

# Objektno orijentisano programiranje

Kompozicija

# Kompozicija

- Da bismo kreirali informacijski sistem koji opisuje bilo koji složeniji sistem iz realnog sveta, pored kreiranja klasa koje opisuju entitete iz realnog sveta neophodno je uvesti mehanizme koji opisuju veze između klasa.
- U realnom životu, kompleksne klase su često izgrađene od manjih, prostijih delova.
- Kompozicija je veza između klasa koja opisuje odnos “poseduje”, “ima”.
- Primeri:
  - Klasa automobil je izgrađena iz školjke automobila, motora, menjača, guma, itd. Da bismo opisali automobil koristimo klasu, ali isto tako da bismo opisali, na primer, njegov motor koristimo klasu.
  - Klasa računar je izgrađena od procesora, memorije, matične ploče, itd. Da bismo opisali računar koristimo klasu, ali isto tako da bismo opisali njegov procesor koristimo klasu.

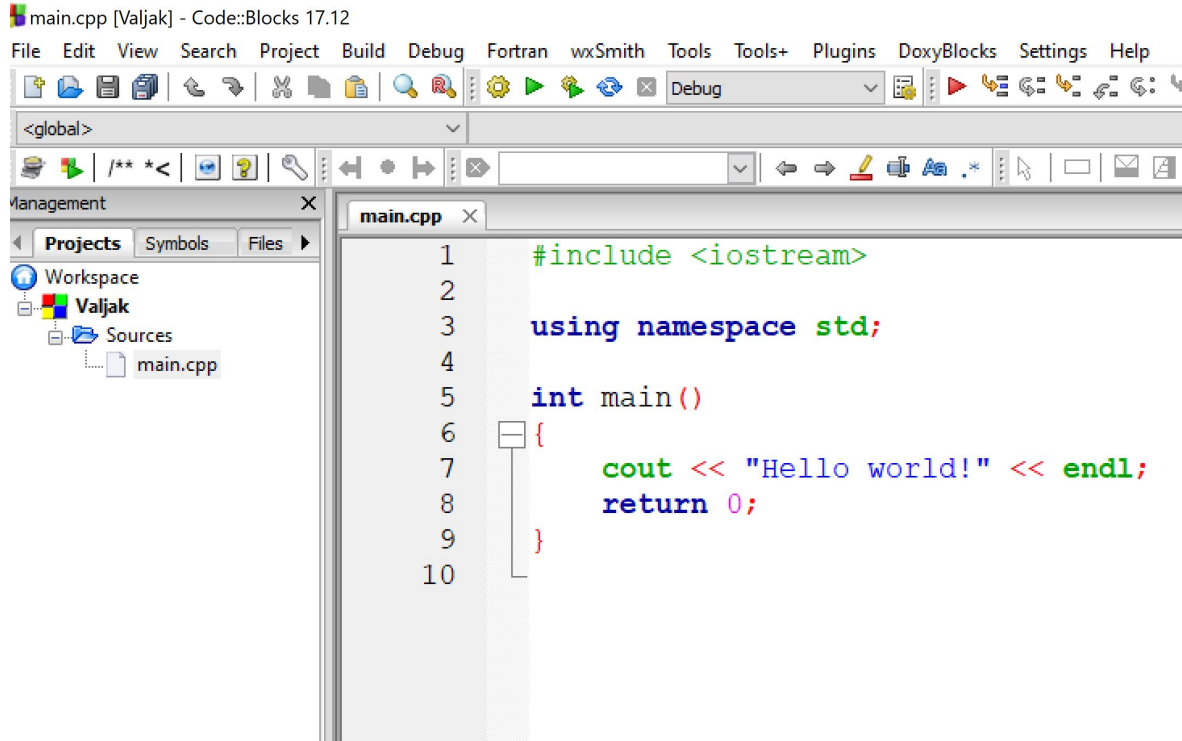
# Kompozicija

- U fazi implementacije, klasa koja predstavlja celinu naziva se **vlasnik**, a klasa koja odgovara delu naziva se **komponenta**. Isti termini se koriste i za pojedinačne objekte.
- Kompozicija jeste takva veza klasa, za koju važi to da vlasnik “poseduje” komponentu, pri čemu komponenta ne može postojati pre kreiranja i posle uništenja vlasnika. Drugim rečima, životni vek komponente sadržan je u životnom veku vlasnika.

