

Programiranje 2, letnji semestar 2023/4

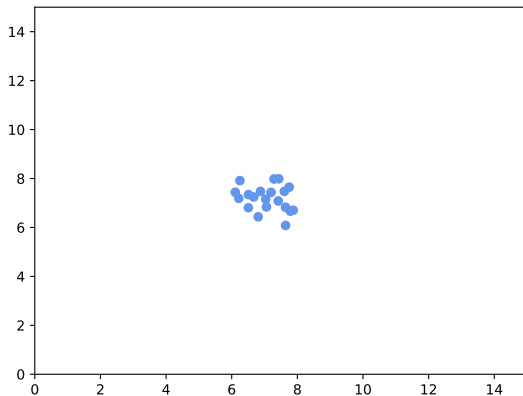
Potprogrami

Stefan Nikolić

Prirodno-matematički fakultet, Novi Sad

08.04.2024.

Pretpostavimo da imamo sledeći problem



Kako da nađemo približan centar grupe tačaka?

Prvo pitanje

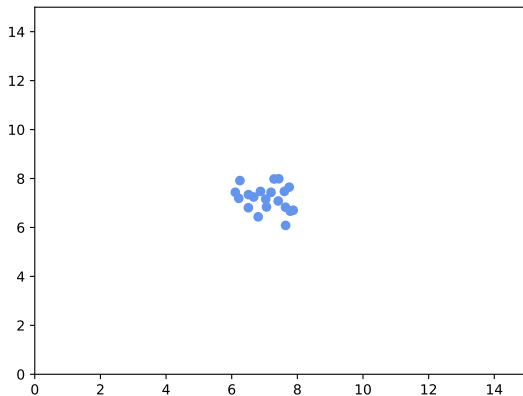
Kako da predstavimo tačke koristeći samo ono znanje koje smo do sada stekli na predavanjima?

Odvojeni nizovi koordinata

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

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

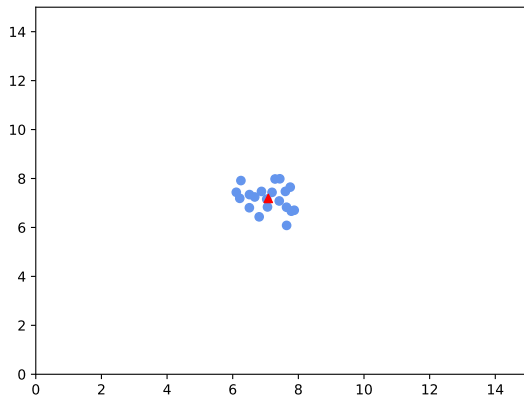
Da se vratimo na početni problem



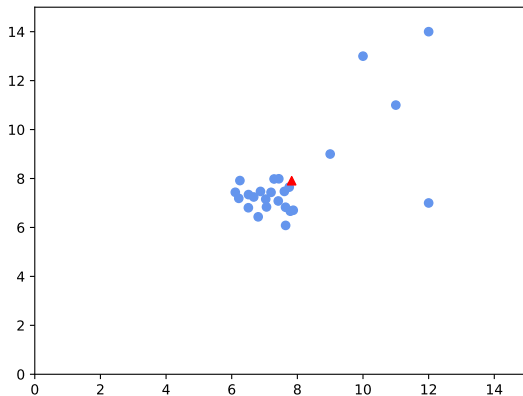
Kako da nađemo približan centar grupe tačaka?

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

To je razumna aproksimacija

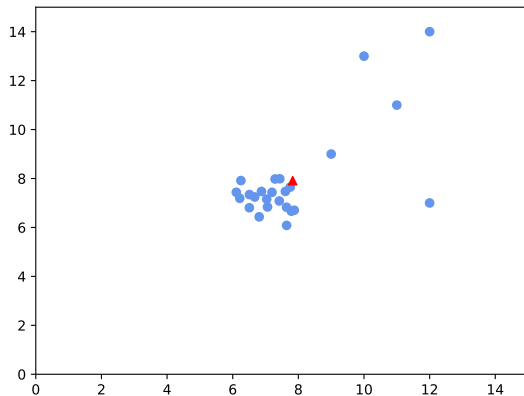


To je razumna aproksimacija



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

Da li možemo da smislimo nešto robusnije?

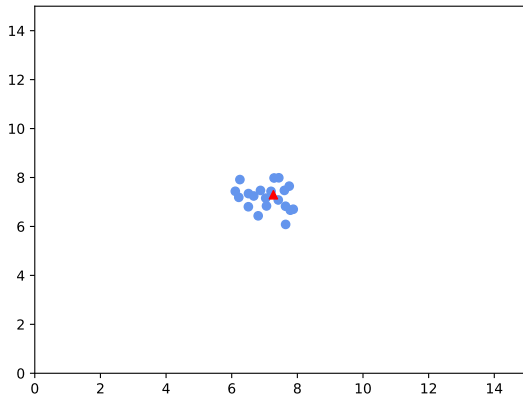


Tako da smanjimo uticaj udaljenih tačaka

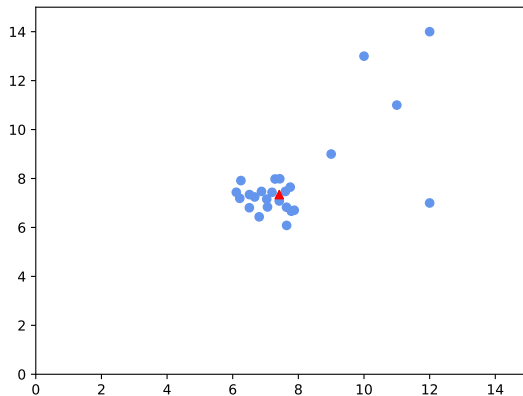
Druga ideja

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}

$$\text{medijana} = \frac{2.7+3.1}{2} = 2.9$$

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

Najpre inicijalizujemo nizove koordinata

```
1 #include <iostream>
2 #define NUM_PTS 25
3
4 using namespace std;
5
6 int main()
7 {
8     float xs[NUM_PTS] = {7.7857203028720035, 7.642458246115663, 6.215313359871936, 7.059634724385643, 6.670815698546658, 6.876282852217749, 7.036072
823998538, 7.290710191791188, 7.639716393843683, 7.752535310413255, 6.1089490156447095, 7.6043411248489505, 7.418263501562334, 6.2496483487612196, 6.806
5125974698395, 7.434551692906639, 6.51122811120167, 7.198011826278233, 7.874699069289809, 6.507268194942792, 12, 9, 12, 11, 10};
9
10    float ys[NUM_PTS] = {6.663959610602355, 6.0833932514505, 7.1901041284124805, 6.837614857113309, 7.24503886439218, 7.47176421269738, 7.1577172004
52314, 7.980448542746833, 6.826401869632361, 7.647518865141043, 7.437274473128388, 7.472813287622878, 7.081873708605056, 7.915294590939337, 6.4339023206
4712, 7.988414876843875, 7.3426188638785606, 7.434664291111027, 6.703619537079404, 6.804945018073546, 14, 9, 7, 11, 13};
11 }
```

Zatim koristimo insertion sort sa prošlog časa da sortiramo x koordinate

```
13 //Koristimo insertion sort sa proslog casa da sortiramo x koordinate
14 for(unsigned i = 1; i < NUM_PTS; ++i)
15 {
16     float val = xs[i];
17     unsigned j = i;
18     while((j > 0) && (val < xs[j - 1]))
19     {
20         xs[j] = xs[j - 1];
21         --j;
22     }
23     xs[j] = val;
24 }
```

Pa izračunamo medijanu

```
26     float median_x = 0;
27     if(NUM_PTS % 2)
28         median_x = xs[NUM_PTS / 2];
29     else
30         median_x = 0.5 * (xs[NUM_PTS / 2] + xs[NUM_PTS / 2 + 1]);
31
```

Ponovimo sve to za y koordinate

```
32 //Koristimo insertion sort sa proslog casa da sortiramo y koordinate
33 for(unsigned i = 1; i < NUM_PTS; ++i)
34 {
35     float val = ys[i];
36     unsigned j = i;
37     while((j > 0) && (val < ys[j - 1]))
38     {
39         ys[j] = ys[j - 1];
40         --j;
41     }
42     ys[j] = val;
43 }
44
45 float median_y = 0;
46 if(NUM_PTS % 2)
47     median_y = ys[NUM_PTS / 2];
48 else
49     median_y = 0.5 * (ys[NUM_PTS / 2] + ys[NUM_PTS / 2 + 1]);
50
51
52 cout << "centar = (" << median_x << ", " << median_y << ")\n";
53
54 return 0;
55 }
```

I u samo 55 linija koda dobijamo rezultat

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

Da li možemo da pojednostavimo program?

