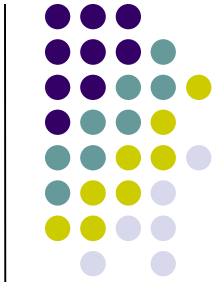
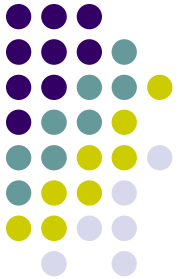


Функције у Цеу

NIT



Функције?

- Основна идеја је да се сложене операције издвоје у посебну целину. Последица:
  - Једноставнији, и јаснији, код на месту примене
  - Једноставније, и сигурније, коришћење истог кода на више места

- Декларација – назив, тип повратне вредности, параметри, спецификатори
- Дефиниција – исто као декларација али укључује и тело функције

Шта ради овај код?

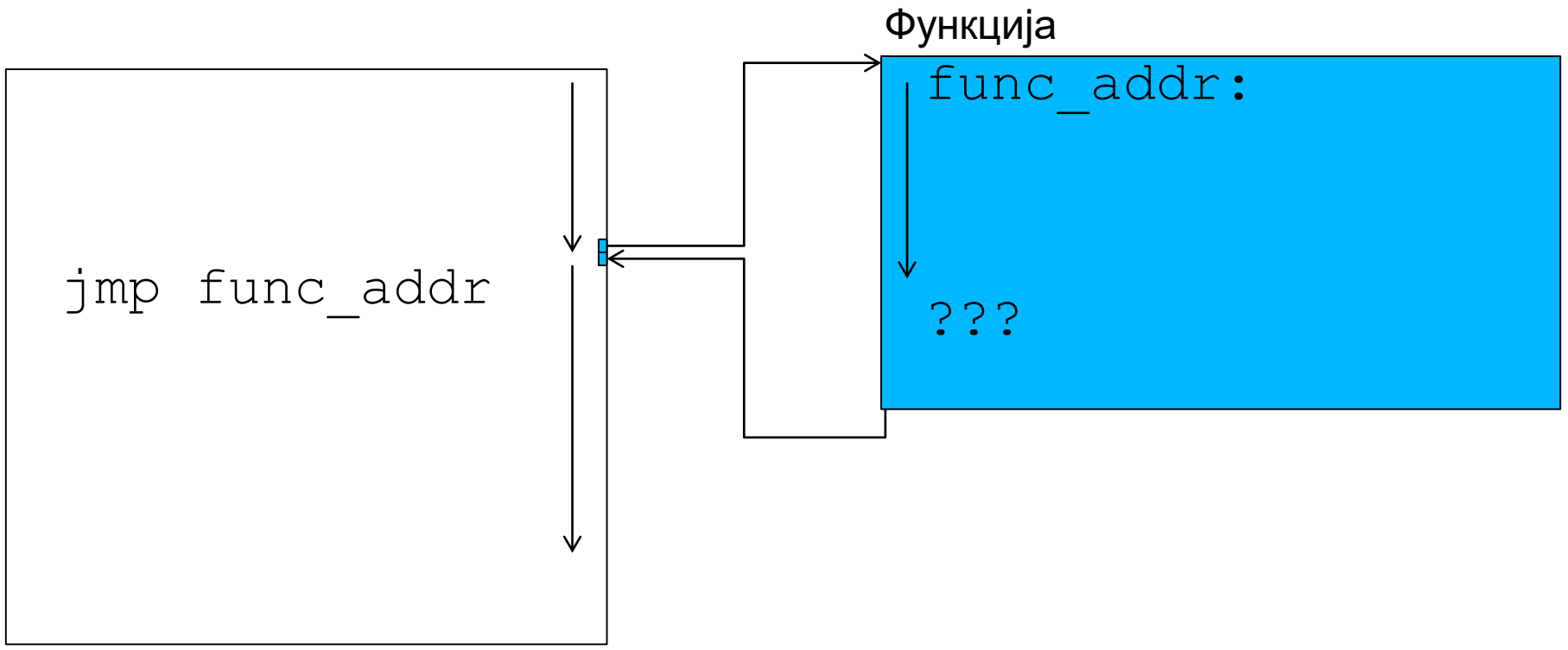
```
void foo(int** mat, int n, int m)
{
  int k;
  for (k = 0; k < m; k += 2)
  {
    int i;
    for (i = 0; i < (n - 1); i++)
    {
      int j;
      for (j = i + 1; j < n; j++)
      {
        if (mat[k][i] < mat[k][j])
        {
          int tmp;
          tmp = mat[k][i];
          mat[k][i] = mat[k][j];
          mat[k][j] = tmp;
        }
      }
    }
  }
  for (k = 1; k < m; k += 2)
  {
    int i;
    for (i = 0; i < n; i++)
    {
      mat[k][i] = 0;
    }
  }
}
```

NIT

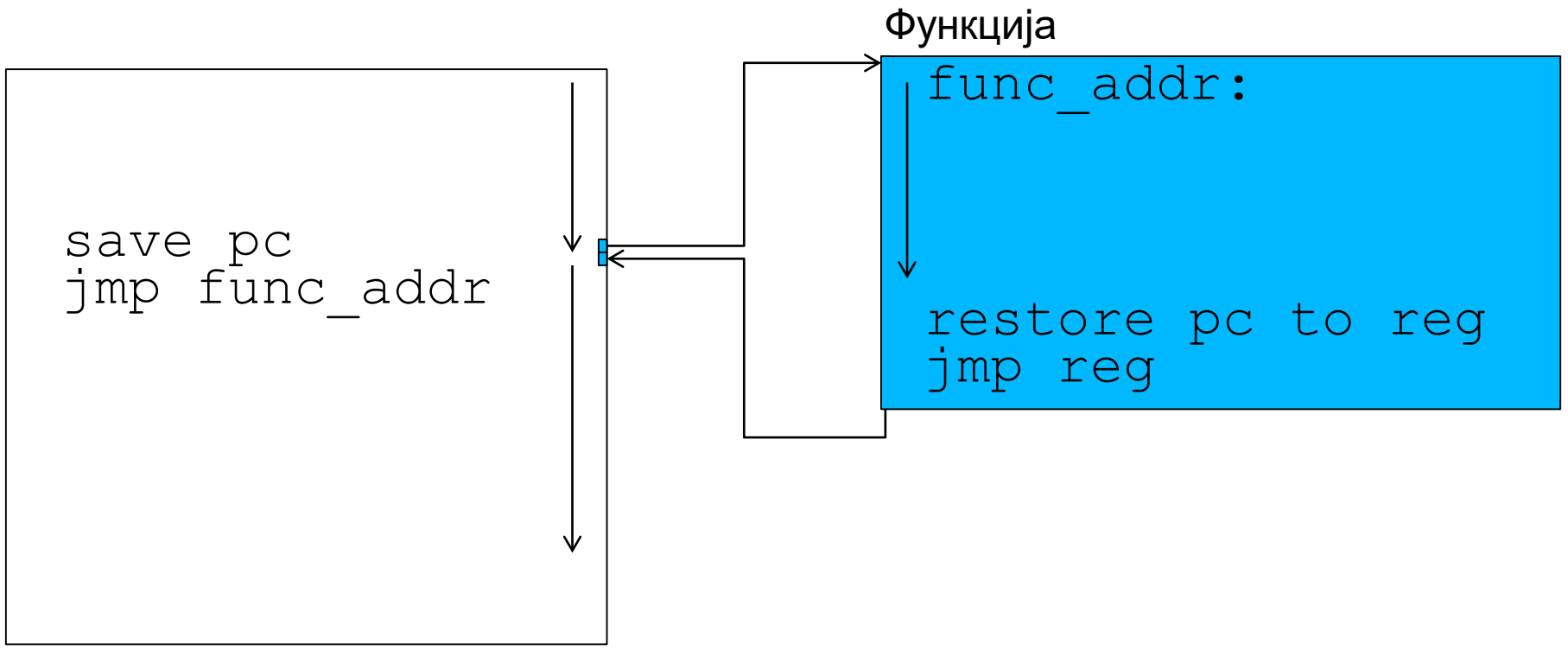
Ради ово

```
void foo(int** mat, int n, int m)
{
  int k;
  for (k = 0; k < m; k += 2)
  {
    sort(mat[k], n);
  }
  for (k = 1; k < m; k += 2)
  {
    zero(mat[k], n);
  }
}
```

Функције



Функције


Функција
func_addr:
save pc
jmp func_addr
restore pc to reg
jmp reg

Дефиниција и декларација функције

```
type-specifier return-type function-name(parameters)
{
  declarations
  statements
  return value;
}
```

