

# Programiranje 2, letnji semestar 2023/4

## Više o objektno-orijentisanom programiranju

---

Stefan Nikolić

Prirodno-matematički fakultet, Novi Sad

22.04.2024.

U proceduralnom programiranju smo delili program na potprograme sa jasno definisanim interfejsima koji su odvajali upotrebu potprograma od njegove implementacije

# Podsetnik

Zamena implementacije ne bi promenila ništa u ostatku programa

```
void sort(int* a);
```

```
void sort(int* a){/*Implementiramo  
Insertion sort*/}
```

```
int main()
```

```
{
```

```
    ...
```

```
    sort(niz);
```

```
    ...
```

```
}
```

```
void sort(int* a);
```

```
void sort(int* a){/*Implementiramo  
Shell sort*/}
```

```
int main()
```

```
{
```

```
    ...
```

```
    sort(niz);
```

```
    ...
```

```
}
```

Da bismo pored lake promene implementacije programske logike omogućili i laku promenu načina organizacije podataka, razdvojili smo i implementaciju strukture podataka od interfejsa pomoću kojih joj pristupamo

## Primer sa prošlog časa: čuvanje tačaka u polarnim koordinatama

Kada smo promenili čuvanje sa Dekartovih na polarne koordinate, zahvaljujući zadržanim interfejsima, nismo morali da menjamo ni jedan segment spoljnog koda koji pristupa strukturi Tačka:

```
#ifndef POINT_H
#define POINT_H

#include <string>
#include <cmath>

typedef struct Point
{
    float r = 1.0;
    float t = 1.0;

    std::string label;
} Point;

//Getters

//Funkcija vraca x-koordinatu tacke p
float get_x(const Point& p);
//Funkcija vraca y-koordinatu tacke p
float get_y(const Point& p);
//Funkcija vraca naziv tacke p
const std::string& get_label(const Point& p);

//Setters

//Funkcija postavlja x-koordinatu tacke p
void set_x(Point& p, float x);
//Funkcija postavlja y-koordinatu tacke p
```

# Primer sa prošlog časa: čuvanje tačaka u polarnim koordinatama

Kada smo promenili čuvanje sa Dekartovih na polarne koordinate, zahvaljujući zadržanim interfejsima, nismo morali da menjamo ni jedan segment spoljnog koda koji pristupa strukturi Tačka:

```
float x;
float y;
string label;
for(unsigned i = 0; i < cnt; ++i)
{
    cout << "Unesite ime tacke " << i << ": ";
    getline(cin >> ws, label);
    set_label(points[i], label);

    cout << "Unesite x koordinatu tacke " << i << ": ";
    cin >> x;
    set_x(points[i], x);

    cout << "Unesite y koordinatu tacke " << i << ": ";
    cin >> y;
    set_y(points[i], y);
}

struct Point c;

cout << "Unesite x koordinatu tacke c: ";
cin >> x;
set_x(c, x);

cout << "Unesite y koordinatu tacke c: ";
cin >> y;
set_y(c, y);
```

## Poslednji korak u modularizaciji

Sada smo pored podele manipulisanja podacima na module podelili i samo skladištenje podataka na module

Preostao nam je problem utvrđivanja koji modul za obradu (skup funkcija) da primenimo na koji modul za skladištenje

Da bismo rešili i taj problem, uveli smo pojam klase, koji kombinuje modul za skladištenje podataka sa odgovarajućim modulom za manipulisanje tim podacima



Klase

---

# Deklaracija klase

```
class Identifikator  
{  
    deklaracija polja i funkcija članica klase  
};
```

# Ograničenja pristupa

Sva polja i funkcije članice pripadaju jednoj od tri kategorije pristupa:

- **public:** vidljivo i izvan klase
- **private:** vidljivo samo unutar klase
- **protected:** vidljivo unutar date klase i svih klasa izvedenih iz nje (uskoro više o tome)

# Primer sa prošlog časa

```
#ifndef POINT_H
#define POINT_H

#include <string>

class Point
{
    private:
        float x;
        float y;

        std::string label;

    public:
        //Getters
        float get_x() const;
        float get_y() const;
        const std::string& get_label() const;

        //Setters
        void set_x(float x);
        void set_y(float y);
        void set_label(const std::string& label);

        //Other members
        float cheby_dist(const Point& center);
};

#endif
```

Privatnom segmentu mogu pristupiti samo funkcije koje su deo klase

Tu su obično sadržani svi podaci (setimo se apstrakcije podataka)

# Primer sa prošlog časa

```
#ifndef POINT_H
#define POINT_H

#include <string>

class Point
{
    private:
        float x;
        float y;

        std::string label;

    public:
        //Getters
        float get_x() const;
        float get_y() const;
        const std::string& get_label() const;

        //Setters
        void set_x(float x);
        void set_y(float y);
        void set_label(const std::string& label);

        //Other members
        float cheby_dist(const Point& center);
};

#endif
```

U javnom segmentu su polja kojima je moguće pristupiti i izvan klase

Tu se najmanje sadrže get i set metode, ali i sve druge koje predstavljaju interfejs klase i korisnika

# Šta će se dogoditi ako ovo napišemo?

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

using namespace std;

int main()
{
    Point center;

    float x;
    cout << "Unesite x koordinatu centra: ";
    cin >> x;
    center.x = x;

    return 0;
}
```

## Šta će se dogoditi ako ovo napišemo?

```
test_point.cpp: In function 'int main()':
test_point.cpp:13:16: error: 'float Point::x' is private within this context
   13 |         center.x = x;
      |                ^
In file included from test_point.cpp:2:
point.h:9:23: note: declared private here
    9 |         float x;
      |                ^
shell returned 1
```

# Kako da rešimo taj problem?

