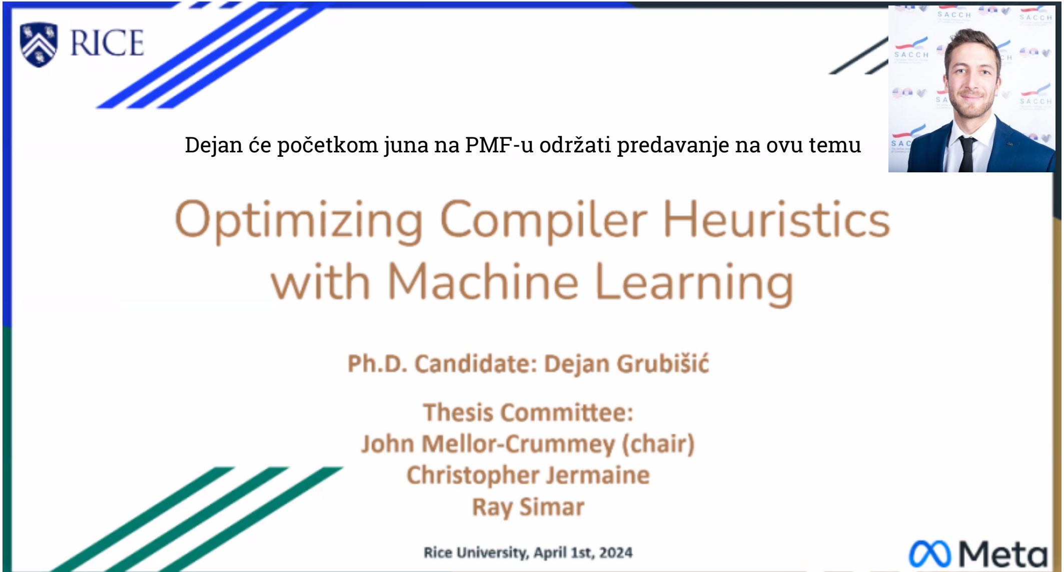


An optimizing compiler performs a sequence of **transformation passes** on the code.

- Each transformation pass analyzes and edits the code to try to optimize the code's performance.
- A transformation pass might run multiple times.
- Passes run in a predetermined order that seems to work well most of the time.

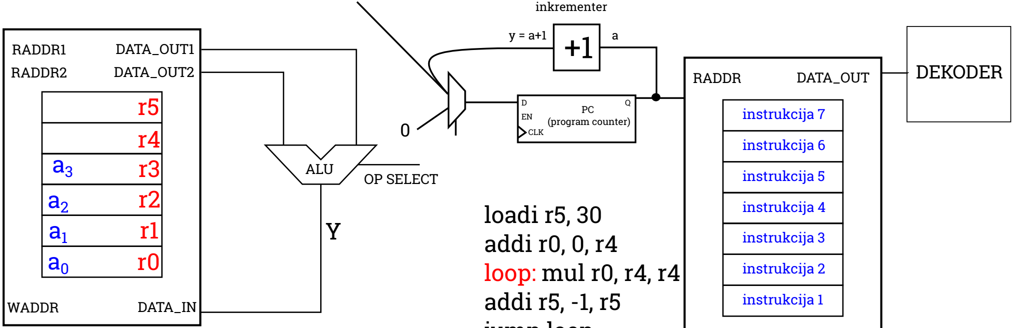
Mašinsko učenje nam danas dozvoljava da otklonimo ovo ograničenje

Dejan će početkom juna na PMF-u održati predavanje na ovu temu



Kako su funkcije implementirane?

Setimo se principa rada fon Nojmanovog računara



podatak upisan u registar

simbolička adresa registra

Setimo se još i opsega vidljivosti promenljivih

Promenljive su vidljive samo u bloku u kom su deklarisane, uključujući i sve blokove unutar tog bloka

I argumenti funkcije su takođe lokalni

Pri pozivu funkcije, pravi se kopija svih prosleđenih parametara koje funkcija može da menja

Kada ne bi bilo tako, rekurzija ne bi bila moguća

```
int fib(int n)
{
	if (n <= 1)
		return n;
	return fib(n - 1) + fib(n - 2);
}
```

Kako izgleda kopiranje?

To zavisi od konkretne arhitekture procesora

No opšti princip je sledeći

Procesor ima određen broj registara koji služe za prenos parametara i povratnih vrednosti[1]

[1]Povratna vrednost je samo jedna, ali procesor može imati više registara u kojima može biti smeštena

No opšti princip je sledeći

Pozivna funkcija kopira parametre u registre za prenos parametara

Po završetku računanja, pozvana funkcija kopira povratnu vrednost u registar za vraćanje vrednosti

To je princip koji nam je poznat i iz svakodnevnog života

Pozivna funkcija : poštar
Pozvana funkcija: građanin

Poštar ostavlja
pismo tamo
gde zna da će ga
građanin naći

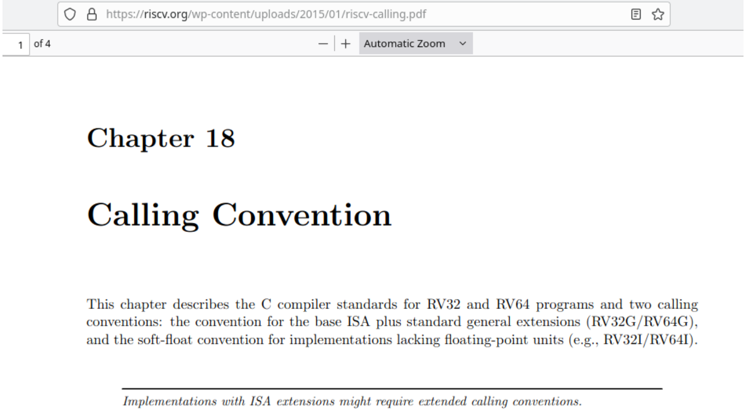
Građanin ostavlja
pismo tamo
gde zna da će ga
poštar naći

Kako poštar i građanin znaju gde da ostave pisma?