- **type-specifier** – одређује видљивост функције (`static` или ништа)
- **return-type** - тип повратне вредности; void ако нема повратне вредности
- **function-name** - јединствено име функције (функција и променљива истог досега не могу се звати исто)
- **parameters** - листа декларација променљивих које представљају параметре функције, међусобно су одвојене зарезом
- **return value;** - вредност која ће бити враћена након завршетка функције (није неопходно ако је тип повратне вредности `void`)

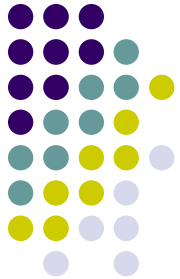
```
return-type function-name(parameter-types)
```

- **parameter-types** - листа типова параметара; за разлику од листе у декларацији са дефиницијом, овде се не морају навести имена параметара

```
int foo(float, int, const long*);
```

NIT

Преношење параметара



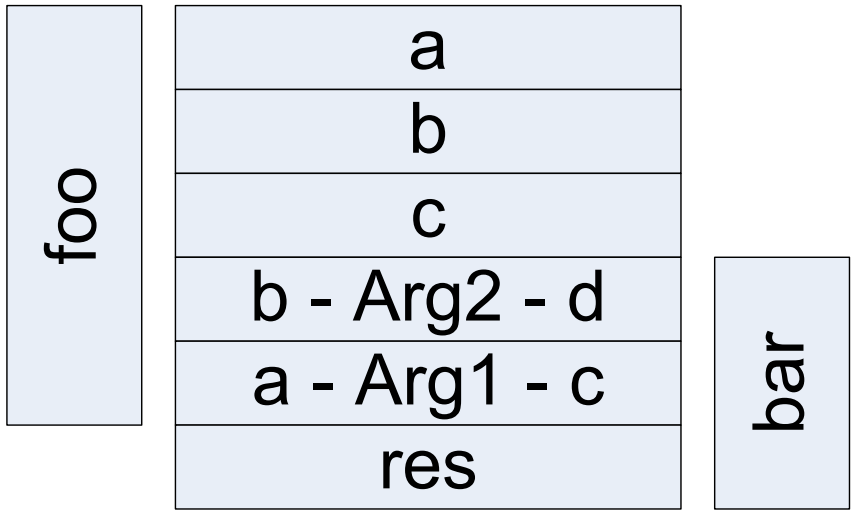
- При позивању функције формални параметри замењују се стварним.

```
void foo(float x, int y);
foo(15, 6);
foo(40, a);
```

- Мора постојати механизам за преношење вредности, тј. параметара функцији при њеном позиву.
- Концептуално постоје два могућа начина преношења параметара:
  - По вредности - функцији се прослеђује копија стварног параметра. Последица: промена параметра унутар функције није видљива у коду који је позива.
  - По референци - функцији се прослеђује баш стварни параметар. Последица: свака промена параметра унутар функције јесте виљива у коду који је позива.

Прослеђивање параметара по вредности

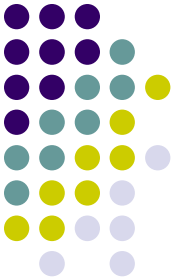
```
int bar(int c, int d)
{
    int res = c + d;
    c = 3;
    d = 7;
    return res;
}

int foo()
{
    int a = 5, b = 9, c;
    c = bar(a, b);
    c = c + a + b;
    return c;
}
```



- **bar не може променити променљиве a и b из функције foo. c и d су сасвим друге променљиве које само при позиву функције добијају вредност коју у том тренутку имају a и b.**
- **Повратна вредност из foo: 28**

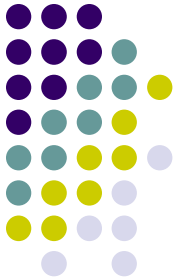
NIT

Прослеђивање параметара по референци



- **Не постоји у Цеу!**
- У Цеу сви параметри се прослеђују по вредности.
- И сада питање гласи: како функција утиче на „спољни свет”, тј. како га мења?
  1. Преко повратне вредности - позивајућа функција може нешто корисно урадити са повратном вредношћу... али и не мора.
  2. Преко глобалних променљивих - свака измена глобалне променљиве утицаће и на свет ван функције. Ово се назива „бочни ефекат” или „споредни ефекат”. Математичари то сматрају великим грехом, али њихова учења се у инжењерској цркви не сматрају догмом.
- Међутим, то није довољно. Преко повратне вредности се може утицати само на једну променљиву, а глобална променљива на коју се утиче је закуцана - не може се мењати од позива до позива.

Прослеђивање параметара по референци



- Решења:
  - Структуре као повратне вредности
  - **Коришћење показивача**
- Када се у Цеу користи механизам показивача за приступ објектима ван функције онда се жаргонски то назива пренос по референци, јер је ефекат сличан.

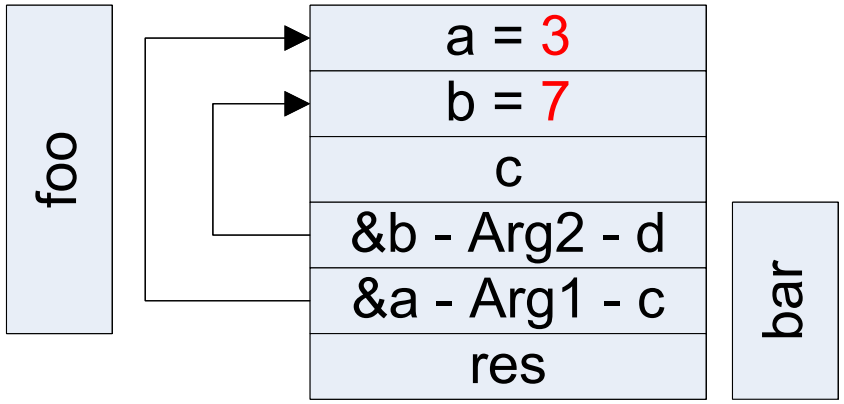
```
void foo(int* x)
{
  *x = 5;
}
```

```
int bar(const int* x)
{
  return *x + 5;
}
```

```
int fiz(int* x)
{
  return x[3] + x[6];
}
```

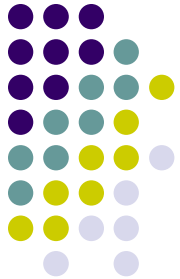
Прослеђивање параметара по референци

```
int bar(int* c, int* d)
{
    int res = *c + *d;
    *c = 3;
    *d = 7;
    return res;
}

int foo()
{
    int a = 5, b = 9, c;
    c = bar(&a, &b);
    c = c + a + b;
    return c;
}
```



- **Друга и трећа линија функције bar утичу на спољни свет**
- **Повратна вредност из foo: 24**

Ефикасност преношења по вредности



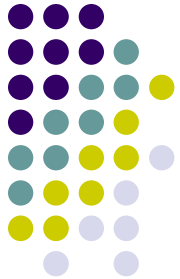
- Стварни параметри морају бити копирани на неко место у меморији, или у одговарајући регистар.
- Било како било - инструкције се троше на то, да не говоримо о меморији.
- Посебно постаје приметно када су променљиве велике.
- Пример за MIPS архитектуру:

```
struct s
{
  int array[7];
};

int func(int a, struct s p, int e)
{
  return e;
}
```

```
struct s
{
  int array[70000];
};

int func(int a, struct s p1, int e)
{
  return e;
}
```

```
    lw      $2,32($sp)
```

```
    li      $2,262144
    addu    $2,$sp,$2
    lw      $2,17860($2)
```



Повратна вредност

- Слично као преношење параметара при позиву, само обрнуто.
- Свака функција, осим `void`, мора имати `return` наредбу.
- Да ли се повратна вредност може вратити „по референци”?

```
struct S
{
  int val1;
  float val2;
};

struct S func()
{
  struct S res;
  res.val1 = 1;
  res.val2 = 2.0;
  return res;
}
```

```
struct S
{
  int val1;
  float val2;
};

struct S* func()
{
  struct S res;
  res.val1 = 1;
  res.val2 = 2.0;
  return &res;
}
```

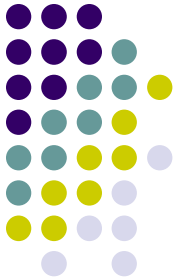
```
struct S
{
  int val1;
  float val2;
};

struct S* func(struct S* res)
{
  res->val1 = 1;
  res->val2 = 2.0;
  return res;
}
```

```
struct S* func()
{
  static struct S res;
  res.val1 = 1;
  res.val2 = 2.0;
  return &res;
}
```

```
struct S* func()
{
  struct S* res;
  res = (struct S*)malloc(
    sizeof(struct S));
  res->val1 = 1;
  res->val2 = 2.0;
  return res;
}
```

NIT

Наравоучение

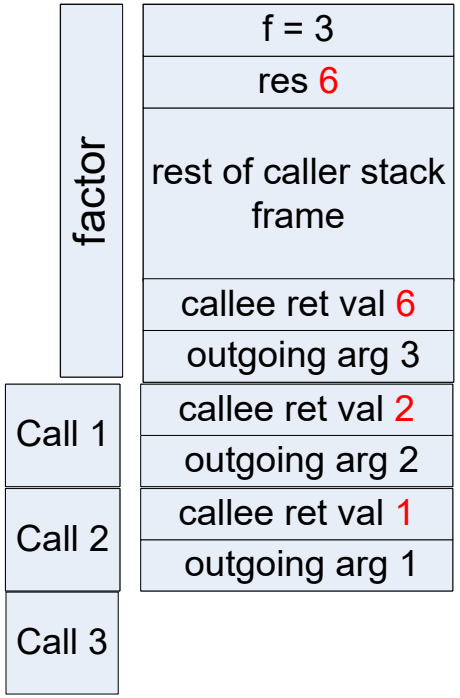


- Добро размислити како је најповољније пренети параметре и повратне вредности.
- За правилну одлуку потребно је добро познавање циљне архитектуре и позивне конвенције.
- Генерално правило је да код наменских система, услед њихових ограничених ресурса, не треба нагомилавати параметре, а веће објекте треба преносити или „преко референце” или путем глобалних променљивих, ако је могуће.