Izdvojimo računanje medijane u potprogram

```
1 #include <iostream>
2 #define NUM PTS 25
3
4 using namespace std;
5
6 float median(float a[], unsigned len)
7 {
8     //Koristimo insertion sort sa proslog casa da sortiramo niz
9     for(unsigned i = 1; i < len; ++i)
10     {
11         float val = a[i];
12         unsigned j = i;
13         while((j > 0) && (val < a[j - 1]))
14         {
15             a[j] = a[j - 1];
16             --j;
17         }
18         a[j] = val;
19     }
20
21     if(len % 2)
22         return a[len / 2];
23
24     return 0.5 * (a[len / 2] + a[len / 2 + 1]);
25 }
26
```

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

```
27 int main()
28 {
29     float xs[NUM_PTS] = {7.7857203028720035, 7.642458246115663, 6.215313359871936, 7.059634724385643, 6.670815698546658, 6.876282852217749, 7.036072
823998538, 7.290710191791188, 7.639716393843683, 7.752535310413255, 6.1089490156447095, 7.6043411248489505, 7.418263501562334, 6.2496483487612196, 6.806
5125974698395, 7.434551692906639, 6.51122811120167, 7.198011826278233, 7.874699069289809, 6.507268194942792, 12, 9, 12, 11, 10};
30
31     float ys[NUM_PTS] = {6.663959610602355, 6.0833932514505, 7.1901041284124805, 6.837614857113309, 7.24503886439218, 7.47176421269738, 7.1577172004
52314, 7.980448542746833, 6.826401869632361, 7.647518865141043, 7.437274473128388, 7.472813287622878, 7.081873708605056, 7.915294590939337, 6.4339023206
4712, 7.988414876843875, 7.3426188638785606, 7.434664291111027, 6.703619537079404, 6.804945018073546, 14, 9, 7, 11, 13};
32
33
34     cout << "centar = (" << median(xs, NUM_PTS) << ", " << median(ys, NUM_PTS) << ")\n";
35
36     return 0;
37 }
```

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 potprogram 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

```
17         }
18         a[j] = val;
19     }
20
21     if(len % 2)
22         return a[len / 2];
23
24     return 0.5 * (a[len / 2] + a[len / 2 + 1]);
25 }
26
27 int main()
28 {
29     float xs[NUM_PTS] = {7.7857203028720035, 7.642458246115663, 6.215313359871936
30 , 7.059634724385643, 6.670815698546658, 6.876282852217749, 7.036072823998538, 7.29071
31 0191791188, 7.639716393843683, 7.752535310413255, 6.1089490156447095, 7.6043411248489
32 505, 7.418263501562334, 6.2496483487612196, 6.8065125974698395, 7.434551692906639, 6.
33 51122811120167, 7.198011826278233, 7.874699069289809, 6.507268194942792, 12, 9, 12, 1
34 1, 10};
35
36     float ys[NUM_PTS] = {6.663959610602355, 6.0833932514505, 7.1901041284124805,
37 6.837614857113309, 7.24503886439218, 7.47176421269738, 7.157717200452314, 7.980448542
38 746833, 6.826401869632361, 7.647518865141043, 7.437274473128388, 7.472813287622878, 7.
39 .081873708605056, 7.915294590939337, 6.43390232064712, 7.988414876843875, 7.342618863
40 8785606, 7.434664291111027, 6.703619537079404, 6.804945018073546, 14, 9, 7, 11, 13};
41
42     cout << "centar = (" << median(xs, NUM_PTS) << ", " << median(ys, NUM_PTS) <<
43     ")\n";
44
45     return 0;
46 }
47
48 //32% kraci kod
```

center_with_function.cpp [4][R0]

39,1

Bot

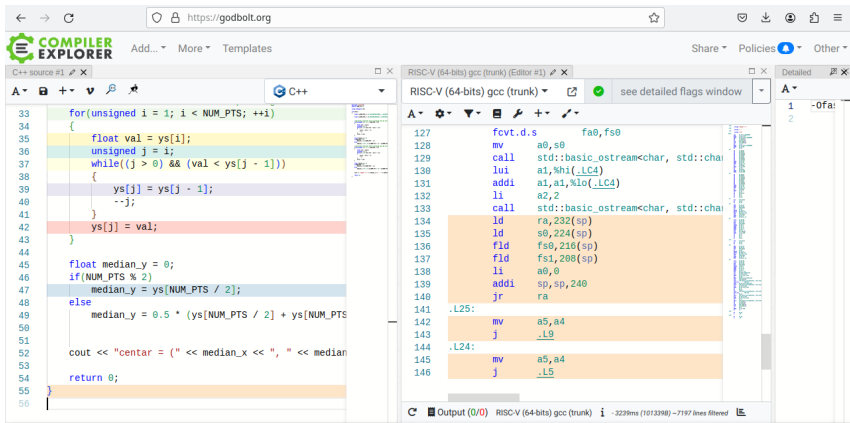
```
23         xs[j] = val;
24     }
25
26     float median_x = 0;
27     if(NUM_PTS % 2)
28         median_x = xs[NUM_PTS / 2];
29     else
30         median_x = 0.5 * (xs[NUM_PTS / 2] + xs[NUM_PTS / 2 + 1]);
31
32     //Koristimo insertion sort sa proslog casa da sortiramo y koordinate
33     for(unsigned i = 1; i < NUM_PTS; ++i)
34     {
35         float val = ys[i];
36         unsigned j = i;
37         while((j > 0) && (val < ys[j - 1]))
38         {
39             ys[j] = ys[j - 1];
40             --j;
41         }
42         ys[j] = val;
43     }
44
45     float median_y = 0;
46     if(NUM_PTS % 2)
47         median_y = ys[NUM_PTS / 2];
48     else
49         median_y = 0.5 * (ys[NUM_PTS / 2] + ys[NUM_PTS / 2 + 1]);
50
51     cout << "centar = (" << median_x << ", " << median_y << ")\n";
52
53     return 0;
54 }
55 }
```

center.cpp [R0]

55,1

Bot

Ali i kraći mašinski kod



The screenshot displays the Compiler Explorer interface at <https://godbolt.org>. The left pane shows C++ source code for a program that calculates the median of an array. The right pane shows the corresponding RISC-V assembly code generated by gcc (trunk). The assembly code includes instructions for floating-point conversion, memory access, and arithmetic operations to calculate the median.

```
33 for(unsigned i = 1; i < NUM_PTS; ++i)
34 {
35     float val = ys[i];
36     unsigned j = i;
37     while((j > 0) && (val < ys[j - 1]))
38     {
39         ys[j] = ys[j - 1];
40         --j;
41     }
42     ys[j] = val;
43 }
44
45 float median_y = 0;
46 if(NUM_PTS % 2)
47     median_y = ys[NUM_PTS / 2];
48 else
49     median_y = 0.5 * (ys[NUM_PTS / 2] + ys[NUM_PTS / 2 - 1]);
50
51 cout << "centar = (" << median_x << ", " << median_y << ")" << endl;
52
53 return 0;
54
55
56
```

```
127 fcvtd.s      fa0,fs0
128 mv          a0,s0
129 call       std::basic_ostream<char, std::char_traits<char>>::operator<< (std::basic_ostream<char, std::char_traits<char>>*, const char*)@plt
130 lui        a1,%hi(.L24)
131 addi       a1,a1,%lo(.L24)
132 li         a2,2
133 call       std::basic_ostream<char, std::char_traits<char>>::operator<< (std::basic_ostream<char, std::char_traits<char>>*, const char*)@plt
134 ld         ra,232(sp)
135 ld         s0,224(sp)
136 fld        fs0,216(sp)
137 fld        fs1,208(sp)
138 li         a0,0
139 addi       sp,sp,240
140 jr         ra
.L25:
142 mv         a5,a4
143 j          .L9
.L24:
145 mv         a5,a4
146 j          .L5
```

Asembli bez upotrebe funkcije za izračunavanje medijane

Ali i kraći mašinski kod

The screenshot displays the Compiler Explorer web application. The left pane shows C++ source code for a function named `median` that implements an insertion sort. The right pane shows the corresponding RISC-V assembly code generated by GCC (trunk). The assembly code includes instructions for stack frame setup, calling the `median` function, and returning the result. The output window at the bottom shows 0/0 lines of output.

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

using namespace std;

float median(float a[], unsigned len)
{
    //Koristimo insertion sort sa proslog casa da sort
    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]);
}
```

```
109      addi    a0,sp,104
110      li      a1,25
111      call    median(float*, unsigned int)
112      fcvtd.s  fa0,fa0
113      mv      a0,s0
114      call    std::basic_ostream<char, std::cha
115      lui     a1,%hi(.LC5)
116      addi    a1,a1,%lo(.LC5)
117      li      a2,2
118      call    std::basic_ostream<char, std::cha
119      ld      ra,216(sp)
120      ld      s0,208(sp)
121      li      a0,0
122      addi    sp,sp,224
123      jr      ra
124      .LC2:
125      .word   1056964088
```

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

```
7 void sort(float a[], unsigned len)
8 {
9     //Koristimo insertion sort sa proslog casa da sortiramo niz
10    for(unsigned i = 1; i < len; ++i)
11    {
12        float val = a[i];
13        unsigned j = i;
14        while((j > 0) && (val < a[j - 1]))
15        {
16            a[j] = a[j - 1];
17            --j;
18        }
19        a[j] = val;
20    }
21 }
22
23 float median(float a[], unsigned len)
24 {
25     sort(a, len);
26
27     if(len % 2)
28         return a[len / 2];
29
30     return 0.5 * (a[len / 2] + a[len / 2 + 1]);
31 }
32
33 int main()
34 {
```

```
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



The screenshot shows a web browser window with the URL `https://cplusplus.com/reference/cmath/atan/`. The page title is "Reference : <cmath> : atan". On the left, there is a navigation menu with links to "C++", "Tutorials", "Reference", "Articles", and "Forum". Under the "Reference" section, a list of C library headers is shown, with "<cmath> (math.h)" selected. The main content area displays the function "atan" with its signature: `double atan (double x); float atan (float x); long double atan (long double x);`. Below the signature, the text reads: "Compute arc tangent", "Returns the principal value of the arc tangent of x , expressed in radians.", and "In trigonometrics, *arc tangent* is the inverse operation of *tangent*." A notice at the bottom states: "Notice that because of the sign ambiguity, the function cannot determine with certainty in which quadrant the angle falls only by its tangent value. See [atan2](#) for an alternative that takes a fractional argument instead."

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

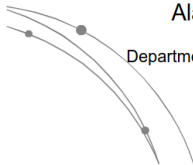
Abstract:

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*

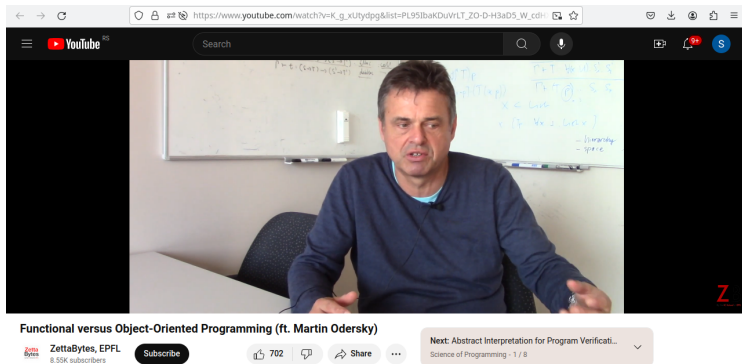
Šta smo radili do sada?

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

Šta ćemo raditi od sledeće nedelje?

