Универзитет „Св. Кирил и Методиј“ во Скопје
ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

# Функции

## Структурно програмирање

ФИНКИ 2013

# Функции во C

- Програмите претставуваат комбинација од кориснички и библиотечни функции
  - стандардните библиотеки содржат голем број различни функции
  - функциите го прават програмирањето поедноставно
  - секоја функција извршува точно дефинирана работа, што е корисна за останатите делови на програмата или за други програми
    - остатокот од програмата и не мора „да знае" како се извршува задачата
  - користењето на функциите е слично со шефот што задава задача на вработените
    - секој работник ја добива информацијата, ја извршува задачата и ги враќа резултатите
    - криење на информации: шефот не мора да ги знае деталите како е извршена работата, него го интересираат резултатите

# Повик на функција

- користењето на функциите во програмите се нарекува повик на функција
  - при повикување на функција се обезбедуваат име и аргументи за функцијата (податоци)
- Формат за повикување на функциите

Име (листа на аргументи)

- Ако има повеќе аргументи – тие се одделуваат со запирки

  printf(“%f”, sqrt( 900.0 ) );

  Во наредбата се **повикува** функцијата sqrt, која ја пресметува и враќа вредноста на квадратен корен од **аргументот**
- Аргументите може да се
  - Константи
  - Променливи
  - Изрази

# Предности при користење на функции

- поедноставно одржување на програмата
- можност за користење претходно развиени функции
  - се користат постоечки функции како градбени блокови за новите програми
- апстракција – се кријат внатрешните детали
- се одбегнува повторување на еден ист код на различни места на програмата

# Дефинирање на функции

- Формат

```
vidVratenaVrednost imeFunkcija(vid Pr1, vid Pr2, ...)
{
        /* telo na funkcijata */
}
```

- imeFunkcija – име на програмски елемент

# Дефинирање на функции

- vidVratenaVrednost – тип на вредност што функцијата ја враќа како резултат
  - може да биде:
    - стандарден податочен тип (int, float, double итн.)
    - сложен тип
    - кориснички дефиниран тип (набројувачки, ...)
    - void - функцијата не враќа вредност
  - ако не е дефиниран, се смета дека типот е **`int`**
    - пример:

      ```
      int praviNesto(){...}
      ```

      е исто со

      ```
      praviNesto(){...}
      ```

Препорака: За да се одбегне конфузијата, секогаш декларирајте го типот.

# Дефинирање на функции

- vid Pr1, vid Pr2, ...- листа на параметри (формални параметри, формални аргументи) одделени меѓусебе со запирки
  - во оваа листа се декларираат променливи
  - функцијата може и да не содржи формални параметри (листата е празна)
  - променливите декларирани во листата на параметри важат и може да се употребат само во телото на функцијата
  - теориски бројот на формални параметри не е ограничен, но во практиката е ограничен

    Препорака: одбегнувајте го користењето на голем број формални параметри

- во K&R стилот заглавјето на функцијата се дефинира

```
vidVratenaVrednost  imeFunkcija(Pr1, Pr2, ... )
   vid Pr1;
   vid Pr2; ...
{   telo na funkcijata   }
```

# Дефинирање на функции

- Тело на функцијата
  - ги содржи истите елементи како и main() функцијата
    - декларација на променливи
      - променливите декларирани за функцијата се локални за истата
    - наредби

# Пример

```
/* Programa za presmetuvanje faktoriel */
#include <stdio.h>
void calc_factoriel( int n ) {
    int  i, fact_num = 1;
    for( i = 1; i <= n; ++i ) fact_num *= i;
    printf("Faktoriel od %d iznesuva %d\n", n, fact_num);
}
int main() {
    int number = 0;
    printf("Vnesi broj\n");
    scanf("%d", &number );
    calc_factoriel( number );
    return 0;
}
```

```
Vnesi broj
3
Faktoriel od 3 iznesuva 6
```

# Напуштање на **функцијата**

- Ако во заглавјето на функцијата е дефинирано дека истата не враќа вредност тогаш од функцијата се излегува кога ќе се стигне до изразот

```
return;
```

или до

```
}
```

```
void print_answer(int answer) {
        if (answer < 0) {
                printf("Answer corrupt\n");
                return;
        }
        printf("The answer is %d\n", answer);
}
```

# Напуштање на функцијата

- Ако во заглавјето на функцијата е дефинирано дека истата враќа вредност тогаш од функцијата се излегува кога ќе се стигне до изразот

**return** *expression;*

Типот на оваа вредност мора да соодветствува на типот на вредноста што е декларирано дека ќе ја врати функцијата

```
float add_numbers( float n1, float n2 ) {
      return n1+n2;
      /* ispravno, zbirot e od vidot float */
}
```

# Вредност што ја враќа функцијата

```
int add_numbers( float n1, float n2 ) {
      return 6.0;    /* ne e vo red */
}
```

```
float add_numbers( float n1, float n2 ) {
      return 6.0; /* ispravno,
                     vraka realna vrednost */
}
```

# Вредност што ја враќа функцијата

- Можно е една функција да има повеќе наредби за враќање на вредност:

```
int validate_input( char command ) {
    switch( command ) {
        case '+' : case '-' : return 1;
        case '*' : case '/' : return 2;
        default  : return 0;
    }
}
```

# Дефинирање и користење на функции

```
#include <stdio.h>
float triangle(float width, float height) {
    float area;
    area = width * height / 2.0;
    return (area);
}
int main() {
    float size;
    printf("Triangle #1 %f\n", size = triangle(1.3, 8.3));
    printf("Triangle #2 %f\n", size = triangle(4.8, 9.8));
    printf("Triangle #3 %f\n", size = triangle(1.2, 2.0));
    return (0);
}
```

Изразот `size = triangle(1.3, 8.3)` е повик кон функцијата. На променливата **`width`** ќе и биде придружена вредност 1.3, а на **`height`** 8.3. Функцијата враќа вредност 5.4, што се сместува во променливата **`size`**.

# Пример за погрешно користење на функции

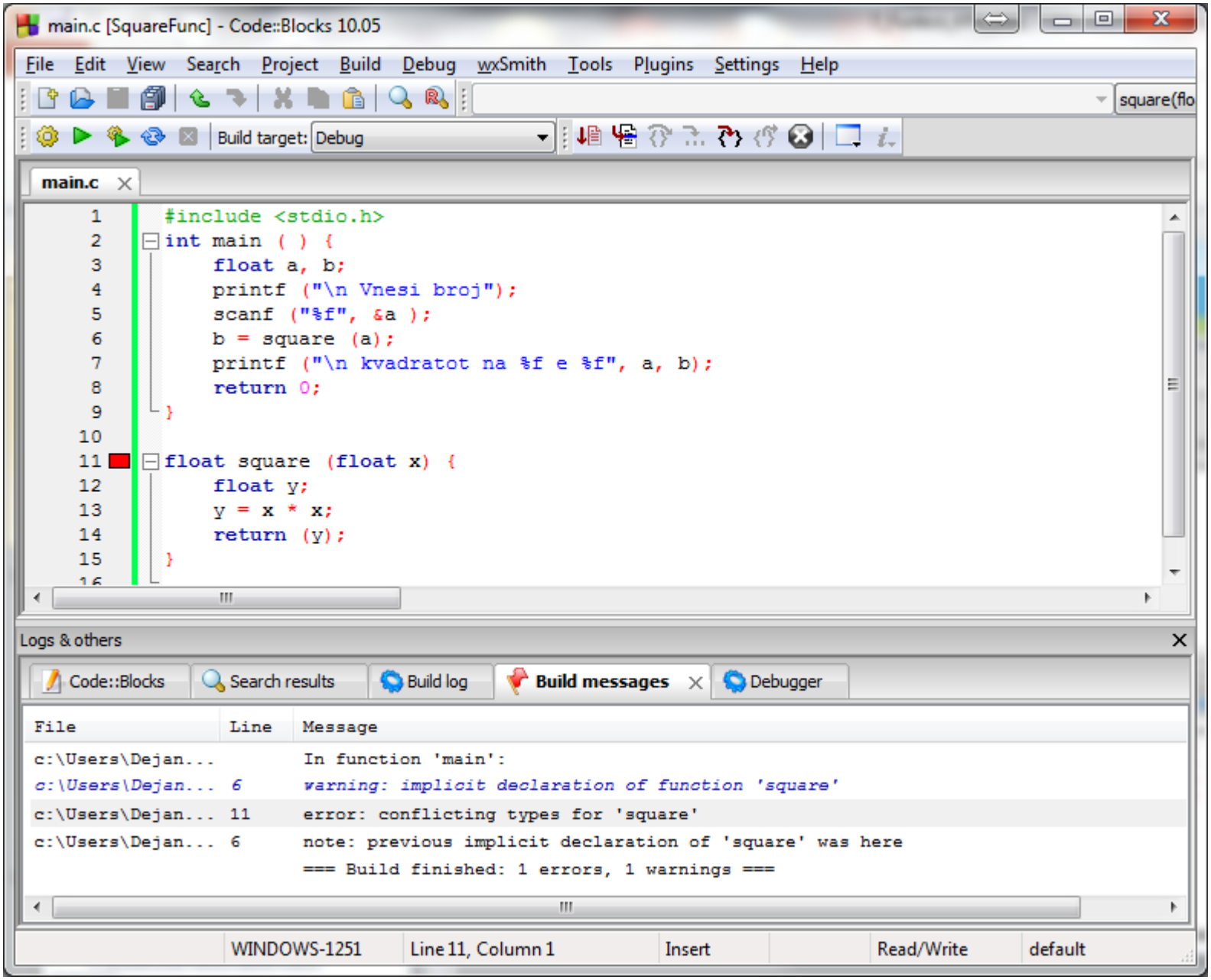
```
square (float x) {
    float y;
    y = x * x;
    return (y);
}
int main ( ) {
    float a, b;
    printf("\n Vnesi broj");
    scanf("%f", &a );
    b = square (a);
    printf ("\n kvadratot na %f e %f", a, b);
return 0;
}
```