```
#ifndef POINT_H
#define POINT_H

#include <string>

class Point
{
    private:
        float x;
        float y;

        std::string label;

    public:
        //Getters
        float get_x() const;
        float get_y() const;
        const std::string& get_label() const;

        //Setters
        void set_x(float x);
        void set_y(float y);
        void set_label(const std::string& label);

        //Other members
        float cheby_dist(const Point& center);
};

#endif
```

U javnom segmentu su polja kojima je moguće pristupiti i izvan klase

Tu se najmanje sadrže get i set metode, ali i sve druge koje predstavljaju interfejs klase i korisnika



# Koristimo javne interfejse

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

using namespace std;

int main()
{
    Point center;

    float x;
    cout << "Unesite x koordinatu centra: ";
    cin >> x;
    center.set_x(x);
    cout << center.get_x() << endl;

    return 0;
}
```

```
Unesite x koordinatu centra: 7  
7
```

# Da li će ovo raditi?

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

using namespace std;

void inc_x(Point& p, float inc_amount)
{
    p.x += inc_amount;
}

int main()
{
    Point center;

    float x;
    cout << "Unesite x koordinatu centra: ";
    cin >> x;
    center.set_x(x);
    cout << center.get_x() << endl;

    inc_x(center, 6);
    cout << center.get_x() << endl;

    return 0;
}
```

## Da li će ovo raditi?

```
test_point.cpp: In function 'void inc_x(Point&, float)':
test_point.cpp:8:11: error: 'float Point::x' is private within this context
   8 |         p.x += inc_amount;
      |         ^
In file included from test_point.cpp:2:
point.h:9:23: note: declared private here
   9 |         float x;
      |         ^
shell returned 1
```

## Registracija prijateljskih funkcija / klasa

U svakoj klasi možemo registrovati prijateljske funkcije (slobodne, ili one koje su deo neke druge klase) / klase, koje će imati posebnu dozvolu za pristup poljima sa inače ograničenim pristupom

# Registracija prijateljskih funkcija / klasa

Zasebna funkcija:

```
friend povratni_tip identifikator(parametri);
```

Funkcija koja je deo neke druge klase:

```
friend povratni_tip identifikator_druge_klase::identifikator(parametri);
```

Druga klasa (pristup će biti moguć iz svih njenih metoda):

```
friend class identifikator_druge_klase;
```

# Registraciju vršimo u telu date klase

```
class Point
{
    private:
        float x;
        float y;

        std::string label;

    public:
        //Getters
        float get_x() const;
        float get_y() const;
        const std::string& get_label() const;

        //Setters
        void set_x(float x);
        void set_y(float y);
        void set_label(const std::string& label);

        //Other members
        float cheby_dist(const Point& center);

        //Friend functions
        friend void inc_x(Point& p, float inc_amount);
};
```

#endif

"point.h" 33L, 485B written

## Registraciju vršimo u telu date klase

```
Unesite x koordinatu centra: 13  
13  
19
```



## Registraciju vršimo u telu date klase

No, primetimo da data funkcija time ne postaje članica klase

Iz tog razloga, prosleđivanje objekta funkciji je eksplicitno (u ovom slučaju preko prvog parametra)

# Implementacija funkcija članica

Definicije funkcija članica klase možemo napisati direktno u telu klase

Međutim, kao i kod drugih funkcija, preporuka je da ih pišemo odvojeno, u odgovarajućem .cpp fajlu

# Implementacija funkcija članica

Da bismo naznačili da je određena funkcija deo klase, koristimo sledeću sintaksu:

```
povratni_tip identifikator_klase::identifikator(parametri)
{
    ...
}
```

Osim toga, implementacija funkcije članice je ista kao i implementacija bilo koje druge funkcije

## U našem primeru

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

float Point::get_x() const
{
    return this -> x;
}

float Point::get_y() const
{
    return this -> y;
}

const std::string& Point::get_label() const
{
    return label;
}

void Point::set_x(float x)
{
    this -> x = x;
}

void Point::set_y(float y)
{
    this -> y = y;
}
```

## this pokazivač

Sve (nestatičke; videćemo uskoro šta to znači) funkcije članice implicitno kao prvi parametar dobijaju objekat na kom se vrši poziv date funkcije članice

Tom objektu je iz funkcije članice moguće pristupiti pomoću pokazivača **this**

# Šta je uopšte objekat?

Ako klasu zamislimo kao korisnički tip podataka

Onda je objekat naprosto promenljiva tog tipa

Kažemo još da je objekat instanca klase koja je naznačena kao njegov tip prilikom deklaracije

# Statička polja

**static** tip polje;

Deklaracija statičkog polja čini da ono ne bude vezano za bilo koji konkretan objekat, već za samu klasu

Time će njegova vrednost biti deljena između svih objekata date klase

# Primer

```
class Point
{
    private:
        float x;
        float y;

        std::string label;

        static bool weight_points;

    public:
        //Getters
```

point.h

5,0-1

Deklarišemo jednu statičku zastavicu koja će nam govoriti da li svim tačkama treba da pridružimo težinu ili ne



```
public:
    //Getters
    float get_x() const;
    float get_y() const;
    const std::string& get_label() const;
    static bool get_weight_points();

    //Setters
    void set_x(float x);
    void set_y(float y);
    void set_label(const std::string& label);
    static void set_weight_points(bool do_weight);
```

point.h

28,0-1

Za pristup statičkim poljima moramo deklarirati i pristupne funkcije kao static

# Primer

```
bool Point::get_weight_points()  
{  
    return weight_points;  
}  
  
void Point::set_weight_points(bool do_weight)  
{  
    weight_points = do_weight;  
}  
  
const std::string& Point::get_label() const  
{  
    return label;  
}  
point.cpp
```

1,16

Ključnu reč static koristimo samo prilikom deklaracija, ne i prilikom definicija

```
bool Point::get_weight_points()  
{  
    return this -> weight_points;  
}
```

Pošto su statičke funkcije vezane za klasu a ne za konkretan objekat, one ne dobijaju objekat kao implicitan parametar, pa ni `this` pokazivač nije dostupan

# Primer

```
point.cpp: In static member function 'static bool Point::get_weight_points()':  
point.cpp:16:16: error: 'this' is unavailable for static member functions  
   16 |         return this -> weight_points;  
      |                ^  
shell returned 1
```

Pošto su statičke funkcije vezane za klasu a ne za konkretan objekat, one ne dobijaju objekat kao implicitan parametar, pa ni `this` pokazivač nije dostupan

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

using namespace std;

void inc_x(Point& p, float inc_amount)
{
    p.x += inc_amount;
}

bool Point::weight_points = true;
int main()
test_point.cpp
```

12,0-1

Svako statičko polje moramo i definisati (deklarisati globalnu promenljivu sa datim identifikatorom i postaviti je na neku vrednost)

