Objektno orjentisano programiranje – C++

Funkcije

Teme

- Argumenti funkcije
- Povratak iz funkcije
- Doseg promenljivih
- Nizovi i pokazivači kao argumenti funkcija
- Reference
- Prototipovi funkcija
- Preklapanje funkcija
- Rekurzivne funkcije

Funkcije

- Funkcija je potprogram koji sadrži jednu C++ naredbu ili više njih, a izvodi određen zadatak
  – sve naredbe koje zaista nešto “rade” u programu nalaze se unutar funkcija
  – i najjednostavniji program mora da ima barem jednu funkciju: main()
- Sve funkcije u jeziku C++ su istog oblika:

```
povratni_tip ime(lista_argumenata) {
    //telo funkcije
}
```

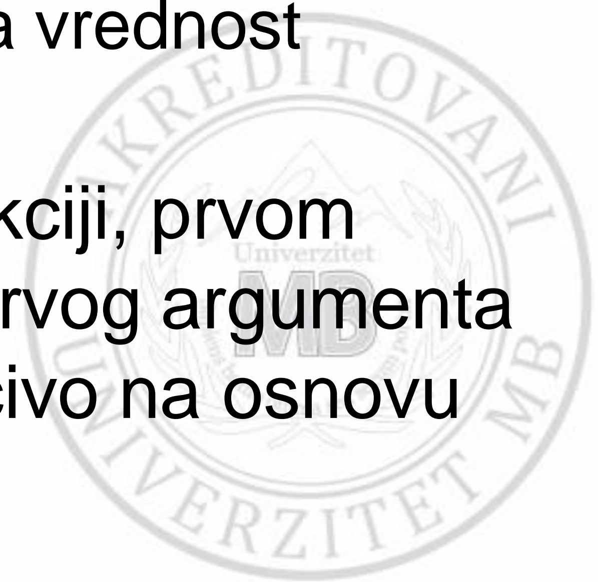
```
int addition (int a, int b){
  int r;
  r=a+b;
  return r;
}
```

Funkcije (2)

- Povratni tip definiše tip podataka koji funkcija “vraća”
  - funkcija ne vraća nikakvu vrednost: povratni tip je **void**
  - argumenti funkcije su vrednosti koje joj se prosleđuju, a u listi se razdvajaju zarezima
  - u telu funkcije nalaze se naredbe koje čine njen izvršni deo
  - izvršavanje funkcije se (po pravilu) završava kada stigne do zatvorene vitičaste zagrade }, kada se kontrola programa vraća u deo koda iz koga je funkcija pozvana

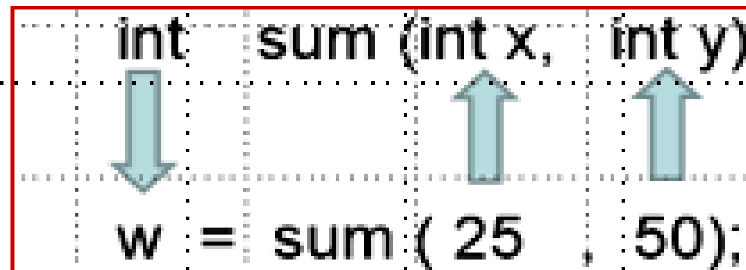
Argumenti funkcije

- Argumenti funkcije su vrednosti koje se koriste za njeno pozivanje
  - funkcija može, ali ne mora da ima argumente
  - promenljiva u funkciji koja prihvata vrednost argumenta zove se parametar
- Kada se argumenti prosleđuju funkciji, prvom parametru se dodeljuje vrednost prvog argumenta itd. (vrednosti se prosleđuju isključivo na osnovu pozicije)



Primer pozivanja funkcije

```
#include <iostream>
using namespace std;
int sum (int x, int y) {
   int y;
   y=x+y;
   return y;
}
int main (){
   int w;
   w = sum (25,50);
   cout << " Rezultat sumiranja = " << w;
  return 0;
}
```



Stvarni i formalni argumenti

- Stvarni parametri (engl. actual parameters) su oni koji se pojavljuju u pozivu funkcije, na primer:

w = sum (25,50);

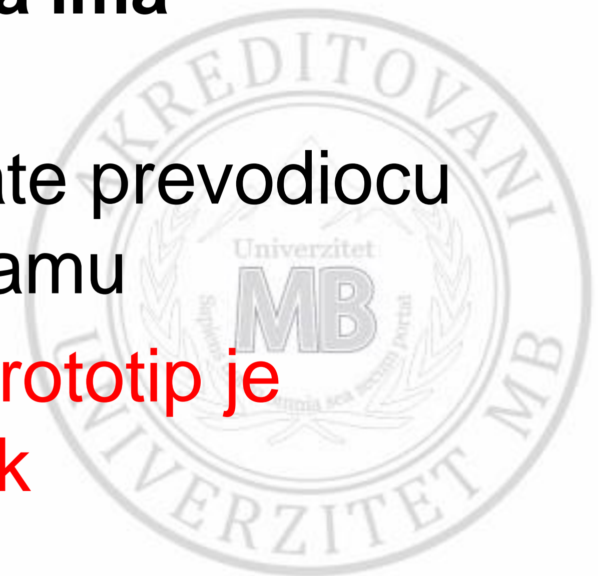
- Formalni parametri (engl. formal parameters) su oni koji se pojavlju u definiciji funkcije, na primer:

int sum (int x, int y)

int main ()

Prototip funkcije

- Prototip funkcije deklariše funkciju pre njene definicije
  - na osnovu prototipa prevodilac zna **koji je povratni tip funkcije**, **koliko ona ima argumenata i kog su oni tipa**
- Ove informacije moraju biti poznate prevodiocu pre prvog poziva funkcije u programu
- Jedina funkcija koja ne zahteva prototip je main(), jer je ona ugrađena u jezik



Primer prototipa funkcije

- Prototip funkcije koja ne vraća vrednost
- Ima dva argumenta od kojih je jedan tipa int,a drugi tipa float

void f1 (int, float);

```
/* prototip funkcije koja ne vraca vrednost
 a ima dva argumenta od kojih je jedan tipa int
 a drugi tipa double */
void mojaFunkcija(int, double);
```

Primer pozivanja funkcija (sa prototipovima)

```
#include <iostream>
using namespace std;
void f1 ();
int f2 (int);
void main () {
    cout << " U funkciji main " <<endl;
    f1 ();
    int i =  f2 (4);
    int k;
    cin>> k;
}
```

```
void f1 () {
    cout<< "U f1"<<endl;
}
int f2 (int j) {
    cout << "U f2, j = 0 " <<j;
    return j;
}
```

Primer: argumenti nekompatibilni

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

void funkcija (int *p); //protopip očekuje pokazivač kao argument
int main (int argc, char *argv[]) {
    int x=10;
    funkcija (x); //greška
        cin.ge
    return 0;
}
void funkcija (int *p){
    //kod funkcije
}
```

Error: argument of type "int" is incompatible with parameter of type "int *"

Funkcija bez tipa (void)

