Funkcije

- zasebni blokovi naredbi koji se pišu prije ili poslije funkcije main()

Deklaracija funkcija:
- funkciju treba deklarirati prije prvog poziva
- sintaksa *(function prototype):* **<povratni_tip> ime_funkcije (<tip> arg_1, .. <tip> arg_n);**

gdje je:
<povratni_tip> - određuje kakav će biti tip podatka kojeg će funkcija vraćati pozivatelju naredbom **return**
- konkretna vrijednost koju funkcija vraća određuje se naredbom **return** u definiciji funkcije
- parametar naredbe **return** može biti: broj, varijabla ili izraz koji se izračunava prije završetka funkcije izuzev polja i funkcija (može vraćati pokazivače i reference na njih)
- parametar naredbe **return** po tipu moraju odgovarati povratnom tipu funkcije u deklaraciji
- ako funkcija ne vraća vrijednost pozivatelju definira se kao **void**

(<tip> arg_1, .. <tip> arg_n) - formalni podaci koji se predaju funkciji prilikom njenog poziva;
- njihov je broj proizvoljan, funkcija može biti i bez argumenata ili sa neodređenim brojem argumenata;

Broj argumenata, njihov redosljed i broj nazivaju se potpisom funkcije (*function signature*).

Definicija funkcije smješta se samo u jedan dio koda. Definicija funkcije mora oblikom u potpunosti odgovarati deklaraciji (tip funkcije, broj, redoslijed i tip argumenata); mogu se razlikovati imena argumenata. Svi pozivi funkcije će se prilikom povezivanja programa usmjeriti na tu definiciju.

Pr 1.
```
double kvadrat (float x);          // deklaracija

double kvadrat (float x) {         // definicija
        return x*x ;               // tijelo funkcije
}
```

Pr 1b.
```
double kvadrat (float);            // u deklaraciji se imena argumenata mogu izostaviti

double kvadrat (float x) {         // definicija
        return x*x ;               // tijelo funkcije
}
```

U definiciji funkcije implicitno je smještena i njena deklaracija, pa se njena deklaracija podrazumjeva ako se funkcija definira prije funkcije main(). Prema tome u jednostavnim programima moguće je i dovoljno definirati sve funkcije prije funkcije main(), dok će u složenijim programima deklaracija funkcije najčešće biti smještena u datoteci zaglavlja koje se predprocesorskom naredbom **#include** uključuju u sve datoteke izvornog koda u kojima se neka od deklariranih funkcije poziva..

Poziv funkcije

- nakon što je funkcija deklarirana, možemo ju pozvati iz bilo kojeg dijela programa navođenjem naziva funkcije sa odgovarajućom listom argumenata
- sintaksa: **ime_funkcije (arg_1a, .. arg_na);**
- prilikom poziva funkcije formalni argumenti zamjene se sa stvarnim argumentima tj. sa konkretnom vrijednošću. Pritom:
  1. broj argumenata u pozivu funkcije mora biti jednak broju argumenata u definiciji funkcije
  2. tipovi stvarnih argumenata moraju se podudarati sa tipovima odgovarajućih formalnih argumenata (istog tipa ili se konverzijom mogu svesti na tipove formalnih argumenata).
  3. imena formalnih argumenata ne moraju biti jednaka imenima stvarnih argumenata
- funkciju koj anije tipa **void** možemo koristiti u aritmetičkom izrazu s desne strane operatora pridruživanja
- funkcija može biti argument poziva funkcije (druge ili iste)

Pr 2.
```
#include <iostream.h>
double kvadrat (float);            // deklaraciji funkcije bez imena argumenata
int main() {
    cout << "Kvadrati ";
    for (int i=1; i<=10, i++)
    cout <<  i << " * " << i << " = " << kvadrat (i) << endl;    //poziv funkcije sa argumentom i
    return 0;
}
double kvadrat (float x) {        // definicija funkcije
        return x*x ;              // tijelo funkcije
}
```

Pr 3.
```
// u jednostavnijim programima dovoljno je definicije funkcije staviti ispred funkcije main()
#include <iostream.h>
double kvadrat (float x) {        // svaka definicija funkcije je ujedno i deklaracija funkcije
        return x*x ;
}
void main() {
    cout << "Kvadrati ";
    for (int i=1; i<=10, i++)
        cout <<  i << " * " << i << " = " << kvadrat (i) << endl;          //poziv funkcije sa argum. i
    cout << "Kubovi ";
    for (int i=1; i<=10, i++)
        cout <<  i << " * " << i << " = " << kvadrat (i)*i << endl;        //poziv funkcije kao arit. izraz
    cout << "Kvadrati kvadrata - četvrta potencija ";
    for (int i=1; i<=10, i++)                                  //poziv funkcije kao argument poziva fije
        cout <<  i << " * " << i << " = " << kvadrat( kvadrat (i)) << endl;
}
```

Pr 4.
```
#include <iostream.h>
double Pkrug (float);              // deklaraciji funkcije bez imena argumenata
void main() {
    cout <<  "Površina i opseg kruga \n"
    for (int i=1; i<=5, i++)       //poziv funkcije tretira se kao podatak tipa double
    cout <<  " za r = " <<  i << " \t P = " << Pkrug (i) << "\t O = " << Pkrug(i)/i*2 << endl;
}
double Pkrug (float r) {                          // definicija funkcije
        return r*r*3.1415 ;
}
```

Pr 5.
```
#include <iostream.h>
double Pkrug (float);              // deklaraciji funkcije bez imena argumenata
void main() {
    cout <<  "Površina i opseg kruga \n"
    for (int i=1; i<=5, i++)
        Pkrug (i) ;                //funkcija koja je tipa void ne može se koristiti u arit. izrazima
}
void Pkrug (float r) {             // definicija funkcije
const float pi = 3.1415f;
    cout <<  " za r = " <<  i << " \t P = " << r*r*pi << "\t O = " << 2*r*pi << endl;
    return ;
}
```

Prijenos argumenata

- po vrijednosti - (*by value*): formalni argument vrijedi samo u funkciji, pa se njegova vrijednost pri izlasku iz funkcije gubi - promjena vrijednosti argumenta u funkciji neće se odraziti na stvarni argument s kojim smo pozvali funkciju
- po referenci - (*by reference*): korištenjem referenci i pokazivača prilikom poziva funkcije kada se umjesto stvarnih vrijednosti argumenata funkciji prosljeduju adrese tih argumenata **mjenja** se vrijednost stvarnim argumentima u pozivnom dijelu programa; **parametri** koji se proslijeđuju **ne** mogu biti **konstante** jer konstante nemaju adresu; tipovi referenci (pokazivača) koji se proslijeđuju funkciji moraju točno odgovarati tipovima u deklaraciji funkcije jer **konverzija se ne provodi** !!!

Uvijek kada je moguće potrebno je izbjegavati pokazivače i reference kao argumente funkcija, odnosno preporuča se funkciju pozivati po vrijednosti. Upotreba pokazivača i referenci neizbježna je kada funkcija treba istovremeno promjeniti dva ili više objekata ili kada joj se moraju prenjeti velike strukture podataka kao npr polja.