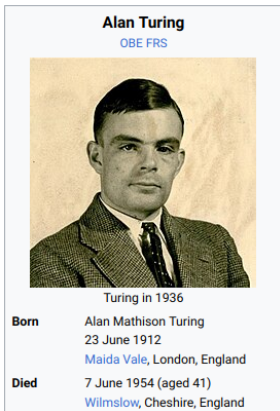
Počevši od sledeće nedelje, bavićemo se objektno-orijentisanim programiranjem, u kom potprograme pridružujemo podacima nad kojima operišu, stvarajući posebnu vrstu modula koju nazivamo *objektima*

Postoji i funkcionalno programiranje

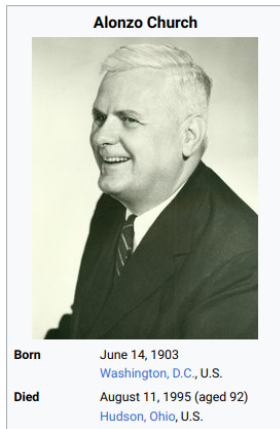


https://www.youtube.com/watch?v=K_g_xUtydpg&list=PL95IbaKDuVrLT_ZO-D-H3aD5_W_cdH

I proceduralno i funkcionalno programiranje sežu u 1930-te



Proceduralno programiranje



Funkcionalno programiranje

Nagradno pitanje

Šta su Tjuring i Čerč bili po obrazovanju?

Alan Mathison Turing

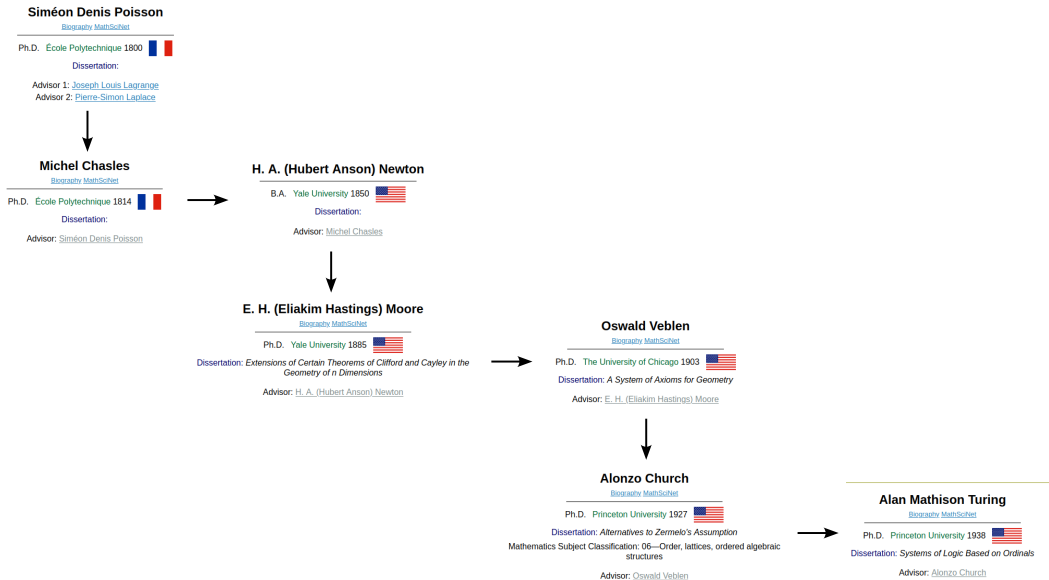
[Biography](#) [MathSciNet](#)

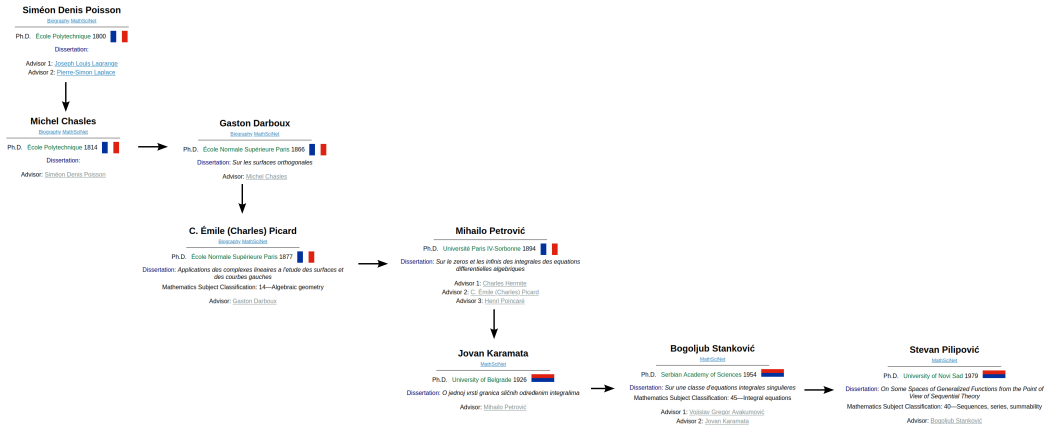
Ph.D. [Princeton University](#) 1938



Dissertation: *Systems of Logic Based on Ordinals*

Advisor: [Alonzo Church](#)





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

PEDIA
ncyclopedia

Search Wikipedia

Create account Log in

Pentium FDIV bug

Article Talk

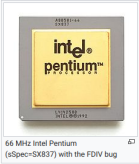
Read Edit View history Tools

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=5X837) 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?

ulazi

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

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

a = {13, 7}



Funkcija koju testiramo

```
23 float median(float a[], unsigned len)
24 {
25     sort(a, len);
26
27     if(len % 2)
28         return a[len / 2];
29
30     return 0.5 * (a[len / 2] + a[len / 2 + 1]);
31 }
```



proizvedeni
rezultat

?

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

The screenshot shows the Wikipedia page for 'Unit testing'. The browser address bar displays 'https://en.wikipedia.org/wiki/Unit_testing'. The Wikipedia logo and search bar are at the top. The article title 'Unit testing' is prominently displayed with a language selector showing '29 languages'. Below the title are tabs for 'Article' and 'Talk'. The main text explains that unit testing is a software testing method for individual units of source code. A 'History' section follows, detailing the origins of unit testing in the early days of software engineering, mentioning the SAGE project and the Mercury project. A sidebar on the right, titled 'Part of a series on Software development', lists various related topics like Core activities, Paradigms and models, and Tools.

WIKIPEDIA
The Free Encyclopedia

Search Wikipedia

Unit testing 29 languages

Article Talk

From Wikipedia, the free encyclopedia

In [computer programming](#), **unit testing** is a [software testing](#) method by which individual units of [source code](#)—sets of one or more computer program [modules](#) together with associated control data, usage [procedures](#), and operating procedures—are tested to determine whether they are fit for use.^[1] It is a standard step in [development](#) and [implementation](#) approaches such as [Agile](#).

History [[edit](#)]

Unit testing, as principle for testing separately smaller parts of large software systems dates back to the early days of software engineering. In June 1956, H.D. Benington presented at US Navy's Symposium on Advanced Programming Methods for Digital Computers the [SAGE](#) project and its specification based approach where the coding phase was followed by "parameter testing" to validate component subprograms against their specification, followed then by an "assembly testing" for parts put together.^{[2][3]}

In 1964, a similar approach is described for the software of the [Mercury project](#), where individual units developed by different programmes underwent "unit tests" before being integrated together.^[4] In 1969, testing methodologies appear more structured, with unit tests, component tests and

Part of a series on
Software development

Core activities [[hide](#)]
Data modeling · Processes · Requirements · Design · Construction · Engineering · Testing · Debugging · Deployment · Maintenance

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

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

Testove bi trebalo automatski pokretati pri svakoj izmeni

The screenshot shows the Wikipedia page for 'Regression testing'. The browser address bar displays 'https://en.wikipedia.org/wiki/Regression_testing'. The Wikipedia logo and search bar are at the top. The article title 'Regression testing' is prominently displayed, with a language selector showing '23 languages'. Below the title are tabs for 'Article' and 'Talk'. A summary line states 'From Wikipedia, the free encyclopedia'. The main text begins with a note that the article is about software development and refers to 'Regression analysis' for statistical analysis. The definition of 'Regression testing' is provided, noting it is the re-running of functional and non-functional tests to ensure software still performs as expected after a change. It mentions that if not, it would be called a 'regression'. A paragraph follows, listing changes that may require regression testing: bug fixes, software enhancements, configuration changes, and substitution of electronic components (hardware). It states that as regression test suites grow with each found defect, test automation is frequently involved. An exception is GUI regression testing, which is normally manual. Sometimes a change impact analysis is performed to determine an appropriate subset of tests (non-regression analysis). A 'Background' section is also visible, stating that as software is updated or changed, or reused on a modified target, the emergence of new faults and/or re-emergence of old faults is quite common. On the right side, there is a 'Part of a series on Software development' box with links to various topics like Core activities, Paradigms and models, Methodologies and frameworks, Supporting disciplines, Practices, Tools, Standards and bodies of knowledge, Glossaries, and Outlines.

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

WIKIPEDIA
The Free Encyclopedia

Search Wikipedia

Create account Log in

23 languages

Article Talk

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

```
23 float median(float a[], unsigned len)
24 {
25     sort(a, len);
26
27     if(len % 2)
28         return a[len / 2];
29
30     return 0.5 * (a[len / 2] + a[len / 2 + 1]);
31 }
--
```

proizvedeni
rezultat

?

očekivani rezultati

4

1

10

Kako nalazimo očekivane rezultate?

Jedna mogućnost je da ih ručno izračunamo

Kako nalazimo očekivane rezultate?

ulazi

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

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

a = {13, 7}

Funkcija koju testiramo

```
23 float median(float a[], unsigned len)
24 {
25     sort(a, len);
26
27     if(len % 2)
28         return a[len / 2];
29
30     return 0.5 * (a[len / 2] + a[len / 2 + 1]);
31 }
--
```

proizvedeni
rezultat

?