```
#include <iostream>
using namespace std;
void myPrint (){
    cout << "Primer funkcije bez tipa!";
    cin.get();
}
int main (){
    myPrint();
}
```

Kako C++ prosleđuje argumente

- Prenos po vrednosti (engl. pass-by-value)
  – vrednost argumenta kopira se u parametar funkcije
  – promene parametra nemaju nikakvog efekta na argument koji se koristi za poziv funkcije
  – poziv po vrednosti je default način prenosa parametara
- Prenos po adresi (engl. pass-by-reference)
  – u parametar se kopira adresa, a ne vrednost argumenta
  – promene parametra menjaju i argument
  – postiže se tako što je pokazivač argument funkcije

Prenos argumenata po vrednosti

```
#include <iostream>
using namespace std;
void calc (int x);
int main(){
   int x = 10;
   calc(x);
   cout<<"x = "<<x<<endl;
   cin.get();
  return 0;
}
```

```
void calc (int x){
    x = x + 10 ; //!!!
}
```

```
x = 10
```

Prenos argumenata po referenci

```
#include <iostream>
using namespace std;
void mul (int& a, int& b, int& c) {
   a*=2;
   b*=2;
   c*=2;
}
```

```
int main () {
    int x=1, y=3, z=7;
    mul (x, y, z);
    cout << "x=" << x
         << ", y=" << y
         << ", z=" << z;
    return 0;
}
```

x=2, y=6, z=14

Prenos argumenata po referenci pomoću pokazivača

```
#include <iostream>
using namespace std;
void calc (int *p);
int main() {
   int x = 10;
   calc(&x);
   cout<<"x = "<<x<<endl;
   cin.get();
   return 0;
}
```

```
void calc (int *p) {
   *p = *p + 10;
}
```

```
x = 20
```

Prenos argumenata po referenci pomoću pokazivača (2)

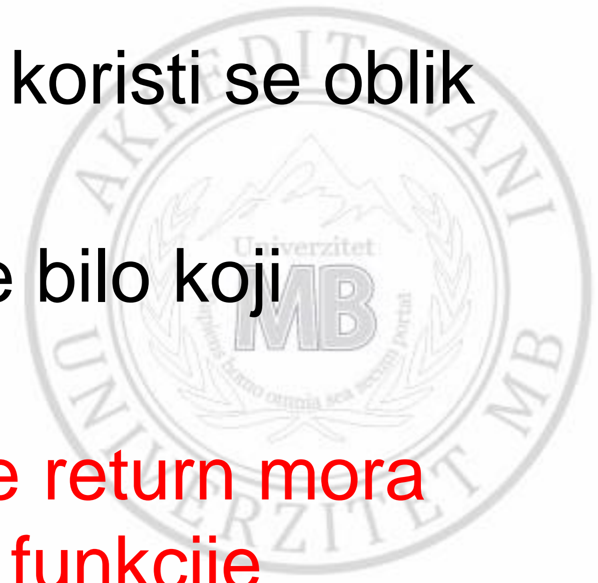
```
#include <iostream>
using namespace std;
void square(int *);
int main() {
   int n = 8;
   cout <<"&n="<< &n<< endl;
   cout <<n<< endl; // 8
   square(&n);
   cout << n << endl;   // 64
   cin.get();
   return 0;
}
```

```
void square(int * pN) {
     cout <<"pN = "
            << pN << endl;
    *pN *= *pN;
}
```

```
&n= 001CFEC4
8
pN = 001CFEC4
64
```

Povratak iz funkcije

- Naredba return služi za precizno kontrolisanje trenutka povratka iz funkcije
- Ako je povratni tip funkcije void, koristi se samo naredba return;
- Ako funkcija vraća neku vrednost, koristi se oblik return povratna_vrednost;
- Povratni tip funkcije može da bude bilo koji validni C++ tip, **osim niza**
- Tip povratne vrednosti iza naredbe return mora da se poklapa sa povratnim tipom funkcije



Povratak iz funkcije (2)

```
#include <iostream>
using namespace std;
int max(int n1, int n2);
int main () {
   int x = 100;
   int y = 200;
   int mn;
   mn = max(x, y);
   cout<< mn;
   cin.get();
   return 0;
}
```

```
int max(int n1, int n2) {
   int res;
   if (n1 > n2)
      res = n1;
   else
      res = n2;
   return res;
}
```

Vraćanje niza iz funkcije

```
#include <iostream>
using namespace std;
int * f1(int arr[], int length){
   for (int i = 0;i < length;++i){
      arr[i] = arr[i] * 4;
   }
   return arr;
}
```

```
*arr2 =4
*(arr2+1) =8
```

```
int main(int argc, char* argv[]){
   int arr[] = { 1,2,3,4,5 };
   int *arr2;
   arr2 = f1(arr, 5);
   cout <<"*arr2 ="
          <<*arr2 <<endl;
   cout<<"*(arr2+1) ="
          <<*(arr2+1) <<endl;
   cin.get();
   return 0;
}
```

Vraćanje niza iz funkcije pomoću pokazivača

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

int * f1 (int a1[], int a2[]){
    for (int i = 0;i < 5;++i){
     a1[i] = a1[i] + a2[i];
    }
    return a1;
}
```

```
*arr3[0] =7
*arr3[1] =9
*arr3[2] =11
*arr3[3] =13
*arr3[4] =15
```

```
int main(){
    int arr1[] = { 1,2,3,4,5 };
    int arr2[] = { 6,7,8,9,10 };
    int *arr3;
    arr3 = f1(arr1,arr2);
    for (int i = 0;i < 5;++i)
        cout<<"*arr3["<<i <<"] ="
              <<*(arr3+i)<<endl;
    cin.get();
    return 0;
}
```

Vraćanje niza iz funkcije pomoću pokazivača (2)

```
#include <iostream>
using namespace std;
double * f1 (double a1[], double a2[]){
   for (int i = 0;i < 5;++i){
     a1[i] = a1[i] + a2[i];
   }
   return a1;
}
```

```
*arr3[0] =7.8
*arr3[1] =10.1
*arr3[2] =12.3
*arr3[3] =13.5
*arr3[4] =15.7
```

```
int main(){
   double arr1[] =
   { 1.1,2.3,3.4,4.5,5.6 };
   double arr2[] =
   { 6.7,7.8,8.9,9.0,10.1 };
   double *arr3;
   arr3 = f1(arr1,arr2);
   for (int i = 0;i < 5;++i)
      cout<<"*arr3["<<i <<"] ="
             <<*(arr3+i)<<endl;
   cin.get();
   return 0;
}
```

Prenos argumenata po referenci (nizovi)

```
#include <iostream>
using namespace std;
void test (int* a, int* b, int* c, int len) {
  for (int i = 0; i < len; ++i)
      c[i] = a[i] + b[i];
}
int main() {
   int a[5] = {1,2,3,4,5}, b[5] = {6,7,8,9,10}, c[5] = {};
   test(a, b, c, 5);
   for (int i = 0; i < 5; ++i)
        cout <<"c["<<i<<"] = "<<c[i]<<endl;
   cin.get();
   return 0;
}
```

```
c[0] = 7
c[1] = 9
c[2] = 11
c[3] = 13
c[4] = 15
```