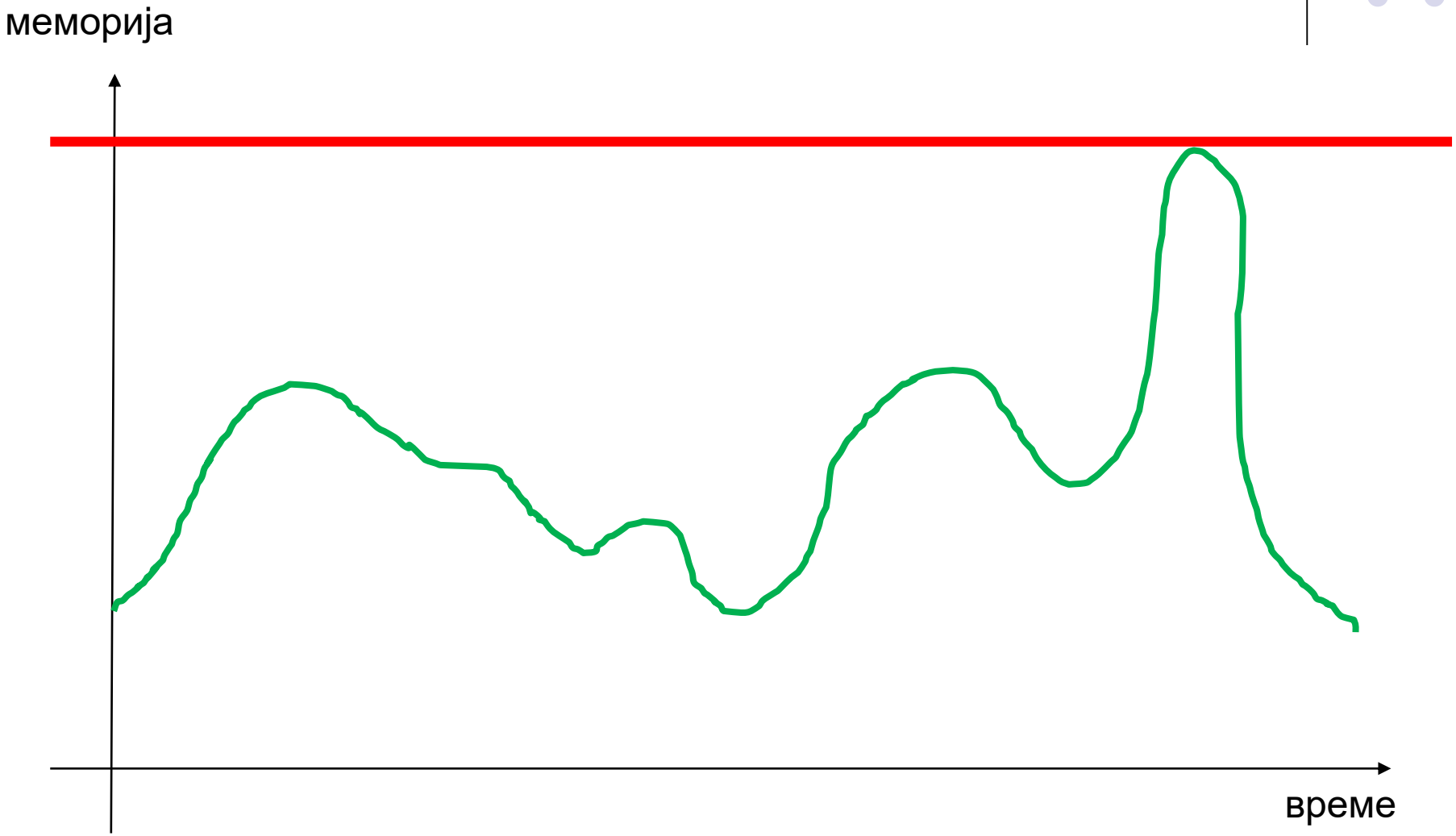
Рекурзија

- Када функција зове саму себе
- Може бити директна и индиректна
- Један од два разлога зашто се користи стек (који је други?)

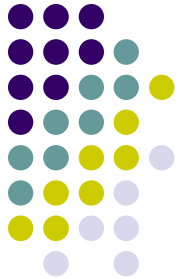
```
void caller()
{
  int f = 3, res;
  res = factor(f);
  printf("factorial %d = %d\n", f, res);
}


long factor(long n)
{
  if (n <= 1) /* terminal condition*/
    return 1;
  else
    return(n * factor(n - 1));
}
```



Колико програму треба меморије

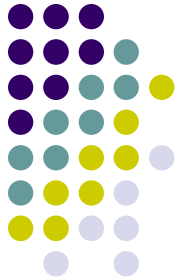


Колико програму треба меморије



- Три главне групе меморије:
- Меморија за променљиве статичке трајности (Меморија за податке)
- Радна меморија за сваку функцију
- Меморија која се динамички заузима

NIT



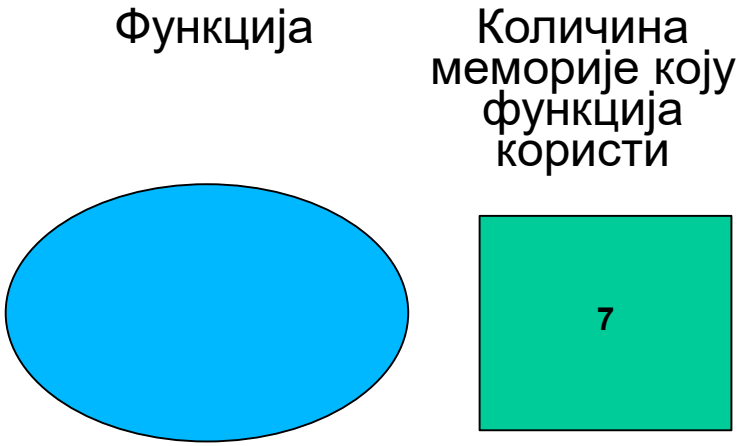
Меморија за податке

- Чине је све променљиве статичке трајности (све глобалне, плус локалне са `static` испред).
- Може се срачунати на основу знања основних карактеристика платформе.
- Меморија за променљиве нитске трајности само треба да се помножи са бројем нити које се креирају.

```
char a;
int b;
char c;
short d;
```