Prijenos argumenata po vrijednosti

Pr. // **područje važenja varijabli**

```
#include <iostream.h>
void mojaF();
int main() {
        int x = 5;
        cout << "\n u main x: " << x;
        mojaF();
        cout << "\nPonovno u main, x je: " << x;
        return 0;
}
void mojaF() {
        int x = 8;
        cout << "\n u mojaF, localni x: " << x << endl;
        {                       cout << "\n unutar bloka u funkciji, x je: " << x;
                                int x = 9;
                                cout << "\n Lokalni x bloka unutar funkcije: " << x;
        }
        cout << "\n Izvan bloka u funkciji x: " << x << endl;
}
```

Pr. 1.

```
#include <iostream.h>
float koef () {
        int x1, y1, x2, y2;             // lokalne varijable stvoriti će se na stogu
        cout << " Koordinate prve točke: \n\t x1 = \t";
        cin >> x1;
        cout << "\n \t y1 = \t";
        cin >> y1;
        cout << " \n Koordinate druge točke: \n\t x2 = \t";
        cin >> x2;
        cout << "\n \t y2 = \t";
        cin >> y2;
        return (y2-y1)/(x2-x1);         //izračunati rezultat funkcija pohranjuje na stog
}
void  main() {
        cout << " Koeficijent pravca ";
        koef();                 // prilikom izvršavanja pozvane funkcije svi podaci na stogu se gube
```

```
        float k = koef();          // rezultat funkcije neće se izgubiti ako se sadržaj stoga prilikom poziva
        cout << k;                 // funkcije pohrani na mjesto varijable
        cout << koef();
}
```

Pr.2.

```
#include <iostream.h>
void Inicijal () {
          int a, d, n;
          cout << " Upisi početni element: \t";
          cin >> a;
          cout << "\n Diferencija je: \t";
          cin >> d;
          cout << "\n Upisi broj elemenata niza: \t";
          cin >> n;
}                                  // deklaracija lokalnih varijabli a,d i n pri izlasku iz funkcije se gubi
void  main() {
        Inicijal ();
        cout << "\n Elementi niza ";
        for (int i=0; i<n; i++)            // GREŠKA !! varijable n,a i d nisu deklarirane !!
          cout <<  "a"<< i << " = " << a+i*d << endl;
}
```

Pr. 3.

```
#include <iostream.h>
void Inicijal () {
          cout << " Upisi početni element: \t";
          cin >> a;
          cout << "\n Diferencija je: \t";
          cin >> d;
          cout << "\n Upisi broj elemenata niza: \t";
          cin >> n;
}                                          // vrijednost varijabli a,d i n pri izlasku iz funkcije se gubi
void  main() {
        int a, d, n;
        Inicijal ();
        cout << "\n Elementi niza ";
        for (int i=0; i<n; i++)            // GREŠKA !! varijable n,a i d nisu inicijalizirane !!
          cout <<  "a"<< i << " = " << a+i*d << endl;
}
```

Pr.4.

```
#include <iostream.h>
int Inicijal (char x[]) {
          int a;
          cout << "\n Upisi " << x <<"\t";
          cin >> a;
          return a;                // upotreba naredbe return omogućuje povrat vrijednosti pozivatelju
}
void  main() {
          cout << " Aritmetički niz \n ";
          int a, d, n;
          a= Inicijal ("pocetni element niza \t a = ");     //pridruživanjem povratne vrijednosti varijabli
          d= Inicijal ("diferenciju \t d = ");              //će biti dodjeljena vrijednost
          n= Inicijal ("broj elemenata niza \t n = ");
```

```
        cout << "\n Elementi niza ";
        for (int i=0; i<n; i++)
              cout <<  "a"<< i << " = " << a+i*d << endl;
}
```

Deklaraciji funkcije sa više argumenata i sa više naredbi return

Pr.7.

```
        #include <iostream.h>
        float Pravokutnik (int,float,float);                    //deklaracija fije sa 3 argumenta
        float Init(char);
        void main() {
        cout <<  "Površina ili opseg pravokutnika \n";
        cout <<  " Odaberi 1- Površina, 2- Opseg \t Izbor: \t";
        int izb;
        cin >> izb;
        float a = Init('a');                          // poziv funkcije kojoj je argument tipa char  tj. 'a',
        float b = Init('b');                          // fija vraća vrijednost tipa float
        cout << Pravokutnik(izb,a,b);
        }
        float Init(char x) {
            float y;                                   // y je lokalna varijabla
            cout <<  " stranica " << x << " =  \t";
            cin >> y;
            return y;
        }

        float Pravokutnik (int i, float a, float b) {                    // definicija funkcije sa 3 argumenta
        switch (i) {
                case 1:
                        cout <<  " \n P = ";
                        return a*b;
                case 2:
                        cout <<  " \n O = ";
                        return 2*(a+b);
                default:
                        cout << "krivi odabir";
                        return 0;
                }
        }
```

Korištenje datoteka zaglavlja

Pr.7a. Funkcije iz zadatka 7. deklarirane /definirane korištenjem datoteka zaglavlja

```
#include <iostream.h>
#include "Pravokutnik.h"    // zaglavlje Pravokutnik.h procesor će najprije tražiti u tekućem direktoriju
#include "Init.h"
void main() {
        cout <<  "Površina ili opseg pravokutnika \n";
        cout <<  " Odaberi 1- Površina, 2- Opseg \t Izbor: \t";
        int izb;
        float a, b;
        cin >> izb;
        a = Init('a');
        b = Init('b');
        cout << Pravokutnik(izb,a,b);
}

// Deklaracija funkcije Pravokutnik u datoteci zaglavlja Pravokutnik.h
float Pravokutnik (int,float,float);

// Funkcija Pravokutnik pohranjena je u zasebnoj izvornoj (source) datoteci Pravokutnik.cpp
#include <iostream.h>
float Pravokutnik (int i, float a, float b) {
        switch (i) {
        case 1:
                cout <<  " \n P = ";
                return a*b;
        case 2:
                cout <<  " \n O = ";
                return 2*(a+b);
        default:
                cout << "krivi odabir";
        return 0;
        }
}

// Definicija funkcije Init u datoteci zaglavlja Init.h
#include <iostream.h>
float Init(char x) {
        float y;
        cout <<  " stranica " << x << " =  \t";
        cin >> y;
        return y;
}
```

Prijenos argumenata po njihovoj adresi (reference, pokazivači)

Pr. 5.

```
// upotreba referenci omogućuje promjenu vrijednosti stvarnim argumentima u pozivnom kodu
#include <iostream.h>
void Inicijal (int &a, int &d, int &n) {
        cout << " Upisi početni element: \t";
        cin >> a;
        cout << "\n Diferencija je: \t";
        cin >> d;
        cout << "\n Upisi broj elemenata niza: \t";
        cin >> n;
}
void  main() {
        int a, d, n;
        Inicijal (a,d,n);
        cout << "\n Elementi niza ";
        for (int i=0; i<n; i++)
             cout <<  "a"<< i << " = " << a+i*d << endl;
}
```