```
int main()  
{  
  
    Point::set_weight_points(false);  
  
    Point center;  
  
    float x;  
    cout << "Unesite x koordinatu centra: ";  
    cin >> x;  
    center.set_x(x);  
    cout << center.get_x() << endl;
```

test\_point.cpp

25,0-1

Statičke funkcije možemo pozivati i direktno iz klase, bez ijednog deklarisanog objekta

```
Point corner;  
  
cout << "Center: " << center.get_weight_points() << endl;  
cout << "Corner: " << corner.get_weight_points() << endl;  
  
corner.set_weight_points(true);  
cout << "Center: " << center.get_weight_points() << endl;  
cout << "Corner: " << corner.get_weight_points() << endl;  
  
return 0;
```



test\_point.cpp

40,1

Bot

A možemo ih pozivati i iz objekata

```
Unesite x koordinatu centra: 6  
6  
12  
Center: 0  
Corner: 0  
Center: 1  
Corner: 1
```



# Konstruktori

---

Da bismo mogli da koristimo deklarisan objekat, često je potrebno da izvršimo alokaciju neke memorije i da inicijalizujemo neka polja

Za to nam služe specijalne funkcije koje nazivamo konstruktorima

# Deklaracija konstruktora

```
identifikator_klase(spisak_parametara);
```

Primetimo da konstruktor ne može da ima proizvoljan identifikator te da nema nikakav povratni tip; čak ni void

# Tipovi konstruktora

- Konstruktor bez parametara (podrazumevani konstruktor)
- Konstruktor sa parametrima
- Kopirajući konstruktor

## Podrazumevani konstruktor

```
        std::string label;  
  
        static bool weight_points;  
  
public:  
    //Konstruktori  
    Point();  
  
    //Getters  
    float get_x() const;  
    float get_y() const;  
    const std::string& get_label() const;  
    static bool get_weight_points();
```

point.h

18,10-24

## Podrazumevani konstruktor

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

Point::Point()
{
    x = -3;
    y = -3;
}
```

## Podrazumevani konstruktor

```
Point corner;
```

```
cout << "Center: " << center.get_weight_points() << endl;  
cout << "Corner: " << corner.get_weight_points() << endl;
```

```
corner.set_weight_points(true);  
cout << "Center: " << center.get_weight_points() << endl;  
cout << "Corner: " << corner.get_weight_points() << endl;
```

```
Point stack_point;
```

```
cout << "stack_point = (" << stack_point.get_x()  
      << ", " << stack_point.get_y() << ")\n";
```

test\_point.cpp

40,45-52

90%



## Podrazumevani konstruktor

```
Unesite x koordinatu centra: 6  
6  
12  
Center: 0  
Corner: 0  
Center: 1  
Corner: 1  
stack_point = (-3, -3)
```

## Konstruktor sa parametrima

```
static bool weight_points;  
  
public:  
    //Konstruktori  
    Point(float x_, float y_);  
  
    //Getters  
    float get_x() const;  
    float get_y() const;  
    const std::string& get_label() const;  
    static bool get_weight_points();
```

point.h

19,0-1

## Konstruktor sa parametrima

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

Point::Point(float x_, float_y)
{
    x = x_;
    y = y_;
}
```

## Zašto se ovo desilo?

```
test_point.cpp: In function 'int main()':  
test_point.cpp:18:15: error: no matching function for call to 'Point::Point()'  
   18 |         Point center;  
      |                ^~~~~~
```

## Zašto se ovo desilo?

```
In file included from test_point.cpp:2:
point.h:18:17: note: candidate: 'Point::Point(float, float)'
   18 |         Point(float x_, float y_);
      |         ^~~~~~
point.h:18:17: note: candidate expects 2 arguments, 0 provided
```

Zašto se to nije ranije dešavalo?

## Zašto se to nije ranije dešavalo?

Kada korisnik ne deklariše nikakav konstruktor, kompajler sam generiše podrazumevani konstruktor. On u opštem slučaju ne garantuje inicijalizaciju polja, ali osigurava da će objekat biti kreiran

Kada deklarišemo objekat pisanjem `identifikator_klase identifikator;`, kompajler će zapravo pozvati podrazumevani konstruktor `identifikator_klase::identifikator_klase();`

# Kako da rešimo naš problem?



# Kako da rešimo naš problem?

Prosleđivanjem parametara u pozivu

# Kako da rešimo naš problem?

A možemo i upotrebom podrazumevanih parametara

## Podrazumevani parametri

```
std::string label;  
  
static bool weight_points;  
  
public:  
    //Konstruktori  
    Point(float x_ = -3, float y_ = -3);  
  
    //Getters  
    float get_x() const;  
    float get_y() const;  
    const std::string& get_label() const;  
    static bool get_weight_points();
```

point.h

18,36-50

Slično kao i u Python-u, podrazumevane vrednosti parametara

## Podrazumevani parametri

```
std::string label;  
  
static bool weight_points;  
  
public:  
    //Konstruktori  
    Point(float x_ = -3, float y_ = -3);  
  
    //Getters  
    float get_x() const;  
    float get_y() const;  
    const std::string& get_label() const;  
    static bool get_weight_points();
```

point.h

18,36-50

Pošto je prosleđivanje parametara poziciono, ako param<sub>i</sub> ima deklarisanu

## Podrazumevani parametri

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

Point::Point(float x_, float y_)
{
    x = x_;
    y = y_;
}
```

Podrazumevane vrednosti navodimo samo u deklaraciji, ne u definiciji

## Podrazumevani parametri

```
cout << "Center: " << center.get_weight_points() << endl;
cout << "Corner: " << corner.get_weight_points() << endl;

corner.set_weight_points(true);
cout << "Center: " << center.get_weight_points() << endl;
cout << "Corner: " << corner.get_weight_points() << endl;

Point stack_point;
cout << "center = (" << center.get_x()
    << ", " << center.get_y() << ")\n";
cout << "stack_point = (" << stack_point.get_x()
    << ", " << stack_point.get_y() << ")\n";
```

test\_point.cpp

43,0-1

93%

Ako pri pozivu funkcije ne prosledimo neki parametar, on će zadržati svoju

## Podrazumevani parametri

```
Unesite x koordinatu centra: 6  
6  
12  
Center: 0  
Corner: 0  
Center: 1  
Corner: 1  
center = (12, -3)  
stack_point = (-3, -3)
```