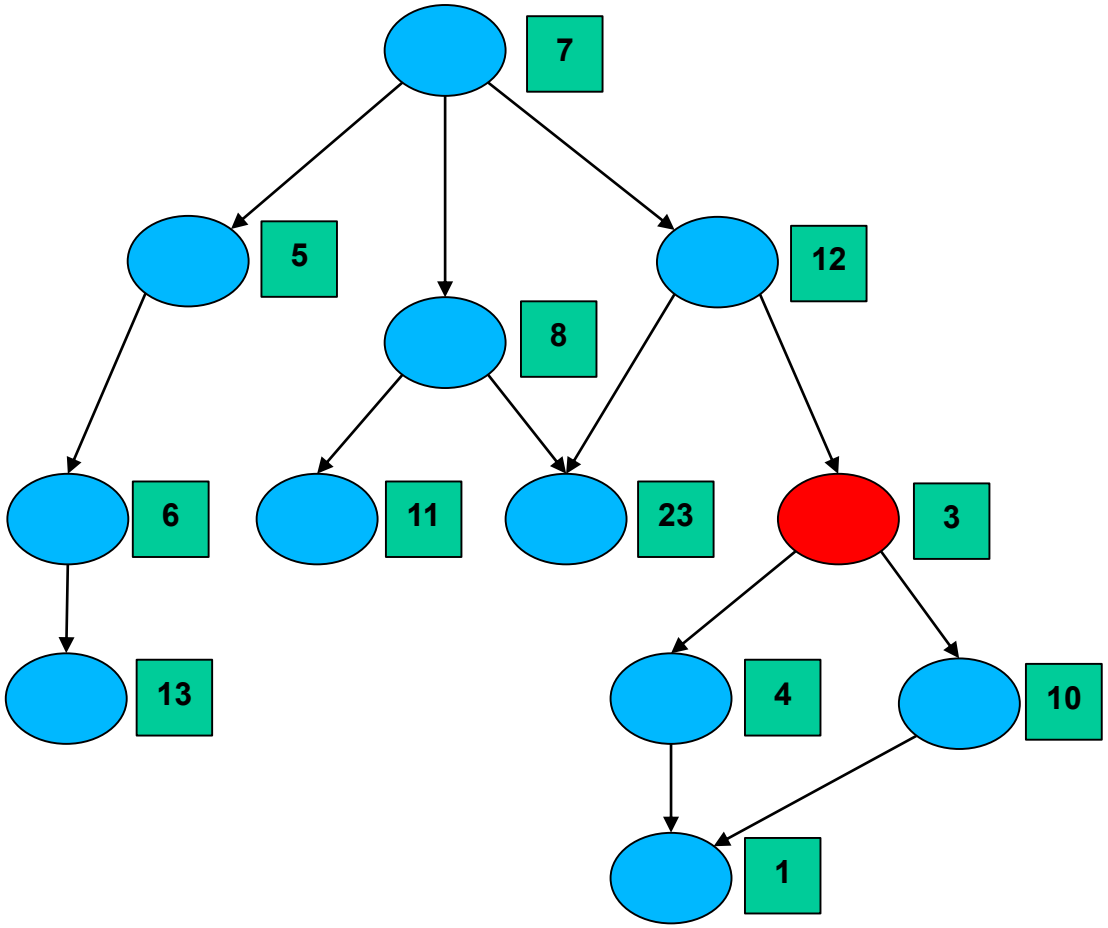
```
struct S {
  char a;
  int b;
  char c;
  short d;
};
```

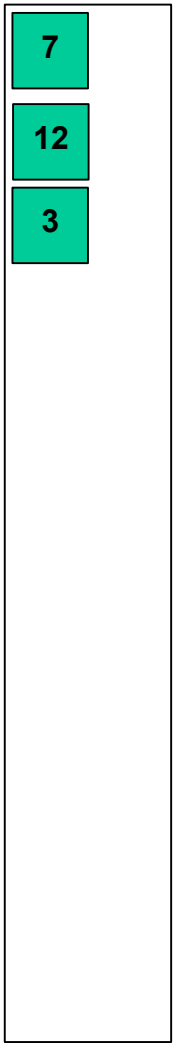
Меморија коју користи функција



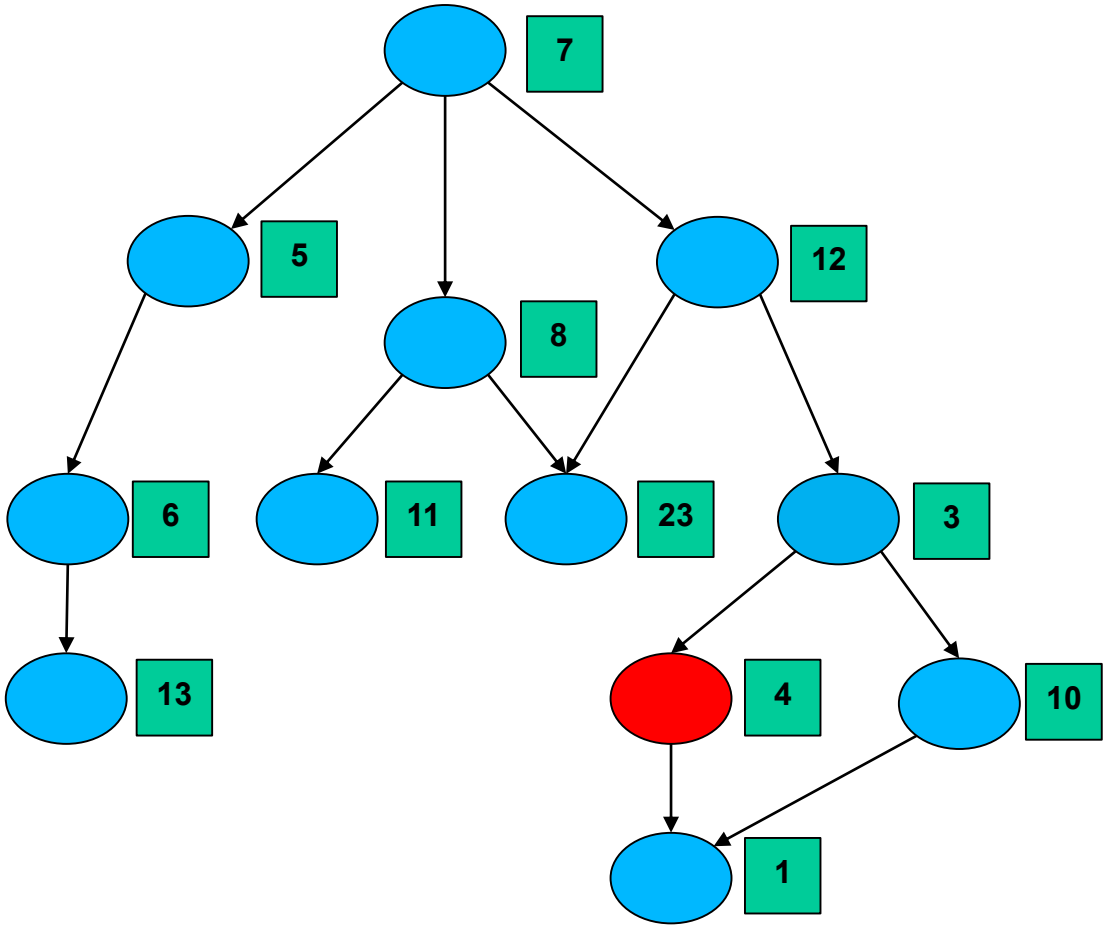
- Функцији треба меморија за:
  - Повратну адресу
  - Локалне променљиве
  - Привремено складиштење вредности
- Због компајлерских оптимизација не можемо гледањем кода бити у потпуности сигурни колико меморије треба којој функцији
- Али можемо имати оквирни осећај
- Прецизне податке добављамо од компајлера (или неког другог алата)