Pr. 6.

```
// primjer neuspješne zamjene jer se varijable prenose po vrijednosti
#include <iostream.h>
void zamjena(float a, float b) {
        cout << " Zamjena. Prije zamjene, a: " << a << " b: " << b << "\n";
        float p = a;
        a = b;
        b = p;
        cout << " Zamjena. Nakon zamjene, a: " << a << " b: " << b << "\n";
}
void main() {
        cout <<  "Zamjeni \n";
        cout <<  "\n Upiši prvi broj\t x = \t";
        float x, y;
        cin >> x;
        cout <<  "\n Upiši drugi broj\t y = \t";
        cin >> y;
        cout << "Main. Prije zamjene, x: " << x << " y: " << y << "\n";
        zamjena(x,y);
        cout << "Main. Nakon zamjene, x: " << x << " y: " << y << "\n";
}
```

Pr. 7. // **upotreba referenci**

```
#include <iostream.h>
void zamjena(float &a, float &b) {
        cout << " Zamjena. Prije zamjene, a: " << a << " b: " << b << "\n";
        float p = a;
        a = b;
        b = p;
        cout << " Zamjena. Nakon zamjene, a: " << a << " b: " << b << "\n";
}
```

```
void main() {
        cout <<  "Zamjeni \n";
        cout <<  "\n Upiši prvi broj\t x = \t";
        float x, y;
        cin >> x;
        cout <<  "\n Upiši drugi broj\t y = \t";
        cin >> y;
        cout << "Main. Prije zamjene, x: " << x << " y: " << y << "\n";
        zamjena(x,y);           //funkcija se poziva argumentima, a ne njihovim adresama
        cout << "Main. Nakon zamjene, x: " << x << " y: " << y << "\n";
}
```

Pr. 8. // **upotreba pokazivača**

```
#include <iostream.h>
void zamjena(float *a, float *b) {      // umjesto vrijednosti funkciji se prosljeđuju pokazivači na varijable;
                                        // formalni argumenti a i b sadržavati će kopije pokazivača koje
                                       // pokazuju na istu adresu na kojoj pokazuju i stvarni argumenti
        cout << "zamjena. Prije zamjene, *a: " << *a << " *b: " << *b << "\n";
        float p = *a;
        *a = *b;                //zamjena se provodi na adresama na koje pokazivači (a i b) pokazuju
        *b = p;
        cout << "zamjena. Nakon zamjene, *a: " << *a << " *b: " << *b << "\n";
}
void main() {
        cout <<  "Zamjeni \n";
        cout <<  "\n Upiši prvi broj\t x = \t";
        float x, y;
        cin >> x;
        cout <<  "\n Upiši drugi broj\t y = \t";
        cin >> y;
        cout << "Main. Prije zamjene, x: " << x << " y: " << y << "\n";
        zamjena(&x,&y);                 //prilikom poziva prosljeđuju se adrese argumenata
        cout << "Main. Nakon zamjene, x: " << x << " y: " << y << "\n";
}
```

Promjena pokazivača unutar funkcija

- pokazivača koji je proslјeđen kao parametar funkcije prenosi se po vrijednosti tj. promjene na parametru (u ovom slučaju pokazivaču) unutar funkcije ne utječu na stvarni parametar naveden u pozivajućem kodu.

```
#include <iostream.h>
void Upis(char *im) {                   // vrijednost pokazivača ime kopira se u privremenu var im
        im = new char[100];             // funkcija radi sa privremenom vrijednosti im a ne sa
        cin >> im;                      //sadržajem varijable ime
}                                       //kad funkcija završi vrijednost varijable im se gubi, a
                                        //vrijednost var korisnik je nepromjenjena
void main() {
        cout <<  "Unesi ime \n";
        char *ime;                      // pokazivač ime na tip char
        Upis(ime);                      //loš poziv; // prenosi se po vrijednosti
        cout <<  "\n Pozdrav, " << ime << endl;
        delete [] ime;
}
```

Pr. 1 // **Promjena pokazivača unutar funkcija korištenjem pokazivača na pokazivače**

```
#include <iostream.h>
void Upis(char **im) {                  // parametar je pokazivača na pokazivač na znak
        *im = new char[100];            // vrijednosti pokazivača ime pristupa se preko
        cin >> *im;                     //pokazivača *im i na taj način ju se može mijenjati
}

void main() {
        cout <<  "Unesi ime \n";
        char *ime;                      //ime je pokazivač na znak
        Upis(&ime);                     // prosljeđuje se adresa pokazivača ime
        cout <<  "\n Pozdrav, " << ime << endl;
        delete [] ime;
}
```

Pr.1a // **Promjena pokazivača unutar funkcija korištenjem pokazivača i referenci**

```
#include <iostream.h>
void Upis(char *&im) {                  // parametar je referenca na pokazivač
        im = new char[100];             //stvarnoj varijabli pristupa se preko reference
        cin >> im;
}

void main() {
        cout <<  "Unesi ime \n";
        char *ime;                      //ime je pokazivač na znak
        Upis(ime);              // prosljeđuje se adresa koja se kod referenci ne navodi eksplicitno
        cout <<  "\n Pozdrav, " << ime << endl;
        delete [] ime;
}
```

Polja kao argumenti

- polja se, iako tako na prvi pogled izgleda, ne prenose po vrijednosti već se prenosi pokazivač na njegov prvi član pa se preko tog pokazivača rukuje sa stvarnim vrijednostima elemenata polja

Ravnopravne deklaracije:

```
void Nuliraj(int polje[], int duljina);        void Nuliraj(int polje[]);
void Nuliraj(int polje[3], int duljina);       void Nuliraj(int polje[3]);
void Nuliraj(int *polje, int duljina);         void Nuliraj(int *polje);
```

Prilikom deklaracije nije potrebno navesti duljinu polja. Ona se može navesti kao dodatni argument kada se želi pozvati funkciju za polje proizvoljne duljine.

Pr.2

```
#include <iostream.h>
void Nuliraj(int polje[], int duljina) {          // prenosi se samo pokazivač na prvi član
        while (duljina--)
          polje [duljina] = 0;
}
void main() {
          int p[] = {3,6,9};
          Nuliraj(p,3);
          for (int i = 0; i<3; i++)
          cout <<  p[i] << " \n";
}
```

Pr.2a. //ravnopravna definicija funkcije

```
#include <iostream.h>
void Nuliraj(int *polje, int duljina) {           // upotreba pokazivača na polje
          while (duljina--)
            *(polje++) = 0;
}
void main() {
          int p[] = {3,6,9};
          Nuliraj(p,3);
          for (int i = 0; i<3; i++)
          cout <<  p[i] << " \n";
}
```

Znakovni niz je isto polje pa i kod njih treba prenjeti pokazivač na prvi član

Pr.3.

```
#include <iostream.h>
int Duljina(char *rj) {                           // upotreba pokazivača na znakovni niz
          int i = 0;
          while (*(rj+i))
            i++;
          return i;
}
void main() {
          char *rijec = new char[20];
          cin >> rijec;
          cout << "duljina rijeci je " << Duljina(rijec);
}
```