Kako poštar i građanin znaju gde da ostave pisma?

Postoji dogovor (konvencija) kog se pridržavaju

Pozivne konvencije


https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf

1 of 4

Automatic Zoom

Chapter 18

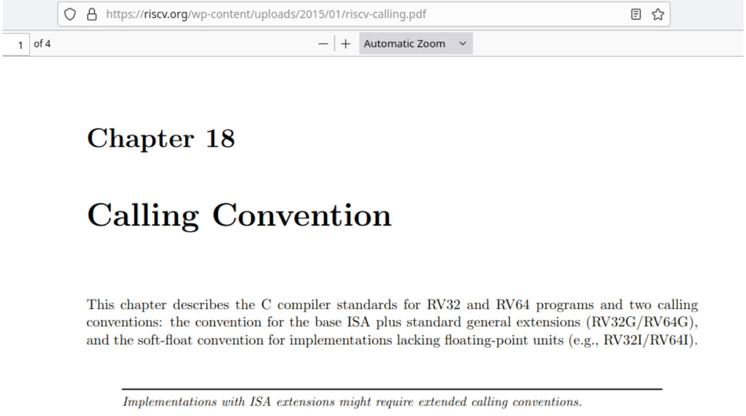
Calling Convention

This chapter describes the C compiler standards for RV32 and RV64 programs and two calling conventions: the convention for the base ISA plus standard general extensions (RV32G/RV64G), and the soft-float convention for implementations lacking floating-point units (e.g., RV32I/RV64I).

Implementations with ISA extensions might require extended calling conventions.

I arhitekte procesora pišu pozivne konvencije i određuju „sandučiće” (registre) za prenos

Pozivne konvencije



https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf

1 of 4 — + Automatic Zoom

Chapter 18

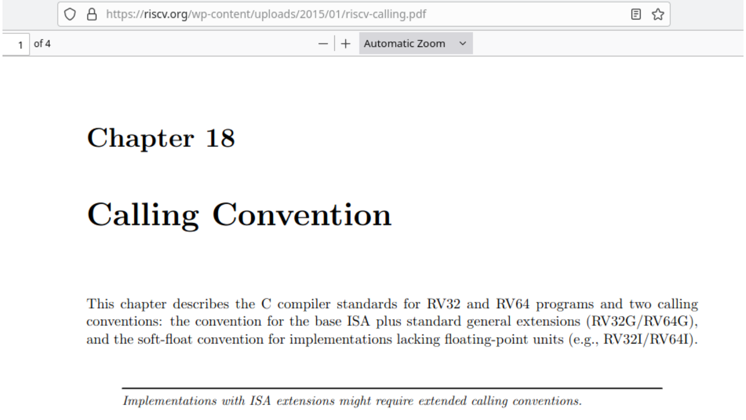
Calling Convention

This chapter describes the C compiler standards for RV32 and RV64 programs and two calling conventions: the convention for the base ISA plus standard general extensions (RV32G/RV64G), and the soft-float convention for implementations lacking floating-point units (e.g., RV32I/RV64I).

Implementations with ISA extensions might require extended calling conventions.

Pisci kompajlera su dužni da osiguraju da su konvencije uvek poštovane

Pozivne konvencije


https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf

1 of 4

Automatic Zoom

Chapter 18

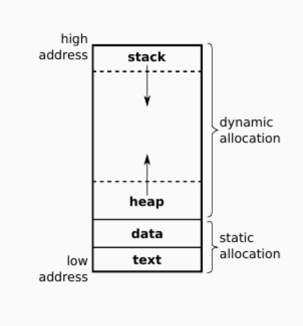
Calling Convention

This chapter describes the C compiler standards for RV32 and RV64 programs and two calling conventions: the convention for the base ISA plus standard general extensions (RV32G/RV64G), and the soft-float convention for implementations lacking floating-point units (e.g., RV32I/RV64I).

Implementations with ISA extensions might require extended calling conventions.

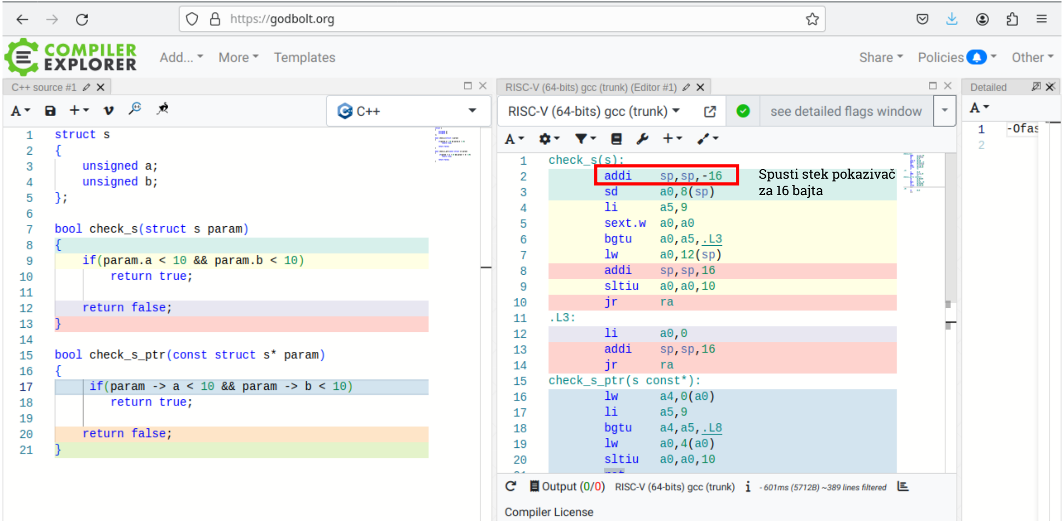
Kompajler je taj koji obezbeđuje primenu konvencije tako što generiše odgovarajuće instrukcije i u pozivnoj i u pozvanoj funkciji

Kako izgleda kopiranje parametara?