Prenos argumenata po referenci (nizovi)

```
#include <iostream>
using namespace std;
int * test (int* a,int* b,int*c,int len){
  for (int i = 0; i < len; ++i)
       c[i] = a[i] + b[i];
       return c;
}
```

```
int main() {
   int a[5] = {1,2,3,4,5},
        b[5] = {6,7,8,9,10},
        c[5] = {};
   int * p= test(a, b, c, 5);
    for (int i = 0; i < 5; ++i)
     cout <<"p["<<i<<"] ="
             <<p[i]<<endl;
   cin.get();
   return 0;
```

Vanjske varijable (engl. extern variable)

File1.cpp

```
#include<iostream>
using namespace std;
int globe ;
void func();
int main()
{
 ......
 .....
}
```

File2.cpp

```
extern int globe ;
int b = globe + 10 ;
```

Global variable in one file is used in other file by **extern keyword**

Funkcija exit ()

- Funkcija exit() iz C++ biblioteke odmah okončava proces.
- Definicija funkcije

void exit(int status)

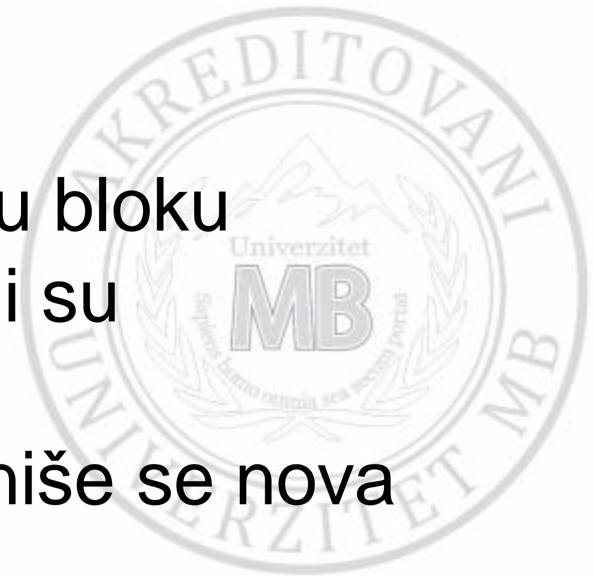
Funkcija exit () (2)

```
#include <iostream>
using namespace std;
int main (){
   cout<<"Početak programa...."<<endl;
   cout<<"Izlazak iz programa"<<endl;
   exit(0);
   cout<<"Kraj programa"<<endl;
   return(0);
}
```

```
Početak programa
Izlazak iz programa
```

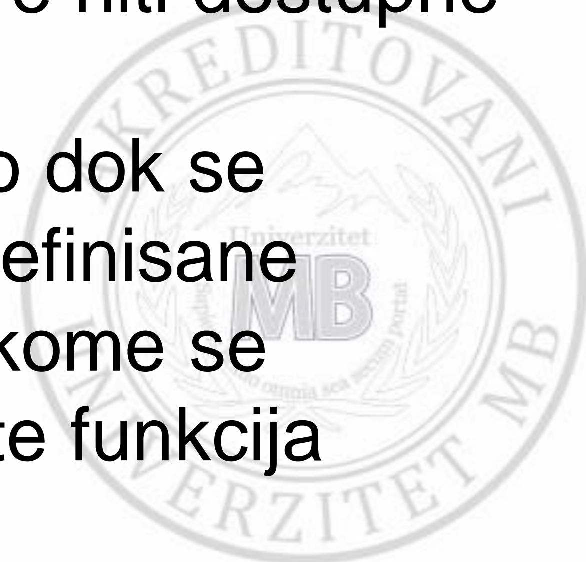
Oblast važenja (engl. scope)

- Pravila oblasti važenja (dosega) promenljivih upravljaju vidljivošću i životnim vekom objekata
- Razlikujemo dva osnovna opsega (oblasti) važenja (scope):
  - lokalni
  - globalni
- Lokalni opseg važnosti se definiše u bloku (početak i kraj bloka koda definisani su zagradama)
- Kad god se započne nov blok, definiše se nova oblast važnosti (scope)



Lokalne promenljive

- Promenljiva koja je definisana unutar bloka zove se lokalna promenljiva
- Sve promenljive definisane unutar bloka su lokalne za taj blok, tj. nisu vidljive niti dostupne izvan njega
- Lokalne promenljive "žive" samo dok se izvršava blok koda u kome su definisane
- Najčešće korišćen blok koda u kome se definišu lokalne promenljive jeste funkcija



Lokalne promenljive (2)

```
#include <iostream>
using namespace std;
void funkcija ();
int main () {
     int var =10;
    cout<<"vrednost var u funkciji main()=" <<var<<endl;
    funkcija ();
    cout<<"vrednost var u funkciji main() nakon poziva funkcije="
    <<var<<endl; cin.get();
    return 0;
}
void funkcija () {
    int var=88;       //var je lokalna promenljiva
    cout<<"vrednost var u funkciji="<<var<<endl;
}
```

Lokalne promenljive (2)

```
#include <iostream>
using namespace std;
void funkcija ();
int main () {
     int var =10;
     cout<<"vrednost var u funkciji main()=" <<var<<endl;
     funkcija ();
```

```
vrednost var u funkciji main()=10
vrednost var u funkciji=88
vrednost var u funkciji main() nakon poziva funkcije=10
```

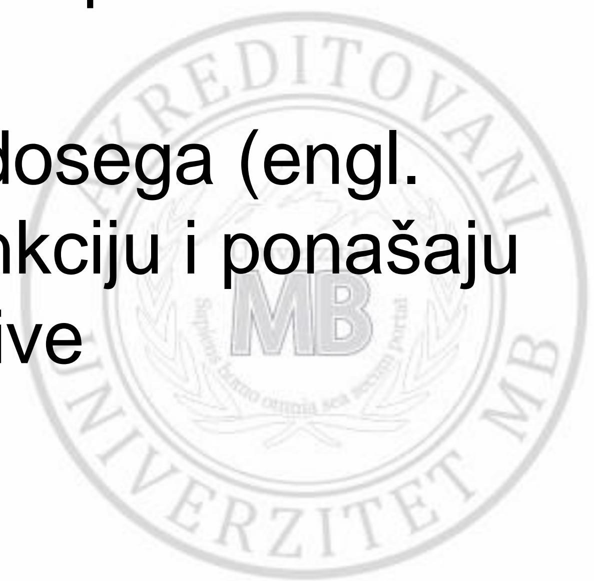
```
}
void funkcija () {
     int var=88;          //var je lokalna promenljiva
     cout<<"vrednost var u funkciji="<<var<<endl;
}
```

Lokalne promenljive (3)

- Pošto se **lokalna promenljiva pravi i uništava** pri svakom ulasku i izlasku iz bloka u kome je definisana, njena vrednost neće biti zapamćena između dva izvršavanja
  - lokalne promenljive u funkciji se prave pri ulasku u funkciju, a uništavaju po izlasku (ne mogu da zadrže vrednost između dva poziva funkcije)
  - ako se lokalna promenljiva inicijalizuje, onda se inicijalizacija ponavlja pri svakom izvršavanju bloka