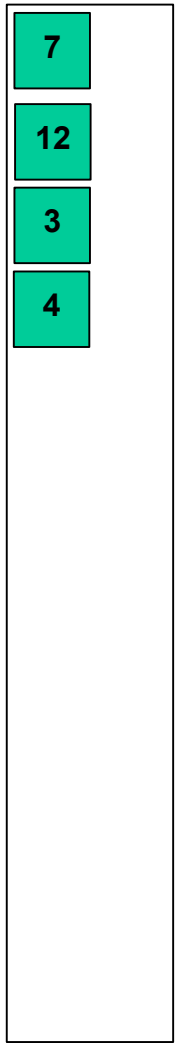
Граф позива функција и стек





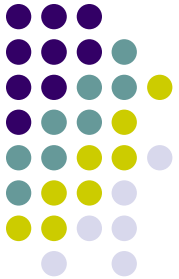
Граф позива функција и стек





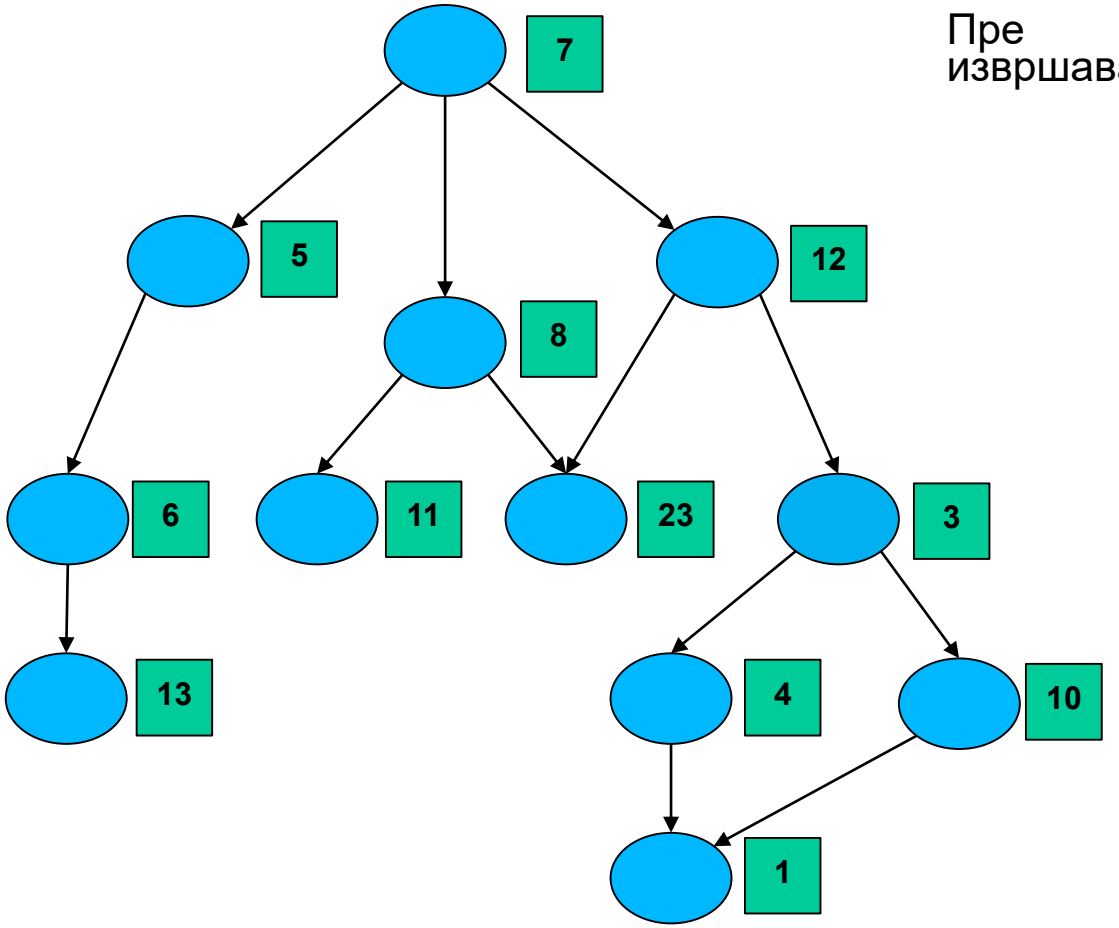
NIT

Колика нам величина стека треба?



- Два приступа:
- Експериментални
- Аналитички
- Плус, организација без стека

Експериментални



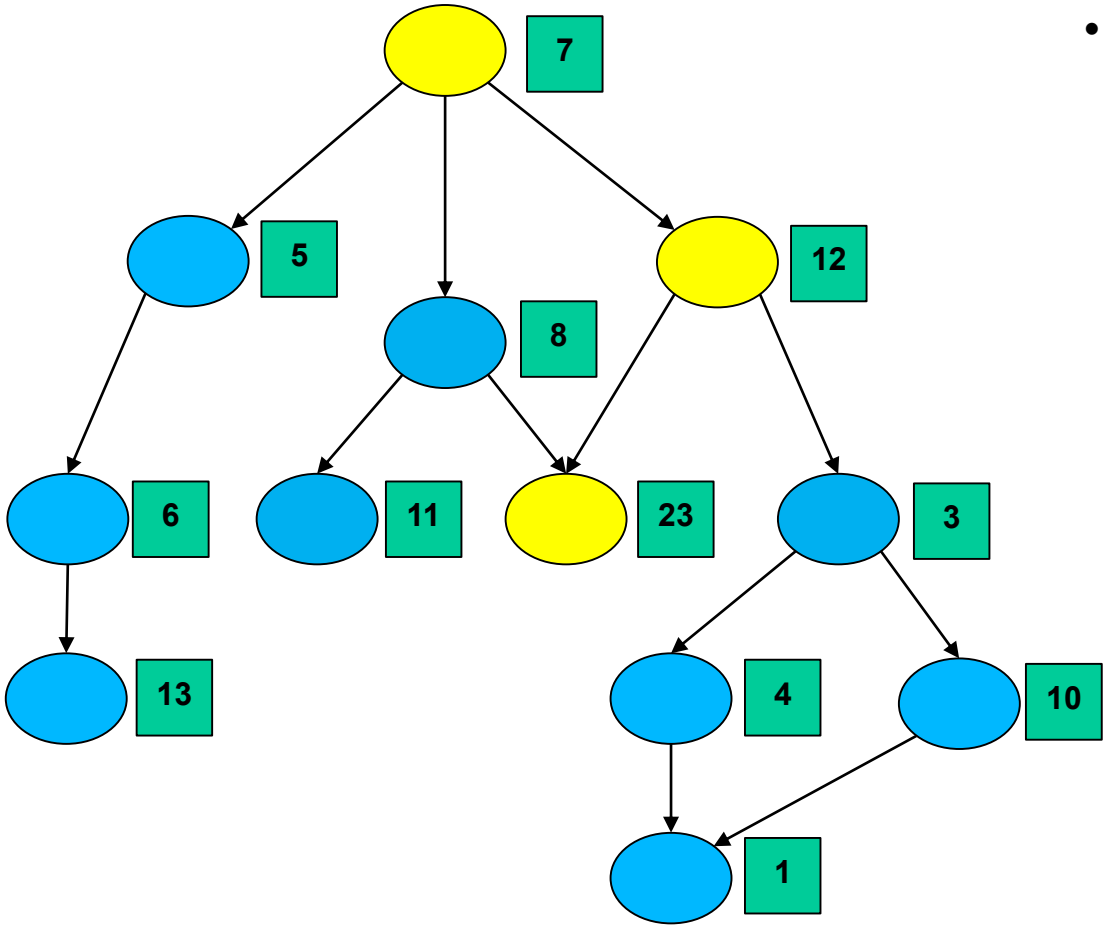
Пре извршавања

```
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
```

```
0xF123
0x5235
0x2134
0xA43E
0x35B4
0x2232
0x7CCE
0xFEA3
0x7593
0xA234
0xCF56
0xDD37
0x88A9
0xCF2C
0x9146
0xBA34
0x7593
0xA234
0xCF56
0xDD37
0x7593
0xA234
0xCF56
0xDD37
0x8821
0x32CD
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
0xBABA
```