Višedimenzionalana polja – nizovi jednodimenzionalnih polja

- prva dimenzija polja nije bitna za pronalaženje određenog člana u polju pa ju se može isključiti iz deklaracije parametara

Ravnopravne deklaracije za višedimenzionalno polje čije su dimenzije poznate:

```
void ispis(float m[redak][stupac]);
void ispis(float m[][stupac]);              //radi za proizvoljno redaka, broj stupaca mora biti poznat
void ispis(float m[][stupac], int redak);   //broj redaka prenesen kao zasebni argument
```

Pr.4. // vise dimenzionalno polje kojem su poznate dimenzije

```
#include <iostream.h>
#include <iomanip.h>

const int redak = 2;                        //varijable redak i stupac deklarirane ispred funkcije ispis() i
const int stupac = 3;                       //main() pa su dohvatljive iz obiju funkcija

void ispis(float m[redak][stupac]) {        // prijenos argumenata preko pokazivača na prvi član polja
        for (int r = 0; r < redak; r++) {
            for (int s = 0; s < stupac; s++)
                  cout << setw(10) << m[r][s];
            cout << endl;
        }
}

void main() {
        float mat [redak][stupac] = { {1.1f,1.2f,1.3f},  //dimenzije polja zadane u izvornom kodu
                                      {2.1f,2.2f,2.3f}};
        ispis(mat);                         // matrica se prenosi preko pokazivača na prvi član
}
```

Pr. 4a.

```
#include <iostream.h>
#include <iomanip.h>
const int redak = 5;
const int stupac = 3;
void ispis(float m[][stupac], int redak) {     // više dim polja = nizovi jednodim polja, prva dim. je nebitna
        for (int r = 0; r < redak; r++) {
              for (int s = 0; s < stupac; s++)
                    cout << setw(10) << m[r][s];
              cout << endl;
        }
}

void main() {
        float mat [redak][stupac] = { {1.1f,1.2f,1.3f},  {2.1f,2.2f,2.3f}};
        ispis(mat, 5);                      // raditi će za proizvoljan broj redaka pa se broj redaka prenosi
}                                           //kao posebni argument
```

Ravnopravne deklaracije za višedimenzionalno polje čije se dimenzije odabiru u izvornom kodu :

```
void ispis(float **m, int stupac, int redak);      //dohvaćanje preko pokazivača na pokazivače
```

Pr. 4b.

```
#include <iostream.h>
#include <iomanip.h>
```

```
void ispis(float **m, int redak, int stupac) {// pristup el. polja preko pokazivača na pokazivače
        for (int r = 0; r < redak; r++) {
            for (int s = 0; s < stupac; s++) {
                ((float*)m)[r*stupac+s] = float(r+s);
                cout << setw(10) << ((float*)m)[r*stupac+s];
            }
            cout << endl;
        }
}

void main() {
        int redak = 5;
        const int stupac = 3;                    //broj stupaca mora biti poznata u tijeku prevođenja koda
        float (*mat) [stupac] = new float [redak][stupac];
        ispis((float**)mat, redak, stupac);
}
```

Pokazivači i reference kao povratne vrijednosti

- kao rezultat funkcije ne smije se vraćati referenca ili pokazivač na lokalni objekt generiran unutar funkcije, već se kao parametar prosljeđuje pokazivač na varijablu u koju se želi pohraniti rezultat funkcije.
- ukoliko funkcije vraća više vrijednosti pozivnom kodu, funkcija mora biti pozvana sa argumentima koji su reference varijabli u koje se želi pohraniti rezulate funkcije

Pr. // **Vraćanje većeg broja podataka iz funkcije**

```
#include <iostream.h>
short faktor(int n, int* pkvad, int* pkub);
void main(){
        int broj, kvad, kub;
        short greška;
        cout << "Upiši broj (0 - 20): ";
        cin >> broj;
        greška = faktor(broj, &kvad, &kub);
        if (!greška)      {
            cout << "broj: " << broj << "\n";
            cout << "kvadrat: " << kvad << "\n";
            cout << "kub: "  << kub   << "\n";
        }
        else
            cout << "Greška!!\n";
}

short faktor(int n, int *pkvad, int *pkub){
        short v = 0;
        if (n > 20)
          v = 1;
        else {
            *pkvad = n*n;
            *pkub = n*n*n;
            v = 0;
        }
        return v;
}
```

Konstantni argument

- argumenti funkcije koji se prenose po vrijednosti ne mijenjaju stvarne argumente u pozivajućem kodu, ali one podatke koji se na taj način ne mogu prenjeti (npr. polja) možemo zaštititi od promjena deklarirajući ih kao konstantne parametre

- u listu parametara funkcije umetne se modifikator **const** ispred parametara prenjetog po referenci koji ne mijenja vrijednost u fukciji:

deklaracija: povratni_tip naziv_fije (**const** tip_1 arg_1, **const** tip_2 arg_2, tip_3 arg_3 ….);

- prilikom pokušaja promjene vrijednosti konstantnog parametara u pozvanoj funkciji kompajler će javiti grešku.

Pr.1

```
#include <iostream.h>
void Nuliraj(const int polje[], int duljina) {
        while (duljina--)
          polje [duljina] = 0;                  // GREŠKA!! nemoguće promjeniti vrijedost konstantnom argumentu
}
void main() {
          int p[] = { 3,6,9};
          Nuliraj(p,3);
}
```

Podrazumjevani argumenti

- ukoliko se prilikom poziva funkcije izostavi stvarni argument, formalni argument biti će inicijaliziran na podrazumjevanu vrijednost (*default argument value*) koja je zapisana u **deklaraciji** funkcije:

povratni_tip naziv_fije (tip1 arg1, tip2 arg2, **tip3 arg3=a**, **tip4 arg4=b**);

- podrazumjevane vrijednosti moraju se navesti u deklarciji funkcije **na kraju liste argumenata**, ako su deklaracija i definicija funkcije odvojene, podrazumjevane vrijednosti navode se samo u deklaraciji funkcije, a u definiciji se ne smiju ponavljati..

Pr.1 // korištenje podrazumjevanih (default) vrijednosti parametara

```
#include <iostream.h>
int Kvadar(int duljina, int sirina = 25, int visina = 1);
void main() {
          int duljina = 100, sirina = 50, visina = 2;
          int P;
          P = Kvadar(duljina, sirina, visina);
          cout << "Površina 1: " << P << "\n";
          P = Kvadar(duljina, sirina);          // koristiti će se podrazumijevana vrijednost 1 za visinu
          cout << "Površina 2: " <<P << "\n";
          P = Kvadar(duljina);                  // koristiti će se podrazumijevana vrijednost 25 i 1
          cout << "Površina 3: " << P << "\n";
}

Kvadar(int duljina, int sirina, int visina){
          return (duljina * sirina * visina);
}
```

Pr.2.

```
#include <iostream.h>
int suma (int n, int p = 1) {
```

```
        for (int i=p, sum =0; i<=n; sum+=i++);
        return sum;
}
void main() {
        int a,b;
        cout << "Suma brojeva u intervalu \n ";
        cout <<" a = \t";
        cin>> a;
        cout <<" b = \t";
        cin>> b;
        cout << " \n suma prvih " << b << "brojeva  je " << suma(b);
        cout << "\n  suma u intervalu (" << a << "," << b << ") je " << suma(b,a);
}
```