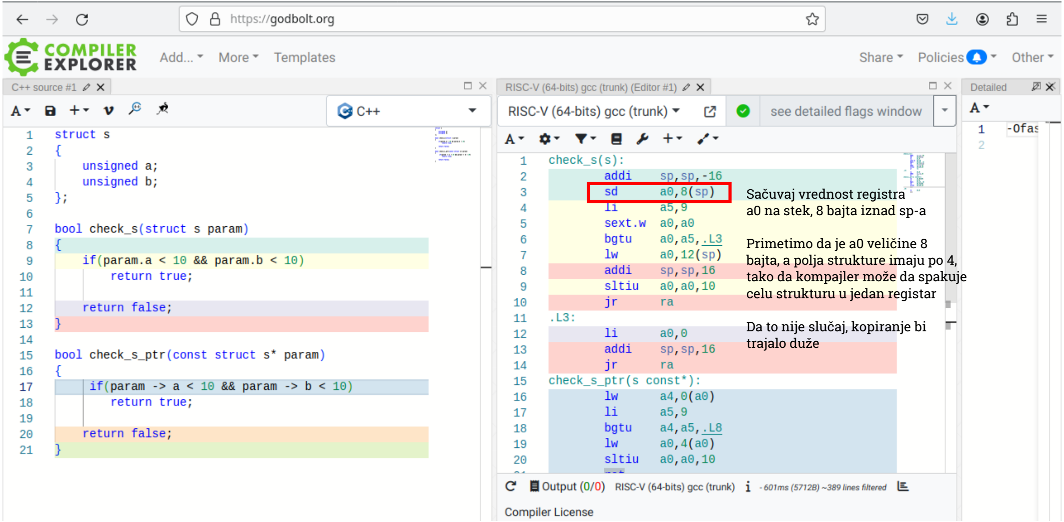


Ako su parametri veći/brojniji od registara za prenos, višak se preliva na **STEK** (eng. stack)

Jedan konkretan primer



Jedan konkretan primer

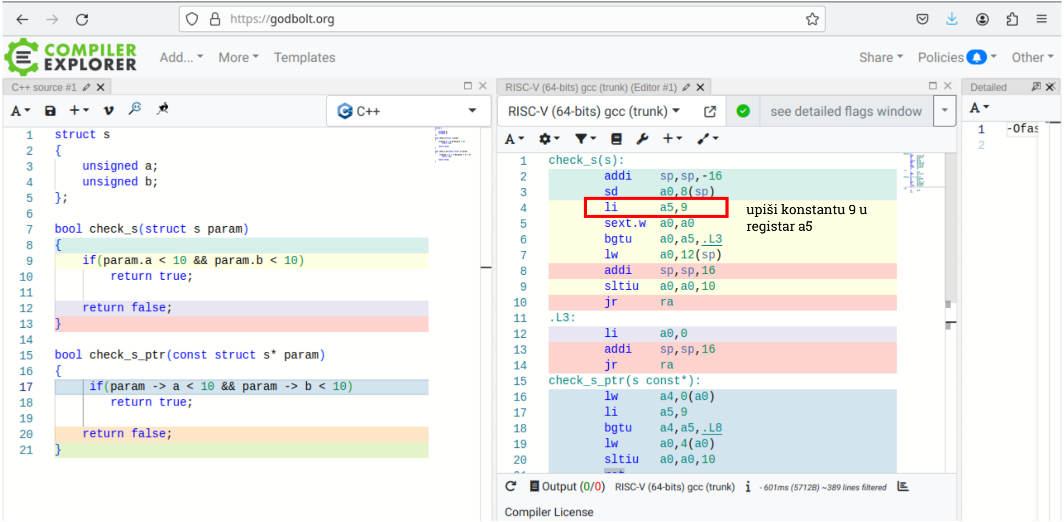


Sačuvaj vrednost registra a0 na stek, 8 bajta iznad sp-a

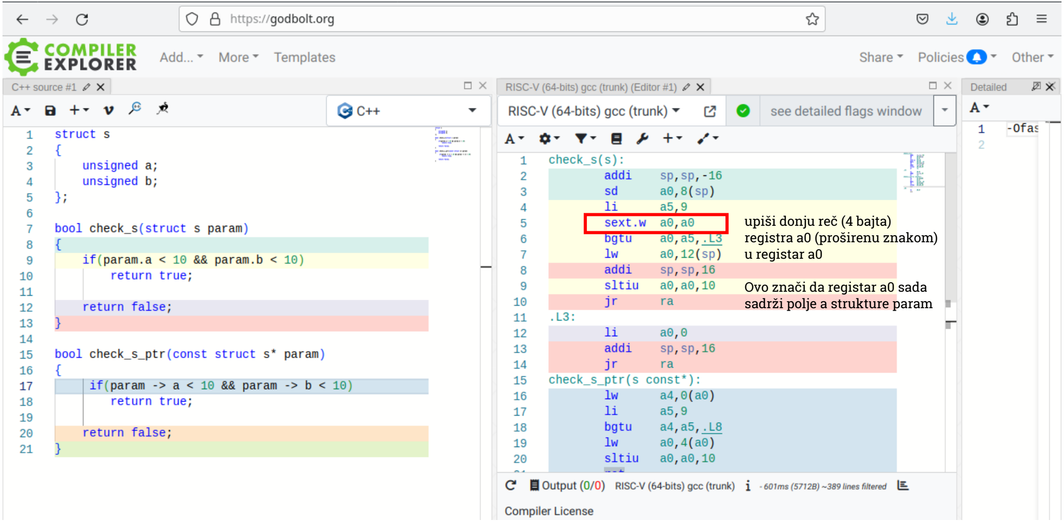
Primetimo da je a0 veličine 8 bajta, a polja strukture imaju po 4, tako da kompajler može da spakuje celu strukturu u jedan registar