**За примерот компајлерот подразбира дека функцијата враќа целобројна вредност!!!**

Vnesi broj 2.5
Kvadratot na 2.5 e 6.00

# Што ќе се случи сега?

```
int main ( ) {
    float a, b;
    printf ("\n Vnesi broj");
    scanf ("%f", &a );
    b = square (a);
    printf ("\n kvadratot na %f e %f", a, b);
    return 0;
}
float square (float x) {
    float y;
    y = x * x;
    return (y);
}
```


main.c [SquareFunc] - Code::Blocks 10.05
File Edit View Search Project Build Debug wxSmith Tools Plugins Settings Help
Build target: Debug
main.c
1 #include <stdio.h>
2 int main ( ) {
3 float a, b;
4 printf ("\n Vnesi broj");
5 scanf ("%f", &a );
6 b = square (a);
7 printf ("\n kvadratot na %f e %f", a, b);
8 return 0;
9 }
10
11 float square (float x) {
12 float y;
13 y = x * x;
14 return (y);
15 }
Logs & others
Code::Blocks
Search results
Build log
Build messages
Debugger
File Line Message
c:\Users\Dejan... In function 'main':
c:\Users\Dejan... 6 warning: implicit declaration of function 'square'
c:\Users\Dejan... 11 error: conflicting types for 'square'
c:\Users\Dejan... 6 note: previous implicit declaration of 'square' was here
=== Build finished: 1 errors, 1 warnings ===
WINDOWS-1251
Line 11, Column 1
Insert
Read/Write
default

# Функциски прототипови

- Ако функција се користи пред да биде дефинирана, потребно е да се декларира на идентичен начин како и секоја променлива.
- На тој начин се информира компајлерот и се овозможува проверка на типот на вредноста што се враќа и листата на параметри при користење на функцијата.
- формат

***vid imeFunkcija(listaParametri);***

*imeFunkcija* е името на функцијата

*listaParametri* – податоци што се проследуваат во функцијата

*vid* – податочен тип на вредноста што ја враќа функцијата (default **`int`**)

прототипот завршува со ; и со тоа се означува дека станува збор за декларација на функција, а не нејзина дефиниција.

# Функциски прототипови

- Ако функција се користи пред да биде дефинирана, потребно е да се декларира на идентичен начин како и секоја променлива.
- На тој начин се информира компајлерот и се овозможува проверка на типот на вредноста што се враќа и листата на параметри при користење на функцијата.
- формат

***vid imeFunkcija(listaParametri);***

*imeFunkcija* е името на функцијата

*listaParametri* – податоци што се проследуваат во функцијата

*vid* – податочен тип на вредноста што ја враќа функцијата (default **`int`**)

прототипот завршува со ; и со тоа се означува дека станува збор за декларација на функција, а не нејзина дефиниција.

# Функциски прототипови - примери

# Функциски прототипови - примери

```
float triangle (float width, float height);
```

# Функциски прототипови - примери

```
float triangle (float width, float height);
```

- се проследуваат две реални вредности, враќа реална вредност

# Функциски прототипови - примери

```
float triangle (float width, float height);
```

- се проследуваат две реални вредности, враќа реална вредност

```
int prim(void);
```

# Функциски прототипови - примери

```
float triangle (float width, float height);
```

- се проследуваат две реални вредности, враќа реална вредност

```
int prim(void);
```

- не се проследуваат вредности, враќа целобројна вредност

# Функциски прототипови - примери

```
float triangle (float width, float height);
```
- се проследуваат две реални вредности, враќа реална вредност

```
int prim(void);
```
- не се проследуваат вредности, враќа целобројна вредност

```
void prim1(void);
```

.
.
.

.
.
.
.

# Функциски прототипови - примери

```
float triangle (float width, float height);
```
- се проследуваат две реални вредности, враќа реална вредност

```
int prim(void);
```
- не се проследуваат вредности, враќа целобројна вредност

```
void prim1(void);
```
- не се проследуваат вредности, не враќа вредност

.

.

.

.

.

.

# Функциски прототипови - примери

```
float triangle (float width, float height);
```
- се проследуваат две реални вредности, враќа реална вредност

```
int prim(void);
```
- не се проследуваат вредности, враќа целобројна вредност

```
void prim1(void);
```
- не се проследуваат вредности, не враќа вредност

```
void prim(int, float);
```

.

.

.

.

.

# Функциски прототипови - примери

**`float triangle (float width, float height);`**

- се проследуваат две реални вредности, враќа реална вредност

**`int prim(void);`**

- не се проследуваат вредности, враќа целобројна вредност

**`void prim1(void);`**

- не се проследуваат вредности, не враќа вредност

**`void prim(int, float);`**

- се проследуваат една целобројна и една реална вредност, не враќа вредност

.

.

.

.

# Функциски прототипови - примери

**`float triangle (float width, float height);`**

- се проследуваат две реални вредности, враќа реална вредност

**`int prim(void);`**

- не се проследуваат вредности, враќа целобројна вредност

**`void prim1(void);`**

- не се проследуваат вредности, не враќа вредност

**`void prim(int, float);`**

- се проследуваат една целобројна и една реална вредност, не враќа вредност

**`float triangle(float, float);`**

.

.

.

# Функциски прототипови - примери

**`float triangle (float width, float height);`**

- се проследуваат две реални вредности, враќа реална вредност

**`int prim(void);`**

- не се проследуваат вредности, враќа целобројна вредност

**`void prim1(void);`**

- не се проследуваат вредности, не враќа вредност

**`void prim(int, float);`**

- се проследуваат една целобројна и една реална вредност, не враќа вредност

**`float triangle(float, float);`**

- се проследуваат две реални вредности, враќа реална вредност

.

.

# Функциски прототипови - примери

```
float triangle (float width, float height);
```
- се проследуваат две реални вредности, враќа реална вредност

```
int prim(void);
```
- не се проследуваат вредности, враќа целобројна вредност

```
void prim1(void);
```
- не се проследуваат вредности, не враќа вредност

```
void prim(int, float);
```
- се проследуваат една целобројна и една реална вредност, не враќа вредност

```
float triangle(float, float);
```
- се проследуваат две реални вредности, враќа реална вредност

```
int maximum( int, int, int );
```

# Функциски прототипови - примери

**`float triangle (float width, float height);`**

- се проследуваат две реални вредности, враќа реална вредност

**`int prim(void);`**

- не се проследуваат вредности, враќа целобројна вредност

**`void prim1(void);`**

- не се проследуваат вредности, не враќа вредност

**`void prim(int, float);`**

- се проследуваат една целобројна и една реална вредност, не враќа вредност

**`float triangle(float, float);`**

- се проследуваат две реални вредности, враќа реална вредност

**`int maximum( int, int, int );`**

- се проследуваат три целобројни вредности, враќа целобројна вредност

# Функциски прототипови - пример

```
int main ( ) {
      float square(float);
      float a, b;
      printf("\n Vnesi broj ");
      scanf("%f", &a);
      b = square(a);
      printf ("\n kvadratot na %f e %f", a, b);
      return 0;
}
float square (float x) {
      float y;
      y = x * x;
      return (y);
}
```

```
Vnesi broj 2.5
kvadratot na 2.5 e 6.2500000
```