Globalne varijable

- objekti koji trebaju biti dohvaćeni iz više različitih funkcija deklariraju se kao globalni; treba ih koristiti samo kada je to opravdano

- varijabla je globalna ako se deklarira izvan tijela funkcije
- ako nije eksplicitno navedeno, inicijalizirana je na vrijednost 0
- globalna varijabla vidljiva je u svim funkcijama čije definicije slijede u datoteci izvornog koda
- globalna varijabla čija se vrijednost ne smije mijenjati deklarira se kao konstanta (**const**)
- lokalna varijabla funkcije koja ime isto ime kao i globalna nadjačati će globalnu varijablu, globalnoj varijabli u tom slučaju pristupa se pomoću operatora **:: (operator za određivanje područja)**
- sještajna klasa: - vanjska (u deklaraciji ključna riječ **extern** ispred naziva objekta koji je definiran u drugom modulu)
  - statička (u deklaraciji ključna riječ **static** ispred naziva objekta koji može biti korišten samo unutar jednog modula)

Pr. 1.

```
#include <iostream.h>
int x1, y1, x2, y2;                 // globalne varijable
void init() {
        cout << " Koordinate prve točke: \n\t x1 = \t";
        cin >> x1;
        cout << "\n \t y1 = \t";
        cin >> y1;
        cout << " \n Koordinate druge točke: \n\t x2 = \t";
        cin >> x2;
        cout << "\n \t y2 = \t";
        cin >> y2;
}
float koef() {
        return (y2-y1)/(x2-x1);  //funkcija koristi globalne varijable pa se ne prenose kao argumeti
}
void supr() {
        x1 = -x1;                   //u funkciji se mijenja vrijednost lokalnim varijablama
        x2 = -x2;
}
void  main() {
        cout << " Koeficijent pravca ";
        init();
```

```
        cout << koef();
        supr();
        cout << 1/koef();
}
```

Pr.2.

```
#include <iostream.h>
void mojaFunk();          // deklaracija

int x = 5, y = 7;         // globalne variable
int main(){
        cout << "x u main: " << x << "\n";
        cout << "y u main: " << y << "\n\n";
        mojaFunk();
        cout << "Povratak iz funkcije mojaFunk!\n\n";
        cout << "x u main: " << x << "\n";
        cout << "y u main: " << y << "\n";
        return 0;
}
void mojaFunk() {
        int y = 10;
        x++;
        cout << "x u funkciji mojaFunk: " << x << "\n";
        cout << "y u funkciji mojaFunk: " << y << "\n\n";
}
```

Pr.3.

```
#include <iostream.h>
int x = 10;                         // globalne varijable

void  main() {
        float x = 25.5f;                    //lokalna varijabla nadjačala je globalnu
        cout << "\n  x = " << x;            // ispis vrijednosti lokalne varijable
        cout << "\n  ::x = " << ::x;        // ispis vrijednosti globalne varijable
}
```

Statički objekti /varijable

- varijable koje se trebaju inicijalizirati samo prvi put prlikom poziva funkcije, a zatim da se njihova vrijednost čuva, slično globalnoj varijabli samo što nije dohvatljiva od drugih objekata iz drugih funkcije
- varijabla je statička dodavanjem ključne riječi **static** ispred oznake tipa
- statična varijabla živi tokom cijelog izvođenja programa, ali dostupna je samo unutar funkcije u kojoj je deklarirana

Pr.1.

```
#include <iostream.h>

void Loto() {
        static bool Dobitak =false;          // statički objekt inicijalizira se samo prilikom prvog poziva
        Dobitak = !Dobitak;
        if (Dobitak)
        cout << "\n Dobitak!";
        else
        cout << "\n Gubitak!";
```

```
}
void  main() {
        int i = 5;
        while(i--)
        Loto();
}
```

Umetnute funkcije (*inline function*)

- funkcijama koje imaju kratko tijelo može se dodavanjem ključne riječi **inline** dati prvoditelju naputak da svaki poziv funkcije jednostavno zamijeniti samim kodom funkcije;

najčešće se tako definiraju funkcije koje vraćaju vrijednost ili referencu

- definicija umetnute funkcije mora predhoditi prvom pozivu funkcije (deklaracija funkcije nije dovoljna je nailaskom na prvi poziv funkcije mora znati čime će ga nadomjestiti)
- moguće je ubrzati izvođenje, ali obzirom da se nadomještanjem poziva funkcije njenim tijelom povećava duljina izvedbenog koda produljuje se njeno prevođenje

Pr. 1.

```
#include <iostream.h>
inline float recip (int n) {
        return -1.f/n;
}

void main() {
        int a;
        cout <<" a = \t";
        cin>> a;
        cout << "Recipročni broj broja \t "<<a << " je " << recip(a);
}
```

Pr.2.

```
#include <iostream.h>
inline int Dvostruki(int);

int main(){
        int broj;
        cout << "Upiši broj: ";
        cin >> broj;
        cout << "\n";
        broj = Dvostruki(broj);
        cout << "Broj: " << broj << endl;
        broj = Dvostruki(broj);
        cout << "Broj: " << broj << endl;
        broj = Dvostruki(broj);
        cout << "Broj:: " << broj << endl;
        return 0;
}

int Dvostruki(int broj){
        return 2*broj;
}
```

Preopterećivanje funkcija

- ako tip stvarnog argumenta u pozivu funkcije ne odgovara tipu navedenom u deklaraciji funkcije provodi se konverzija tipa stvarnog argumenta tj. tip stvarnog argumenta svodi se na tip formalnog

Pr.

```
#include <iostream.h>
double Dvostruki(double);
void main(){
    int     i = 6500; long    l = 65000;
    float   f = 6.5F; double  d = 6.5e20;
    int 2_i = Dvostruki(i);          //konverzija tipa stvarnog argumenta (int) na tip formalnog (double),
                                     a zatim konverzija rezultata funkcije na tip varijable 2_i (int)
    long 2i_l = Dvostruki(l);        //konverzija tipa stvarnog argumenta (long) na tip formalnog (double)
                                     a zatim konverzija rezultata funkcije na tip varijable 2_l (long)
    float 2_f = Dvostruki(f);        //konverzija tipa stvarnog argumenta (float) na tip formalnog (double)
                                     a zatim konverzija rezultata funkcije na tip varijable 2_f (float)
    double 2_d = Dvostruki(d);
    cout << "dvostruki  " << i << " je  "<< 2_i << "\n";
    cout << "dvostruki  " << l << " je  "<< << 2_l << "\n";
    cout << "dvostruki  " << f << " je  "<< 2_f << "\n";
    cout <<"dvostruki  " << d << " je  "<< 2_d << "\n";
}

double Dvostruki(double original){       //unutar funkcije barata se sa tipom double
    return 2 * original;                 //i rezultat funkcije je tipa double
}
```