Da to nije slučaj, kopiranje bi trajalo duže

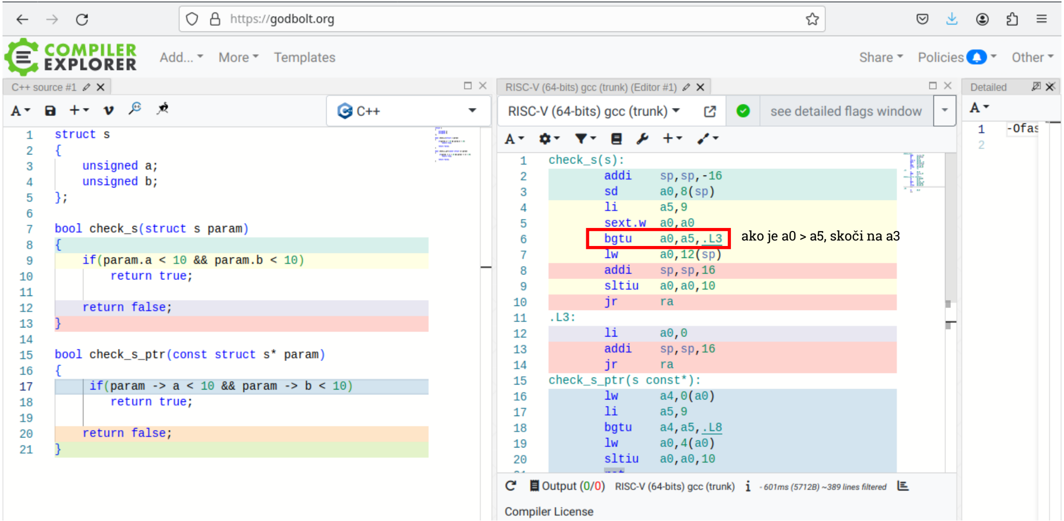
Jedan konkretan primer



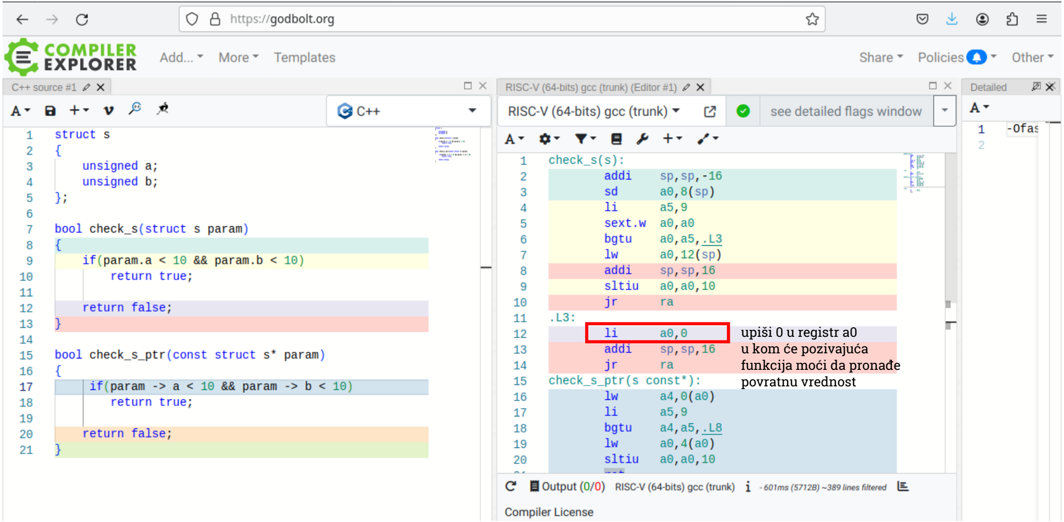
Jedan konkretan primer



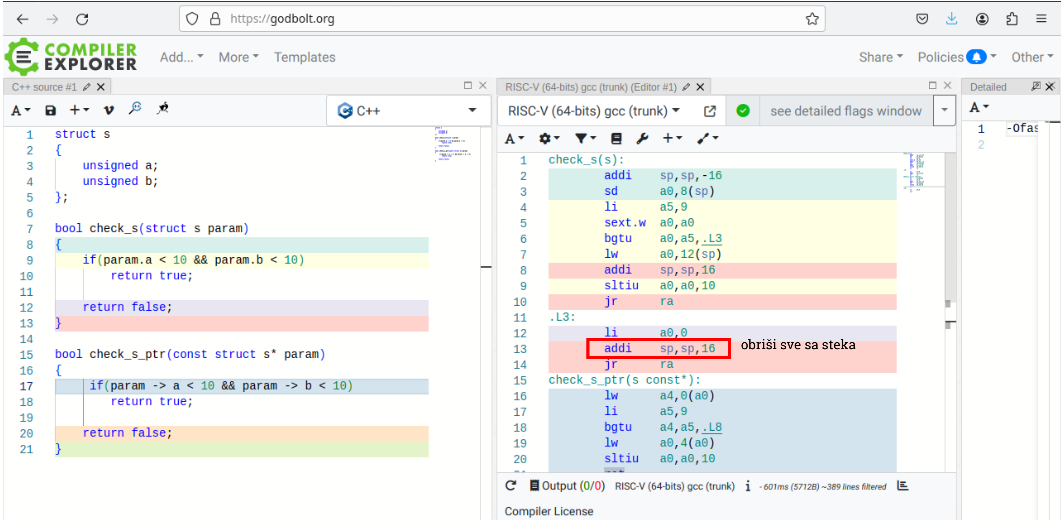
Jedan konkretan primer



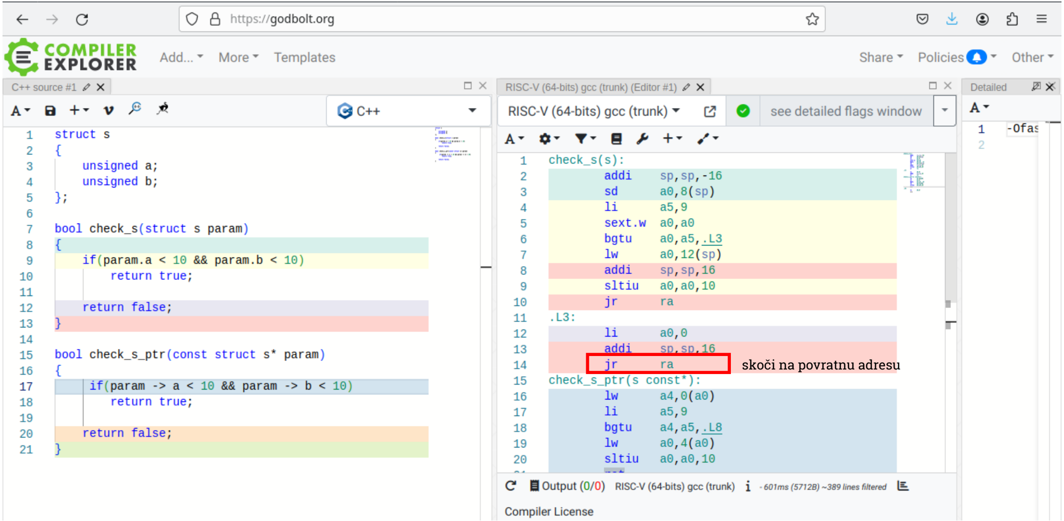
Jedan konkretan primer



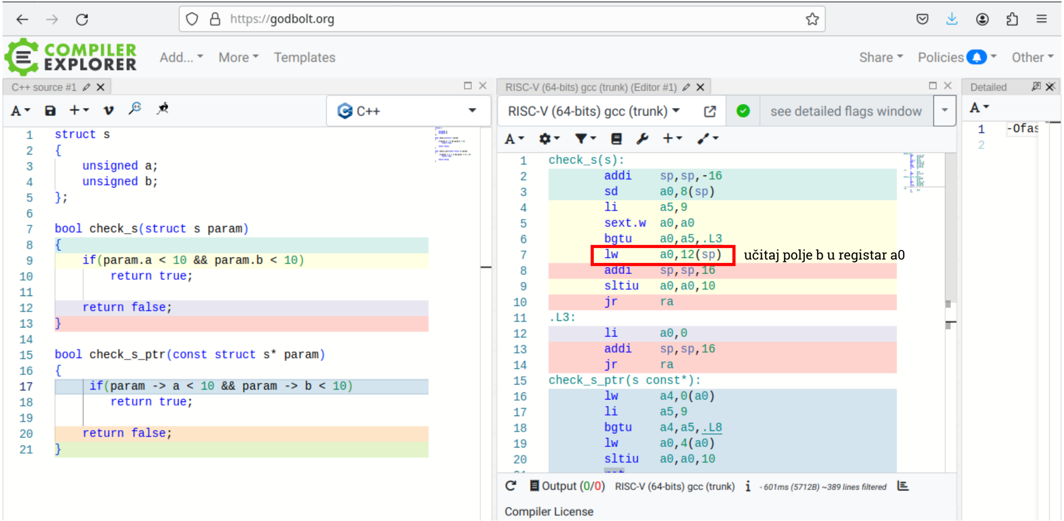
Jedan konkretan primer



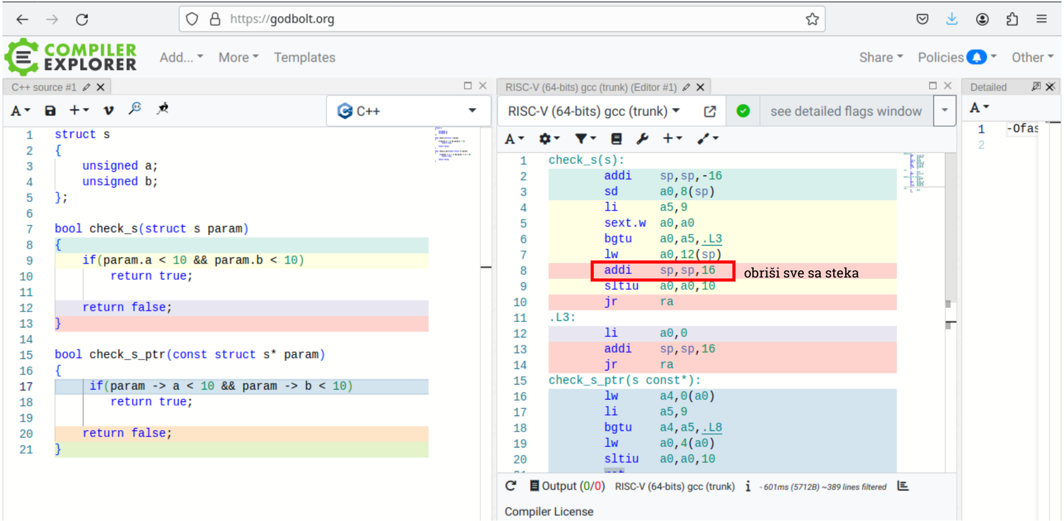
Jedan konkretan primer



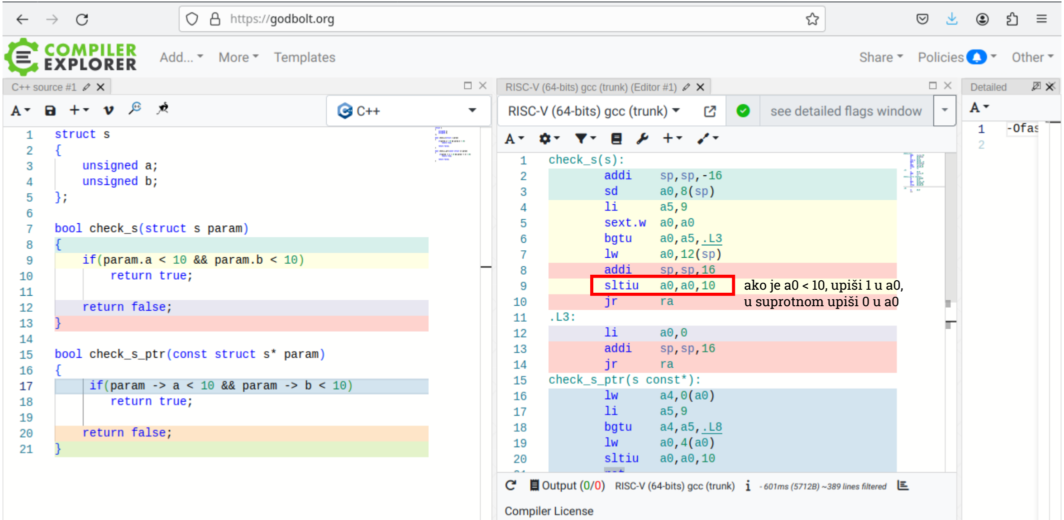
Jedan konkretan primer



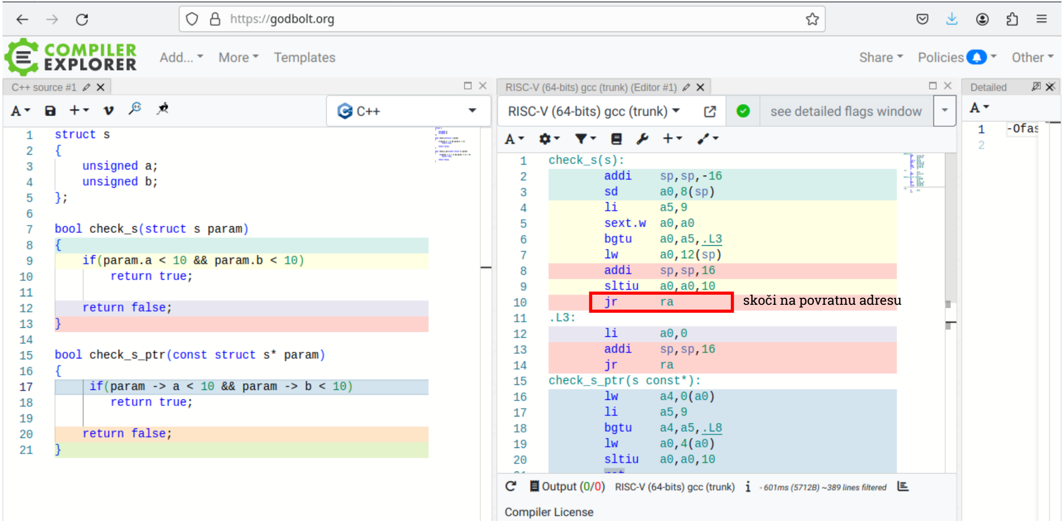
Jedan konkretan primer



Jedan konkretan primer



Jedan konkretan primer