# Функциски прототипови - пример

```
void print_message( void );
int main() {
 print_message();
 return 0;
}
void print_message( void ) {
 printf("Ova e funkcija narecena print_message\n");
}
```

```
Ova e funkcija narecena print_message
```

# Функциски прототип- пример

```
/* Primer: Opredeluvanje na najgolemiot od tri broja */
#include <stdio.h>
int maximum( int, int, int ); /* funkciski prototip */
int main(){
   int a, b, c;
   printf( "Vnesi tri celi broevi: " );
   scanf( "%d%d%d", &a, &b, &c );
   printf( "Najgolemiot e: %d\n", maximum( a, b, c ));
   return 0;
}
/* definicija na funkcijata */
int maximum( int x, int y, int z ){
   int max = x;
   if ( y > max ) max = y;
   if ( z > max ) max = z;
   return max;
}
```

```
Vnesi tri celi broevi: 22 85 17
Najgolemiot e: 85
```

# Пренос на вредност

Формалните аргументи ги прифаќаат вредностите кои се задаваат при повикувањето на функцијата (напишани со задебелени букви во примерот)

Кај прототипот

```
double stipendija(int osnovna_stipendija, float prosecna_ocena);
```

Кај дефинирањето

```
double stipendija(int osnovna_stipendija, float prosecna_ocena) {
  float vkupno;
  vkupno = osnovna_stipendija + 200*(prosecna_ocena-6);
  return vkupno;
  }
```

# Пренос на вредност

- Се наведуваат при ПОВИКУВАЊЕТО на функцијата (задебелни букви во примерот)

iznos=stipendija(**osnstip, prosoc**);

- При повикување на функцијата...

  ... вредностите на вистинските аргументи се пренесуваат во формалните аргументи.

# Пренос на вредност...

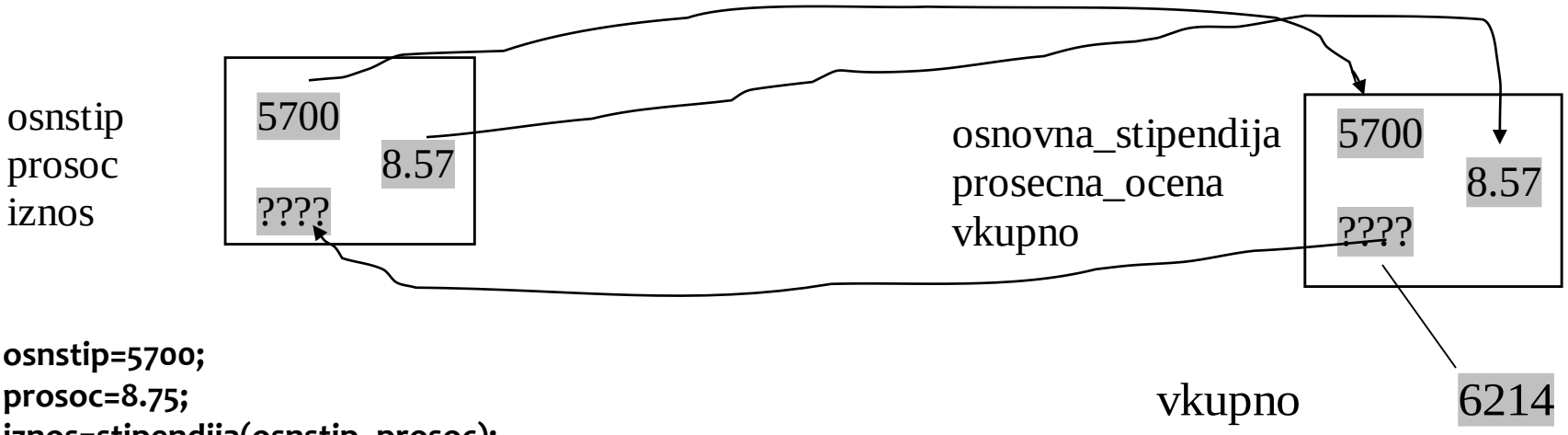
- при секој повик на функцијата се проверува бројот и типот на аргументите за време на преведувањето
- ако бројот на вистинските аргументи е различен од бројот на формалните, се јавува порака за грешка.
- ако типот на формалнте аргументи е различен од типот на вистинските аргументи соодветно,
  - прво компајлерот се обидува да го „преведе" едниот тип во друг (на пример double во int). Притоа, може да се изврши повикот, но резултатот да биде погрешен.
  - Ако не успее „преведувањето" на типот, тогаш компајлерот јавува порака за грешка.

# Пренос на вредност

- За секој *формален аргумент при пренесување на вредност*
  - се креира нова променлива,
  - се иницијализира променливата со вредноста на вистинскиот аргумент,
  - **измените во оваа променлива не влијаат на вистинскиот аргумент.**

# Пренос на вредност...

- Формалните аргументи **osnovna_stipendija, prosecna_ocena** се третираат како обични целобројна и реална променлива




**osnstip=5700;**
**prosoc=8.75;**
**iznos=stipendija(osnstip, prosoc);**

```
double stipendija(int osnovna_stipendija, float prosecna_ocena){
        float vkupno;
        vkupno = osnovna_stipendija + 200*(prosecna_ocena-6);
        return vkupno;
}
```

# Пренос на вредности во функцијата

- Со овој метод сите промени што ќе ги претрпи вредноста на формалниот аргумент при извршувањето на функцијата немаат никаков ефект на вредноста на аргументот со кој е повикана функцијата.
- со тоа се одбегнуваат несаканите ефекти поради промена на вредноста на формалниот аргумент

# Пренос на вредности во функцијата

пример:

```
int swapv (int x, int y) {
    int t;
    t = x; x = y; y = t;
    printf ("\n x = %d y = %d", x, y); return 0;
}
int main ( ) {
    int a = 10, b =20;
    printf ("\n a = %d b = %d", a, b);
    swapv (a, b);
    printf ("\n a = %d b = %d", a, b);
}
```

```
a = 10 b =20
x = 20 y =10
a = 10 b =20
```

# Области на важење на променливите

- Променливите се разликуваат според типот, областа на важење и начинот на креирање
- Според областа на важење на променливите, тие грубо може да се поделат на:
  - Глобални променливи
  - Локални променливи

# Локални и глобални променливи

- **Локални променливи**
  - постојат само во рамките на функциите и блоковите наредби во кои се креирани.
  - непознати се за сите останати функции или блокови наредби.
  - се уништуваат кога ќе се напушти функцијата или блокот наредби во кои се креирани.
  - повторно се креираат при секој следен повик на функцијата или блокот наредби.
- **Глобални променливи**
  - се дефинираат надвор од секоја функција и нив може да ги користи секоја функција во програмата.
  - променливите постојат се додека се извршува програмата

# Правила за подрачјето на важење на променливите

1. Променливата не може да се користи надвор од подрачјето на важење,
2. Глобалните променливи можат да се користат во целата програма,
3. Променливите декларирани во еден блок можат да се користат само во него,
4. Променлива декларирана во една функција не може да се користи во друга,
5. Променливата може да биде скриена во некој дел од нејзиното подрачје на важење,
6. Не може две различни променливи со исто име да имаат исто подрачје на важење.

# Пример - глобални променливи

```
#include <stdio.h>
int add_numbers( void );
int value1, value2, value3;
int add_numbers( void ) {
   auto int result;
   result = value1 + value2 + value3;
   return result;
}
int main() {
   auto int result;
   value1 = 10;
   value2 = 20;
   value3 = 30;
   result = add_numbers();
   printf("%d + %d + %d = %d\n", value1, value2, value3,
result);
   return 0;
}
```

```
10 + 20 + 30 = 60
```

# Дефинирање на глобални променливи

Областа на важење на глобалните променливи може да се ограничи со внимателно поставување на декларациите на променливите во датотеката.
Пример

```
#include <stdio.h>
void no_access( void );
void all_access( void );
int n2; /*n2 e poznata od ovaa tocka*/
void no_access( void ) {
   n1 = 10; /* netocno, n1 ne e deklarirana, i
   kompajlerot pri preveduvanje na programata vo
                ovaa tocka ke javi greska*/
   n2 = 5;    /* tocno */
}
int n1;         /* n1 e deklarirana i vazi od ovaa tocka
   vo ostatokot od programata */

void all_access( void ) {
   n1 = 10;    /* tocno */
   n2 = 3;     /* tocno */
}
```

# Покривање (криење) на глобални променливи

```c
#include <stdio.h>
void prikazi();
int x = 20;
int main() {
   printf("%d vo glavnata programa\n",
x);
   prikazi();
   return 0;
}
void prikazi() {
   printf("%d vo funkcija\n", x);
}
```

```
20 vo glavnata programa
20 vo funkcija
```

```c
#include <stdio.h>
void prikazi();
int x = 20;
int main() {
   int x =10;
   printf("%d vo glavnata programa\n",
x);
   prikazi();
   return 0;
}
void prikazi() {
   printf("%d vo funkcija\n", x);
}
```

```
10 vo glavnata programa
20 vo funkcija
```

# Локални променливи

Локални променливи може да се декларираат и за секој блок наредби и за нив важат истите правила - променливите се креираат при секое повторно влегување во блокот и важат за блокот и сите останати блокови вгнездени во истиот.

```
for(i=0; i<10; i++) {
      float x=0.0;
      . . .
}
```

**x** ќе важи само во рамките на **for** блокот и тоа при секое повторување на циклусот: на почетокот ќе се резервира простор за истата и ќе се иницијализира, а на крајот се уништува.