Izbjegavanje suvišnih konverzija koje ne utječu na točnost rezultata i produljuju poziv i izvođenje funkcije moguće je deklariranjem funkcija sa odgovarajućim tipom argumenata – u tu svrhu se koriste preopterećene funkcije – funkcije <u>istog imena ali različitim brojem i/ili tipom argumenta</u>

Pr.// nadjačavanje funkcije

```
#include <iostream.h>

int Dvostruki(int);
long Dvostruki(long);
float Dvostruki(float);
double Dvostruki(double);

void main(){
        int     i = 6500; long    l = 65000;
        float   f = 6.5F; double  d = 6.5e20;
        int 2_i = Dvostruki(i);
        long 2i_l = Dvostruki(l);
        float 2_f = Dvostruki(f);
        double 2_d = Dvostruki(d);
    cout << "dvostruki  " << i << " je  "<< 2_i << "\n";
    cout << "dvostruki  " << l << " je  "<< << 2_l << "\n";
    cout << "dvostruki  " << f << " je  "<< 2_f << "\n";
    cout <<"dvostruki  " << d << " je  "<< 2_d << "\n";
}
int Dvostruki(int original){
        return 2 * original;
}
```

```
long Dvostruki(long original){
        return 2 * original;
}
float Dvostruki(float original){
        return 2 * original;
}
double Dvostruki(double original){
        return 2 * original;
}
```

Ukoliko ne postoji odgovarajuća funkcija prevoditelj provodi odgovarajuću konverziju tipa stvarnog parametra pridržavajući se pravila pridruživanja:

1. traženje funkcije kod koje postoji egzaktno slaganje parametara (bez konverzije tipova)
2. traži se funkcije kod koje treba prevesti samo cjelobrojnu konverziju (char u int, short u int, enumerated tipovi u int) i float u double
3. traži se funkcije kod koje treba prevesti standardnu konverziju (npr. int u double,…)

Ukoliko ima više funkcija koje odgovaraju uvjetu, poziv se proglašava neodređenim i prevoditelj prijavljuje grešku.
Kod funkcija sa više argumenata postupak nalaženja odgovarajuće funkcije provodi se za svaki argument.

- funkcije se **mogu** preopterećivati funkcijama koje imaju jednak povratni tip rezultata

npr. **double kvadrat**(float);
double kvadrat(double); //jednak je povratni tip, ali tip argumenata je različit

- funkcije se **ne mogu** preopterećivati po povratnom tipu:

npr. float **kvadrat**(**float**);
double **kvadrat**(**float**); //kompajler će javiti grešku jer su tipovi argumenata obiju f_ija jednaki

- funkcije se **ne mogu** preopterećivati ako se razlikuju samo u podrazumjevanim argumentima

npr. int **kvadrat**(**int i=1**);
long **kvadrat**(**int**); //jednaki su tipovi argumenata

- funkcije se **ne mogu** preopterećivati ako se razlikuju samo po vrijednosti ili referenci na nju

npr. int **kvadrat**(**int i**);
long **kvadrat**(**int &ref_i**); //imaju jednake inicijalne vrijednosti – nedovoljna razlika

Napomena: Osim funkcija moguće je opterećivati i operatore. Operatori su definirani za sve ugrađene tipove podataka, mogu se preopterećivati jedino na korisnički definiranim tipovima. Npr operator + primjenjen na podacima tipa int daje vrijednost int, a ako je primjenjen npr. na varijablama tipa float daje vrijednost tipa float.

Predlošci funkcija

Ukoliko su algoritni funkcija za različiti tip podataka potpuno identični, umjesto preopterećivanja funkcija daleko praktičnije je korisiti predloške funkcija (eng. *function templates*) koji omogućavaju da se jednom definirani kod prevede više puta za različite tipove argumenata.
Deklaracija:

template <arg1_ pred, arg2_ pred,..> deklaracija funkcije(arg1_pred, arg2_pred,..);

svaki argument predloška sastoji se od ključne riječi class i imena argumenta i mora se pojaviti u listi parametara funkcije:
arg1_pred … <class ime_arg1>
arg2_pred… <class ime_arg2>

Pr. umjesto preopterećivanja funkcije Dvostruki koristi se predložak:

```
template <class Tip>
Tip Dvostruki(Tip original){                    //funkcija Dvostruki je parametrizirana tako da se
        return 2 * original;                    //tip argumenta i tip fije mogu po potrebi mijenjati
}

#include <iostream.h>
void main(){
      int     i = 6500; long     l = 65000;
      float    f = 6.5F; double   d = 6.5e20;
      int 2_i = Dvostruki(i);                   //generira se oblik fije: int Dvostruki(int);
      long 2i_l = Dvostruki(l);                 // generira se oblik fije: long Dvostruki(long);
      float 2_f = Dvostruki(f);                 // generira se oblik fije: float Dvostruki(float);
      double 2_d = Dvostruki(d);                // generira se oblik fije: double Dvostruki(double);
      cout << "dvostruki  " << i << " je  "<< 2_i << "\n";
      cout << "dvostruki  " << l << " je  "<< << 2_l << "\n";
      cout << "dvostruki  " << f << " je  "<< 2_f << "\n";
      cout <<"dvostruki  " << d << " je  "<< 2_d << "\n";
}
```

Rekurzije

- neovisnost lokalnih varijabli funkcije od programa izvan funkcije omogućuje da funkcija poziva samu sebe.
- postupak uzastupnog pozivanja funkcije iz funkcije = **rekurzija**

Pr.

```
#include <iostream.h>
int fib (int n);

int main(){
        int n, odg;
        cout << "Odaberi broj: ";
        cin >> n;
        odg = fib(n);
        cout << odg << " je " << n << ". Fibonaccijev broj\n";
        return 0;
}

int fib (int n){
        cout << "\n\n Obrada fib(" << n << ")... ";
        if (n < 3 ) {
            cout << "Vraća 1!\n";
            return (1);
        }
        else {
            cout << "Poziv fib(" << n-2 << ") i fib(" << n-1 << ").\n";
            return( fib(n-2) + fib(n-1));
        }
}
```

Deklariranje klasa

```
class Ime_klase {                                   // klasa može imati i samo varijable ili samo metode
    tip_podatka var_instance_1;
                …                                   // var instance mogu biti deklarirane bilo gdje u klasi,
    tip_podatka var_instance_n;                     // samo ne u metodi jer tada više nisu varijable instanci

    tip_podatka metoda_1 (parametri) {              // tip podatka - tip koji će metoda vračati sa return u dio
        // tijelo metode                            // programa koji ju je pozvao ili void ako ne vrača ništa
    }
                ….
    tip_podatka metoda_n (parametri) {              // parametri - popis svih varijabli i njihovih tipova koji
        // tijelo metode                        // se metodi dostavljaju prilikom poziva
    }                                               // parametri = tip1 var1, tip2 var2, tip3 var3,…
};
```