Ako pri pozivu funkcije ne prosledimo neki parametar, on će zadržati svoju

## Podrazumevani parametri

```
cout << "Center: " << center.get_weight_points() << endl;
cout << "Corner: " << corner.get_weight_points() << endl;

corner.set_weight_points(true);
cout << "Center: " << center.get_weight_points() << endl;
cout << "Corner: " << corner.get_weight_points() << endl;

Point stack_point(9, 9);
cout << "center = (" << center.get_x()
    << ", " << center.get_y() << ")\n";
cout << "stack_point = (" << stack_point.get_x()
    << ", " << stack_point.get_y() << ")\n";
```

test\_point.cpp

38,25-32



## Podrazumevani parametri

```
Unesite x koordinatu centra: 6  
6  
12  
Center: 0  
Corner: 0  
Center: 1  
Corner: 1  
center = (12, -3)  
stack_point = (9, 9)
```

U suprotnom, biće korišćena prosleđena vrednost

## Podrazumevani parametri

```
Point stack_point(9);  
cout << "center = (" << center.get_x( )  
      << ", " << center.get_y( ) << ")\n";  
cout << "stack_point = (" << stack_point.get_x( )  
      << ", " << stack_point.get_y( ) << ")\n";
```

test\_point.cpp

38,22-29

Naravno, možemo zameniti i samo x\_ ali ne i samo y\_, zbog pozicionog navođenja parametara (C++ nema imenovano navođenje kao Python)

## Podrazumevani parametri

```
Unesite x koordinatu centra: 6
6
12
Center: 0
Corner: 0
Center: 1
Corner: 1
center = (12, -3)
stack_point = (9, -3)
```

Naravno, možemo zameniti i samo `x_` ali ne i samo `y_`, zbog pozicionog navođenja parametara (C++ nema imenovano navođenje kao Python)

# Podrazumevani parametri

Podrazumevane parametre možemo koristiti za sve funkcije, a ne samo za konstruktore

## Kako smo ovo još mogli da rešimo?

```
test_point.cpp: In function 'int main()':  
test_point.cpp:18:15: error: no matching function for call to 'Point::Point()'  
   18 |         Point center;  
      |         ^~~~~~
```

## Mogli smo da napišemo dva konstruktora

```
public:
    //Konstruktori
    Point();
    Point(float x_, float y_);

    //Getters
    float get_x() const;
    float get_y() const;
    const std::string& get_label() const;
```

point.h

20,0-1

## Mogli smo da napišemo dva konstruktora

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

Point::Point()
{
    x = -3;
    y = -3;
}

Point::Point(float x_, float y_)
{
    x = x_;
    y = y_;
}
```

## Mogli smo da napišemo dva konstruktora

```
//centar je deklarisan sa Point center;  
Point stack_point(9, 9);  
cout << "center = (" << center.get_x()  
      << ", " << center.get_y() << ")\n";  
cout << "stack_point = (" << stack_point.get_x()  
      << ", " << stack_point.get_y() << ")\n";
```

test\_point.cpp

38,40-47



## Mogli smo da napišemo dva konstruktora

```
Unesite x koordinatu centra: 6  
6  
12  
Center: 0  
Corner: 0  
Center: 1  
Corner: 1  
center = (12, -3)  
stack_point = (9, 9)
```

Kako kompajler zna koji konstruktor da pozove?

# Preklapanje funkcija (function overloading)

[https://en.cppreference.com/w/cpp/language/overload\\_resolution](https://en.cppreference.com/w/cpp/language/overload_resolution)

[cppreference.com](#) [Log in](#)

Page

Discussion

Standard revision: Diff View Edit History

[C++](#) [C++ language](#) [Functions](#)

## Overload resolution

In order to compile a function call, the compiler must first perform [name lookup](#), which, for functions, may involve [argument-dependent lookup](#), and for function templates may be followed by [template argument deduction](#).

If the name refers to more than one entity, it is said to be *overloaded*, and the compiler must determine which overload to call. In simple terms, the overload whose parameters match the arguments most closely is the one that is called.

In detail, overload resolution proceeds through the following steps:

1. Building the set of [candidate functions](#).
2. Trimming the set to only [viable functions](#).
3. Analyzing the set to determine the single [best viable function](#) (this may involve [ranking of implicit conversion sequences](#)).

```
void f(long);
void f(float);

f(0L); // calls f(long)
f(0);  // error: ambiguous overload
```

Besides function calls, overloaded function names may appear in several additional contexts, where different rules apply: see [Address of an overloaded function](#).

If a function cannot be selected by overload resolution, it cannot be used (e.g. it is a [templated entity](#) with a failed [constraint](#)).

## Konstruktor se poziva i prilikom dinamičke alokacije objekata

```
Point* heap_point = new Point(25, -25);  
cout << "heap_point = (" << heap_point -> get_x( )  
      << ", " << heap_point -> get_y( ) << ")\n";  
delete heap_point;
```

test\_point.cpp

48,19-26

Konstruktor se poziva i prilikom dinamičke alokacije objekata

```
Unesite x koordinatu centra: 6  
6  
12  
Center: 0  
Corner: 0  
Center: 1  
Corner: 1  
center = (12, -3)  
stack_point = (9, 9)  
heap_point = (25, -25)
```

# Kopirajući (Copy) konstruktor

Konstruktoru možemo da prosledimo i postojeći objekat date klase, čijim će kopiranjem biti sačinjen kreirani objekat

## Kopirajući (Copy) konstruktor

```
static bool weight_points;  
  
public:  
    //Konstruktori  
    Point();  
    Point(float x_, float y_);  
    Point(const Point& p);  
  
    //Getters  
    float get_x() const;  
    float get_y() const;
```

point.h

20,3-17

# Kopirajući (Copy) konstruktor

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

Point::Point()
{
    x = -3;
    y = -3;
}

Point::Point(float x_, float y_)
{
    x = x_;
    y = y_;
}

Point::Point(const Point& p)
{
    x = p.x;
    y = p.y;
}
```