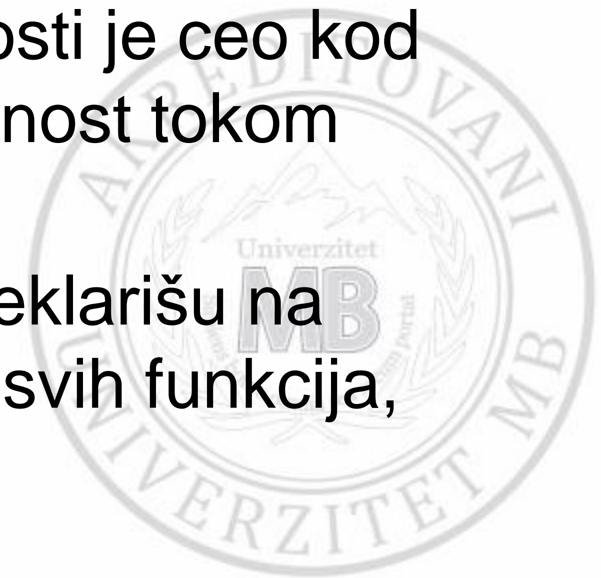
Lokalne promenljive (4)

- Lokalna promenljiva se može definisati bilo gde u bloku, pre prvog korišćenja
  - običnо se sve promenljive definišu na početku funkcije
- Pošto su parametri funkcije unutar dosega (engl. scope) funkcije, oni su lokalni za funkciju i ponašaju se kao i sve druge lokalne promenljive



Globalne promenljive

- Globalni doseg (global scope) je oblast deklarisanja koja se nalazi izvan svih funkcija
  - globalne promenljive dostupne su u celom programu, tj. njihova oblast važnosti je ceo kod programa i zadržavaju svoju vrednost tokom celog izvršavanja
  - obično se globalne promenljive deklarišu na samom početku programa, izvan svih funkcija, pre funkcije main()



Globalne promenljive (2)

- Inicijalizuju se kada program počne da se izvršava
  - ako nije navedena vrednost za inicijalizaciju globalne promenljive, ona se inicijalizuje na 0
- Globalne promenljive smeštaju se u posebnu oblast memorije rezervisanu za tu namenu
  - korisne su kada se isti podatak koristi u više funkcija, ili kada neka promenljiva treba da zadrži vrednost tokom celokupnog izvršavanja programa

Globalne promenljive (3)

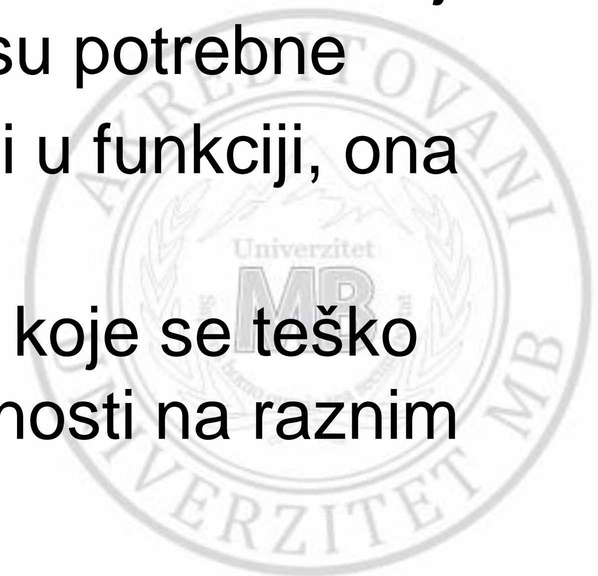
```
#include <iostream>
using namespace std;
void f1();
void f2();
int brojac;  //globalna var
int main () {
    int i;
    for (int i=0; i<10; i++) {
        brojac = i*2;
        f1();
    }
    cin.get();
    return 0;
}
```

```
void f1 () {
    cout<< "brojac =" << brojac;
    f2();
}
void f2 () {
    int brojac; // lokalna var
    for (brojac=0; brojac<2;brojac++ )
      cout <<".";
    cout <<endl;
}
```

```
brojac =0..
brojac =2..
brojac =4..
brojac =6..
brojac =8..
brojac =10..
brojac =12..
brojac =14..
brojac =16..
brojac =18..
```

Globalne promenljive (4)

- Korišćenje globalnih promenljivih treba izbegavati iz više razloga:
  - zauzimaju memoriju sve vreme tokom izvršavanja programa, a ne samo onda kada su potrebne
  - ako se globalna promenljiva koristi u funkciji, ona postaje manje opšta
  - prouzrokuju greške u programima koje se teško otkrivaju, npr. zbog promena vrednosti na raznim mestima u programu



Globalne i lokalne promenljive

```
#include <iostream>
using namespace std;
int func() {
   int i=10;
   i++;      /* lokalna varijabla   */
   return i;
  }
int main () {
   int i=4;
   i++;         /* globalna varijabla    */
   int k=func();
   cin.get();
}
```

Prosleđivanje pokazivača funkciji

- Kada parametri funkcije nisu pokazivači, prilikom poziva funkcije pravi se kopija argumenata sa kojima se radi u funkciji; to se zove prosleđivanje po vrednosti (pass-by-value)
  - funkcija ne može nikako da promeni vrednosti argumenata koji joj se prosledjuju po vrednosti
- Kada je parametar funkcije pokazivač, funkcija može da promeni vrednost promenljive na koju pokazuje pokazivač; to je prosleđivanje po adresi (pass-by-address)

Pokazivači kao argument funkcije

```
#include <iostream>
using namespace std;
//prosleđivanje pokazivača
void funkcija (int *p) {
  *p = 100;
}
```

i = 100

```
int main () {
  int i=0;
  int *p;
  p = &i;
//prosleđivanje pokazivača kao
// argumenta
  funkcija (p);
  cout<<"i = "<<i<<endl;
  cin.get();
  return 0;
}
```

AKREDITOVANI UNIVERZITET MB

Niz kao argument funkcije

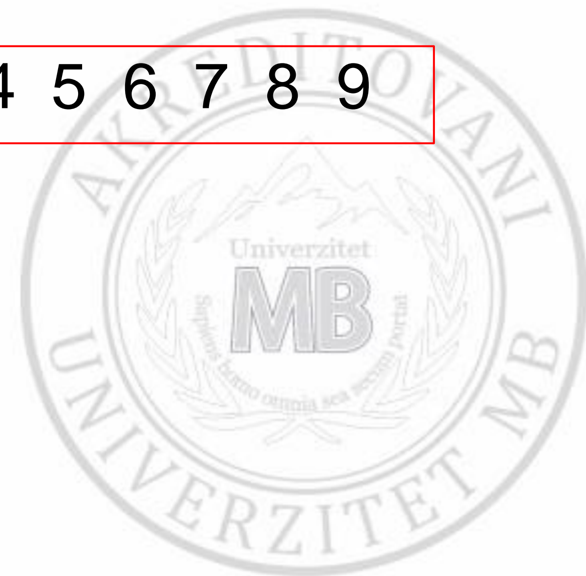
- Kada je argument funkcije niz, prosleđuje se adresa prvog elementa niza, a ne ceo niz
- Pošto je ime niza pokazivač na prvi element niza, to znači da se zapravo prosleđuje pokazivač na niz, pa funkcija može da promeni sadržaj niza koji se koristi za pozivanje funkcije