Zašto ovo moramo dobro da zapamtimo?

https://godbolt.org

C++ source #1

```cpp
struct s
{
    unsigned a;
    unsigned b;
};

bool check_s(struct s param)
{
    if(param.a < 10 && param.b < 10)
        return true;

    return false;
}

bool check_s_ptr(const struct s* param)
{
    if(param -> a < 10 && param -> b < 10)
        return true;

    return false;
}
```

RISC-V (64-bits) gcc (trunk) (Editor #1)

```asm
check_s(s):
        addi    sp,sp,-16
        sd      a0,8(sp)
        li      a5,9
        sext.w  a0,a0
        bgtu    a0,a5,.L3
        lw      a0,12(sp)
        addi    sp,sp,16
        sltiu   a0,a0,10
        jr      ra
.L3:
        li      a0,0
        addi    sp,sp,16
        jr      ra
check_s_ptr(s const*):
        lw      a4,0(a0)
        li      a5,9
        bgtu    a4,a5,.L8
        lw      a0,4(a0)
        sltiu   a0,a0,10
```

obriši sve sa steka

Output (0/0) RISC-V (64-bits) gcc (trunk) - 601ms (5712B) ~389 lines filtered

Compiler License

Lokalne promenljive nestaju nakon završetka funkcije

```
#include <iostream>
using namespace std;

int* return_sum_address(int a, int b)
{
        int sum = a + b;

        return &sum;
}


int main()
{
        cout << *return_sum_address(6, 5) << endl;

        return 0;
}
```