# Локални променливи

```
int main() {
   int x = 1;
   {
      int x = 2;
      {
         int x = 3; printf("x=%d", x);
      }
      printf("x=%d", x);
   }
   printf("x=%d\n", x);
   return 0;
}
```

```
x=3 x=2 x=1
```

# Класи променливи според начинот на креирање

- перманентни или привремени променливи
- перманентните се најчесто глобални променливи
  - се креираат и иницијализираат на почетокот на секоја програма и постојат до завршувањето на програмата
- локалните променливи се најчесто привремени променливи
  - се креираат и иницијализираат при извршувањето на блокот во кој се декларирани
  - се уништуваат при напуштање на блокот во кој се декларирани
- Следните клучни зборови се користат да се опише кога и каде променливите ќе се креираат и уништат

```
auto
static
extern
register
```

# Auto и static променливи

- Променливите декларирани како **`static`** се креираат и иницијализираат еднаш, на првиот повик на функцијата
  - При секој следен повик на функцијата не се креираат ниту се реиницијализираат статичките променливи.
  - Кога ќе заврши функцијата овие променливи се уште постојат, но не може да им се пристапи од другите функции.
- Променливите декларирани како **`auto`** се однесуваат сосема спротивно.
  - Тие се креираат при повик на функцијата и се уништуваат при напуштање на истата.
  - Привремените променливи се нарекуваат *automatic* променливи бидејќи просторот за истите се резервира автоматски.
  - Квалификаторот **`auto`** може да се користи да се означат овој тип на променливи иако тоа во практиката многу ретко се случува.

# Auto и static променливи

```
#include <stdio.h>
void demo( void );
void demo( void ) {
    auto int avar = 0;
    static int svar = 0;
    printf("auto = %d, static = %d\n", avar, svar);
    ++avar; ++svar;
}
int main() {
    int i=0;
    while( i < 3 ) { demo(); i++; }
    return 0;
}
```

```
auto = 0, static = 0
auto = 0, static = 1
auto = 0, static = 2
```

# Локални променливи декларирани како static

```
#include <stdio.h>
int main() {
  int counter;     /* brojac */
  for (counter = 0; counter < 3; ++counter) {
    int temporary = 1;     /* privremena promenliva */
    static int permanent = 1;  /* permanentna promenliva */
    printf("Temporary %d Permanent %d\n",
                              temporary, permanent);
    ++temporary;
    ++permanent;
   }
   return (0);
}
```

```
Temporary 1 Permanent 1
Temporary 1 Permanent 2
Temporary 1 Permanent 3
```

Универзитет „Св. Кирил и Методиј“ во Скопје
ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

# Рекурзија

## Структурно програмирање

ФИНКИ 2013

# Што ако функцијата се повика самата себеси?

```
void Rekurzivno(int i) {
        printf("%d", i);
        Rekurzivno(--i);
        }
```

# Рекурзија

Постојат 2 начина за составување ПОВТОРУВАЧКИ алгоритми:

- Итерација (Iteration)
  - Се повторуваат само параметрите на алгоритмот, а не и самиот алгоритам.
- Рекурзија (Recursion)
  - Повторувачки процес во кој алгоритмот СЕ ПОВИКУВА СЕБЕСИ. Алгоритмот се појавува во сопствената дефиниција .

# Да го разгледаме примерот

Начин на пресметување

5! = 5 * 4! (поедноставување на проблемот)

4! = 4 * 3! ...

По дефиниција 1! = 0! = 1 (основен случај)

Тогаш важи

2! = 2 * 1! = 2 * 1 = 2;

3! = 3 * 2! = 3 * 2 = 6;

```
int factoriel(int n){
    if(!n) return 1;
    else return n*factoriel(n-1);
}
```

# Да го разгледаме примерот

- Факториел на бројот n ?

Начин на пресметување

5! = 5 * 4! (поедноставување на проблемот)

4! = 4 * 3! ...

По дефиниција 1! = 0! = 1 (основен случај)

Тогаш важи

2! = 2 * 1! = 2 * 1 = 2;

3! = 3 * 2! = 3 * 2 = 6;

```
int factoriel(int n){
    if(!n) return 1;
    else return n*factoriel(n-1);
}
```

# Да го разгледаме примерот

- Факториел на бројот n ?
- Производ на целите вредности од 1 до бројот n.

Начин на пресметување

5! = 5 * 4! (поедноставување на проблемот)

4! = 4 * 3! ...

По дефиниција 1! = 0! = 1 (основен случај)

Тогаш важи

2! = 2 * 1! = 2 * 1 = 2;

3! = 3 * 2! = 3 * 2 = 6;

```
int factoriel(int n){
    if(!n) return 1;
    else return n*factoriel(n-1);
}
```

# Да го разгледаме примерот

- Факториел на бројот n ?
- Производ на целите вредности од 1 до бројот n.
- Итеративен алгоритам
  - Factoriel(n)=n (n-1) (n-2) … 2 1
  - Пр. Factoriel(4)=4*3*2*1 = 24

Начин на пресметување

5! = 5 * 4! (поедноставување на проблемот)

4! = 4 * 3! ...

По дефиниција 1! = 0! = 1 (основен случај)

Тогаш важи

2! = 2 * 1! = 2 * 1 = 2;

3! = 3 * 2! = 3 * 2 = 6;

```
int factoriel(int n){
    if(!n) return 1;
    else return n*factoriel(n-1);
}
```

# Да го разгледаме примерот

- Факториел на бројот n ?
- Производ на целите вредности од 1 до бројот n.
- Итеративен алгоритам
  - Factoriel(n)=n (n-1) (n-2) … 2 1
  - Пр. Factoriel(4)=4*3*2*1 = 24
- Рекурзивен алгоритам

Начин на пресметување

5! = 5 * 4! (поедноставување на проблемот)

4! = 4 * 3! ...

По дефиниција 1! = 0! = 1 (основен случај)

Тогаш важи

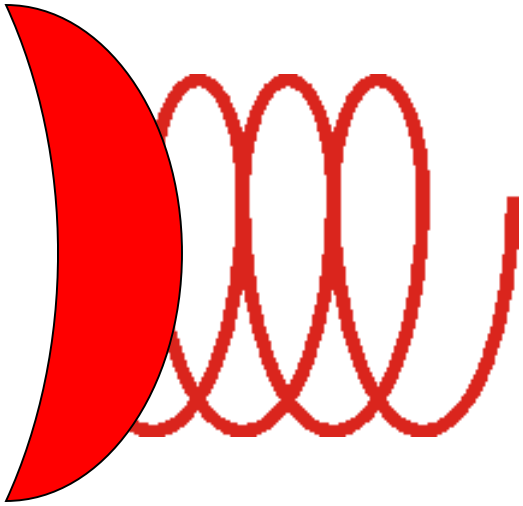
2! = 2 * 1! = 2 * 1 = 2;

3! = 3 * 2! = 3 * 2 = 6;

```
int factoriel(int n){
    if(!n) return 1;
    else return n*factoriel(n-1);
}
```