Prosleđivanje nizova funkciji

```
#include <iostream>
using namespace std;
void prikaziNiz (int brojevi[10]);
int main () {
   int t[10],i;
   for (int i=0;i<10;i++)
      t[i] =i;
   prikaziNiz (t);
   cin.get();
   return 0;
}
```

```
void prikaziNiz (int brojevi[10]) {
  for (int i=0;i<10;i++)
    cout<<brojevi[i]<<"   ";
    cout<<endl;
}
```

```
0 1 2 3 4 5 6 7 8 9
```



Funkcija main

- Prva funkcija koja se poziva kad program počne da se izvršava, bez obzira na to gde se nalazi u kodu
- Programu main možemo da prosledimo parametre iz komandne linije
  - npr. program možemo da kompajliramo iz komandne linije pomoću komande

cl ime_programa;

ovdje je *ime_programa* argument

- Jezik C++ definiše dva (opciona) parametra za funkciju main() koji prihvataju argumente iz komandne linije: argc i argv

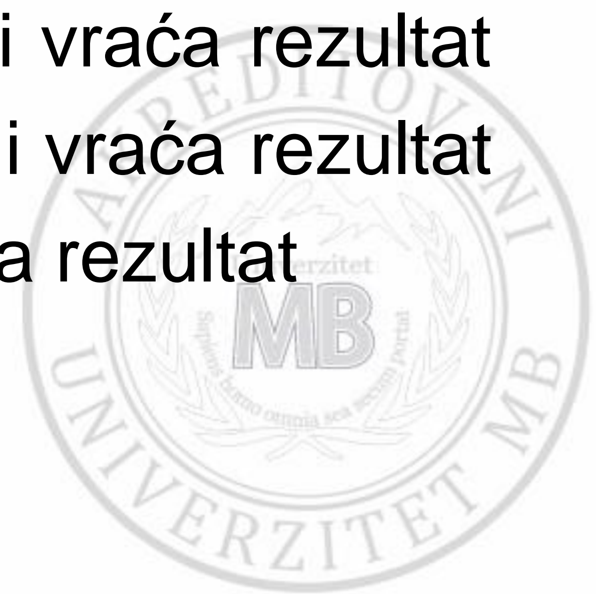
Prosleđivanje numeričkih parametara funkciji main()

- Deklarisane u zaglavlju <cstdlib>

atof()- Konvertuje string u **double** i vraća rezultat

atol()- Konvertuje string u **long int** i vraća rezultat

atoi()- Konvertuje string u **int** i vraća rezultat



Rekurzivne funkcije

- Funkcije koje pozivaju same sebe
- Većina rekurzivnih funkcija se izvršava sporije od svojih iterativnih verzija, zbog usporenja koje izaziva pozivanje funkcija
- Pri svakom pozivanju rekurzivne funkcije prave se nove kopije argumenata i lokalnih promenljivih na steku, pa treba voditi računa o tome da se on ne "istroši"
- Rekurzivne funkcije su pogodne za rešavanje problema koji se elegantnije rešavaju rekurzijom (npr. faktorijel)

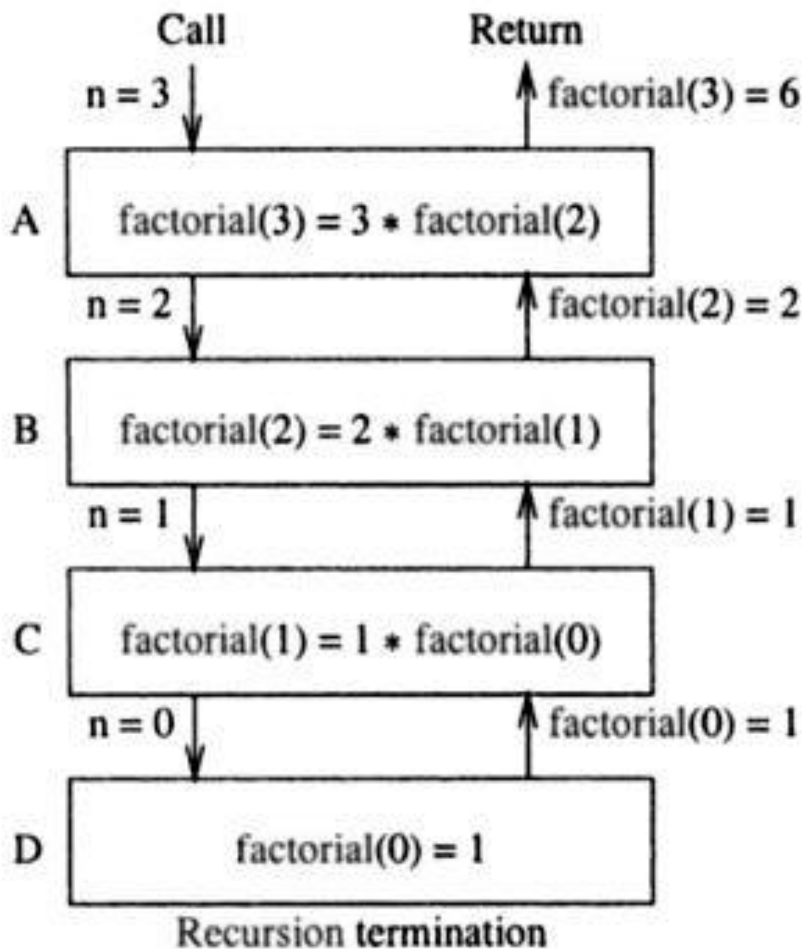
Rekurzivne funkcije (2)

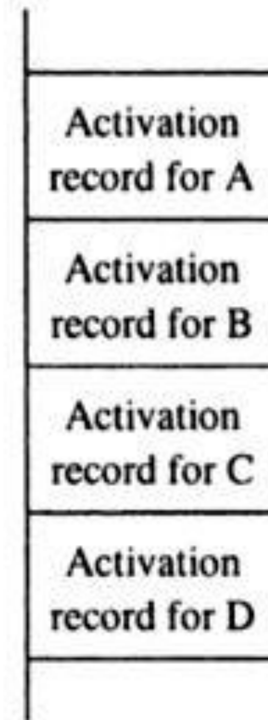
```
#include <iostream>
using namespace std;
void obrniString(char *);
int main () {
   char str[] = "Ovo je test";
   obrniString(str);
   cin.get();
   return 0;
}
```

```
void obrniString(char *p) {
  if (*p)
     obrniString (p+1);
  else
     return;
  cout<<*p;
}
```

```
tset ej ovO
```

Funkcija factorial 3!