Referentni model

očekivani
rezultat

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

Šta može biti referentni model?

ulazi

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

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

a = {13, 7}

Funkcija koju testiramo

```
23 float median(float a[], unsigned len)
24 {
25     sort(a, len);
26
27     if(len % 2)
28         return a[len / 2];
29
30     return 0.5 * (a[len / 2] + a[len / 2 + 1]);
31 }
--
```

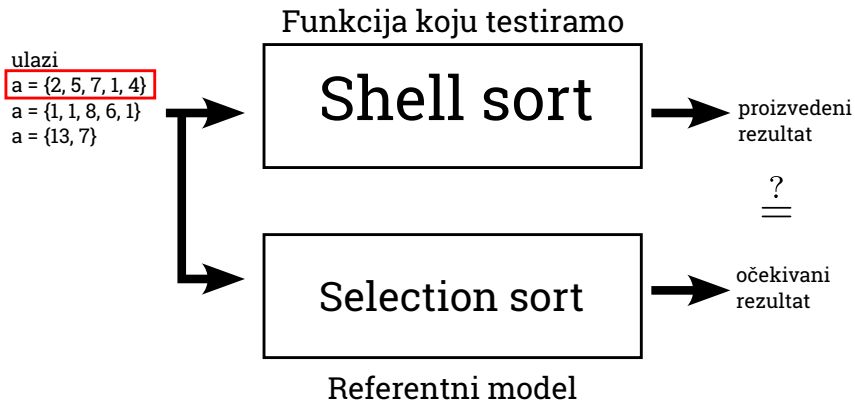
proizvedeni
rezultat

?

Referentni model

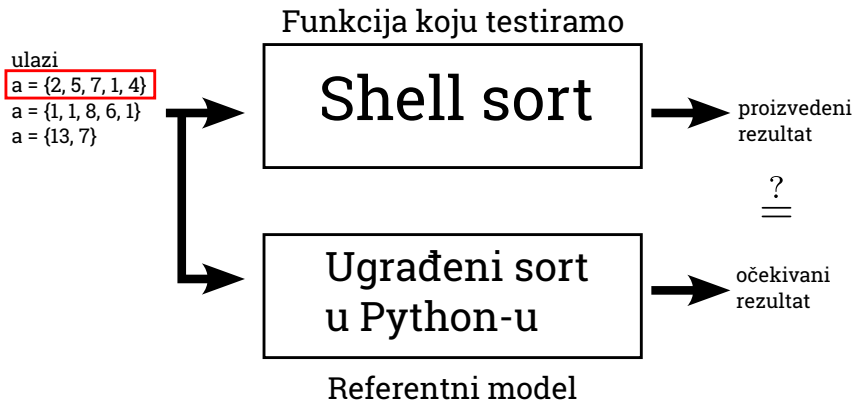
očekivani
rezultat

Šta može biti referentni model?



Referentni model je najčešće sporija ali znatno jednostavnija implementacija iste funkcije

Šta može biti referentni model?

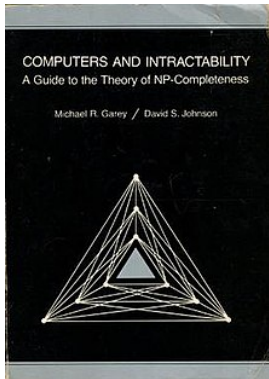


Referentni model je najčešće sporija ali znatno jednostavnija implementacija iste funkcije

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



Stephen Cook
OC 00nt



Cook in 2008

Born Stephen Arthur Cook
December 14, 1939 (age 84)
[Buffalo, New York](#)

Alma mater [Harvard University](#)
[University of Michigan](#)

Known for [NP-completeness](#)
[Propositional proof complexity](#)
[Cook–Levin theorem](#)

Леонид Анатольевич Левин



Дата рождения 2 ноября 1948 (75 лет)
Место рождения [Днепропетровск, Украинская ССР, СССР](#)
Страна СССР, США
Научная сфера информатика
Место работы [Бостонский университет](#)
Альма-матер [МГУ \(мехмат\)](#)

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](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

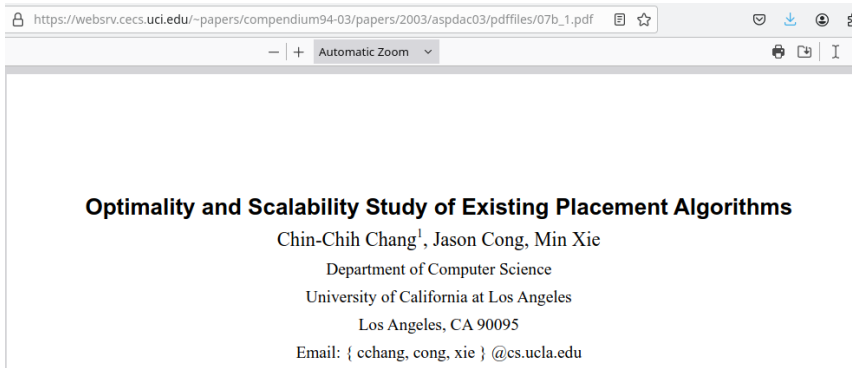
Optimalno rešenje je ono legalno rešenje koje je najbolje spram postavljenog kriterijuma

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

Optimalno rešenje ne mora biti jedinstveno

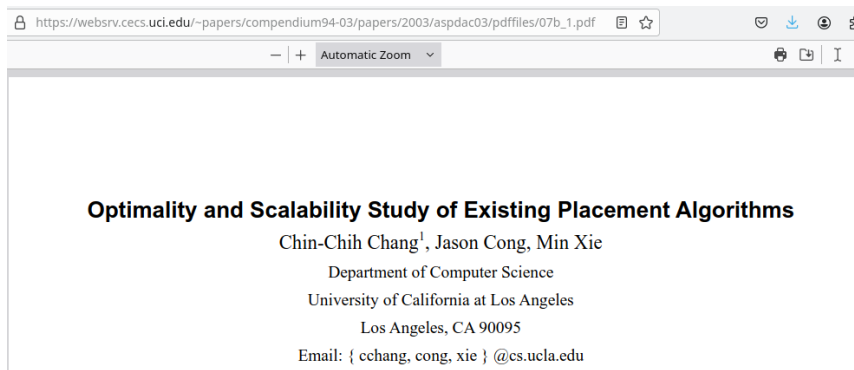
Kako da proverimo koliko je naš algoritam daleko od optimuma?

Konstruišemo instance problema za koje unapred znamo optimalno rešenje



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



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

Sir
Tony Hoare
FRS FREng



Tony Hoare in 2011

Pronunciation [/ˈtɒnihoʊr/](#)

Born Charles Antony Richard Hoare
11 January 1934 (age 90)
[Colombo, British Ceylon](#)

Other names C. A. R. Hoare

Education [University of Oxford \(BA\)](#)
[Moscow State University](#)

Known for [Quicksort](#)
[Quickselect](#)
[Hoare logic](#)
[Null reference](#)
[Communicating sequential](#)

Za to često upotrebljavamo Horovu logiku

Za kratku ilustraciju ćemo pozajmiti jedan primer iz Švedske

Part II. Hoare Logic and Program
Verification

Dilian Gurov

Example

• Program *Abs*

```
if (x > 0) {
  y = x;
} else {
  y = -x;
}
```

can be specified with

preduslov $\langle x = x_0 \rangle$ *Abs* $\langle y = |x_0| \rangle$

naredba

postuslov

37

Proof Tableau

$\langle x = x_0 \rangle$	Precondition
if (x > 0) {	
$\langle x = x_0 \wedge x > 0 \rangle$	If
$\langle x = x_0 \rangle$	Implied ()
y = x;	
$\langle y = x_0 \rangle$	Assignment
} else {	
$\langle x = x_0 \wedge \neg(x > 0) \rangle$	If
$\langle -x = x_0 \rangle$	Implied ()
y = -x;	
$\langle y = x_0 \rangle$	Assignment
}	
$\langle y = x_0 \rangle$	Postcondition

38

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) {  
     $\langle x = x_0 \wedge x > 0 \rangle$   
     $\langle x = |x_0| \rangle$   
    y = x;  
     $\langle y = |x_0| \rangle$   
} else {  
     $\langle x = x_0 \wedge \neg(x > 0) \rangle$   
     $\langle -x = |x_0| \rangle$   
    y = -x;  
     $\langle y = |x_0| \rangle$   
}
```

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

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

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

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

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) {
```

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

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

```
    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

Iz prethodnog iskaza, ovo važi po definiciji

$\langle x = x_0 \rangle$ 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;
```

```
     $\langle y = |x_0| \rangle$ 
```

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 {  
     $\langle x = x_0 \wedge \neg(x > 0) \rangle$   
     $\langle -x = |x_0| \rangle$   
    y = -x;  
     $\langle y = |x_0| \rangle$   
}
```

Ako smo u else grani, x je ≤ 0

$\langle y = |x_0| \rangle$ 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;
```

```
     $\langle y = |x_0| \rangle$ 
```

```
}
```