## Kopirajući (Copy) konstruktor

```
//centar je deklarisan sa Point center;  
Point stack_point(9, 9);  
cout << "center = (" << center.get_x()  
      << ", " << center.get_y() << ")\n";  
cout << "stack_point = (" << stack_point.get_x()  
      << ", " << stack_point.get_y() << ")\n";  
  
Point* heap_point = new Point(center);  
cout << "heap_point = (" << heap_point -> get_x()  
      << ", " << heap_point -> get_y() << ")\n";  
delete heap_point;  
  
return 0;
```

test\_point.cpp

50,10-17

## Kopirajući (Copy) konstruktor

```
Unesite x koordinatu centra: 6
6
12
Center: 0
Corner: 0
Center: 1
Corner: 1
center = (12, -3)
stack_point = (9, 9)
heap_point = (12, -3)
```

## Da li će ovo raditi?

```
std::string label;  
  
static bool weight_points;  
  
public:  
    //Konstruktori  
    Point();  
    Point(float x_, float y_);  
    //Point(const Point& p);  
  
    //Getters  
    float get_x() const;  
    float get_y() const;
```

point.h

20,4-18

## Da li će ovo raditi?

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

Point::Point()
{
    x = -3;
    y = -3;
}

Point::Point(float x_, float y_)
{
    x = x_;
    y = y_;
}

//Point::Point(const Point& p)
//{
//    x = p.x;
//    y = p.y;
//}
```

## Da li će ovo raditi?

```
//centar je deklarisan sa Point center;
Point stack_point(9, 9);
cout << "center = (" << center.get_x()
      << ", " << center.get_y() << ")\n";
cout << "stack_point = (" << stack_point.get_x()
      << ", " << stack_point.get_y() << ")\n";

Point* heap_point = new Point(center);
cout << "heap_point = (" << heap_point -> get_x()
      << ", " << heap_point -> get_y() << ")\n";
delete heap_point;

return 0;
```

test\_point.cpp

50,10-17

Da li će ovo raditi?

```
Unesite x koordinatu centra: 7
7
13
Center: 0
Corner: 0
Center: 1
Corner: 1
center = (13, -3)
stack_point = (9, 9)
heap_point = (13, -3)
```

# Implicitni kopirajući konstruktor

https://en.cppreference.com/w/cpp/language/copy\_constructor

- function return: `return a;` inside a function such as `T f()`, where `a` is of type `T`, which has no `move constructor`.

## Implicitly-declared copy constructor

If no user-defined copy constructors are provided for a class type, the compiler will always declare a copy constructor as a non-`explicit inline public` member of its class. This implicitly-declared copy constructor has the form

`T::T(const T&)` if all of the following are true:

- each direct and virtual base `B` of `T` has a copy constructor whose parameters are of type `const B&` or `const volatile B&`;
- each non-static data member `M` of `T` of class type or array of class type has a copy constructor whose parameters are of type `const M&` or `const volatile M&`.

Otherwise, the implicitly-declared copy constructor is `T::T(T&)`.

Due to these rules, the implicitly-declared copy constructor cannot bind to a `volatile lvalue` argument.

A class can have multiple copy constructors, e.g. both `T::T(const T&)` and `T::T(T&)`.

Even if some user-defined copy constructors are present, the user may still force the implicit copy constructor declaration with the keyword `default`. (since C++11)

The implicitly-declared (or defaulted on its first declaration) copy constructor has an exception specification as described in `dynamic exception specification(until C++17)` `noexcept specification(since C++17)`.

# Implicitni kopirajući konstruktor

Implicitni kopirajući konstruktor će rekurzivno pozvati kopirajući konstruktor nad svim objektima unutar datog objekta (više o tome uskoro), a ostala polja će preslikati prostom dodelom



## Plitka naspram duboke kopije (Shallow vs Deep Copy)

```
int a[10];  
int* b = &a;  
int* c = b;
```

Jesu li nizovi u b i c isti ili različiti objekti u memoriji?

## Plitka naspram duboke kopije (Shallow vs Deep Copy)

```
int a[10];  
int* b = &a;  
int* c = b;
```

Isti su. `b[3] = 10` će prouzrokovati da i `c[3]` bude 10.

## Plitka naspram duboke kopije (Shallow vs Deep Copy)

```
int a[10];  
int* b = &a;  
int* c = b;
```

obj1.p = obj2.p dovodi do istog efekta: polje p oba objekta će pokazivati na isti objekat u memoriji

## Plitka naspram duboke kopije (Shallow vs Deep Copy)

Ako nam je potrebno da polja pokazuju na različite kopije objekta u memoriji, potrebno je da ručno izvršimo to kopiranje

Na taj način pravimo takozvanu *duboku kopiju*

# Operator dodele

Implicitno definisan operator dodele (=) takođe pravi plitku kopiju

# Preklapanje operatora

---

## U C++-u možemo da preklapamo i operatore

Sve je isto kao i kod deklaracije i definicije ostalih funkcija članica, samo je identifikator zamenjen sa **operator**naziv\_operatora

Ako je preklapanje deklarirano unutar klase, objekat nad kojim pozivamo binarni operator je levi operand, a onaj koji prosleđujemo je desni operand

```
Point operator+(const Point& p) const;

//Friend functions
friend void inc_x(Point& p, float inc_amount);

};

std::ostream& operator<<(std::ostream& os, const Point& p);
#endif
point.h
```

38,1-8

Bot



## Primer

```
Point Point::operator+(const Point& p) const
{
    Point sum(x + p.x, y + p.y);

    return sum;
}

std::ostream& operator<<(std::ostream& os, const Point& p)
{
    os << "(" << p.get_x() << ", " << p.get_y() << ")";

    return os;
}
```

```
//centar je deklarisan sa Point center;  
Point stack_point(9, 9);  
cout << center + stack_point << endl;  
  
return 0;
```

```
}
```



```
~
```

```
~
```

```
~
```

```
~
```

```
~
```

test\_point.cpp [ + ]

45,1

```
Unesite x koordinatu centra: 6  
6  
12  
Center: 0  
Corner: 0  
Center: 1  
Corner: 1  
(21, 6)
```

```
Point operator+(const Point& p) const;

//Friend functions
friend void inc_x(Point& p, float inc_amount);

};

std::ostream& operator<<(std::ostream& os, const Point& p);
#endif
point.h 38,1-8 Bot
```

Primetimo da je levi operand za « ostream (na primer cout), pa nam je zato preklopljen operator funkcija koja nije deo klase Point.

I operator + možemo implementirati kao spoljnu funkciju. Tada moramo oba sabirana objekta da prosledimo eksplicitno, prilikom deklaracije i definicije. Pozivi ostaju nepromenjeni.