Reference

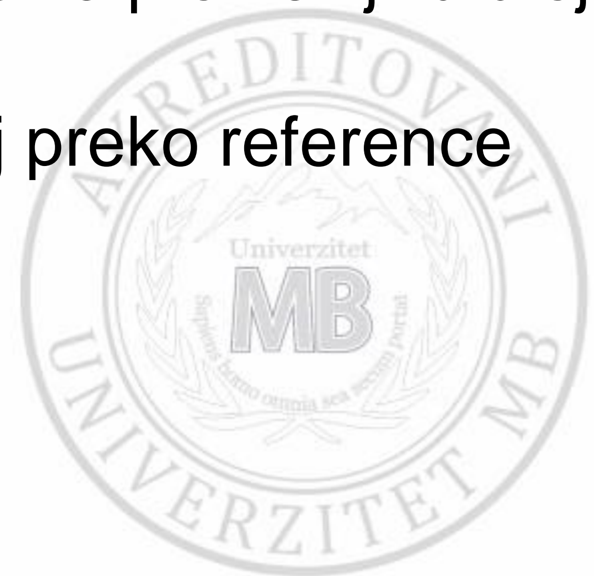
- Umesto “ručnog” poziva po adresi pozivanjem funkcije sa argumentima pokazivačima, može se C++ kompajleru reći da automatski koristi prenos po adresi
- To se postiže tako što se kao argument funkcije navode reference
- Unutar funkcije, operacije sa parametrom se automatski dereferenciraju, što znači da se ne radi sa adresom promenljive nego sa njenom vrednošću

Reference (2)

```
#include <iostream>
using namespace std;
int main () {
  long broj =10;
    long &rboj = broj; //definisanje reference na promenljivu broj
    cout<<"broj pre: " <<broj<<endl;
    rboj+=10; //povećava promenljivu broj preko reference
    cout<<"broj posle: "<<broj<<endl;
    cin.get();
    return 0;
}
```

- **Referenca se deklariše tako što se ispred njenog imena navodi &**

```
broj pre: 10
broj posle: 20
```



Reference (3)

- Referenca se koristi kao alijas (drugo ime) za promjnljivu, tj. može se koristiti kao alternativni način za pristup promjnljivoj
- Prilikom deklaracije, nezavisna referenca se uvek inicijalizuje i tokom celog svog životnog veka je vezana za objekat kojim je inicijalizovana
  - reference se ne mogu “preusmeravati” na druge promenljive kao pokazivači
- Retko se reference koriste samostalno; njihova najvažnija upotreba je kao argumenti funkcija

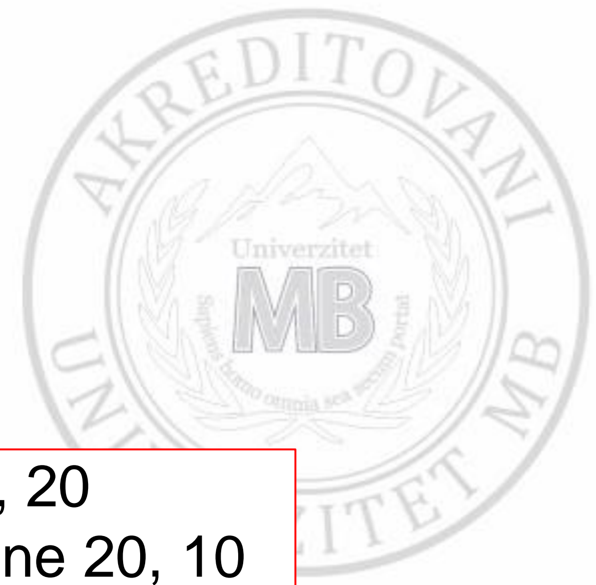
Reference kao argumenti funkcija

- Prosleđivanje reference postiže se navođenjem operatora & ispred imena promenljive
- Nije potrebno u funkciji koristiti operator dereferenciranja (*) kao što je slučaj sa pokazivačima
- Operacije koje se izvode nad parametrom koji je referenca utiču na samu vrednost parametra, a ne na njegovu adresu

Primer

```
#include <iostream>
using namespace std;
void swap(int&, int&);
int main(){
  int i=10, j=20;
  cout<<"Pocetne vrednosti i,j "
        << i <<", "<< j <<endl;
  swap(i, j);
  cout<<"Vrednosti i,j posle zamene„
        << i <<", "<< j <<endl;
  cin.get();
  return 0;
}
```

```
void swap(int&x, int&y) {
   int temp;
   temp = x;
   x = y;
   y = temp;
}
```



```
Pocetne vrednosti i,j 10, 20
Vrednosti i,j posle zamene 20, 10
```

Reference (4)

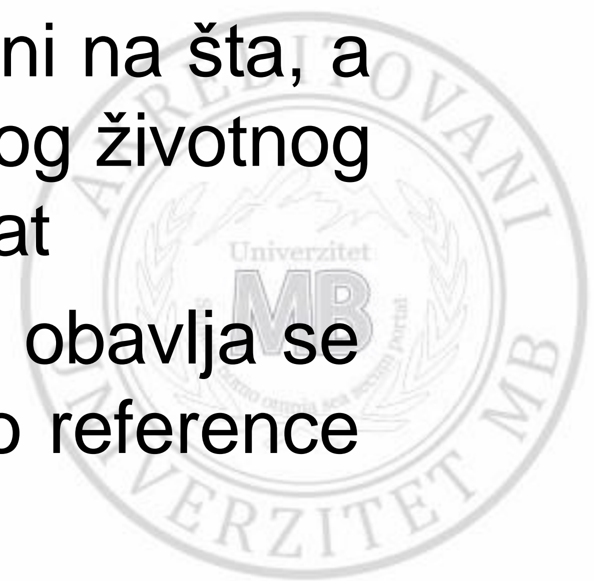
- Funkcija čiji je argument referenca poziva se na isti način kao da se radi o prenosu po vrednosti
- Parametar se inicijalizuje adresom argumenta, bez ikakvog kopiranja vrednosti
- Referenca kao argument funkcije pravi se pri svakom pozivu funkcije, i uništava po završetku izvršavanja funkcije

Pravila u radu sa referencama

- Ne može se definisati referenca promenljive koja je sama referenca (tj. ne postoje reference na reference)
- Ne postoje nizovi referenci
- Ne postoje pokazivači na reference

Pokazivači i reference

- Pokazivač se može preusmeriti tako da pokazuje na neki drugi objekat (promenljivu), a referenca ne može
- Pokazivač može da ne pokazuje ni na šta, a referenca od početka do kraja svog životnog veka pokazuje na jedan isti objekat
- Pristup objektu preko pokazivača obavlja se pomoću operatora *, dok je preko reference pristup neposredan



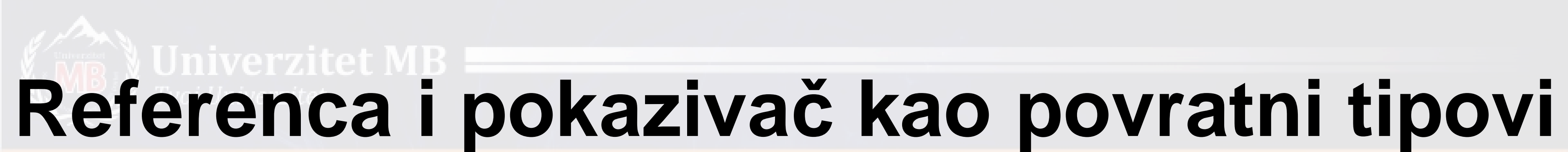
Referenca kao povratni tip funkcije

- Kada je povratni tip funkcije referenca, ona vraća implicitni pokazivač na povratnu vrednost
- To znači da se funkcija u tom slučaju može koristiti i sa leve strane operatora dodele (lvalue)
- Kada je povratni tip referenca, treba voditi računa o životnom veku promenljive povezane sa referencom