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

Tada ovo važi po definiciji apsolutne vrednosti

$\langle x = x_0 \rangle$ 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;  
     $\langle y = |x_0| \rangle$   
}
```

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;  
     $\langle y = |x_0| \rangle$   
} 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

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

https://en.wikipedia.org/wiki/Z3_Theorem_Prover

PEDIA
ncyclopedia

Search Wikipedia

Search

Create account Log in ...

1 language

Z3 Theorem Prover

Article Talk

Read Edit View history Tools

From Wikipedia, the free encyclopedia

Z3, also known as the **Z3 Theorem Prover**, is a [satisfiability modulo theories](#) (SMT) solver developed by [Microsoft](#).^[2]

[predicate logic](#)

Overview [\[edit \]](#)

Z3 was developed in the *Research in Software Engineering* (RISE) group at [Microsoft Research Redmond](#) and is targeted at solving problems that arise in [software verification](#) and [program analysis](#). Z3 supports arithmetic, fixed-size bit-vectors, extensional arrays, datatypes, uninterpreted functions, and [quantifiers](#). Its main applications are [extended static checking](#), test case generation, and [predicate abstraction](#).^[citation needed]

Z3 was open sourced in the beginning of 2015.^[3] The source code is licensed under [MIT License](#) and hosted on [GitHub](#).^[4] The solver can be built using [Visual Studio](#), a [makefile](#) or using [CMake](#) and runs on [Windows](#), [FreeBSD](#), [Linux](#), and [macOS](#).

The default input format for Z3 is [SMTLIB2](#). It also has officially supported [bindings](#) for several [programming languages](#), including [C](#), [C++](#), [Python](#), [.NET](#), [Java](#), and [OCaml](#).^[5]

Z3 Theorem Prover

Z3

Original author(s)	Microsoft Research
Developer(s)	Microsoft
Initial release	2012; 12 years ago
Stable release	4.13.0 ^[1] / 7 March 2024; 28 days ago
Repository	github.com/Z3Prover/z3
Written in	C++
Operating system	Windows, FreeBSD, Linux (Debian, Ubuntu), macOS
Platform	IA-32, x86-64, WebAssembly, arm64

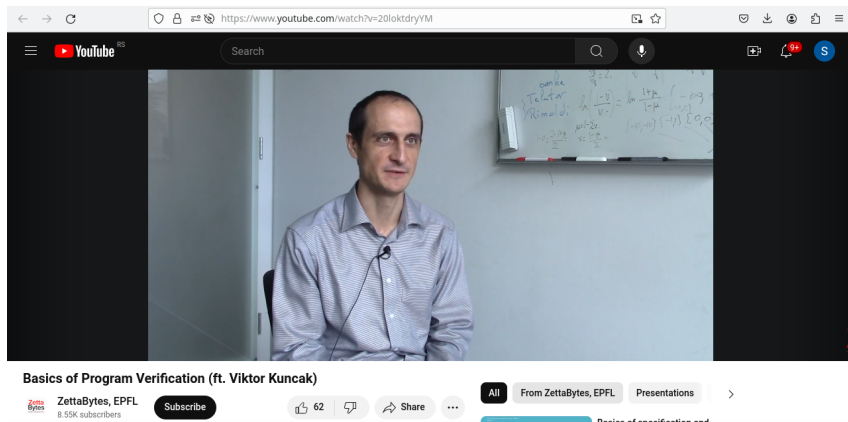
Međutim, automatski dokazivači imaju ograničenu moć

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

Više o verifikaciji možemo čuti od Viktora Kunčaka



<https://www.youtube.com/watch?v=20loktdryYM>

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



Šta su zajedničko imali Ljevin i Hor?

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](#).^[6] 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 Emdein^[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

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

Андрей Николаевич Колмогоров



Имя при рождении фр. *Andrei 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

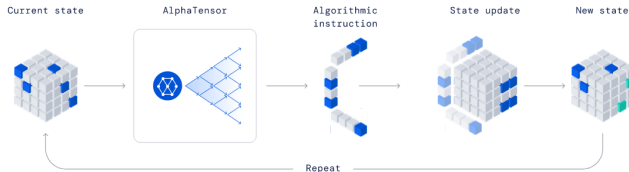
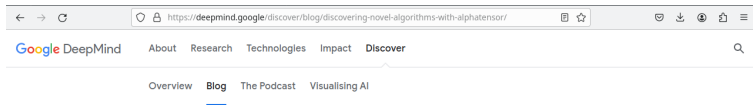
History [edit]

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.

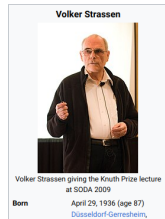
In 1960, Kolmogorov organized a seminar on mathematical problems in *cybernetics* at the *Moscow State University*, where he stated the $\Omega(n^2)$ conjecture and other problems in the *complexity of computation*. Within a week, Karatsuba, then a 23-year-old student, found an algorithm that multiplies 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.



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)

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

Preporuka



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 deklarirana 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)

```
1  /* Copyright (C) 1991-2024 Free Software Foundation, Inc.
2     This file is part of the GNU C Library.
3
4     The GNU C Library is free software; you can redistribute it and/or
5     modify it under the terms of the GNU Lesser General Public
6     License as published by the Free Software Foundation; either
7     version 2.1 of the License, or (at your option) any later version.
8
9     The GNU C Library is distributed in the hope that it will be useful,
10    but WITHOUT ANY WARRANTY; without even the implied warranty of
11    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
12    Lesser General Public License for more details.
13
14    You should have received a copy of the GNU Lesser General Public
15    License along with the GNU C Library; if not, see
16    <https://www.gnu.org/licenses/>.  */
17
18    #include <stdlib.h>
19
20    #undef» abs
21
22    /* Return the absolute value of I.  */
23    int
24    abs (int i)
25    {
26        return i < 0 ? -i : i;
27    }
28
```

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

```
#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: █
```

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

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

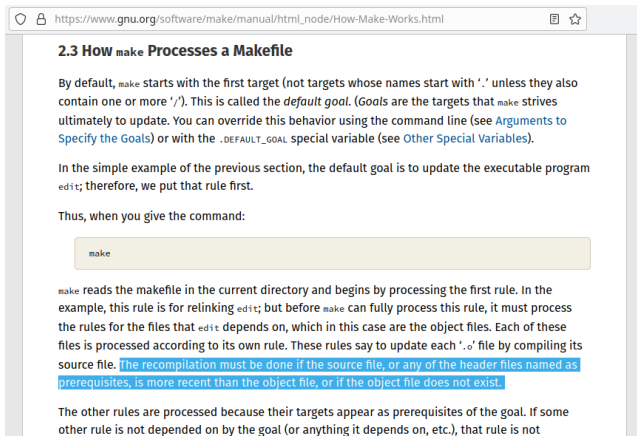
Graf poziva

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



The screenshot shows a web browser window with the address bar displaying `https://www.gnu.org/software/make/manual/html_node/How-Make-Works.html`. The page content is titled "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
```

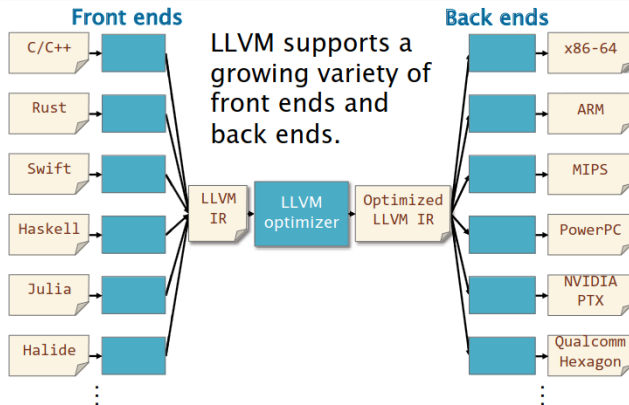
`make` reads the makefile in the current directory and begins by processing the first rule. In the example, this rule is for relinking `edit`; but before `make` can fully process this rule, it must process the rules for the files that `edit` depends on, which in this case are the object files. Each of 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?

Larger Context of the Compiler

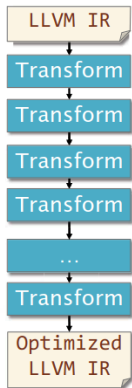


© 2008–2018 by the MIT 6.172 Lecturers

3

Kako izgleda proces kompilacije?

Simple Model of the Compiler



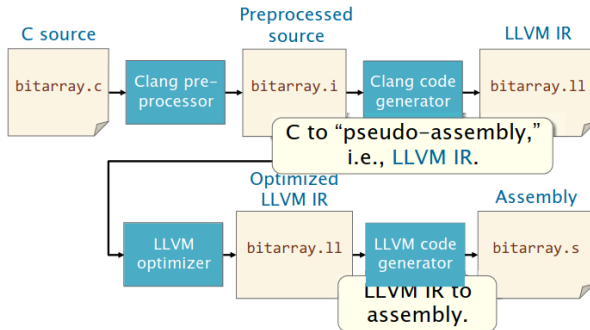
An optimizing compiler performs a sequence of **transformation passes** on the code.

- 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.

Kako izgleda proces kompilacije?

Clang/LLVM Compilation Pipeline

To understand this translation process, let us see how the **compiler** reasons about it.



© 2008-2018 by the MIT 6.172 Lecturers

6

Linker povezuje više objektnih fajlova u jedan izvršni

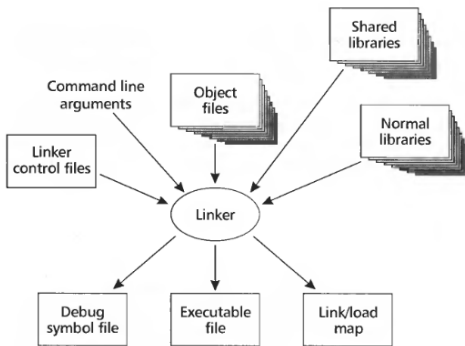


FIGURE 1.1 • The linker process.

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"
```

```
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

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

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

```
~
~
~
~
~
~
~
~
```

```
tacka_2.h
```