```
//Other members
float cheby_dist(const Point& center);

//Friend functions
friend void inc_x(Point& p, float inc_amount);

};

Point operator+(const Point& a, const Point& b);
std::ostream& operator<<(std::ostream& os, const Point& p);
#endif
```

~

point.h

41,47

Bot

## Primer

```
Point operator+(const Point& a, const Point& b)
{
    Point sum(a.get_x() + b.get_x(), a.get_y() + b.get_y());

    return sum;
}

std::ostream& operator<<(std::ostream& os, const Point& p)
{
    os << "(" << p.get_x() << ", " << p.get_y() << ")";

    return os;
}

float Point::get_x() const
{
    return this -> x;
}
```

```
//centar je deklarisan sa Point center;  
Point stack_point(9, 9);  
cout << center + stack_point << endl;  
  
return 0;
```

```
}
```

```
~
```

```
~
```

```
~
```

```
~
```

```
~
```

test\_point.cpp

45,0-1



```
Unesite x koordinatu centra: 7
7
13
Center: 0
Corner: 0
Center: 1
Corner: 1
(22, 6)
```

# Destruktori

---

Kao što smo već više puta videli, neophodno je da uklonimo iz memorije sve što smo ranije rezervisali, a nije nam više potrebno

Za dealokaciju prostora rezervisanog za objekat, zadužen je destruktor

Kao što se i konstruktor poziva prilikom alokacije, čak i kada je ona sasvim prosta i ne sadrži nikakvu inicijalizaciju, tako se i destruktor poziva čak i kada nema potrebe za dealokacijom dinamički alocirane memorije

Sintaksa je ista kao i za podrazumevani konstruktor, samo ispred identifikatora stavljamo i znak ~

```
        static bool weight_points;

public:
    //Konstruktor
    Point();
    Point(float x_, float y_);

    //Destruktor
    ~Point();

    //Getters
    float get_x() const;
```

point.h

22,11-25

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

Point::Point()
{
    x = -3;
    y = -3;
}

Point::Point(float x_, float y_)
{
    x = x_;
    y = y_;
}

Point::~Point()
{
    std::cout << "Destruktor je pozvan nad tackom "
                << *this << std::endl;
}

Point operator+(const Point& a, const Point& b)
{
    Point sum(a.get_x() + b.get_x(), a.get_y() + b.get_y());
    return sum;
}
```

```
corner.set_weight_points(true);  
cout << "Center: " << center.get_weight_points() << endl;  
cout << "Corner: " << corner.get_weight_points() << endl;  
  
//centar je deklarisan sa Point center;  
Point* heap_point = new Point(9, 9);  
cout << center + *heap_point << endl;  
  
delete heap_point;  
  
return 0;  
}
```

test\_point.cpp

39,30-37

# Primer

```
Unesite x koordinatu centra: 6
6
12
Center: 0
Corner: 0
Center: 1
Corner: 1
(21, 6)
Destruktor je pozvan nad tackom (21, 6)
Destruktor je pozvan nad tackom (9, 9)
Destruktor je pozvan nad tackom (-3, -3)
Destruktor je pozvan nad tackom (12, -3)
```



# Šta se ovde sve desilo?

```
Unesite x koordinatu centra: 6
6
12
Center: 0
Corner: 0
Center: 1
Corner: 1
(21, 6)
Destruktor je pozvan nad tackom (21, 6)
Destruktor je pozvan nad tackom (9, 9)
Destruktor je pozvan nad tackom (-3, -3)
Destruktor je pozvan nad tackom (12, -3)
```

# Šta se ovde sve desilo?

```
Unesite x koordinatu centra: 6
```

```
6
```

```
12
```

```
Center: 0
```

```
Corner: 0
```

```
Center: 1
```

```
Corner: 1
```

```
(21, 6)
```

```
Destruktor je pozvan nad tackom (21, 6) oslobađa se rezultat poziva operatora +
```

```
Destruktor je pozvan nad tackom (9, 9) poziv operatora delete
```

```
Destruktor je pozvan nad tackom (-3, -3)
```

```
Destruktor je pozvan nad tackom (12, -3) čišćenje preostalih promenljivih sa steka,  
nakon završetka poziva funkcije main
```

# Kompozicija

---

Često je korisno da jedna klasa kao polje sadrži objekat druge klase

# Primer

```
#ifndef DISTANCES_H
#define DISTANCES_H

#include <iostream>

class Trace
{
public:
    Trace(){std::cout << "Pozvan je konstruktor klase Trace.\n";}
    ~Trace(){std::cout << "Pozvan je destruktor klase Trace.\n";}
};

class Point
{
private:
    float x;
    float y;

    std::string label;

    static bool weight_points;

    Trace trace;

public:
    //Konstruktori
```

Kako ćemo dobiti objekat trace?

## Kako ćemo dobiti objekat trace?

Prilikom kreiranja objekta klase A, kompajler će prvo pozvati konstruktore svih objekata deklariranih u klasi A, redom kojim su navedeni

Ukoliko ima više nivoa kompozicije, uradiće to rekursivno

```
Pozvan je konstruktor klase Trace.  
Pozvan je konstruktor klase Point.  
Unesite x koordinatu centra: 6  
6  
12  
Pozvan je konstruktor klase Trace.  
Pozvan je konstruktor klase Point.  
Center: 0  
Corner: 0  
Center: 1  
Corner: 1  
Pozvan je konstruktor klase Trace.  
Pozvan je konstruktor klase Point.  
Pozvan je konstruktor klase Trace.  
Pozvan je konstruktor klase Point.  
(21, 6)  
Destruktor je pozvan nad tackom (21, 6)  
Pozvan je destruktorklase Trace.  
Destruktor je pozvan nad tackom (9, 9)  
Pozvan je destruktorklase Trace.  
Destruktor je pozvan nad tackom (-3, -3)  
Pozvan je destruktorklase Trace.  
Destruktor je pozvan nad tackom (12, -3)  
Pozvan je destruktorklase Trace.
```



# Nasleđivanje

---

# Implementacija pottipova (Subtyping)

Setimo se definicije definicije:

Vrsni pojam + klasna razlika

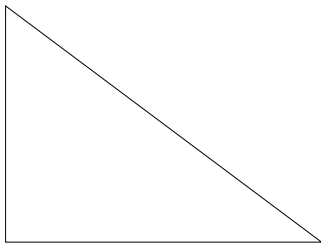
Pravougli trougao

Šta je vrsni pojam, a šta klasna razlika?