Referenca kao povratni tip funkcije (2)

- Pokazivač na lokalnu promenljivu, kao ni referenca na lokalnu promenljivu u funkciji nikada ne smeju da budu povratna vrednost funkcije
- Pošto referenca nikada ne postoji samostalno, već je uvek alijas neke druge promenljive, objekat s kojim je povezana mora da postoji i nakon završetka izvršavanja funkcije

```
int& f()= {
    int i=10;
     return i;  //promenljiva i više ne postoji
                //kada se funkcija završi
}
```



Referenca i pokazivač kao povratni tipovi funkcije – primer slajd 1

```
#include <iostream>
using namespace std;
void swap(int* x, int* y) { // prenos pokazivačem
    int z = *x;
    *x=*y;
    *y=z;
}

void swap(int& x, int& y){  // prenos referencom
    int z = x;
    x=y;
    y=z;
}
```

Referenca i pokazivač kao povratni tipovi funkcije – primer slajd 2

```
int main(){
    int a = 45;
    int b = 35;
    cout<<"Before Swap\n";
    cout<<"a="<<a <<" b="<<b<<endl;
    swap(&a, &b);
    cout<< "Prije swap, prenos pokazivačem"<<endl;
    cout<<"a="<<a<<" b="<<b<<endl;;
    swap(a, b);
    cout<<"Posle  swap, prenos referencom"<<endl;
    cout<<"a="<<a<<" b="<<b<<endl;
    cin.get();
}
```

Korišćenje modifikatora const u funkciji

- Kada se modifikator const navede ispred imena parametra funkcije, to znači da odgovarajući argument nikako ne sme biti promenjen
  - kompajler proverava da li se argument unutar funkcije menja i upozorava na to
  - korišćenjem referenci može se izbeći nepotrebno kopiranje argumenata prilikom poziva funkcije, a ako se one koriste uz modifikator const, obezbedjuje se dodatna zaštita od slučajne izmene argumenta

Korišćenje modifikatora const u funkcijama

```
#include <iostream>
using namespace std;
int incr10 (const int &broj);
int main () {
   const int primer=20;
   cout<<"pre poziva funkcije: "<<primer<<endl;
   cout<<"vrednost koja se vraca iz funkcije: "
         <<incr10(primer)<<endl;
   cout<<"posle poziva funkcije: "<<primer<<endl;
   cin.get();
   return 0;
}
```

```
int incr10 (const int &broj) {
  //return broj +=10, ovo je bila greška
   return (broj *10);
}
```

```
pre poziva funkcije: 20
vrednost koja se vraca iz funkcije: 200
posle poziva funkcije: 20
```

Univerzitet MB

Statičke promenljive u funkcijama

- Umesto korišćenja globalnih promenljivih u funkcijama kada je potrebna promenljiva koja pamti vrednosti između različitih poziva iste funkcije bolje je koristiti statičke promenljive
- Deklaracija statičke promenljive postiže se dodavanjem modifikatora static ispred tipa promenljive

Statičke promenljive u funkcijama (2)

- Statička promenljiva inicijalizuje se samo jednom, pri prvom izvršavanju funkcije, pamti svoje vrednosti tokom celog izvršavanja programa, a dostupna je u bloku u kome je deklarisana

Modifikator static

```
#include <iostream>
using namespace std;
void zapamti();
int main () {
   for (int i=0;i<5;i++)
      zapamti();
   cin.get();
   return 0;
}
void zapamti(){
  static int brojac =0;
  cout<<"Ovo je: "<<brojac++<<" put da sam pozvan"<<endl;
}
```

```
Ovo je: 1 put da sam pozvan
Ovo je: 2 put da sam pozvan
Ovo je: 3 put da sam pozvan
Ovo je: 4 put da sam pozvan
Ovo je: 5 put da sam pozvan
```

Sa modifikatorom static

```
#include <iostream>
using namespace std;
void counter() {
   static int count=0;
   cout << count++;
}
int main(){
   for(int i=0;i<5;i++)
      counter();
   cin.get();
   return 0;
}
```

0 1 2 3 4

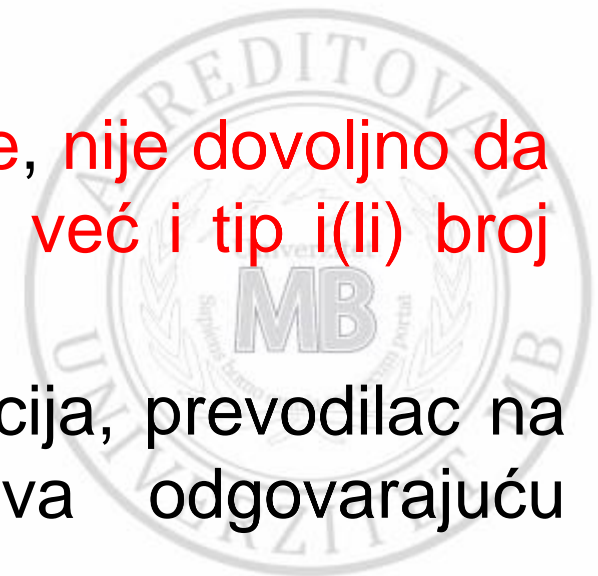
Bez modifikatora static

```
#include <iostream>
using namespace std;
void counter() {
    int count=0;
   cout << count++;
}
int main(){
   for(int i=0;i<5;i++)
       counter();
   cin.get();
   return 0;
}
```

0 0 0 0 0

Preklapanje funkcija

- U jeziku C++ mogu da postoje dve (ili više) funkcija istog imena, sa različitim brojem i(li) tipovima argumenata; to se zove preklapanje (preopterećivanje) funkcija (engl. function overloading)
  - da bi dve funkcije bile preklopljene, nije dovoljno da im se razlikuje samo povratni tip, već i tip i(li) broj argumenata moraju biti različiti
  - kada se poziva preklopljena funkcija, prevodilac na osnovu tipa argumenata poziva odgovarajuću funkciju



Preklapanje funkcija (2)

- Preklopljene funkcije predstavljaju zajednički interfejs za funkciju koja radi isto, ali sa različitim tipovima podataka. Ista imena treba koristiti samo za funkcije koje rade isto
- Prilikom preklapanja funkcija moguće je napraviti funkcije čije se telo donekle razlikuje zato što je funkcija prilagođena tipu parametara koje prihvata

```
int Double(int);
long Double(long);
float Double(float);
double Double(double);
```

Preklapanje funkcija – primer slajd 1

```
#include <iostream>
using namespace std;
int min (int a, int b);
char min (char a, char b);
int *min (int *a, int *b);
int main () {
   int i=9, j=3;
   cout<<min (i,j)<<endl;
   cout<<min ('X', 'a')<<endl;
   int * pointer=min (&i, &j);
   cout <<*pointer<<endl;
   cin.get();  return 0;
}
```