10,6

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;
```

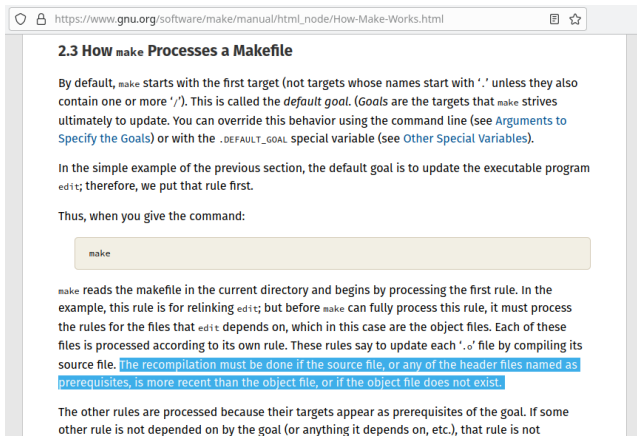
All include_1.cpp

3,0-1

Isti princip primenjujemo i ovde

$(-6, 11)$

Čak i kada nisu potrebne, deklaracije su preporučljive



The screenshot shows a web browser window displaying the GNU Make manual page for section 2.3, titled "How make Processes a Makefile". The browser's address bar shows the URL: https://www.gnu.org/software/make/manual/html_node/How-Make-Works.html. The page content explains the default behavior of the 'make' command, which starts with the first target (the 'default goal') unless overridden. It mentions that the default goal is to update the executable program 'edit', and thus that rule is placed first. A code block shows the command 'make'. The text then explains that 'make' reads the makefile in the current directory and begins by processing the first rule. It notes that 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. A highlighted sentence states: "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." Finally, it states that 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.

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:

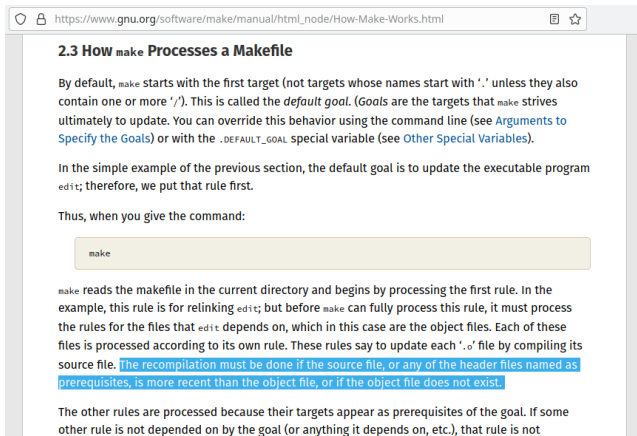
```
make
```

`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



The screenshot shows a web browser window with the URL `https://www.gnu.org/software/make/manual/html_node/How-Make-Works.html`. The page title is "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:

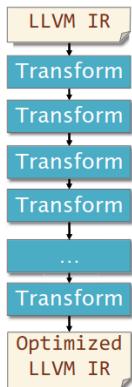
```
$ clang -cc1 -include-pch test.h.pch test.c -o test.s
```

Design Philosophy

Precompiled headers are meant to improve overall performance. 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

Simple Model of the Compiler



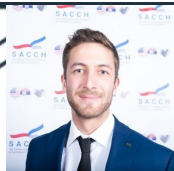
An optimizing compiler performs a sequence of **transformation passes** on the code.

- 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



Optimizing Compiler Heuristics with Machine Learning

Ph.D. Candidate: Dejan Grubišić

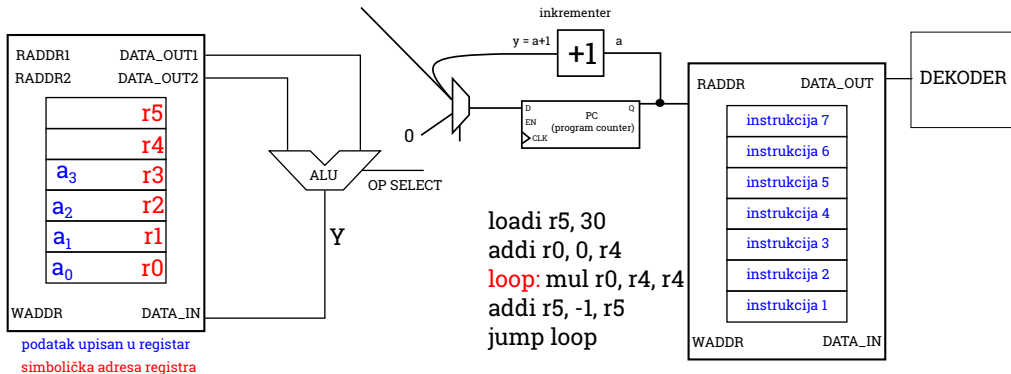
Thesis Committee:
John Mellor-Crummey (chair)
Christopher Jermaine
Ray Simar

Rice University, April 1st, 2024



Kako su funkcije implementirane?

Setimo se principa rada fon Nojmanovog računara



Setimo se još i opsega vidljivosti promenljivih

Promenljive su vidljive samo u bloku u kom su deklarisanе, 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 vrednosti¹

¹Povratna 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

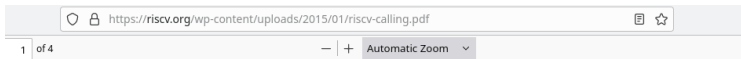


Građanin ostavlja
pismo tamo
gde zna da će ga
poštar naći

Kako poštari i građani znaju gde da ostave pisma?

Kako poštar i građanin znaju gde da ostave pisma?

Postoji dogovor (konvencija) kog se pridržavaju



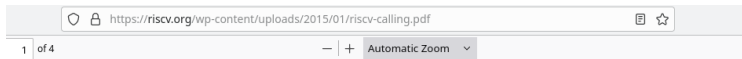
Chapter 18

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 arhitekture procesora pišu pozivne konvencije i određuju „sandučice” (registre) za prenos



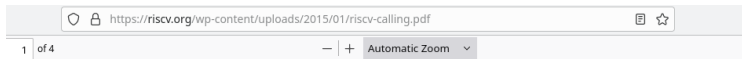
Chapter 18

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



Chapter 18

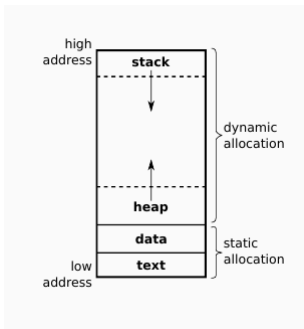
Calling Convention

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?



Compiler Design

Lecture 13: Code generation : Memory management and function call
(EaC Chapter 6&7)

Christophe Dubach
22 February 2021

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

Jedan konkretan primer

The image shows the Godbolt Compiler Explorer interface. The left pane displays the C++ source code for a program that checks if a struct's fields are less than 10. The right pane shows the generated RISC-V assembly for the function `check_s(s)`. A red box highlights the instruction `addi sp, sp, -16`, which is annotated with the text "Spusti stek pokazivač za 16 bajta" (Lower the stack pointer by 16 bytes). The assembly also shows stack frame setup, loading of constants, and a jump to `.L3`. The bottom status bar indicates the compiler is RISC-V (64-bits) gcc (trunk) and the output is 0/0.

```
1 struct s
2 {
3     unsigned a;
4     unsigned b;
5 };
6
7 bool check_s(struct s param)
8 {
9     if(param.a < 10 && param.b < 10)
10         return true;
11
12     return false;
13 }
14
15 bool check_s_ptr(const struct s* param)
16 {
17     if(param -> a < 10 && param -> b < 10)
18         return true;
19
20     return false;
21 }
```

```
1 check_s(s):
2     addi    sp, sp, -16      Spusti stek pokazivač
3     sd      a0, 8(sp)       za 16 bajta
4     li      a5, 9
5     sext.w  a0, a0
6     bgtu    a0, a5, .L3
7     lw      a0, 12(sp)
8     addi    sp, sp, 16
9     sltiu   a0, a0, 10
10    jr      ra
11 .L3:
12    li      a0, 0
13    addi    sp, sp, 16
14    jr      ra
15 check_s_ptr(s const*):
16    lw      a4, 0(a0)
17    li      a5, 9
18    bgtu    a4, a5, .L8
19    lw      a0, 4(a0)
20    sltiu   a0, a0, 10
```

Jedan konkretan primer

The image shows the Compiler Explorer interface with the following components:

- Browser:** Address bar shows `https://godbolt.org`.
- Compiler:** RISC-V (64-bits) gcc (trunk) (Editor #1).
- Source Code (C++):**

```
1 struct s
2 {
3     unsigned a;
4     unsigned b;
5 };
6
7 bool check_s(struct s param)
8 {
9     if(param.a < 10 && param.b < 10)
10         return true;
11
12     return false;
13 }
14
15 bool check_s_ptr(const struct s* param)
16 {
17     if(param -> a < 10 && param -> b < 10)
18         return true;
19
20     return false;
21 }
```
- Assembly Output (RISC-V):**

```
1 check_s(s):
2     addi    sp,sp,-16
3     sd      a0,8(sp)
4     li      a5,9
5     sext.w  a0,a0
6     bgtu    a0,a5,.L3
7     lw      a0,12(sp)
8     addi    sp,sp,16
9     sltiu   a0,a0,10
10    jr      ra
11 .L3:
12     li      a0,0
13     addi    sp,sp,16
14     jr      ra
15 check_s_ptr(s const*):
16     lw      a4,0(a0)
17     li      a5,9
18     bgtu    a4,a5,.L8
19     lw      a0,4(a0)
20     sltiu   a0,a0,10
```
- Annotations:**
 - Line 3: `sd a0,8(sp)` is highlighted with a red box. Text: "Sačuvaj vrednost registra a0 na stek, 8 bajta iznad sp-a".
 - Line 6: `bgtu a0,a5,.L3`. Text: "Primetimo da je a0 veličine 8 bajta, a polja strukture imaju po 4, tako da kompajler može da spakuje celu strukturu u jedan registar".
 - Line 11: `.L3:`. Text: "Da to nije slučaj, kopiranje bi trajalo duže".
- Output:** Output (0/0) RISC-V (64-bits) gcc (trunk) i - 601ms (5712B) ~389 lines filtered.

Jedan konkretan primer

The image shows the Godbolt compiler explorer interface. On the left, the C++ source code is displayed in a file named 'source #1'. It defines a struct 's' with two unsigned integers 'a' and 'b', and two functions: 'check_s' and 'check_s_ptr'. The 'check_s' function takes a struct 's' by value and returns true if both 'a' and 'b' are less than 10. The 'check_s_ptr' function takes a pointer to struct 's' and returns true if the pointed-to struct's 'a' and 'b' are less than 10. On the right, the RISC-V assembly output for 'gcc (trunk)' is shown. The assembly is divided into sections for 'check_s(s):', '.L3:', and 'check_s_ptr(s const*)'. In the 'check_s(s):' section, line 4 contains the instruction 'li a5,9', which is highlighted with a red box. A comment next to it says 'upiši konstantu 9 u registar a5'. The assembly also shows stack frame setup with 'addi sp, sp, -16' and 'sd a0, 8(sp)', and various comparisons and jumps. The bottom status bar indicates the output was generated by 'RISC-V (64-bits) gcc (trunk)' in 601ms, with 389 lines filtered.