# Da li je ovo ipsravan odgovor?

Učitaljica: „Nacrtajte trougao”

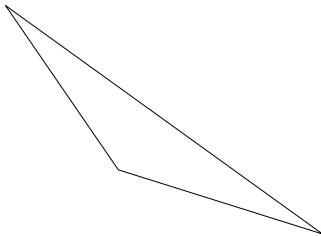
Učenik:



# A ovo?

Učiteljica: „Nacrtajte pravougli trougao”

Učenik:



# Šta je to pottip

Kadgod imamo tip B koji uvek može biti korišćen umesto nekog drugog tipa A, kažemo da je B pottip tipa A

Uslov za ovo je da je skup interfejsa tipa A striktno podskup interfejsa tipa B

## Kada je ovo korisno?

Ako smo razvijali neki kod, a zatim se pojavila potreba za proširenom funkcionalnošću tipa A, možemo napraviti pottip B koji će sav pređašnji kod moći da koristi kao da se ništa nije ni promenilo, a samo novi kod kom je potrebna proširena funkcionalnost će morati da koristi i nove interfejse

## U našem primeru

Klasa Point sa poljem trace je pottip klase Point, jer gde god smo koristili objekat klase Point, možemo koristiti i objekat klase Point sa poljem trace



# Da bismo mogli da iskoristimo i stari kod

C++ omogućuje nasleđivanje (eng. Inheritance):

```
class identifikator_izvedene_klase : vrsta_nasleđivanja identifikator_bazne_klase
```

# U našem primeru

```
class DerivedPoint : public Point
{
    private:
        Trace trace;
    public:
        DerivedPoint() {x = -6; y = -6; std::cout << "Pozvan je konstruktor DerivedPoint klase.\n";}
        DerivedPoint(float x_, float y_) {x = x_; y = y_; std::cout << "Pozvan je konstruktor DerivedPoint klase.\n";}
        ~DerivedPoint() {std::cout << "Pozvan je destruktor klase DerivedPoint\n";}
};

#endif
```

## U našem primeru

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

using namespace std;

int main()
{
    DerivedPoint center;

    float x;
    cout << "Unesite x koordinatu centra: ";
    cin >> x;
    center.set_x(x);
    cout << center.get_x() << endl;

    //centar je deklarisan sa DerivedPoint center;
    DerivedPoint* heap_point = new DerivedPoint(9, 9);
    cout << center + *heap_point << endl;

    delete heap_point;

    return 0;
}
```

# U našem primeru

```
In file included from point.cpp:2:
point.h: In constructor 'DerivedPoint::DerivedPoint()':
point.h:51:33: error: 'float Point::x' is private within this context
   51 |         DerivedPoint() {x = -6; y = -6; std::cout << "Pozvan je konstruktor DerivedPoint klase.\n";}
      |                             ^
point.h:16:23: note: declared private here
   16 |         float x;
      |         ^
point.h:51:41: error: 'float Point::y' is private within this context
   51 |         DerivedPoint() {x = -6; y = -6; std::cout << "Pozvan je konstruktor DerivedPoint klase.\n";}
      |                             ^
point.h:17:23: note: declared private here
   17 |         float y;
      |         ^
point.h: In constructor 'DerivedPoint::DerivedPoint(float, float)':
point.h:52:51: error: 'float Point::x' is private within this context
   52 |         DerivedPoint(float x_, float y_) {x = x_; y = y_; std::cout << "Pozvan je konstruktor DerivedPoint klase.\n";}
      |                                         ^
point.h:16:23: note: declared private here
   16 |         float x;
      |         ^
```

# U našem primeru

```
class DerivedPoint : public Point
{
    private:
        Trace trace;
    public:
        DerivedPoint() {this -> set_x(-6); this-> set_y(-6); std::cout << "Pozvan je konstruktor DerivedPoint klase.\n";}
        DerivedPoint(float x_, float y_) {this -> set_x(x_); this -> set_y(y_); std::cout << "Pozvan je konstruktor DerivedPoint klase.\n";}
        ~DerivedPoint() {std::cout << "Pozvan je destruktorklase.\n";}
};

#endif
point.h
```

56,6

97%

## U našem primeru

```
Pozvan je konstruktor klase Point.  
Pozvan je konstruktor klase Trace.  
Pozvan je konstruktor DerivedPoint klase.  
Unesite x koordinatu centra: 6  
6  
Pozvan je konstruktor klase Point.  
Pozvan je konstruktor klase Trace.  
Pozvan je konstruktor DerivedPoint klase.  
Pozvan je konstruktor klase Point.  
(15, 3)  
Destruktor je pozvan nad tackom (15, 3)  
Pozvan je destruktorklase DerivedPoint  
Pozvan je destruktorklase Trace.  
Destruktor je pozvan nad tackom (9, 9)  
Pozvan je destruktorklase DerivedPoint  
Pozvan je destruktorklase Trace.  
Destruktor je pozvan nad tackom (6, -6)
```

## U našem primeru

```
class Point
{
    protected:
        float x;
        float y;

        std::string label;

    public:
        //Konstruktori
}
point.h
```

# U našem primeru

```
class DerivedPoint : public Point
{
    private:
        Trace trace;
    public:
        DerivedPoint() {x = -6; y = -6; std::cout << "Pozvan je konstruktor DerivedPoint klase.\n";}
        DerivedPoint(float x_, float y_) {x = x_; y = y_; std::cout << "Pozvan je konstruktor DerivedPoint klase.\n";}
        ~DerivedPoint() {std::cout << "Pozvan je destruktor klase DerivedPoint\n";}
};
#endif
```



## U našem primeru


