



NAMESPACE

sintaksa

```
using namespace s1; → ako imas to ne treba
namespace s1 {
    float const pi = 3.14;
}
```

definacija

Broj/Ref. No.

s1::pi;

→ vrijedi od mesta navođenja do kraja bloka / datoteke u kojoj je navедena.

→ može se navesti više, ne smije biti poklapanja u nazivu

FUNCTION OVERLOADING

→ funkcije s istim nazivom, drugačijim parametrima
→ obavija se ona kojoj pripadaju parametri ili ona koja je najbliža za konverziju

REFERENCA &

→ pokazivač koji se ne mijenja → ne može prvo pokazivati na var a i onda na var b.
→ pokazivač ima svoju adresu, dok referenca dijeli s var
→ referenca ne može pokazivati na NULL, pointer da
→ pointer može pokazivati na pointer, referenca na referenci ne

```
int *p=a;
int **q=p;
```

Hrvatska kontrola zračne plovidbe d.o.o. / Croatia Control, Croatian Air Navigation Services, Ltd.

```
int &r=a;
int &&s=r;
```

X

■ Privredna banka Zagreb d.d.
SWIFT: PBZGHR2X
2340009-1100198272
IBAN: HR6023400091100198272

■ Zagrebačka banka d.d.
SWIFT: ZABAHR2X
2360000-1101513818
IBAN: HR9323600001101513818

■ Trgovački sud u Zagrebu / Commercial Court in Zagreb
MBS / Registration No. 080328617
Temeljni kapital / Subscribed Capital
352.759.600,00HRK
Matični broj / Company No. 1475886
OIB / VAT No. 33052761319

■ Direktor / Director General: Dragan Bilać
Predsjednik NO-a / President of the SB: Darko Prebežac

REFERENCA VS. POINTER

- kao parametri funkcije
- kao return types

↳ kad ti treba aritmetika pokazivača, NULL
najčešće za polja
→ za strukture podataka tipa list, tree ...

```
Void swap(int &a, int &b){  
    int tmp=a;  
    a=b;  
    b=tmp;  
}
```

→ Kad bi pisalo
(int a, int b)
nebi se zamijenilo

RAZRED (CLASS) → na kraju bloka ide ;

→ Članske varijable su po defaultu private

→ Ako instanciram objekt neke klase Point p;
rezervira se memorija, ali se ne inicijaliziraju članske
varijable (u p.x je spremljeno smjeće)

→ Zapravo želim

```
Point p;  
p.x = 7;  
p.y = 8;
```

instancirala sam točku
p(7,8).

→ Ako imam složenu ~~klasu~~ radim sljedeće :

U Point.h deklariram razred

```
class Point {  
public:  
    double x;  
    double y;  
    Void mirrorX (double xValue);  
    Void mirrorY (double yValue);
```

Main.cpp onda izgleda

```
#include <iostream>  
#include <Point.h>  
int main(void){  
    Point p;  
    ...  
    return 0;
```

U Point.cpp definiram funkcije

```
#include "Point.h"  
Void Point::mirrorX(double xValue){ x = xValue - (x - xValue);}  
Void Point::mirrorY(double yValue){ y = yValue - (y - yValue);}
```

`memcpy (&s, &p, sizeof(p));`

↓ ↓ ↓
 destinacija izvor memorija
 pričuvanje duljina izvora

→ tako možemo pridružiti vrijednosti jedne instance klase drugoj, ekivalentno

`s = p;`

KONSTRUKTORI

Broj/Ref. No.

Datum/Date

- služe za inicijalizaciju vrijednosti članstih varijabli

```
class Point {
public:
  double x;
  double y;
  Point() { // default konstruktor
    x = 0;
    y = 0;
  }
  Point(double xIn, double yIn) {
    x = xIn; // parametri
    y = yIn;
  }
}
```

primjeri instanciranja

→ Point p; [p = (0,0)]

→ Point r; [r = (1,2)]

→ Ako ovdje napišem $y=y$;
onda stane →

Treba:

`this → y = y;`

KONSTRUKTORI S INICIJALIZACIJSKIM LISTAMA

```
class Point {  
public:  
    double x;  
    double y;  
    Point() : x(0), y(0) {} //prazan           ↗ //parametarski  
    Point(double xIn, double yIn) : x(xIn), y(yIn) {}  
};
```

↓
ako ovde napisem
x(x) sve je ok

DESTRUKTOR

- funkcija koja briše objekt
- automatski se dogodi ako
 - a) program završi,
 - b) funkcija završi
 - c) blok naredbi s lokalnim var završi
 - d) pozovemo operator delete
- imaju isto ime kao klasa
- "prefix" je tilda ~
- nemaju argumente, ne vraćaju ništva
- ako ga nema, compiler ga automatski stvara
- treba se koristiti kad imas pointer na memoriju alocirani u klasi



UNIFORMNA INICIJALIZACIJA

- konzistentna inicijalizacija (ista sintaksa) i za varijable,
i za objekte i za polja