Npr.

```
*       class Pravokutnik {     // deklaracija klase Pravokutnik, stvaranje novog tipa podataka: Pravokutnik
            double visina;      // deklaracija varijabli instance koje se mogu korisitit u čitavoj klasi, ako nije
            double sirina;      // ako nije naveden tip pristupa podrazumjeva se da je private
        };                      // točka zarez stavlja se na kraju definicije klase

*       class Pravokutnik {             // klasa ima definiranu metodu Povrsina()
        private:
            double visina;
            double sirina;
        public:
            double povrsinaPrav() {     // deklaracija metode bez parametara sa povratnim tipom double
                double povrsina;        // lokalna varijabla metode vrijedi samo u metodi
                povrsina = visina*sirina;       // varijable instance mogu se koristiti u metodi
                return povrsina;                // tip podatka koji metoda vraća (tj. varijabla koja prima
                  }                             // povratnu vrijednos) mora odgovarati povratnom tipu
        };

*       class Pravokutnik {             // klasa ima definiranu metodu sa parametrima
        public:
            double povrsinaPrav(double visina, double sirina) {     // parametri postaju lokalne varijable
                return visina*sirina;           // koje svoju vrijednost dobivaju prilikom poziva metode
              }
        };

*       class Pravokutnik {             // klasa ima metodu sa parametrima koja je definirana izvan klase
        pubilic:
              double povrsinaPrav(double visina, double sirina) ;
        };

            double Pravokutnik::povrsinaPrav(double visina, double sirina) {
                return visina*sirina;
            }
```

Stvaranje instance klase (objekta): **Ime_klase objekt ;**

```
Pravokutnik mali;       // stvaranje instance mali klase Pravokutnik
Pravokutnik novi;       // stvaranje instance novi klase Pravokutnik
```

Pristup varijablama, metodama: objekt. var_instance ili objekt. metoda()

```
novi.visina = 15;       // pristup varijablama instance novi i dodjela vrijednosti (samo ako su public)
mali.visina = 2;        // promjena varijable jednog objekta nikako ne utječe na istoimenu varijablu drugog
double p= mali.povrsinaPrav(3, 7);
```

Primjer korištenja klasa u programu:

* klasa bez metode:

```
#include <iostream.h>
class Pravokutnik {
public:
   double visina;
   double sirina;
};

void main() {
   Pravokutnik mali;

   mali.visina =5;
   mali.sirina = 10;
   double  povrsina = mali.visina * mali.sirina;

   cout<< " Površina pravokutnika je  " << povrsina;
   }
```

* korištenja metode bez parametara

```
#include <iostream.h>
class Pravokutnik {
public:
        double visina;
        double sirina;

   double povrsinaPrav () {
        return visina*sirina;
        }

};

   void main() {
   Pravokutnik mali;

   mali.visina = 5;
   mali.sirina = 10;

   cout<< " Površina pravokutnika je  " << mali.povrsinaPrav()<<endl;
}
```

* korištenja metode sa parametrima

```
#include <iostream.h>
class Pravokutnik {
public:
     double povrsinaPrav (double visina, double sirina) {
```

```
    return visina*sirina;
    }
};
void main() {
    Pravokutnik mali;
    cout<<" Površina pravokutnika je  " << mali.povrsinaPrav(5,10)<< endl;
    }
}
```

Pravilo:
- ako su parametri koji se prenose u metodu jednostavnog tipa tada se prenose po vrijednosti
(vrijednost varijable jednostavnog tipa ostati će nepromjenjena u programskom bloku iz kojeg je metoda pozvana)
- ako se u metodu šalje složeni tip podatka (npr. objekt) tada se prenose preko reference
(sve promjene koje se u metodi dogode na varijabli složenog tipa, odraziti će se na parametru sa kojim se je metoda pozvala)

Primjer:
u header fileu pravokutnik.h deklarirana je slijedeća klasa:

```
class Pravokutnik {
public:
        float a,b;
        float pov() {
                return a*b;
        }
        float opseg(){
                return 2*(a+b);
        }
};
```

```
#include <iostream.h>
#include "pravokutnik.h"

void main() {
        Pravokutnik prav1;
        cout<<"\n Upiši podatke za prvi pravokutnik \n";
        cout<<" Upiši duljinu stranice a: ";
        cin>> prav1.a;
        cout<<" Upiši duljinu stranice b: ";
        cin>> prav1.b;
        cout<<" \n Površina pravokutnika je : " << prav1.pov();
        cout<<" \n Opseg pravokutnika je : " << prav1.opseg();
        cout << endl;
        Pravokutnik prav2; // druga instanca klase Pravokutnik
        cout<<"\n Upiši podatke za drugi pravokutnik \n";
        cout<<" Upiši duljinu stranice a: ";
        cin>> prav2.a;
        cout<<" Upiši duljinu stranice b: ";
        cin>> prav2.b;
        cout<<" \n Površina pravokutnika je : " << prav2.pov();
        cout<<" \n Opseg pravokutnika je : " << prav2.opseg();
        cout << endl;
```

```
        if (prav2.pov() > prav1.pov())
                cout<< "Drugi pravokutnik je veći od prvoga \n ";
        else if (prav2.pov() < prav1.pov())
                cout<< "Drugi pravokutnik je manji od prvoga \n";
        else
                        cout<< "Pravokutnici su jednaki \n";
}
```

Konstruktori i destruktori

- Konstruktori su posebne metode za inicijalizaciju varijabla instance u procesu njenog stvaranja (ne može se direktno pozvati, automatski se poziva prilikom stvaranja novog objekta)
- ime konstruktora mora biti jednako imenu klase u kojoj se nalazi

Primjer konstruktora bez parametara:

```
class Pravokutnik {
                double visina;
                double sirina;

        public:
                Pravokutnik() {     // konstruktor
                        visina = 5;
                        sirina = 10;
                        }
                        ~ Pravokutnik() {}// destruktor


                double povrsinaPrav () {
                        double povrsina;
                        povrsina = visina*sirina;
                        return povrsina;
                        }
};
```

```
#include <iostream.h>
void main() {
   Pravokutnik mali;
   cout<<" Površina pravokutnika je  " << mali.povrsinaPrav ();
}
```

Primjer konstruktora sa parametrima:

```
class Pravokutnik {
                double visina;
                double sirina;
        public:
                Pravokutnik(double v, double s) {          // konstruktor
                        visina = v;
                        sirina = s;
                        }
                ~ Pravokutnik() {} // destruktor
                double povrsinaPrav () {
                        return visina*sirina;
                        }
```

```
}
```

```
#include <iostream.h>
void main() {
    Pravokutnik mali (5,10);
    cout<<" Površina pravokutnika je  " << mali.povrsinaPrav ();
}
```