После извршавања

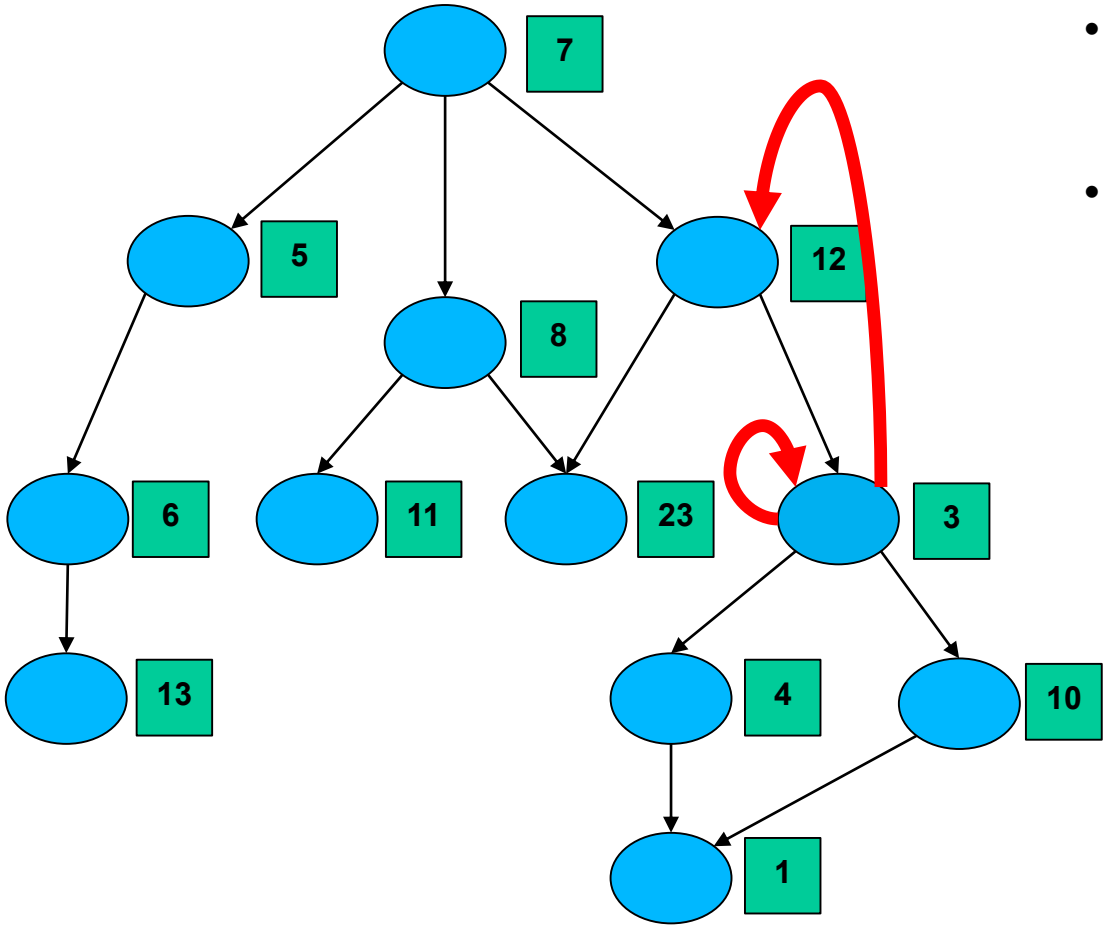
Аналитички



- Потребно познавање целог графа позива – што је тешко осигурати

7
12
23

Аналитички

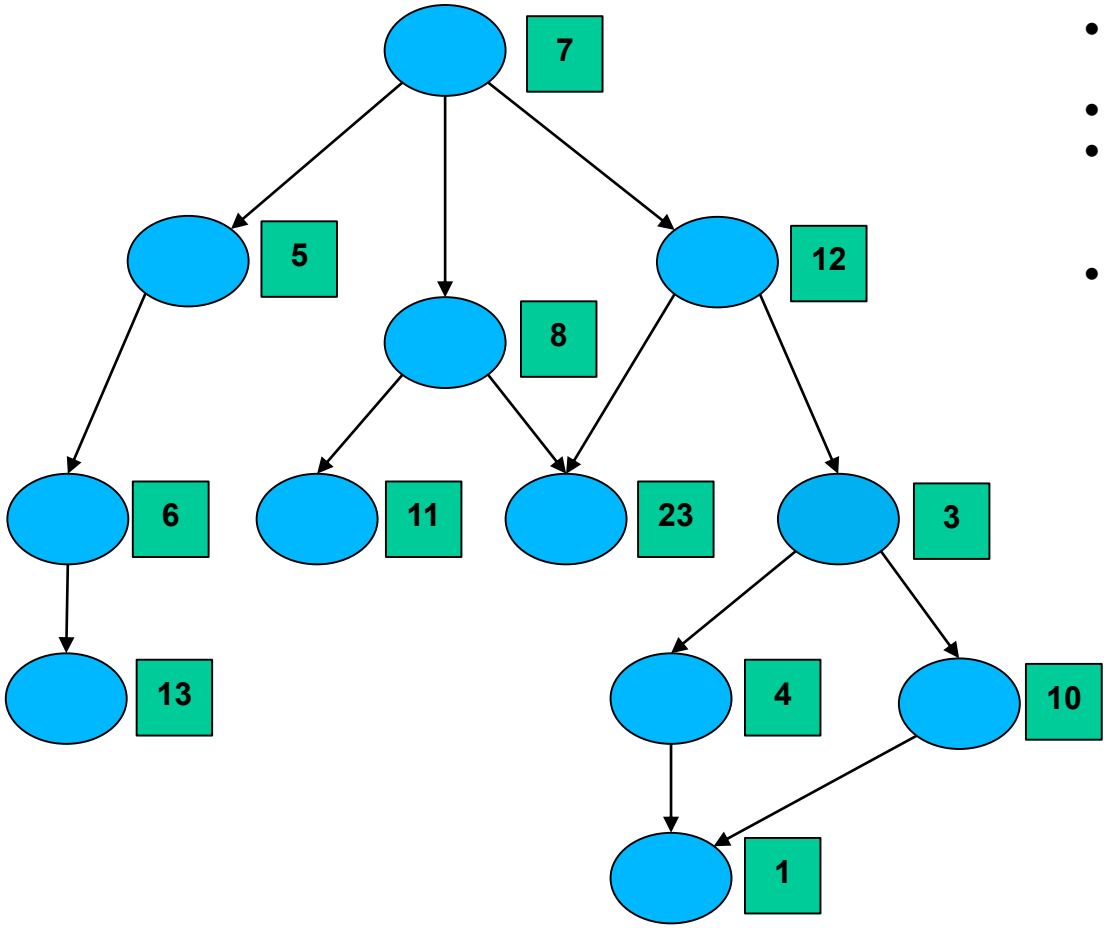


- Потребно познавање целог графа позива – што је тешко осигурати
- Додатни проблеми се јављају уколико постоји:
  - Рекурзија
  - Позиви преко показивача
  - Прекидне рутине, више нити итд.

```
void (*p)(int x);
...
p(17);
```

? ☹

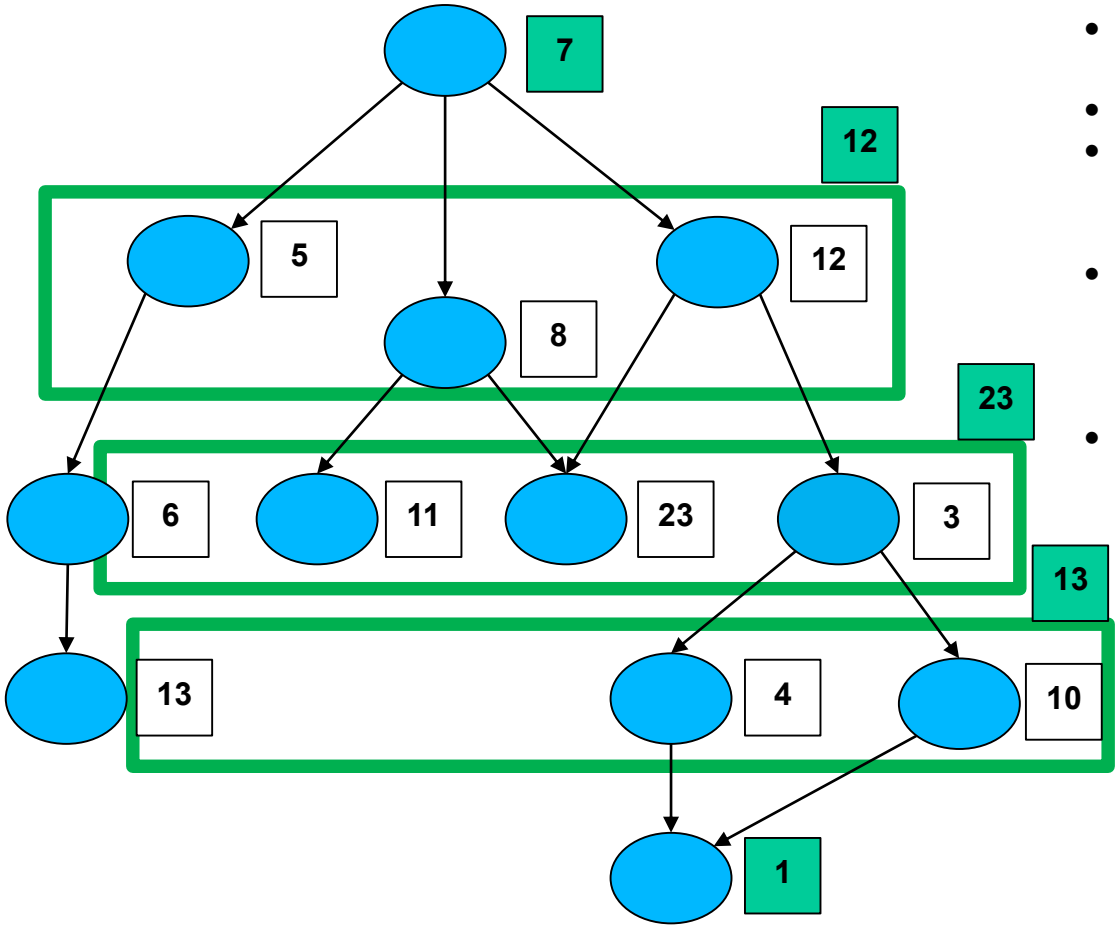
Организација без стека



- Свака функција има своју меморију
- Рекурзија није дозвољена
- Позив преко показивача је мало проблематичнији