# Primer - Klasa Valjak

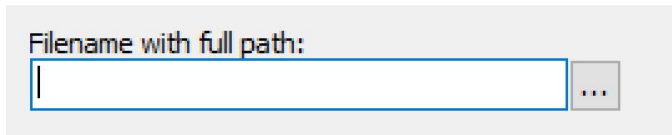
1. Napisati klasu **Krug** opisanu pomoću polja koje predstavlja njegov poluprečnik (double r). Napisati klasu **Pravougaonik** opisanu pomoću polja koja predstavljaju njegove stranice (double a, double b). Napisati klasu **Valjak** koja ima dva objekta člana: B (objekat klase Krug) i M (objekat klase Pravougaonik).

# Rešenje: 1. Kreirati novi projekat u CB



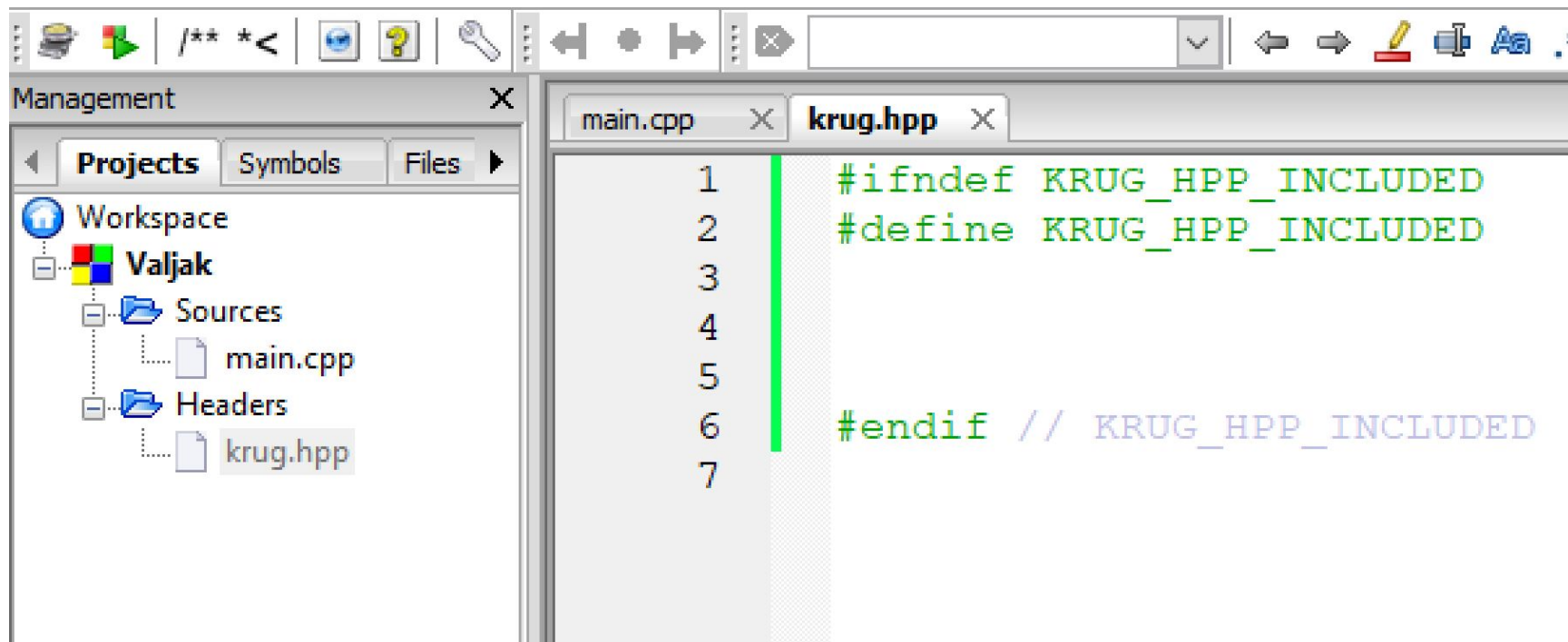
## Rešenje: 2. Dodati krug.hpp fajl u projekat