```
1 struct s
2 {
3     unsigned a;
4     unsigned b;
5 };
6
7 bool check_s(struct s param)
8 {
9     if(param.a < 10 && param.b < 10)
10         return true;
11
12     return false;
13 }
14
15 bool check_s_ptr(const struct s* param)
16 {
17     if(param->a < 10 && param->b < 10)
18         return true;
19
20     return false;
21 }
```

```
1 check_s(s):
2     addi    sp, sp, -16
3     sd      a0, 8(sp)
4     li      a5, 9
5     sext.w  a0, a0
6     bgtu    a0, a5, .L3
7     lw      a0, 12(sp)
8     addi    sp, sp, 16
9     sltiu   a0, a0, 10
10    jr       ra
11 .L3:
12     li      a0, 0
13     addi    sp, sp, 16
14     jr       ra
15 check_s_ptr(s const*):
16     lw      a4, 0(a0)
17     li      a5, 9
18     bgtu    a4, a5, .L8
19     lw      a0, 4(a0)
20     sltiu   a0, a0, 10
```

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

Jedan konkretan primer

The screenshot displays the Godbolt compiler explorer interface. On the left, the C++ source code for a program is shown. It defines a struct `s` with two unsigned integers `a` and `b`. Two functions are provided: `check_s` and `check_s_ptr`, both of which check if `param.a < 10` and `param.b < 10` and return a boolean result. The right pane shows the RISC-V assembly generated by gcc (trunk) for 64-bit. The assembly includes instructions for stack frame setup, loading constants, and conditional jumps. A red box highlights the instruction `sxt.w a0,a0` on line 5, with a comment explaining its purpose: to sign-extend the register `a0` by writing the lower 4 bytes of the register back into it. The bottom status bar indicates the compilation was successful, with an output of 0/0 and a time of 601ms.

```
1 struct s
2 {
3     unsigned a;
4     unsigned b;
5 };
6
7 bool check_s(struct s param)
8 {
9     if(param.a < 10 && param.b < 10)
10         return true;
11
12     return false;
13 }
14
15 bool check_s_ptr(const struct s* param)
16 {
17     if(param->a < 10 && param->b < 10)
18         return true;
19
20     return false;
21 }
```

```
1 check_s(s):
2     addi sp,sp,-16
3     sd a0,8(sp)
4     li a5,9
5     sxt.w a0,a0      upiši donju reč (4 bajta)
6     bgtu a0,a5,.L3   registra a0 (proširenu znakom)
7     lw a0,12(sp)     u registar a0
8     addi sp,sp,16
9     sltiu a0,a0,10   Ovo znači da registar a0 sada
10    jr ra             sadrži polje a strukture param
11 .L3:
12     li a0,0
13     addi sp,sp,16
14     jr ra
15 check_s_ptr(s const*):
16     lw a4,0(a0)
17     li a5,9
18     bgtu a4,a5,.L8
19     lw a0,4(a0)
20     sltiu a0,a0,10
```

Jedan konkretan primer

The screenshot displays the Godbolt compiler explorer interface. The left pane shows the C++ source code for a program with two functions: `check_s` and `check_s_ptr`. The right pane shows the generated RISC-V assembly for the `check_s` function. A red box highlights the instruction `bgtu a0,a5,.L3` on line 6, with a handwritten note in Croatian: "ako je a0 > a5, skoči na a3". The assembly includes stack frame setup, argument passing, and conditional branching logic.

```
1 struct s
2 {
3     unsigned a;
4     unsigned b;
5 };
6
7 bool check_s(struct s param)
8 {
9     if(param.a < 10 && param.b < 10)
10         return true;
11
12     return false;
13 }
14
15 bool check_s_ptr(const struct s* param)
16 {
17     if(param -> a < 10 && param -> b < 10)
18         return true;
19
20     return false;
21 }
```

```
1 check_s(s):
2     addi    sp,sp,-16
3     sd      a0,8(sp)
4     li      a5,9
5     sext.w  a0,a0
6     bgtu    a0,a5,.L3    ako je a0 > a5, skoči na a3
7     lw      a0,12(sp)
8     addi    sp,sp,16
9     sltiu   a0,a0,10
10    jr       ra
11 .L3:
12     li      a0,0
13     addi    sp,sp,16
14     jr       ra
15 check_s_ptr(s const*):
16     lw      a4,0(a0)
17     li      a5,9
18     bgtu    a4,a5,.L8
19     lw      a0,4(a0)
20     sltiu   a0,a0,10
```

Jedan konkretan primer

The screenshot displays the Godbolt compiler explorer interface. The left pane shows the C++ source code for a program that checks if a struct's fields are less than 10. The right pane shows the generated RISC-V assembly code for the same program, compiled with gcc (trunk). A red box highlights the instruction `li a0, 0` in the assembly, which is annotated with the text: "upiši 0 u registr a0 u kom će pozivajuća funkcija moći da pronade povratnu vrednost". The bottom status bar indicates the output is 0/0, the compilation took 601ms, and there are 389 lines filtered.

```
1 struct s
2 {
3     unsigned a;
4     unsigned b;
5 };
6
7 bool check_s(struct s param)
8 {
9     if(param.a < 10 && param.b < 10)
10         return true;
11
12     return false;
13 }
14
15 bool check_s_ptr(const struct s* param)
16 {
17     if(param -> a < 10 && param -> b < 10)
18         return true;
19
20     return false;
21 }
```

```
1 check_s(s):
2     addi    sp, sp, -16
3     sd      a0, 8(sp)
4     li      a5, 9
5     sext.w  a0, a0
6     bgtu    a0, a5, .L3
7     lw      a0, 12(sp)
8     addi    sp, sp, 16
9     sltiu   a0, a0, 10
10    jr      ra
11 .L3:
12    li      a0, 0
13    addi    sp, sp, 16
14    jr      ra
15 check_s_ptr(s const*):
16    lw      a4, 0(a0)
17    li      a5, 9
18    bgtu    a4, a5, .L8
19    lw      a0, 4(a0)
20    sltiu   a0, a0, 10
```

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

Jedan konkretan primer

The screenshot displays the Godbolt compiler explorer interface. The left pane shows the C++ source code for a program that checks if a struct's fields are less than 10. The right pane shows the generated RISC-V assembly code for the same program, compiled with gcc (trunk). The assembly code is color-coded to match the C++ source, and a red box highlights the instruction `addi sp, sp, 16` with the comment "obriši sve sa steka" (erase everything from the stack).

```
1 struct s
2 {
3     unsigned a;
4     unsigned b;
5 };
6
7 bool check_s(struct s param)
8 {
9     if(param.a < 10 && param.b < 10)
10         return true;
11
12     return false;
13 }
14
15 bool check_s_ptr(const struct s* param)
16 {
17     if(param -> a < 10 && param -> b < 10)
18         return true;
19
20     return false;
21 }
```

```
1 check_s(s):
2     addi    sp, sp, -16
3     sd      a0, 8(sp)
4     li      a5, 9
5     sext.w  a0, a0
6     bgtu    a0, a5, .L3
7     lw      a0, 12(sp)
8     addi    sp, sp, 16
9     sltiu   a0, a0, 10
10    jr      ra
11 .L3:
12     li      a0, 0
13     addi    sp, sp, 16    obriši sve sa steka
14     jr      ra
15 check_s_ptr(s const*):
16     lw      a4, 0(a0)
17     li      a5, 9
18     bgtu    a4, a5, .L8
19     lw      a0, 4(a0)
20     sltiu   a0, a0, 10
```

Jedan konkretan primer

The screenshot displays the Godbolt compiler explorer interface. The left pane shows the C++ source code for a program with two functions: `check_s` and `check_s_ptr`. The right pane shows the generated RISC-V assembly for the `check_s` function. A red box highlights the `jr ra` instruction at line 14, with a comment "skoči na povratnu adresu" (jump to return address) next to it. The bottom status bar indicates the compiler used is RISC-V (64-bits) gcc (trunk) and shows the output of the compilation.

```
1 struct s
2 {
3     unsigned a;
4     unsigned b;
5 };
6
7 bool check_s(struct s param)
8 {
9     if(param.a < 10 && param.b < 10)
10         return true;
11
12     return false;
13 }
14
15 bool check_s_ptr(const struct s* param)
16 {
17     if(param -> a < 10 && param -> b < 10)
18         return true;
19
20     return false;
21 }
```

```
1 check_s(s):
2     addi    sp,sp,-16
3     sd      a0,8(sp)
4     li      a5,9
5     sext.w  a0,a0
6     bgtu    a0,a5,.L3
7     lw      a0,12(sp)
8     addi    sp,sp,16
9     sltiu   a0,a0,10
10    jr      ra
11 .L3:
12    li      a0,0
13    addi    sp,sp,16
14    jr      ra      skoči na povratnu adresu
15 check_s_ptr(s const*):
16    lw      a4,0(a0)
17    li      a5,9
18    bgtu    a4,a5,.L8
19    lw      a0,4(a0)
20    sltiu   a0,a0,10
```

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

Jedan konkretan primer

The image shows the Compiler Explorer interface at <https://godbolt.org>. The left pane displays the C++ source code for a program that checks if a struct's fields are less than 10. The right pane shows the generated RISC-V assembly for the same code, with a red box highlighting the instruction `lw a0, 12(sp)` and a comment explaining its purpose.

C++ Source Code (Left Pane):

```
1 struct s
2 {
3     unsigned a;
4     unsigned b;
5 };
6
7 bool check_s(struct s param)
8 {
9     if(param.a < 10 && param.b < 10)
10         return true;
11
12     return false;
13 }
14
15 bool check_s_ptr(const struct s* param)
16 {
17     if(param -> a < 10 && param -> b < 10)
18         return true;
19
20     return false;
21 }
```

RISC-V Assembly (Right Pane):

```
1 check_s(s):
2     addi sp, sp, -16
3     sd a0, 8(sp)
4     li a5, 9
5     sext.w a0, a0
6     bgtu a0, a5, .L3
7     lw a0, 12(sp)    učitaj polje b u registar a0
8     addi sp, sp, 16
9     sltiu a0, a0, 10
10    jr ra
11 .L3:
12    li a0, 0
13    addi sp, sp, 16
14    jr ra
15 check_s_ptr(s const*):
16    lw a4, 0(a0)
17    li a5, 9
18    bgtu a4, a5, .L8
19    lw a0, 4(a0)
20    sltiu a0, a0, 10
```

Jedan konkretan primer

The image shows the Compiler Explorer interface at <https://godbolt.org>. The left pane displays the C++ source code for a program that checks if a struct's fields are less than 10. The right pane shows the generated RISC-V assembly code for the same program using gcc (trunk). A red box highlights the instruction `addi sp, sp, 16` on line 8, with a comment "obriši sve sa steka" (erase everything from the stack) next to it. The bottom status bar indicates the output is 0/0, the compiler is RISC-V (64-bits) gcc (trunk), and the execution time is 601ms (5712B) with ~389 lines filtered.