Lokalne promenljive nestaju nakon završetka funkcije

```
wrong_ret.cpp: In function ‘int* return_sum_address(int, int)’:
wrong_ret.cpp:8:16: warning: address of local variable ‘sum’ returned [-Wreturn-local-addr]
    8 |         return &sum;
      |                ^~~~
wrong_ret.cpp:6:13: note: declared here
    6 |         int sum = a + b;
      |             ^~~
/bin/bash: line 1: 30593 Segmentation fault      (core dumped) ./run
```

Malo više o kopiranjima

Potrogram je u punom smislu te reči potprogram

Pri pozivu potprograma, predaje mu se upravljanje čitavim procesorom

Bitno je da svi registri koje potprogram koristi tokom izvršenja na kraju budu vraćeni u prvobitno stanje

Malo više o kopiranju

Prethodne vrednosti registara se takođe čuvaju na steku

Pozivna konvencija određuje da li je za čuvanje i ponovno postavljanje vrednosti nekog registra zadužena pozivna ili pozvana funkcija

Čuvanje i ponovno postavljanje registara

https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Jedan konkretan primer

https://inst.eecs.berkeley.edu/~cs61c/resources/RISCV_Calling_Convention.pdf

110%

```
sum_squares:

prologue:
    addi sp sp -16
    sw s0 0(sp)
    sw s1 4(sp)
    sw s2 8(sp)
    sw ra 12(sp)

    li s0 1
    mv s1 a0
    mv s2 0

loop_start:
    bge s0 s1 loop_end
    mv a0 s0
    jal square
    add s2 s2 a0
    addi s0 s0 1
    j loop_start

loop_end:
    mv a0 s2

epilogue:
    lw s0 0(sp)
    lw s1 4(sp)
    lw s2 8(sp)
    lw ra 12(sp)
    addi sp sp 16
    jr ra
```

Notice that we store values in the s registers because we need those values fo

na početku koda svake funkcije
se nalazi prolog u kom se vrši čuvanje
registara za koje je pozvana funkcija zadužena
(sw instrukcija znači store word)

na kraju koda svake funkcije
se nalazi epilog u kom se vrši vraćanje vrednosti
sa steka u registre (lw instrukcija znači load word)

Zbog svega ovoga, poziv funkcije je skup

No, to ne znači da treba izbegavati podelu programa na funkcije

Prednosti takve podele su, kao što smo već videli, izuzetne

Ključna reč INLINE

Dodavanje ključne reči INLINE na početku deklaracije i definicije funkcije (ispred povratnog tipa), učiniće da kompajler pokuša da izbegne poziv funkcije prostim ubacivanjem njenog tela u telo pozivne funkcije, umesto naredbe poziva

Ključna reč INLINE

https://gcc.gnu.org/onlinedocs/gcc/Inline.html

Next: Const and Volatile Functions, Previous: Determining the Alignment of Functions, Types or Variables, Up: Extensions to the C Language Family [Contents][Index]

6.45 An Inline Function is As Fast As a Macro

By declaring a function inline, you can direct GCC to make calls to that function faster. One way GCC can achieve this is to integrate that function's code into the code for its callers. This makes execution faster by eliminating the function-call overhead; in addition, if any of the actual argument values are constant, their known values may permit simplifications at compile time so that not all of the inline function's code needs to be included. The effect on code size is less predictable; object code may be larger or smaller with function inlining, depending on the particular case. You can also direct GCC to try to integrate all "simple enough" functions into their callers with the option `-finline-functions`.

Ključna reč INLINE

Međutim, zamena poziva telom funkcije nije uvek isplativa, zbog složenosti današnjih memorijskih sistema

U principu, veće funkcije je bolje zadržati kao obične

Svakako je najbolje proveriti kako se vreme izvršenja programa menja nakon deklaracije funckije kao inline, pa izabrati bolju opciju

I nikada ne treba zaboraviti

Optimizacija performansi koda uvek ima neku cenu: povećava naše vreme razvoja, otežava održavanje...

Zato kod optimizujemo preko neke razumne mere samo kada se uverimo da je to neophodno

Kako možemo da vratimo pozivnoj funkciji više od jedne vrednosti?

Prosleđivanje povratne vrednosti preko adrese

```
#include <iostream>
using namespace std;

int add_sub(int a, int b, int* diff)
{
        *diff = a - b;

        return a + b;
}

int main()
{
        int a = 7;
        int b = 5;

        int amb = 0;
        int apb = add_sub(a, b, &amb);

        cout << a << " + " << b << " = " << apb << endl;
        cout << a << " - " << b << " = " << amb << endl;

        return 0;
}
~
```

Prosleđivanje povratne vrednosti preko adrese

```
7 + 5 = 12
7 - 5 = 2
```

Šta možemo da uradimo ako je neophodno da menjamo vrednost parametera i van funkcije?

I tu koristimo prosleđivanje po adresi

```
int no_ptr_inc(int a, int b)
{
        a += b;

        return a;
}

int with_ptr_inc(int* a, int b)
{
        *a += b;

        return *a;
}

int main()
{
        int a = 7;
        int b = 5;

        cout << a << " + " << b << " = " << no_ptr_inc(a, b) << endl;
        cout << "a = " << a << endl;

        cout << a << " + " << b << " = " << with_ptr_inc(&a, b) << endl;
        cout << "a = " << a << endl;

        return 0;
}
```

I tu koristimo prosleđivanje po adresi

```
7 + 5 = 12
a = 7
7 + 5 = 12
a = 12
```