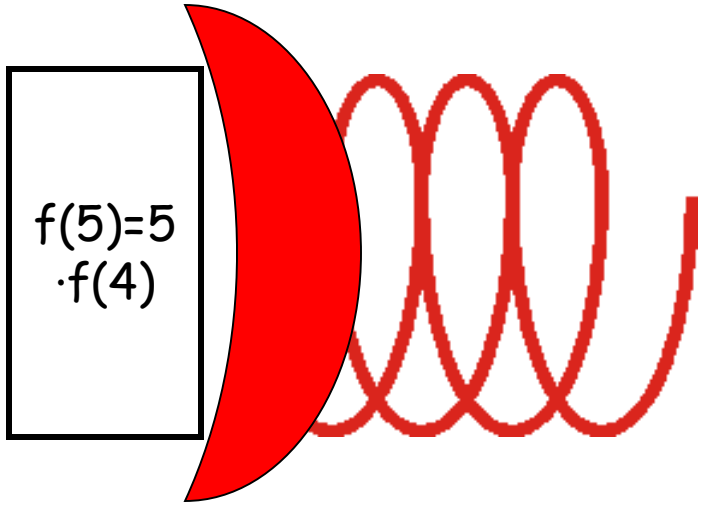
# Патување во две насоки

- Прво – се разложува од врвот кон дното ( од n до 0 )
- Второ- се решава одејќи од дното кон врвот (од 0 до n)



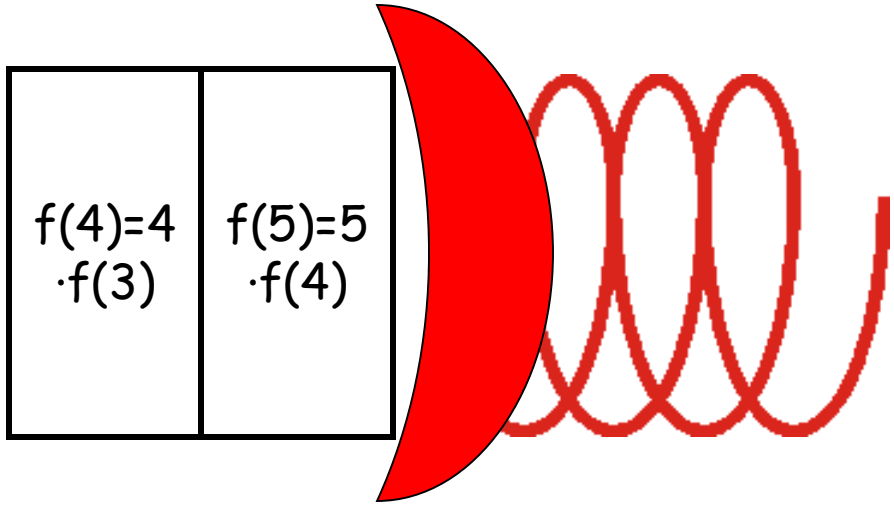
```
int factoriel(int n){
    if(!n) return 1;
    else return n*factoriel(n-1);
}
```

# Патување во две насоки



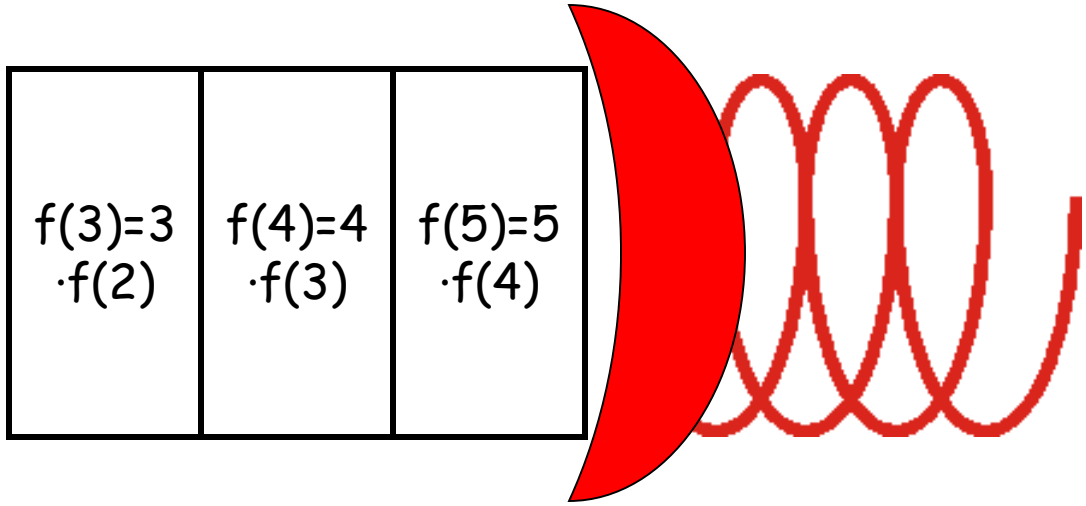

```
int factoriel(int n){
    if(!n) return 1;
    else return n*factoriel(n-1);
}
```

# Патување во две насоки



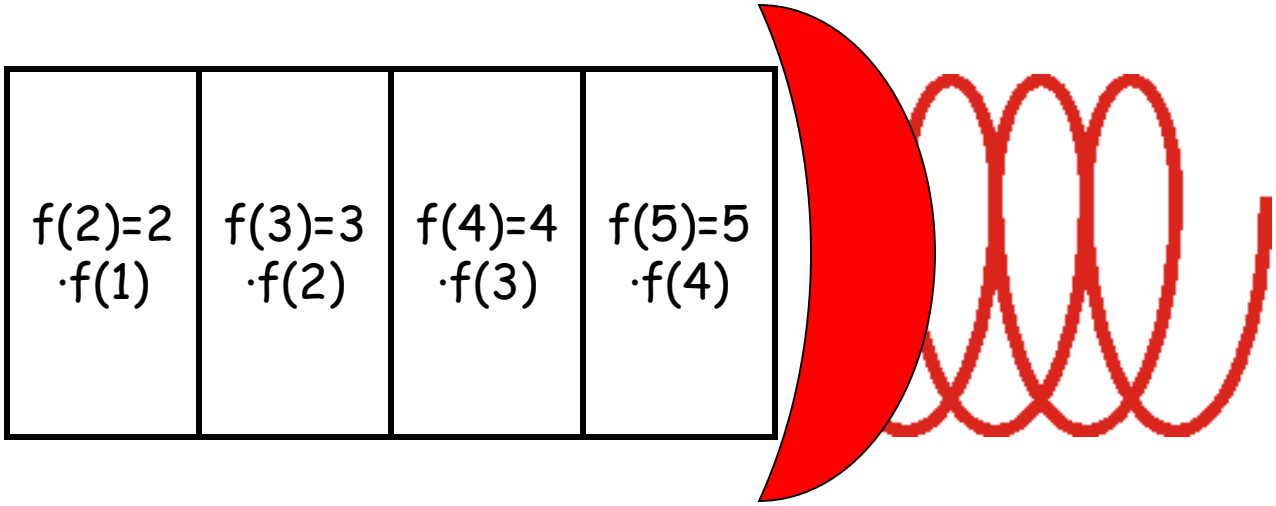

```
int factoriel(int n){
    if(!n) return 1;
    else return n*factoriel(n-1);
}
```

# Патување во две насоки



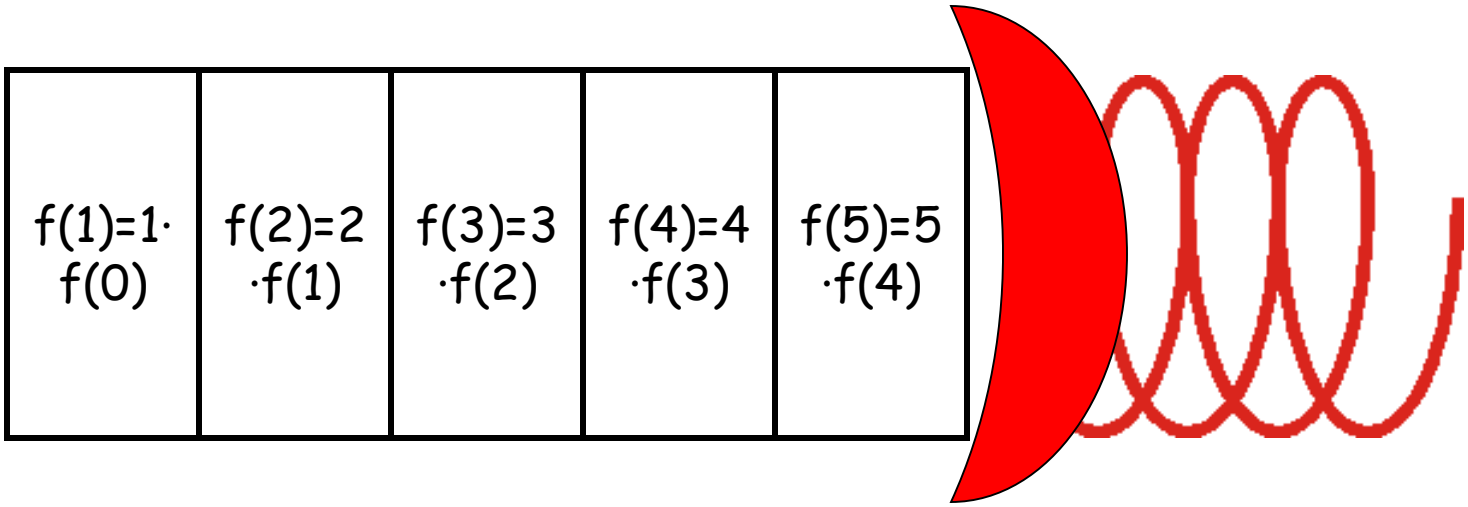

```
int factoriel(int n){
    if(!n) return 1;
    else return n*factoriel(n-1);
}
```