- Али, сигурни смо да нам меморије неће понестати

Организација без стека

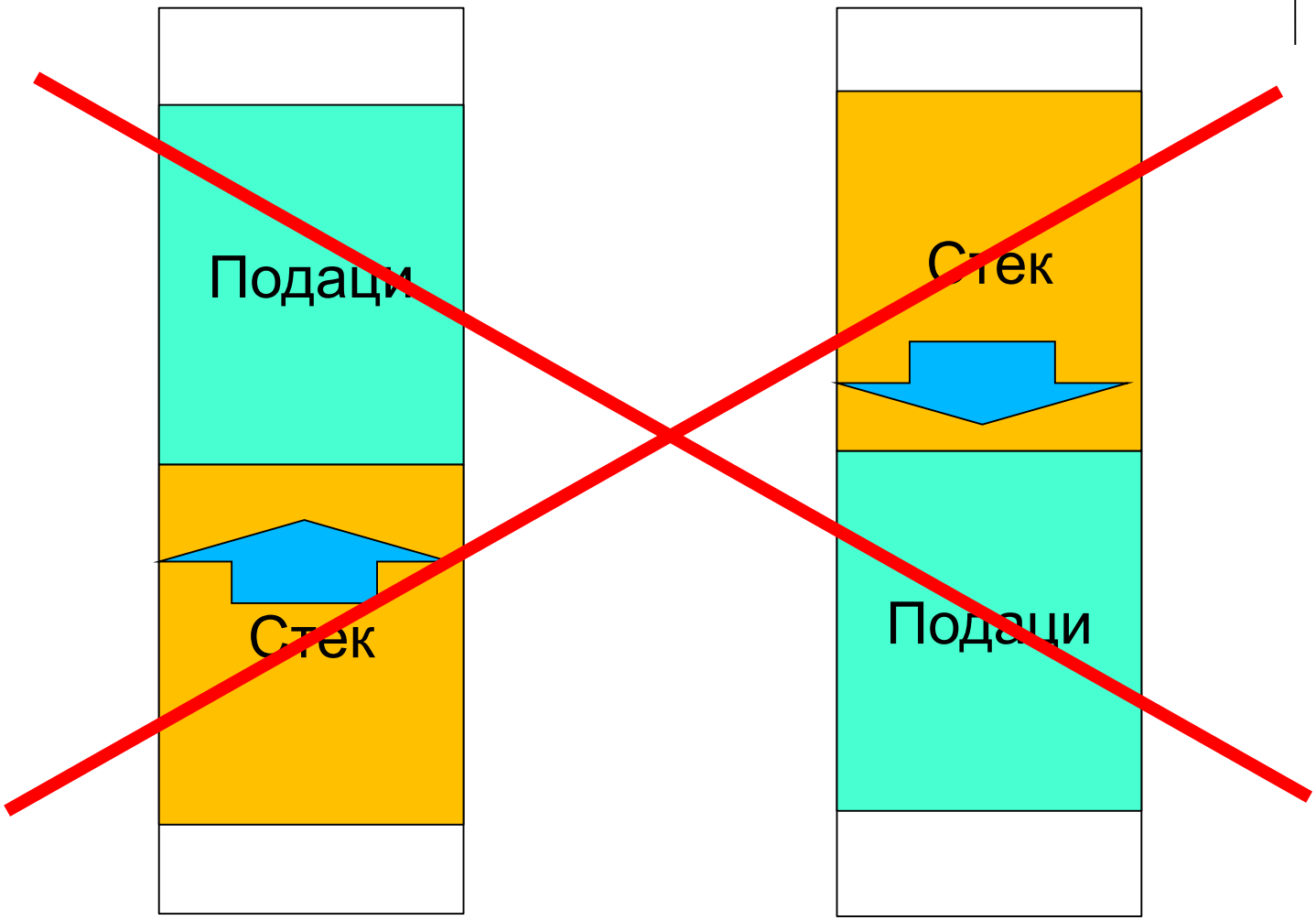


- Свака функција има своју меморију
- Рекурзија није дозвољена
- Позив преко показивача је мало проблематичнија

- Али, сигурни смо да нам меморије неће понестати

- Може се оптимизовати на основу познавања графа позива тако што ће функције које сигурно не могу бити у истој линији позива делити меморију.

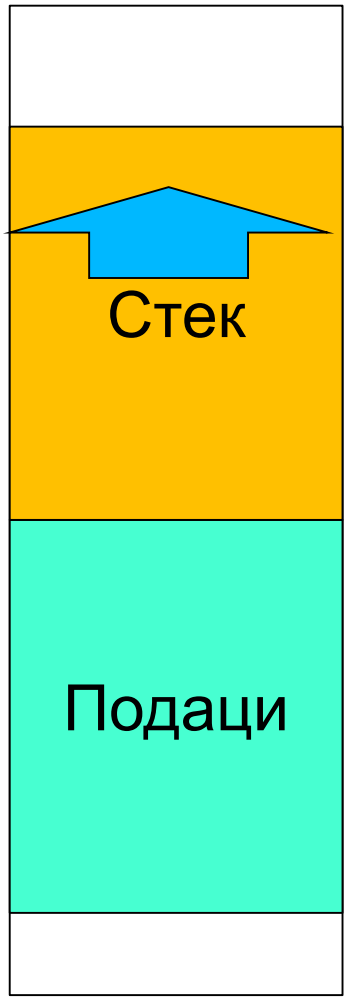
Где ставити стек?



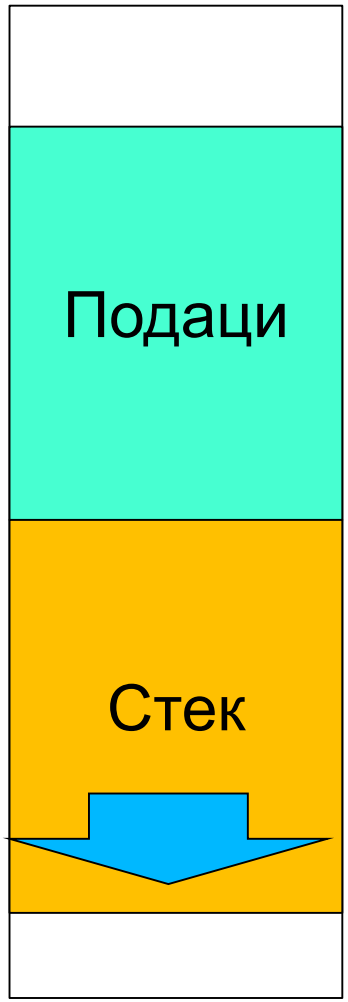
Стек расте на горе

Стек расте на доле

Где ставити стек?



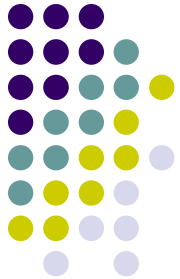
Стек расте на горе



Стек расте на доле

NIT

Рекурзија



- Рекурзија је тесно повезна са стеком, а рад са стеком код неких наменских система није баш најефикаснији.
- Уједно, код неких система постоји физичка подршка за позиве функција и тада је дубина позива ограничена физичким фактором.
- И на крају, рекурзија је ретко потребна и ретко се сусреће у раду са наменским системима. У случају и да је потребна, обично је боље, а и безбедније, направитну програмску конструкцију стека и директно га контролисати.

Рекурзија и петља

```
void foo(int i)
{
    if (i < 10)
    {
        neki_kod(i);
        foo(i + 1);
    }
}
...
foo(0);
...
```

```
...
int i = 0;
while (i < 10)
{
    neki_kod(i);
    i += 1;
}
...
```

```
void foo(kont)
{
    if (uslov(kont))
    {
        neki_kod(kont);
        foo(new_kont);
    }
}
...
foo(init_kont);
...
```

```
...
kont = init_kont;
while (uslov(kont))
{
    neki_kod(kont);
    kont = new_kont;
}
...
```

```
void foo(kont)
{
    if (uslov(kont))
    {
        neki_kod(kont);
        foo(new_kont);
    }
}
...
foo(init_kont);
...
```

```
kont = init_kont;
while (uslov(kont))
{
    neki_kod(kont);
    push(kont);
    kont = new_kont;
}

while (stack != empty)
{
    pop(kont);
}
```

Рекурзија и петља

```
void foo(int i)
{
    if (i < 10)
    {
        foo(i + 1);
        neki_kod(i);
    }
}
...
foo(0);
...
```

```
...
int i = 10;
while (i > 0)
{
    i -= 1;
    neki_kod(i);
}
...
```

```
void foo(kont)
{
    if (uslov(kont))
    {
        foo(new_kont);
        neki_kod(kont);
    }
}
...
foo(init_kont);
...
```