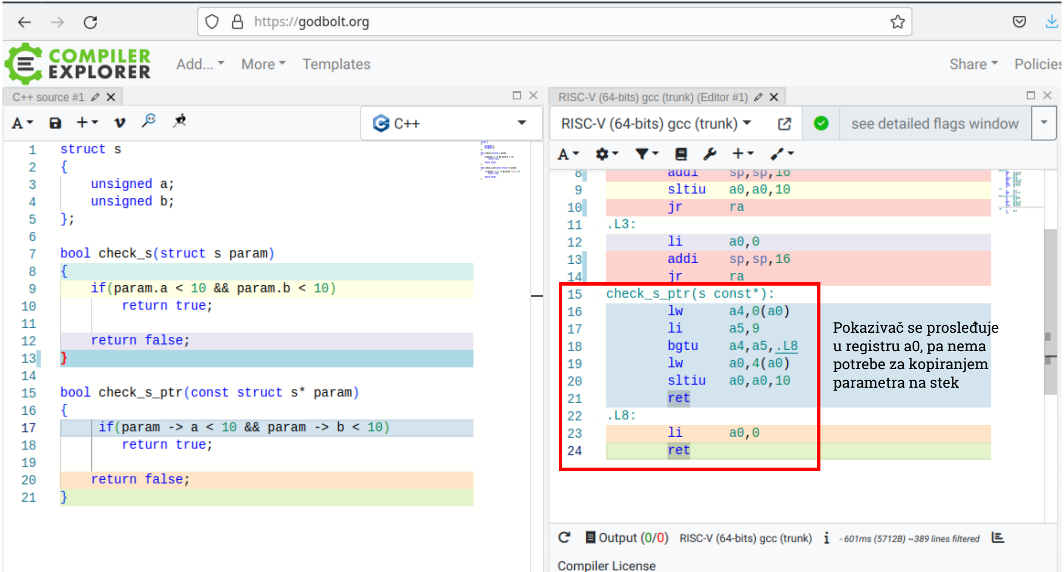
Ne zaboravimo da je i ime niza zapravo adresa njegovog prvog elementa

```
int main()
{
        float xs[NUM_PTS] = {7.7857203028720035, 7.642458246115663, 6.215313359871936, 7.059634724385643, 6.670815698546658,
6.876282852217749, 7.036072823998538, 7.290710191791188, 7.639716393843683, 7.752535310413255, 6.1089490156447095, 7.60434112
48489505, 7.418263501562334, 6.2496483487612196, 6.8065125974698395, 7.434551692906639, 6.51122811120167, 7.198011826278233,
7.874699069289809, 6.507268194942792, 12, 9, 12, 11, 10};

        float ys[NUM_PTS] = {6.663959610602355, 6.0833932514505, 7.1901041284124805, 6.837614857113309, 7.24503886439218, 7.4
7176421269738, 7.157717200452314, 7.980448542746833, 6.826401869632361, 7.647518865141043, 7.437274473128388, 7.4728132876228
78, 7.081873708605056, 7.915294590939337, 6.43390232064712, 7.988414876843875, 7.3426188638785606, 7.434664291111027, 6.70361
9537079404, 6.804945018073546, 14, 9, 7, 11, 13};

        for(unsigned i = 0; i < NUM_PTS; ++i)
                cout << ys[i] << ' ';
        cout << endl;

        cout << "centar = (" << median(xs, NUM_PTS) << ", " << median(ys, NUM_PTS) << ")\n";

        for(unsigned i = 0; i < NUM_PTS; ++i)
                cout << ys[i] << ' ';
        cout << endl;

        return 0;
}
```

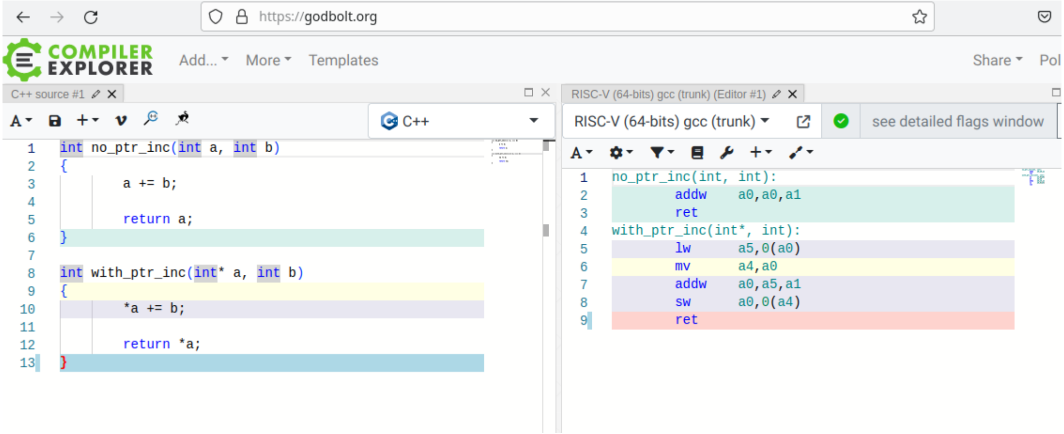
Ne zaboravimo da je i ime niza zapravo adresa njegovog prvog elementa

```
6.66396 6.08339 7.1901 6.83762 7.24504 7.47176 7.15772 7.98045 6.8264 7.64752 7.43727 7.47281 7.08187 7.91529 6.4339 7.98841
7.34262 7.43466 6.70362 6.80494 14 9 7 11 13
centar = (7.41826, 7.34262)
6.08339 6.4339 6.66396 6.70362 6.80494 6.8264 6.83762 7 7.08187 7.15772 7.1901 7.24504 7.34262 7.43466 7.43727 7.47176 7.4728
1 7.64752 7.91529 7.98045 7.98841 9 11 13 14
```

Prenos po adresi štedi vreme kada funkcija ima veće parametre, jer nema kopiranja



Ipak, i dereferenciranje pokazivača košta



No, o svemu tome više u sredu

Hvala na pažnji