# Патување во две насоки




```
int factoriel(int n){
    if(!n) return 1;
    else return n*factoriel(n-1);
}
```

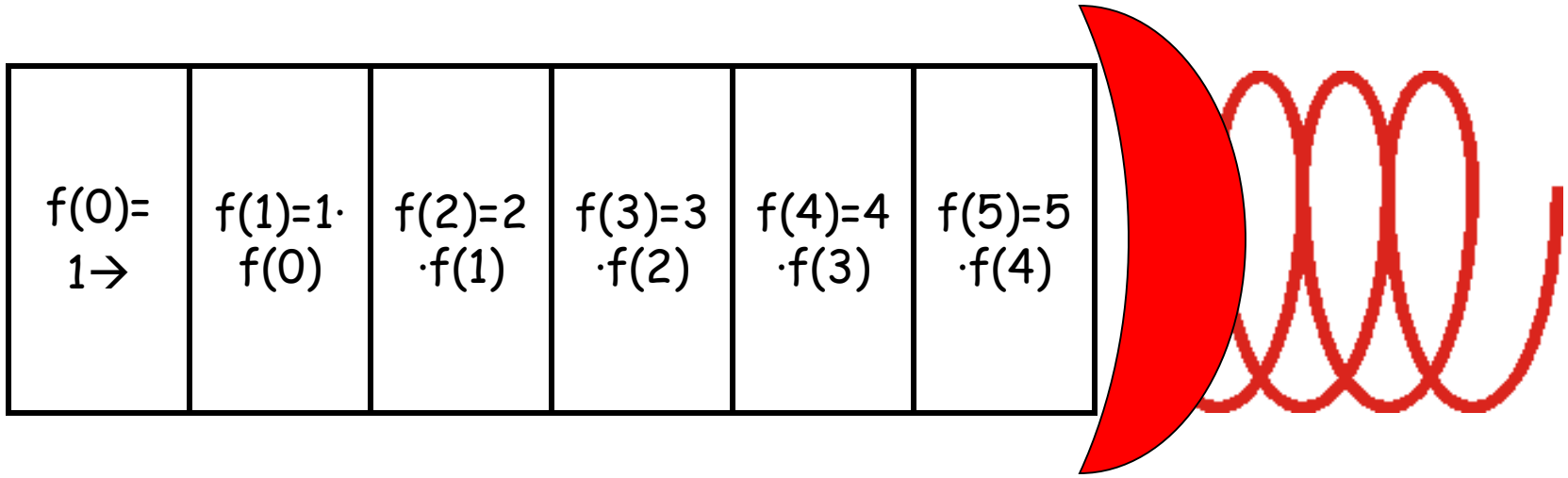
# Патување во две насоки




```
int factoriel(int n){
    if(!n) return 1;
    else return n*factoriel(n-1);
}
```

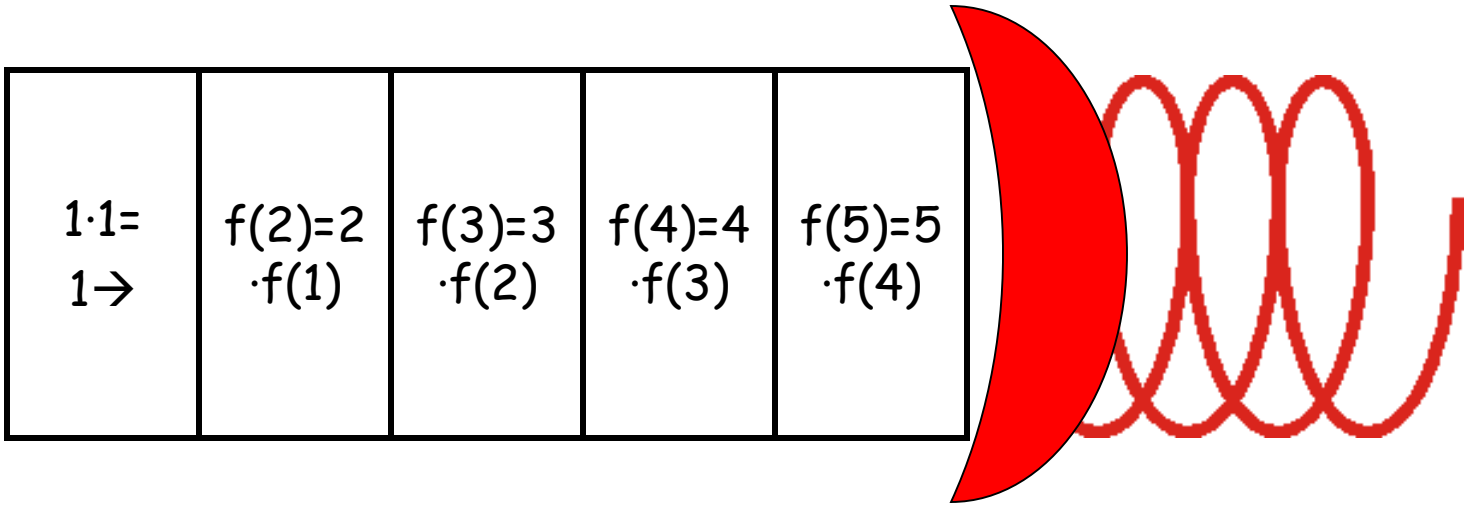
# Патување во две насоки

- Прво – се разложува од врвот кон дното ( од n до 0 )
- Второ- се решава одејќи од дното кон врвот (од 0 до n)



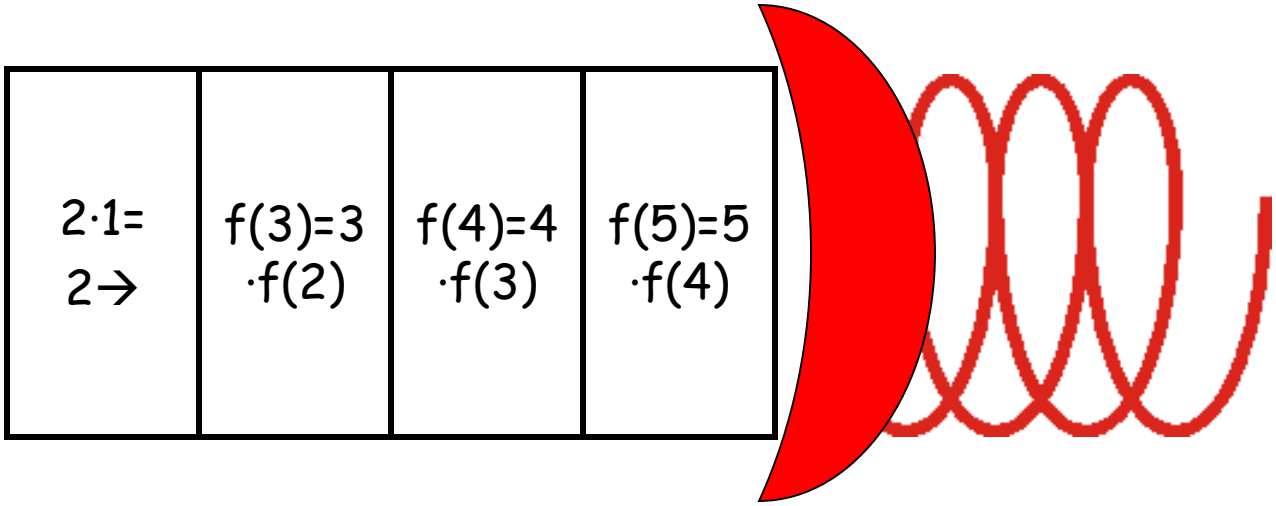

```
int factoriel(int n){
    if(!n) return 1;
    else return n*factoriel(n-1);
}
```