```
31
Pozvan je konstruktor klase Point.
Pozvan je konstruktor klase Trace.
Pozvan je konstruktor DerivedPoint klase.
Unesite x koordinatu centra: 6
6
Pozvan je konstruktor klase Point.
Pozvan je konstruktor klase Trace.
Pozvan je konstruktor DerivedPoint klase.
Pozvan je konstruktor klase Point.
(15, 3)
Destruktor je pozvan nad tackom (15, 3)
Pozvan je destruktorklase DerivedPoint
Pozvan je destruktorklase Trace.
Destruktor je pozvan nad tackom (9, 9)
Pozvan je destruktorklase DerivedPoint
Pozvan je destruktorklase Trace.
Destruktor je pozvan nad tackom (6, -6)
```

# Tipovi nasleđivanja

← → ↺

🔒 https://stackoverflow.com/questions/860339/what-is-the-difference-between-public-private-a 📄 90% ☆

🔔 ⬇️ 👤 📄 ☰

 About Products For Teams 🔍 Search... [Log in](#) [Sign up](#)

🏠 Home

**❏ Questions**

🏷️ Tags

👤 Users

🏢 Companies

**LABS** ⓘ

💬 Discussions

**COLLECTIVES** +  
Communities for your favorite technologies. [Explore all Collectives](#)

**TEAMS**  
    
Ask questions, find answers and collaborate at work with Stack Overflow for Teams.  
[Explore Teams](#)  
[Create a free Team](#)

▲

▼

🔖

🕒

**1789**

```
class A
{
    public:
        int x;
    protected:
        int y;
    private:
        int z;
};

class B : public A
{
    // x is public
    // y is protected
    // z is not accessible from B
};

class C : protected A
{
    // x is protected
    // y is protected
    // z is not accessible from C
};

class D : private A    // 'private' is default for classes
{
    // x is private
    // y is private
    // z is not accessible from D
};
```

ChatGPT is banned  
⚙️ Pausing the 1-rep voting experiment on Stack Overflow: reflecting on the...

16 people chatting  
Lounge<C++>  
7 hours ago - LandonZeKipitOfGreytbrin  
  
C++ Questions and Answers  
13 hours ago - Sildrone  


**Linked**  

50 What are access specifiers? Should I inherit with private, protected or public?

15 c++ inheritance syntax

9 What is difference between protected and private derivation in c++

9 Private inheritance causing problem in c++

2 Protected Inheritance in C++

0 Variable inaccessible despite class

**IMPORTANT NOTE:** Classes B, C and D all contain the variables x, y and z. It is just question of access.

86

# Virtuelne funkcije

---

## Nekada pozivamo metodu preko pokazivača

Pokazivač na objekat bazne klasu može da pokazuje na objekat izvedene klasu

(setimo se potpisivanja: pravougli trougao možemo da koristimo svuda gde se očekuje trougao)

## Nekada pozivamo metodu preko pokazivača

Kompajler bira metodu koju će pozvati, razrešavanjem tipa pokazivača

## Nekada pozivamo metodu preko pokazivača

Ali, to u gornjem slučaju znači da će uvek pozvati metodu bazne klase

(Na primer, računanje površine trougla neće biti moguće formulom  $a * b / 2$ , već će iziskivati traženje visine)

Da bismo to predupredili, koristimo virtuelne metode, koje se razrešavaju tokom izvršenja programa, tako da bude pozvana metoda iz objekta na koji pokazivač pokazuje, a ne iz klase iz tipa pokazivača

# Jedan primer sa StackOverflow-a

https://stackoverflow.com/questions/2391679/why-do-we-need-virtual-functions-in-c

About Products For Teams Search...

Show 1 more comment

28 Answers Sorted by: Highest score (default)

▲ Here is how I understood not just what `virtual` functions are, but why they're required:

3145 ▼ Let's say you have these two classes:

```
class Animal
{
public:
    void eat() { std::cout << "I'm eating generic food."; }
};

class Cat : public Animal
{
public:
    void eat() { std::cout << "I'm eating a rat."; }
};
```

In your main function:



```
Animal *animal = new Animal;
Cat *cat = new Cat;

animal->eat(); // Outputs: "I'm eating generic food."
cat->eat();    // Outputs: "I'm eating a rat."
```

So far so good, right? Animals eat generic food, cats eat rats, all without `virtual`.



# Jedan primer sa StackOverflow-a

 <https://stackoverflow.com/questions/2391679/why-do-we-need-virtual-functions-in-c>  90% 

[/](#) [About](#) [Products](#) [For Teams](#)

Let's change it a little now so that `eat()` is called via an intermediate function (a trivial function just for this example):


```
// This can go at the top of the main.cpp file
void func(Animal *xyz) { xyz->eat(); }
```


Now our main function is:


```
Animal *animal = new Animal;
Cat *cat = new Cat;


func(animal); // Outputs: "I'm eating generic food."
func(cat);    // Outputs: "I'm eating generic food."
```

Uh oh... we passed a Cat into `func()`, but it won't eat rats. Should you overload `func()` so it takes a `Cat*`? If you have to derive more animals from Animal they would all need their own `func()`.


 103 [Do I need perm library?](#)


 [Risk of not capi](#)


 [Why machine v](#)


 [Is there still pro bicycles?](#)


 [Is mathematica](#)


 [What is the poi conservation q](#)

 [Slight cut on fin](#)

 [Would it be pos industrial techn](#)

 [Double bracket](#)

 [Speed up Mani 3d plots in Shor](#)

 [Book on Hilbert](#)

# Jedan primer sa StackOverflow-a

The solution is to make `eat()` from the `Animal` class a virtual function:

```
class Animal
{
    public:
        virtual void eat() { std::cout << "I'm eating generic food."; }
};

class Cat : public Animal
{
    public:
        void eat() { std::cout << "I'm eating a rat."; }
};
```

Main:

```
func(animal); // Outputs: "I'm eating generic food."
func(cat);    // Outputs: "I'm eating a rat."
```

Done.

Hvala na pažnji