Point p{3, 7}

int polje[] {1, 2, 3}

Point poljePointova [{3, 7}, {7, 3}]

NASLEDJIVANJE

class Shape {

private :

int id;

public :

double x;

double y;

double a;

int getId { return id; }

Shape(int id, double x, double y, double a):
id(id), x(x), y(y), a(a) {}

virtual double area () = 0;

virtualna funkcija: razredi koji je
naslijetaju trebaju je stvarno
implementirati

parametarski
~~bezparametarski~~
konstruktor

class Triangle : public Shape {
public:

double b;
double c; → inicijaliziram
dodatne varijable

Triangle(int id, double x, double y, double a, double b, double c)
: Shape(id, x, y, a), b(b), c(c) {}

double area () {

double s = (a+b+c)/2
return sqrt(s*(s-a)*(s-b)*(s-c));

stvarna implementacija
virtualne funkcije

Tiskaj param.
konstruktor

PUBLIC, PRIVATE, PROTECTED

public - dostupno svim dijelovima programa

private - u razredu u kojem je definisano i friend funkcijama i razredima

protected - u razredu u kojem je definisano, razredu koji nasledjuje taj razred, u friend funkcijama i razredima

FRIEND FUNKCIJA

funkcija neke klase koja je definisana izvan klase i ima pristup private i protected članovima.

```
class Shape {  
private:  
    int id;  
    friend void setId(Shape &s,  
                      int newId);  
};  
  
void setId(Shape &s, int newId){  
    s.id = newId; }  
int main(void){ ...  
    setId(s, 10) → koristim fe  
          golje funkcije  
}
```

FRIEND KLASA

razred koji može pristupati private članovima razreda koji ga je navelo kao "prijatelja".

```
class Shape {  
private:  
    int id; }  
friend class SomeClass;  
public:  
    int getId() { return id; }  
  
class SomeClass {  
public:  
    void setId(Shape &s, int newId){  
        s.id = newId; }  
    ...  
}
```

class Podredena : public Nadredena {
 typeof(NadPublic) je public
 typeof(NadProtoc) je Protected
 ~~typeof(NadPriv)~~ je nedostupna }

class Podredena : protected/private Nadredena {
 typeof(NadPublic) je private / protected
 typeof(NadPrivate) je protected / private
 typeof(NadProtoc) je protected / private

```
class Nadredena {  
    public int NadPublic;  
    private int NadPriv;  
    protected int NadProtoc;  
}
```



STATIC ČLANOVI

- varijabla koju sve instance klase dijele (istu vrijednost)
- ona postoji i kad nemainstanciranih objekata
- dobar primjer je brojač („koliko smo objekata instancirali“)
- u konstruktoru definisano povećava se Broj/Ref. No.
Datum/Date
- novog objekta i kad se prouženi vrijednost static člana, povećani se svim objektima – ne samo zadnjem

```
class Shape {
```

private:

```
static int AutoNumber;
```

→ deklaracija unutar razreda

```
... Static void resetAutoNumber {autoNum=0;}
```

→ static funkcija

```
Shape (double x, double y, double a);
```

→ povećaje unutar konstruktora

```
x(x), y(y), a(a) {
```

```
id = ++ autoNumber;
```

→ definicija izvan svega

```
int Shape :: autoNumber;
```

→ poziv static funkcije unutar maina

```
int main (void) {
```

```
... Shape :: resetAutoNumber();
```

```
}
```

STRUKTURA

Struct Point {

```
int x;  
int y;  
};
```

```
int main(void) {
```

```
    Point p = {1, 2};  
    cout << p.x;
```

```
}
```

Struct	Class
default access modifiers	public
kada konstrui?	samo podaci

podaci + funkcije

OPERATOR OVERLOADING

- ako želim da se operator nad objektima neke konkretnе klase ponaša drugačije nego imao.

```
class Complex {  
private:  
    int real, imag;  
public  
    Complex(int r, int i)  
    { real=r;  
      imag=i;  
    }
```

```
Complex operator + (Complex &other){  
    Complex result;  
    result.real = real + other.real;  
    result.imag = imag + other.imag;  
    return result;  
}
```

```
bool operator == (Complex &other){  
    return (imag == other.imag &&  
            real == other.real);  
}
```

- ponašanje operatora opisujem unutar klase

→ za comp1(1, 7) + comp2(3, -1)
vraća objekt tipa Complex
npr comp3(4, 6).