Primjer konstruktora sa parametrima i bez parametara:

```
class Krug {
private:                    //ako se ne navede varijable i metode se isto smatraju privatne
            int r;

 public:
        Krug(int rr) {          // ili Krug(int rr) :r(rr) {}
            r=rr;
            }
         Krug() {               // ili Krug() :r(5) {}
            r=5;
        }
        ~Krug() {}              //destruktor

        void inicijal( int rr) {
            r=rr;
         }
         float povrsina() {
            return r*r*3.1415;
         }
};

// Glavni program u kojem se koristi deklarirana klasa:
#include <iostream.h>
void main() {
        Krug  K1;  // uzeti će se vrijednost radijusa koja je definirana u konstruktoru
        cout<<"\n Povrsina kruga je " <<  K1.povrsina() << endl;
        int r;
        cout << "Upisi radijus :";
        cin>> r;
        Krug K2(r);
        cout<<"\n Povrsina kruga je " <<  K1.povrsina() << endl;
        cout << "Upisi novi radijus :";
        cin>> r;
        Krug.inicijal(r);                   // za naknadne provjene vrijednosti varijable r
        cout<<"\n Povrsina kruga je " <<  K1.povrsina() << endl;
}

ili

class Krug {
private:
        float r;
public:
```

```
  Krug(): r(10) {}
  Krug(float x) : r(x) {}
        ~Krug() {}
        void inicijal (float x);
        float povrsina() ;
};

void Krug::inicijal (float x) {
                r=x;
        }
float Krug::povrsina() {
                return r*r*3.1415f;
        }

#include <iostream.h>

void main() {
   Krug k1;
   cout<<"\n Površina kruga je " <<  k1.povrsina() << endl;
   cout<<"\n Upisi radijus kruga:";
   float x;
   cin>> x;
   Krug k2(x);
    cout<<"\n Površina kruga je " <<  k2.povrsina() << endl;
    cout<<"\n Upisi radijus kruga:";
   cin>> x;
    k2.inicijal(x);
    cout<<"\n Površina kruga je " <<  k2.povrsina() << endl;
}
```

Polje objekata

Pr. deklariranja: djelatnik d1[15]; // 15 instanci klase djelatnik

Pr.

```
class djelatnik {
private:
        char ime[10],pre[10];
        float placa;
public:
        void unos();
        float povecanje(int );
        void ispis();
};

void djelatnik::unos(){
cout<<"Ime: ";
cin>>ime;
cout<<"Prezime: ";
cin>>pre;
cout<<"Iznos place: ";
cin>>placa;
}
```

```
float djelatnik::povecanje(int a) {
	placa+=a;
	return placa;
}
void djelatnik::ispis(){
cout<< ime << pre << " ima placu od " << placa<< endl;
}
#include <iostream.h>
#include  "niz.h"
void main() {
	djelatnik d1[15];
	cout<< "Koliko ima djelatnika";
	int n;
	cin>> n;
	for (int i=0; i<n; i++)
		d1[i].unos();
	cout<< "\n Ispis elemenata : \n ";
	for ( i=0; i<n; i++)
	d1[i].ispis();
	cout<< " \n Povećanj"<< endl;
	char uvjet;
	do {
	cout<< "\n Upiši redni broj djelatnika kojem treba povećati plaću:";
	int br;
	cin>> br;
	cout<< "\n Upiši iznos povećanja plaće:";
	int iznos;
	cin>> iznos;
	cout<<" Uvećana plaća iznosi " << d1[br-1].povecanje(iznos);
	cout<<" \n Treba li jos nekome povećati plaću (d/n): ";
	cin>> uvjet;
	}
	while (uvjet =='d');
	cout<< "\n Stanje nakon povećanja: \n ";
	for ( i=0; i<n; d1[i++].ispis());
}
```

Dinamičko kreiranje objekata // free store

Pr.

```
#include <iostream.h>

class SimpleCat
{
public:
	SimpleCat();
	~SimpleCat();
private:
	int itsAge;
};

SimpleCat::SimpleCat()
```

```
{
        cout << "Constructor called.\n";
        itsAge = 1;
}

SimpleCat::~SimpleCat()
{
        cout << "Destructor called.\n";
}

int main()
{
        cout << "SimpleCat Frisky...\n";
        SimpleCat Frisky;
        cout << "SimpleCat *pRags = new SimpleCat...\n";
        SimpleCat * pRags = new SimpleCat;
        cout << "delete pRags...\n";
        delete pRags;
        cout << "Exiting, watch Frisky go...\n";
        return 0;
}


//Pristu objektima na stogu (the heap)

#include <iostream.h>

class SimpleCat
{
public:
        SimpleCat() {itsAge = 2; }
        ~SimpleCat() {}
        int GetAge() const { return itsAge; }
        void SetAge(int age) { itsAge = age; }
private:
        int itsAge;
};

int main()
{
        SimpleCat * Frisky = new SimpleCat;
        cout << "Frisky is " << Frisky->GetAge() << " years old\n";
        Frisky->SetAge(5);
        cout << "Frisky is " << Frisky->GetAge() << " years old\n";
        delete Frisky;
        return 0;
}


Pr.
// Pokazivači kao varijable klase

#include <iostream.h>
```

```
class SimpleCat
{
public:
        SimpleCat();
        ~SimpleCat();
        int GetAge() const { return *itsAge; }
        void SetAge(int age) { *itsAge = age; }

        int GetWeight() const { return *itsWeight; }
        void setWeight (int weight) { *itsWeight = weight; }

private:
        int * itsAge;
        int * itsWeight;
};

SimpleCat::SimpleCat()
{
        itsAge = new int(2);
        itsWeight = new int(5);
}

SimpleCat::~SimpleCat()
{
        delete itsAge;
        delete itsWeight;
}

int main()
{
        SimpleCat *Frisky = new SimpleCat;
        cout << "Frisky is " << Frisky->GetAge() << " years old\n";
        Frisky->SetAge(5);
        cout << "Frisky is " << Frisky->GetAge() << " years old\n";
        delete Frisky;
        return 0;
}
```

Pr.

```
// References to class objects

#include <iostream.h>

class SimpleCat
{
public:
        SimpleCat (int age, int weight);
        ~SimpleCat() {}
        int GetAge() { return itsAge; }
        int GetWeight() { return itsWeight; }
private:
        int itsAge;
        int itsWeight;
```

```
};

SimpleCat::SimpleCat(int age, int weight)
{
        itsAge = age;
        itsWeight = weight;
}

int main()
{
        SimpleCat Frisky(5,8);
        SimpleCat & rCat = Frisky;

        cout << "Frisky is: ";
        cout << Frisky.GetAge() << " years old. \n";
        cout << "And Frisky weighs: ";
        cout << rCat.GetWeight() << " pounds. \n";
        return 0;
}
```

Operator this

- koristi se kada se želimo koristiti podatke unutar trenutnog objekta; objekta kojem metoda i sama pripada

```
Pravokutnik(double v, double s) {        // konstruktor
        this.visina = v;                 // korištenje operatora this je ispravno ali nije nužno
        this.sirina = s;
}
```

// U istom programskom bloku dvije varijable ne mogu imati jednaka imena, ali dopušteno je preklapanje imena varijabli instanca s imenima lokalnih varijabli metoda.

```
Pravokutnik(double visina, double sirina) {      // lokalne varijable imaju isto ime kao i
        this.visina = visina;                    // varijable instance, pa bi ih prekrile;
        this.sirina = sirina;                    // korištenje operatora this omogućava neposredni
}                                                // pristup varijablam koje pripadaju trenutnoj instanci;
```

// Izraz **this**.sirina = sirina znači da se vrijednost parametra sirina pridružuje vrijednosti varijable instance sirina.