# Патување во две насоки



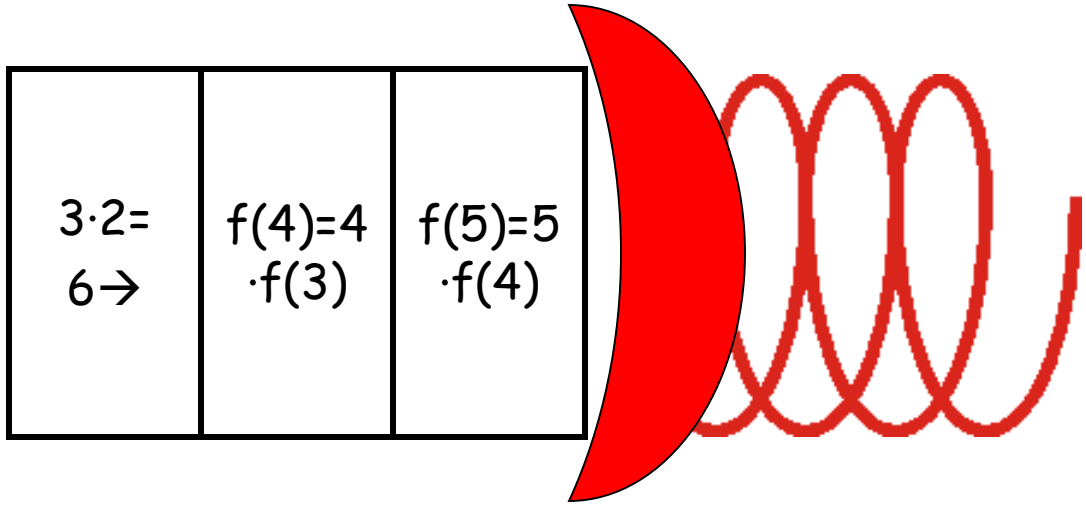

```
int factoriel(int n){
    if(!n) return 1;
    else return n*factoriel(n-1);
}
```

# Патување во две насоки



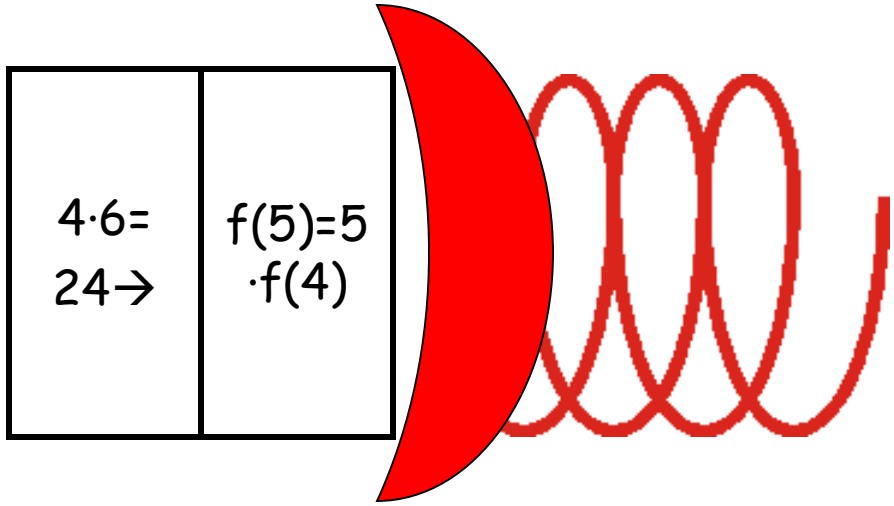

```
int factoriel(int n){
    if(!n) return 1;
    else return n*factoriel(n-1);
}
```

# Патување во две насоки



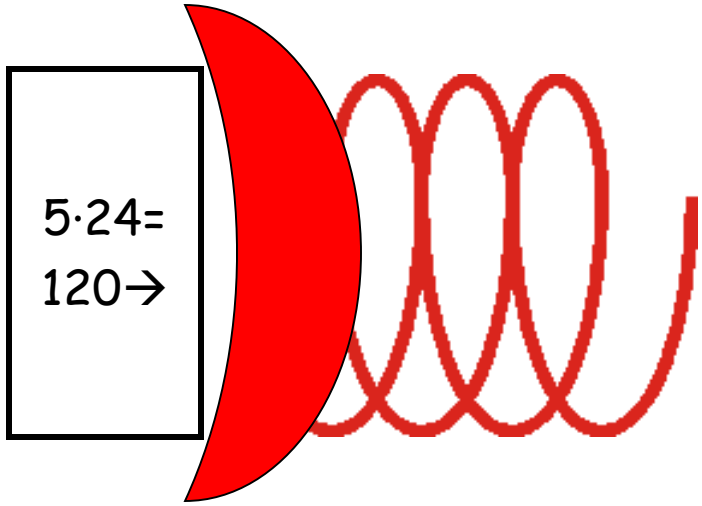

```
int factoriel(int n){
    if(!n) return 1;
    else return n*factoriel(n-1);
}
```

# Патување во две насоки



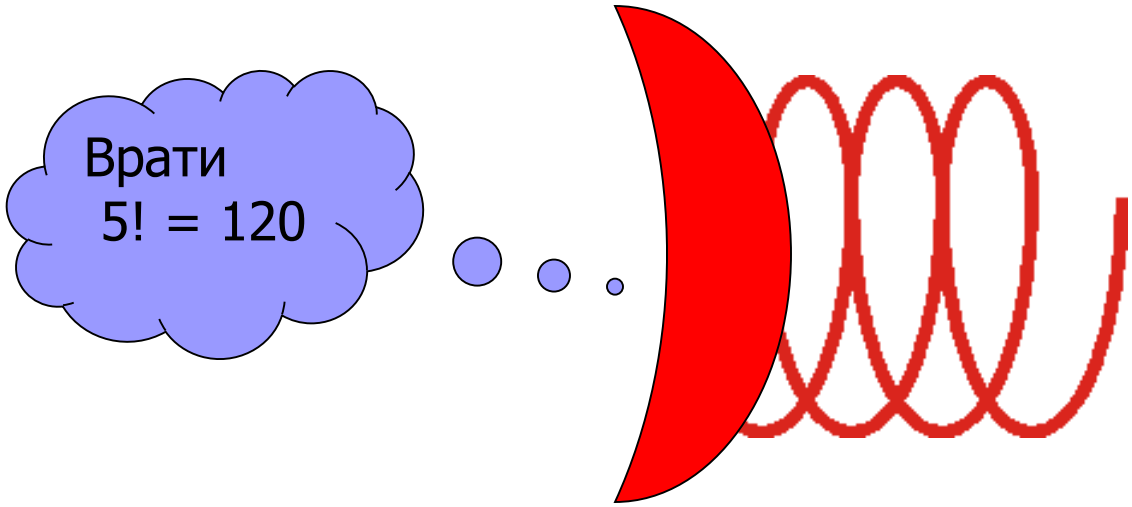

```
int factoriel(int n){
    if(!n) return 1;
    else return n*factoriel(n-1);
}
```

# Патување во две насоки




```
int factoriel(int n){
    if(!n) return 1;
    else return n*factoriel(n-1);
}
```

# Патување во две насоки




```
int factoriel(int n){
    if(!n) return 1;
    else return n*factoriel(n-1);
}
```

# Креирање рекурентни алгоритми

- Секоја рекурзивен алгоритам МОРА да има основен(граничен случај) :
  - Изразот што го решава проблемот
  - пр. factoriel (0)

- Остатокот од алгоритмот се нарекува општ случај (рекурентна врска)
  - n·factoriel(n-1)

# Пример

- За да се напише рекурзивен алгоритам:
  - Одредете го основниот случај factoriel (0)
  - Одредете го општиот случај (рекурентната врска) n factoriel(n-1)
- СЕКОЈ рекурзивен повик мора да реши
  - Дел од проблемот

  ИЛИ
  - Да ја редуцира големината на проблемот

# Пример

- Да се пресмета рекурзивно stepen (m,n) = $m^n$
- m= 2, n= 3, Stepen (2,3) = $2^3$=8
- Решение:
  - основен случај n=0, stepen(x,0)=1
  - Одредување на општиот случај

    n=1, stepen(2,1)=2

    n=2, stepen(2,2)=4

    n=3, stepen(2,3)=2·stepen(2,2)

    ➔stepen(m,n)=m·stepen(m,n-1)

```
int stepen(int m, int n) {
      if(n) return m*stepen(m,n-1);
      else return 1;
}
```

# Примери

# Примери

```
int zbirN(int n) {
.
.
.
```

```
.
.
.
.
```

```
.
.
.
.
.
.
.
.
```

# Примери

```
int zbirN(int n) {
    if (n==0) return 0;
```

# Примери

```
int zbirN(int n) {
    if (n==0) return 0;
    else return n+zbirN(n-1);
```

# Примери

```
int zbirN(int n) {
    if (n==0) return 0;
    else return n+zbirN(n-1);
}
```

# Примери

```
int zbirN(int n) {
    if (n==0) return 0;
    else return n+zbirN(n-1);
}
```

.
.
.
.

```
void svezdi(int n) {
.
.
.
.
.
.
```

# Примери

```
int zbirN(int n) {
    if (n==0) return 0;
    else return n+zbirN(n-1);
}
```

.
.
.
.

```
void svezdi(int n) {
    if (n>0) {
```

.
.
.
.