→ za comp1 == comp2 vraća false.



istream i ostream - vrlo slično toString u Java

... i dalje smo u klasi Complex...

```
ostream &operator<<(ostream &os,
const Complex &c){
```

```
    os << c.real << " +i" << c.imag << endl;
} return os;
```

→ ako u mainu postavim
cout << c1;
dobivam

5 + 3i

Broj/Ref No.
Datum/Date

```
istream &operator>>(istream &is,
Complex &c){
```

```
    cout << "Enter real part: ";
    is >> c.real;
```

```
    cout << "Enter imaginary part: ";
    is >> c.imag;
```

```
    return is;
```

→ ako u mainu postavim
cin >> c1;
dobivam

Enter real part: 5
Enter imaginary part: 3

MEMORIJA

MALLOC - pronalazi na gornjim slobodan dio, vraća pointer na početak rezervirane memorije

pointer = (tip-podataka *) malloc (broj-podataka)

npr.

```
char *p;  
char *r;  
P = (char *)malloc(4);
```

- definiraj mi pokazivač tipa char
- nadi mi i rezerviraj mjesto za 4 bajta ~~na p~~
- na p pridruži adresu ~~na p~~ na kojoj počinje dio memorije koji si rezervirao

REALLOC - ako želim promijeniti količinu rezervirane memorije na koju pokazuje pointer

... nastavak gornjeg kôda...

```
P = realloc(p, 8)  
r = realloc(NULL, 16)
```

- promijeni mi broj rezerviranih bajtova na koje pokazuje P s 4 na 8.
- ako moraš mijenjati adresu rezervacije na p postavi novu adresu
- napravi za r isto što si gore radio za p (malloc = realloc s NULL)

Ako si normalno ljudsko biće i ne znaš koliko bajtova zauzima pojedini tip podataka koristi

P = (tip-podataka *) malloc (broj-podataka * sizeof(tip));

npr.

```
P = (char *) malloc (2 * sizeof(char))
```

„nadi i rezerviraj dovoljno mjesto za dva podatka tipa char i vrati pokazivač s adresom početka dijela memorije“

NULLPTR

- keristim kad želim pozvat funkciju čiji je argument pointer, a želim utrpat pointer NULL vrijednosti → mogu slat NULL i nullptr [primer 1.]
- ako imam function overloading tipa int fun (int var) i tipa int fun (int *var) NULL će bacati error jer ne zna gdje treba ići, nullptr zna gdje treba ići [primer 2.a]
- pouka: kao argument uvijek koristi nullptr, a ne NULL; nikad nećeš morat razmišljati [primer 2.b]

primer 1.

```
int fun ( int *var ) {
    cout << "Ušli smo u fun
            s pointerom" ;
}

fun (NULL);
fun (nullptr);
```

Output 1.

Ušli smo u fun s pointerom
Ušli smo u fun s pointerom

primer 2.

```
int fun (int *var) {
    cout << "fun s pointerom" ;
}

int fun (int var) {
    cout << "fun bez pointeri" ;

    fun(1);
    fun(nullptr);
}

fun(NULL);
```

Output 2a.GREŠKA U PREVODENJU

Output 2b. – zakomentiramo zadnju liniju
fun bez pointeri
fun s pointerom

OPERATORI NEW I DELETE

- bolja verzija malloc-a

`pointer_var = new tip-podataka` npr. `int *p = new int;`

môžemo odnosi i inicijalizirati

`pointer_var = new tip-podataka(vrijednost)` npr. `int *p = new int(7);`

môžemo rezervirati i polje, tada ptr pokazuje na adresu prvog elementa

`pointer_var = new tip-podataka[veličina-polja]` npr. `int *p = new int[10];`

- ako rezerviram memoriju pomoću new, ona se neće sama očistiti
nego se programer mora pobrinuti za to; tu dolazi operator delete

`delete pointer-var;` npr. `delete p;`

ako želim očistiti cijelo rezervirano polje

`delete[] pointer-var;` npr. `delete[] p;`

COPY CONSTRUCTOR



EXCEPTION HANDLING

```
try {
    ...
    throw my_exception ("Baš si luser");
}
catch (exception &e) {
    cin<< e.what();
}
```

- možeš napraviti i klasu MyException i onda ju naslijediti
- ako si tako nesretna da ti to nekad zatreba - guglaj "Exception classes"

Broj/Ref. No.

Datum/Date

GENERICS

funkcija

```
template <typename T>
inline T abs(T a){
    return a<0? -a: a;
}
```

inline

- mislim da se funkcija s inline ne prevedi kao takva, nego kad se pozove samo se taj odsječak kôda koji je u tijelu "zalijepi" na mjesto gdje se funkciju pozove

- ako je kodka, inline daje efikasnost izvođenja

razred

```
template <typename T>
class Point {
public:
    T x;
    T y;
    point(T x, T y) : x(x), y(y) {}
};
```

... main...

```
Point<int> pint(1,2)
Point<double> pdouble(1.2, 3.)
```

SMART POINTERS

- koristiš kada ne želiš/trebaš eksplicitno pozvat delete
- destruktur se zove automatski: čim objek izlazi izvan doseg

C++ STANDARD TEMPLATE LIBRARY (STL)

- generički razredi koji implementiraju česte algoritme i strukture podataka
 - algoritmi (npr. sort)
 - kontejneri (npr. list)
 - Iteratori
 - funkcijski objekti

ITERATOR

```
#include <iterator>
#include <vector>

...
vector<int> v = {1,2,3,4,5}
vector<int>::iterator i;

for(i = v.begin(); i != v.end(); i++){
    cout << *i << ",";
}
```

jednostavno privati zaokružene stvari