Univerzitet MB

Preklapanje funkcija – primer slajd 2

```
int min (int a, int b) {
  if (a>b)
  return b;
  else
    return a;
}
char min (char a, char b) {
   if (tolower (a)>tolower (b))
     return b;
   else
     return a;
}
```

```
int *min (int *a, int *b) {
  if (*a > *b)
  return    b;
  else
    return a;
}
```

```
3
a
3
```

Podrazumevane (engl. default) vrednosti argumenata

- Parametru funkcije može da se dodeli podrazumevana (engl. default) vrednost koja će se koristiti kada u pozivu funkcije nije naveden odgovarajući argument za taj parametar
  - default vrednosti mogu biti navedene samo jednom, i to prilikom prve deklaracije funkcije
  - svi parametri koji prihvataju default vrednosti moraju biti zadati sa desne strane parametara koji nemaju default vrednosti

Podrazumevane vrednosti argumenata (2)

```
void funkcija(int a=10, int b); //pogresno
```

```
void funkcija(int a = 10, int b)
Error: default argument not at end of parameter list
```

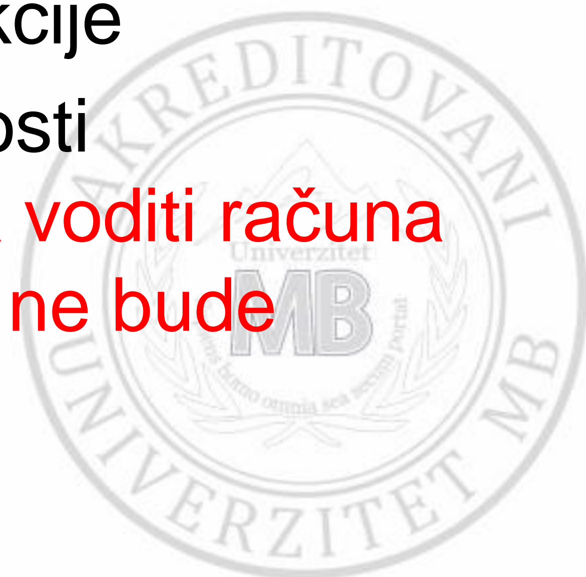
Podrazumevane vrednosti argumenata (3)

```
#include <iostream>
using namespace std;
int saberi (int =10, int=100);
int main () {
   cout<<saberi (20,30)<<endl;
   cout<<saberi (20)<<endl;
   cout<<saberi ()<<endl;;
   cin.get();
   return 0;
}
int saberi (int x, int y){
    return x+y;
}
```

```
50
120
110
```

Podrazumevane vrednosti argumenata (4)

- Default vrednosti argumenata koriste se za pojednostavljivanje poziva složenih funkcija ili kao “prečica” za preklopljene funkcije
- Prilikom zadavanja default vrednosti argumenata u kodu funkcije treba voditi računa da kada se argument ne navede, ne bude nikakvih posljedica



Podrazumevane vrednosti argumenata (5)

```
#include <iostream>
using namespace std;
int divide (int a, int b=2) {
int r;
  r=a/b;
  return (r);
}
int main () {
  cout << divide (12) << '\n';
  cout << divide (20,4) << '\n';
  cin.get();
  return 0;
}
```

```
6
5
```

Podrazumevane vrednosti argumenata (6)

```
#include <iostream>
using namespace std;
int sum(int x, int y=20) {
int result;
  result = x + y;
  return (result);
}
```

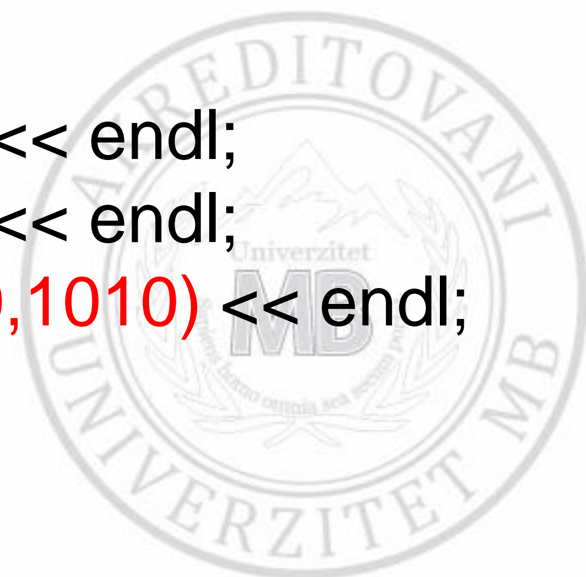
```
Ukupno :300
Ukupno :120
```

```
int main (){
    int x = 100;
    int y = 200;
    int r;
    r = sum(x, y);
    cout << "Ukupno :"  << r
            << endl;
    r = sum(x);
    cout << "Ukupno :" << r
            << endl;
    cin.get();
    return 0;
}
```

Funkcija inline

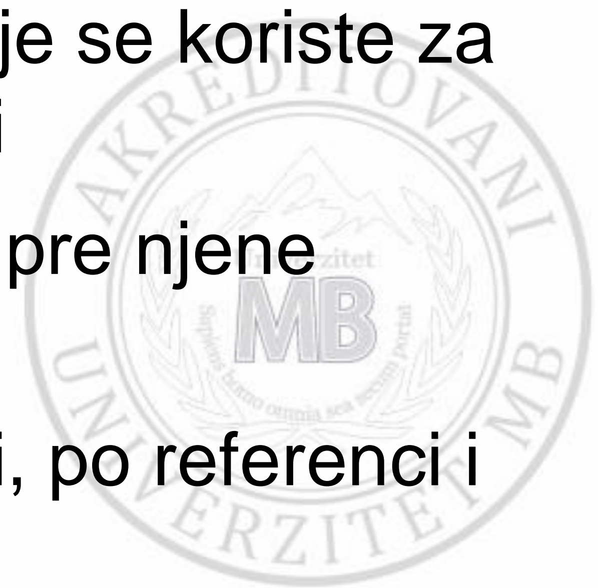
```
#include <iostream>
using namespace std;
inline int Max(int x, int y){
   return (x > y)? x : y;
}
int main( ){
   cout << "Max (20,10): " << Max(20,10) << endl;
   cout << "Max (0,200): " << Max(0,200) << endl;
   cout << "Max (100,1010): " << Max(100,1010) << endl;
   return 0;
}
```

Ako je funkcija inline, prevodilac stavlja primerak koda te funkcije na svakom mestu gde se funkcija zove u vreme prevođenja



Zaključak

- Funkcija je potprogram sa određenim brojem naredbi a izvodi neki zadatak
- povratni_tip ime(lista argumenata)
- Argumenti funkcije su vrednosti koje se koriste za njeno pozivanje – stvarni i formalni
- Prototip funkcije deklariše funkciju pre njene definicije
- Prenos argumenata – po vrednosti, po referenci i po pokazivaču



Zaključak (2)

- Povratak iz funkcije - return
- Oblast važenja (doseg – scope) promenljivih – lokalni i globalni
- Rekurzivne funkcije – pozivaju same sebe
- Statičke promenljive u funkcijama – pamte vrednosti između različitih poziva iste funkcije
- Preklapanje funkcija – 2 ili više funkcija istog imena sa različitim brojem i tipovima argumenata

Kraj prezentacije

HVALA NA PAŽNJI!