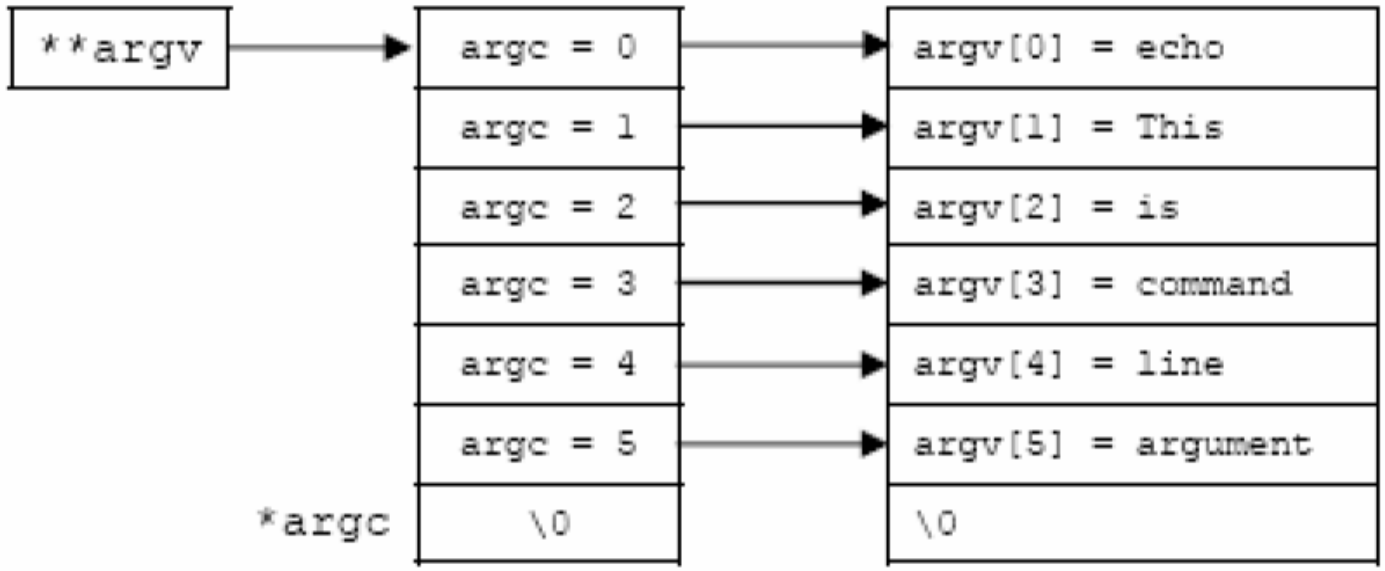
```
kont = init_kont;
while (uslov(kont))
{
    push(kont);
    kont = new_kont;
}

while (stack != empty)
{
    pop(kont);
    neki_kod(kont);
}
```

main функција

- Почетна функција
- Декларација: int main(int argc, char** argv)

void main()
void main(int argc, char** argv)
float main(long djura) ?!?!?!?!?!?
...

```
C:\>echo This is command line argument
This is command line argument
```



NIT

Живот у свету без прототипова

- Прототип је потпуна декларација функције (и никакава друга декларација не би требала да се користи)
- Када изоставимо прототип, или чак целу декларацију, компајлер почне да прави претпоставке, а машине које претпостављају су врло опасна ствар.
- Обично постоји опција у компајлеру да не прихвата позив функције ако њен прототип није наведе. Али пазите - неки пут је та опција подразумевано искључена.

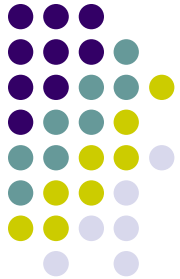
```
res = foo(3.7);
```

Компајлер сматра да је повратна вредност функције int. Ако ваља - ваља, ако не - пробај да провалиш шта је проблем.
Неки компајлери сматрају и да је параметар типа int, али неки ће бити „паметни” па ће на основу типа прослеђеног стварног параметра одредити тип.

```
double foo(double);

res = foo(3.7);
```

NIT

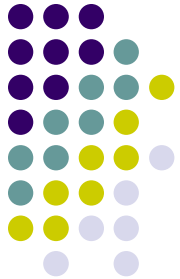
Позивна конвенција 1/2



- Позивна конвенција одређује спрегу између позване и позивајуће функције.
- Дефинише две ствари:
  - Начин преношења стварних параметара и повратне вредности
    - У које ресурсе се смештају
      - Ако у регистре: веза између редног броја параметра и конкретног регистра
      - Ако на стек или у меморију: веза између редног броја параметра и редоследа на стеку
    - Договор где се који параметар смешта зависи од његовог редног броја и типа
  - Ко које ресурсе сме да дира (мења)
    - Који регистри ће гарантовано имати исту вредност пре и после позива, а који не
    - Ко ће заузимати и ослобађати меморију на стеку за параметре и повратну вредност
- Позивна конвенција је део апликационе бинарне спреге - ABI (Application Binary Interface)
- ABI дефинише: величину типова и поравнање у меморији, позивну конвенцију, ствари везане за системске позиве, а некада чак и бинарни формат објектне датотеке

NIT

Позивна конвенција 2/2



- Позивна конвенција је везана за конкретну платформу (комбинација процесора, оперативног система и делимично компајлера)
- На истој платформи може постојати више позивних конвенција
- Разлози:
  - различити програмски језици
  - различити компајлери
  - компајлерске оптимизације
  - сврха кода, итд.

- Различите позивне конвенције доводе до проблема у случају комбинације кода насталог на различите начине.
- Да би се спој могао направити ABI мора бити задовољен, а најважнија је позивна конвенција
- Најчешћи спој са библиотекама, или комбинација асемблерког и Це кода (из Цеа се зове функција написана у асемблеру или обрнуто)

Пример

a.c

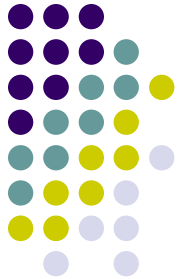
```
int foo(int x) {
  ...
}

...

    a = foo(5);

...

  bar(foo(x));

...
```

a.c

```
static int foo(int x) {
  ...
}

...

    a = foo(5);

...

  bar(foo(x));

...
```

NIT

Мешање Цеа и асемблера



- Разлози за мешање:
  - Нешто је већ написано у асемблеру
  - За одређене ствари компајлер није довољно ефикасан
  - Приступ одређеним могућностима физичке архитектуре којима се из Цеа не може приступити

- Два начина мешања:
  - Писањем асемблера у посебној датотеци
    - Спрега је искључиво на нивоу позива функција
    - Мора се поштовати позивна конвенција
  - Коришћењем уграђеног асемблера
    - Асемблерски код у истој датотеци са Це кодом
    - Мора бити подржано од стране компајлера
    - Разни механизми подршке пошто није део стандарда
    - Асемблер може бити коришћен само у телу функције

Позив цеовске функције из асемблера

- Асемблерски код мора поштовати позивну конвенцију
- Осим тога морамо познавати како Це **декорише** називе
  Најчешће то ради додавањем _ испред имена
- Пример позива функције printf из асемблера

```
    global   _main
    extern   _printf

    section .data
    text     db       "Hello World!", 10, 0
    strformat db      "%s", 0

    section .code
    _main
      push     dword text
      push     dword strformat
      call     _printf
      add      esp, 8
      ret
```

- _main симбол мора бити декларисан као јаван
- _printf симбол је декларисан као спољан

Позив асемблерске функције из Цеа

- Позивна конвенција мора бити поштована.
- Дакле, асемблерски код мора бити написан тако да поштује конвецнију коју користи компајлер
- Пример:

```
int sum(int a1, int a2);

int a1, a2, x;
x = sum(a1, a2);
```

```
_sum
  push ebp            ;save bp
  mov ebp, esp        ;new frame
  mov eax, [ebp+8]    ;take 1. arg
  mov ecx, [ebp+12]   ;take 2. arg
  add eax, ecx        ;
  pop ebp             ;restore bp
  ret                 ;return
```

- Повратна вредност по конвенцији је у EAX регистру
- По коришћеној позивној конвецији

Уграђени асемблер 1/2

- GCC нуди следећи механизам мешања асемблера и Цеа директно у коду.
- Постоје и други механизми таквог мешања у неким другим компајлерима, али GCC-ов мехизам је постао де факто стандард.
- Синтакса GCC-овског асемблерског исказа:

```
asm(assembler template
    : output operands                   /* optional */
    : input operands                    /* optional */
    : list of clobbered registers       /* optional */
    );
```

- assembler template – знаковни низ са асемблерским кодом
- output operands – цеовски операнди који ће бити промењени када се асм код изврши
- input operands – цеовски операнди који чије вредности се користе при извршавању асм кода
- clobbered registers – регистри циљне платформе чије вредности асм код експлицитно мења

Уграђени асемблер 2/2

```
int func(int in)
{
int res;
asm ("movl %1, %%eax; \
      movl %%eax, %0;"
     :"=g"(res)          /* output */
     :"r"(in)            /* input */
     :"%eax"             /* clobbered register */
     );
return res;
}
```

- %n – референца на n-ти операнд из листи излазних и улазних операнада (први индекс је 0)
- %% – стварно стави %
- “xy”(c_expression) – connecting C variable with value used in inline assembler

a,b,c,d	eax, ebx, ecx, edx respectively
S, D	esi and edi respectively
I	constant value (0 to 31)
q	dynamically allocated register: eax, ebx, ecx, edx
r	dynamically allocated register: eax, ebx, ecx, edx, esi, edi
g	eax, ebx, ecx, edx or variable in memory
m	memory
=	will be used for storing data