# Примери

```
int zbirN(int n) {
    if (n==0) return 0;
    else return n+zbirN(n-1);
}
```

.

.

.

.

```
void svezdi(int n) {
    if (n>0) {
      printf("*");
```

.

.

.

# Примери

```
int zbirN(int n) {
    if (n==0) return 0;
    else return n+zbirN(n-1);
}
```

.

.

.

.

```
void svezdi(int n) {
    if (n>0) {
      printf("*");
      svezdi(n-1);
```

.

.

# Примери

```
int zbirN(int n) {
    if (n==0) return 0;
    else return n+zbirN(n-1);
}
```

.

.

.

.

```
void svezdi(int n) {
    if (n>0) {
      printf("*");
      svezdi(n-1);
    }
```

.

# Примери

```
int zbirN(int n) {
    if (n==0) return 0;
    else return n+zbirN(n-1);
}
```

```
void svezdi(int n) {
    if (n>0) {
      printf("*");
      svezdi(n-1);
    }
    else printf("\n");
```

.

.

.

.

# Примери

```
int zbirN(int n) {
    if (n==0) return 0;
    else return n+zbirN(n-1);
}
```

```
void svezdi(int n) {
    if (n>0) {
      printf("*");
      svezdi(n-1);
    }
    else printf("\n");
}
```

.

.

.

.

# Примери

```
int zbirN(int n) {
    if (n==0) return 0;
    else return n+zbirN(n-1);
}
```

```
void svezdi(int n) {
    if (n>0) {
      printf("*");
      svezdi(n-1);
    }
    else printf("\n");
}
```

```
int zbirCifri(int n) {
.
.
.
```

# Примери

```
int zbirN(int n) {
    if (n==0) return 0;
    else return n+zbirN(n-1);
}
```

```
void svezdi(int n) {
    if (n>0) {
      printf("*");
      svezdi(n-1);
    }
    else printf("\n");
}
```

```
int zbirCifri(int n) {
        if (n<10) return n;
    .
    .
```

# Примери

```
int zbirN(int n) {
    if (n==0) return 0;
    else return n+zbirN(n-1);
}
```

```
void svezdi(int n) {
    if (n>0) {
      printf("*");
      svezdi(n-1);
    }
    else printf("\n");
}
```

```
int zbirCifri(int n) {
        if (n<10) return n;
        else return n%10 + zbirCifri(n/10);
}
```

# Примери

```
int zbirN(int n) {
    if (n==0) return 0;
    else return n+zbirN(n-1);
}
```

```
void svezdi(int n) {
    if (n>0) {
      printf("*");
      svezdi(n-1);
    }
    else printf("\n");
}
```

```
int zbirCifri(int n) {
        if (n<10) return n;
        else return n%10 + zbirCifri(n/10);
}
```

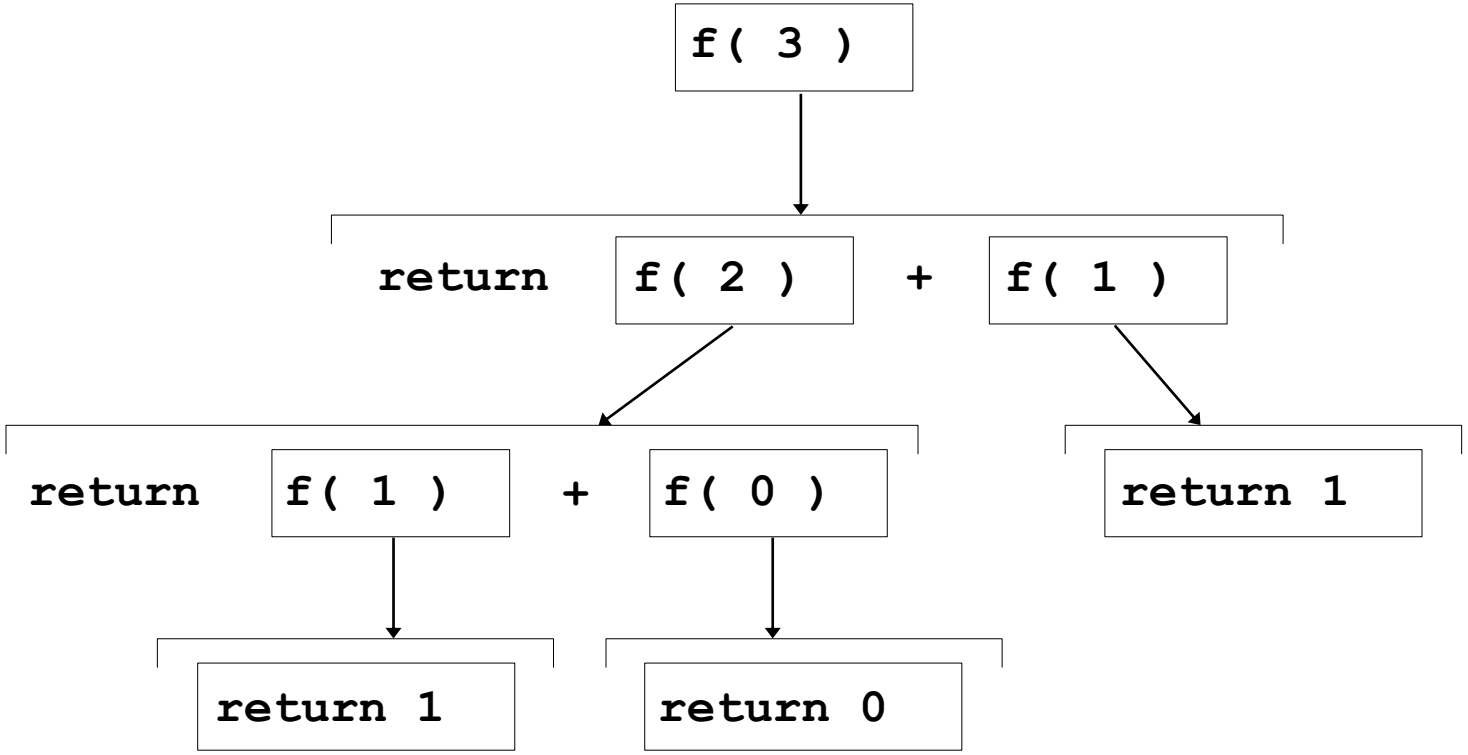
# Фибоначиева низа броеви со рекурзија

- Фибоначиева низа: 1, 1, 2, 3, 5, 8...
  секој број се добива како збир на претходните два

  **fib(n) = fib(n-1) + fib(n-2)** – рекурзивна формула

```
long fibonacci(long n) {
   if (n==0 || n==1) return n; /*osnoven slucaj*/
   else return fibonacci(n-1) + fibonacci(n-2);
}
```

# Начин на извршување на функцијата за пресметка на Фибоначиева низа



# Рекурзија vs Итерација

- Повторување
  - Итерација: експлицитни циклуси
  - Рекурзија: функциски повици
- Прекин на повторувањето
  - Итерација: условот за повторување повеќе не важи
  - Рекурзија: се препознава основниот случај
- И во обата случаи можни се појави на бесконечно повторување

# Кога да НЕ се користи рекурзија

- Ако се одговори со НЕ на било кое од следните прашања:
  - Дали алгоритмот или податочните структури природно се зададени со рекурзивна формула?
  - Дали рекурзивното решение е пократко и поразбирливо?
  - Дали рекурзивното решение се одвива во прифатливи временски и просторни граници?
- Рекурзивните алгоритми генерално се ПОСПОРИ од итеративните алгоритми
  - Функциските повици земаат повеќе време отколку инструкција во циклус

# Прашања?