```
1 struct s
2 {
3     unsigned a;
4     unsigned b;
5 };
6
7 bool check_s(struct s param)
8 {
9     if(param.a < 10 && param.b < 10)
10         return true;
11
12     return false;
13 }
14
15 bool check_s_ptr(const struct s* param)
16 {
17     if(param -> a < 10 && param -> b < 10)
18         return true;
19
20     return false;
21 }
```

```
1 check_s(s):
2     addi    sp, sp, -16
3     sd      a0, 8(sp)
4     li      a5, 9
5     sext.w  a0, a0
6     bgtu    a0, a5, .L3
7     lw      a0, 12(sp)
8     addi    sp, sp, 16      obriši sve sa steka
9     sltiu   a0, a0, 10
10    jr      ra
11 .L3:
12    li      a0, 0
13    addi    sp, sp, 16
14    jr      ra
15 check_s_ptr(s const*):
16    lw      a4, 0(a0)
17    li      a5, 9
18    bgtu    a4, a5, .L8
19    lw      a0, 4(a0)
20    sltiu   a0, a0, 10
```

Jedan konkretan primer

The screenshot displays the Godbolt compiler explorer interface. The left pane shows the C++ source code for a program with two functions: `check_s` and `check_s_ptr`. The right pane shows the generated RISC-V assembly for the `check_s` function. A red box highlights the instruction `sltiu a0,a0,10` on line 9, with a comment in Croatian explaining its purpose: "ako je a0 < 10, upiši 1 u a0, u suprotnom upiši 0 u a0". The bottom status bar indicates the compiler used is RISC-V (64-bits) gcc (trunk) and shows the output as 0/0.

```
1 struct s
2 {
3     unsigned a;
4     unsigned b;
5 };
6
7 bool check_s(struct s param)
8 {
9     if(param.a < 10 && param.b < 10)
10         return true;
11
12     return false;
13 }
14
15 bool check_s_ptr(const struct s* param)
16 {
17     if(param -> a < 10 && param -> b < 10)
18         return true;
19
20     return false;
21 }
```

```
1 check_s(s):
2     addi    sp,sp,-16
3     sd      a0,8(sp)
4     li      a5,9
5     sext.w  a0,a0
6     bgtu    a0,a5,.L3
7     lw      a0,12(sp)
8     addi    sp,sp,16
9     sltiu   a0,a0,10    ako je a0 < 10, upiši 1 u a0,
10     jr      ra           u suprotnom upiši 0 u a0
11 .L3:
12     li      a0,0
13     addi    sp,sp,16
14     jr      ra
15 check_s_ptr(s const*):
16     lw      a4,0(a0)
17     li      a5,9
18     bgtu    a4,a5,.L8
19     lw      a0,4(a0)
20     sltiu   a0,a0,10
```

Jedan konkretan primer

The screenshot displays the Godbolt compiler explorer interface. The left pane shows the C++ source code for a program with two functions: `check_s` and `check_s_ptr`. The right pane shows the generated RISC-V assembly for the `check_s` function. A red box highlights the `jr ra` instruction at line 10, with a comment "skoči na povratnu adresu" (jump to return address) next to it. The bottom status bar indicates the compiler is RISC-V (64-bits) gcc (trunk) and shows compilation statistics.

```
1 struct s
2 {
3     unsigned a;
4     unsigned b;
5 };
6
7 bool check_s(struct s param)
8 {
9     if(param.a < 10 && param.b < 10)
10         return true;
11
12     return false;
13 }
14
15 bool check_s_ptr(const struct s* param)
16 {
17     if(param -> a < 10 && param -> b < 10)
18         return true;
19
20     return false;
21 }
```

```
1 check_s(s):
2     addi    sp,sp,-16
3     sd      a0,8(sp)
4     li      a5,9
5     sext.w  a0,a0
6     bgtu    a0,a5,.L3
7     lw      a0,12(sp)
8     addi    sp,sp,16
9     sltiu   a0,a0,10
10    jr      ra      skoči na povratnu adresu
11 .L3:
12     li      a0,0
13     addi    sp,sp,16
14     jr      ra
15 check_s_ptr(s const*):
16     lw      a4,0(a0)
17     li      a5,9
18     bgtu    a4,a5,.L8
19     lw      a0,4(a0)
20     sltiu   a0,a0,10
```

Zašto ovo moramo dobro da zapamtimo?

The image shows a screenshot of the Compiler Explorer website (https://godbolt.org) with the Compiler Explorer logo in the top left. The interface displays two panels: the source code on the left and the generated assembly on the right.

Source Code (C++):

```
1 struct s
2 {
3     unsigned a;
4     unsigned b;
5 };
6
7 bool check_s(struct s param)
8 {
9     if(param.a < 10 && param.b < 10)
10         return true;
11
12     return false;
13 }
14
15 bool check_s_ptr(const struct s* param)
16 {
17     if(param -> a < 10 && param -> b < 10)
18         return true;
19
20     return false;
21 }
```

Assembly (RISC-V 64-bits gcc (trunk)):

```
1 check_s(s):
2     addi    sp,sp,-16
3     sd      a0,8(sp)
4     li      a5,9
5     sext.w  a0,a0
6     bgtu    a0,a5,.L3
7     lw      a0,12(sp)
8     addi    sp,sp,16
9     sltiu   a0,a0,10
10    jr      ra
11 .L3:
12     li      a0,0
13     addi    sp,sp,16
14     jr      ra
15 check_s_ptr(s const*):
16     lw      a4,0(a0)
17     li      a5,9
18     bgtu    a4,a5,.L8
19     lw      a0,4(a0)
20     sltiu   a0,a0,10
```

The assembly output is color-coded by instruction type. A red box highlights the instruction `addi sp, sp, 16` on line 13, with the comment "obriši sve sa steka" (erase everything from the stack) next to it. The status bar at the bottom indicates the compiler is RISC-V (64-bits) gcc (trunk) and shows the output of the compilation.

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

Potprogram 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

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>

— + Automatic Zoom ▾

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/RISC 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
```

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)

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

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

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`.

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 funkcije 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?

Prosledjivanje 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

Compiler Explorer interface showing C++ source code and its RISC-V assembly output.

C++ source code (C++ source #1):

```
1 struct s
2 {
3     unsigned a;
4     unsigned b;
5 };
6
7 bool check_s(struct s param)
8 {
9     if(param.a < 10 && param.b < 10)
10         return true;
11     return false;
12 }
13
14
15 bool check_s_ptr(const struct s* param)
16 {
17     if(param -> a < 10 && param -> b < 10)
18         return true;
19     return false;
20 }
21
```

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

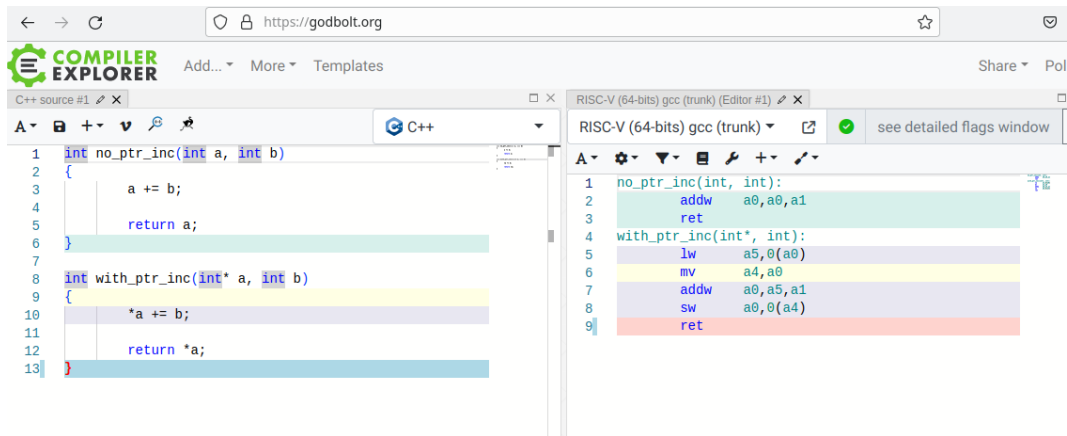
```
8      addi    sp,sp,10
9      sltiu   a0,a0,10
10     jr       ra
11
12     .L3:
13     li       a0,0
14     addi     sp,sp,16
15     jr       ra
16
17     check_s_ptr(s const*):
18     lw       a4,0(a0)
19     li       a5,9
20     bgtu     a4,a5,.L8
21     lw       a0,4(a0)
22     sltiu   a0,a0,10
23     ret
24
25     .L8:
26     li       a0,0
27     ret
```

Pokazivač se prosleđuje u registru a0, pa nema potrebe za kopiranjem parametra na stek

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

Compiler License

Ipak, i dereferenciranje pokazivača košta



The screenshot displays the Godbolt Compiler Explorer interface. The left pane shows the C++ source code for two functions: `no_ptr_inc` and `with_ptr_inc`. The right pane shows the corresponding RISC-V assembly code generated by GCC (trunk) for a 64-bit architecture. The assembly code for `no_ptr_inc` is simple, consisting of an `addw` instruction and a `ret` instruction. The assembly code for `with_ptr_inc` is more complex, involving memory access instructions (`lw`, `mv`, `sw`) to dereference the pointer before performing the addition.

```
1 int no_ptr_inc(int a, int b)
2 {
3     a += b;
4
5     return a;
6 }
7
8 int with_ptr_inc(int* a, int b)
9 {
10    *a += b;
11
12    return *a;
13 }
```

```
1 no_ptr_inc(int, int):
2     addw    a0,a0,a1
3     ret
4 with_ptr_inc(int*, int):
5     lw      a5,0(a0)
6     mv      a4,a0
7     addw    a0,a5,a1
8     sw      a0,0(a4)
9     ret
```

No, o svemu tome više u sredu

Hvala na pažnji