- Podsećanje: .hpp (header) fajl je potrebno kreirati na sledeći način:
  - U gornjem meniju odabrati File -> New -> File
  - Odabrati "C/C++ header"
  - Kliknuti na tri tačkice kao na slici:
  - Ukucati *krug.hpp* i kliknuti "Save"
  - Označiti "Debug" i "Release" i kliknuti "Finish"



## Rešenje 2. Dodati *krug.hpp* fajl u projekat

- Ukoliko je sve urađeno kako treba, dobija se sledeći prikaz:



## Rešenje: 3.1. Kreirati klasu Krug - polja

- polja pišemo u **private** segmentu, a metode u **public**:

```
main.cpp x *krug.hpp x
1  #ifndef KRUG_HPP_INCLUDED
2  #define KRUG_HPP_INCLUDED
3
4  class Krug{
5  private:
6      ///polja
7  public:
8      ///metode
9  };
10
11
12
13  #endif // KRUG_HPP_INCLUDED
14
```

```
main.cpp x krug.hpp x
1  #ifndef KRUG_HPP_INCLUDED
2  #define KRUG_HPP_INCLUDED
3
4  class Krug{
5  private:
6      ///polja
7      double r; // polunrecnik kruga
8  public:
9      ///metode
10 };
11
12
13
14 #endif // KRUG_HPP_INCLUDED
15
```



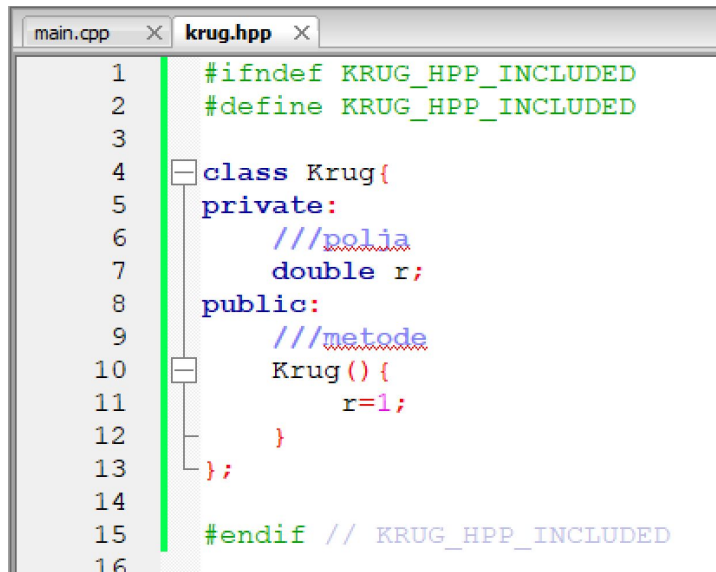
## Rešenje: 3.2. Kreirati klasu Krug - konstruktori

- Potrebno je kreirati konstruktor bez parametara, konstruktor sa parametrima i konstruktor kopije.
- Ispravan način kreiranja konstruktora i ostalih metoda jeste da nakon što svakog kreiramo u fajlu *krug.hpp* izvršimo njegovo testiranje u fajlu *main.cpp* (kreiranjem objekta ili pozivom metode u *main* funkciji).
- Na taj način smanjujemo mogućnost od gomilanja grešaka.

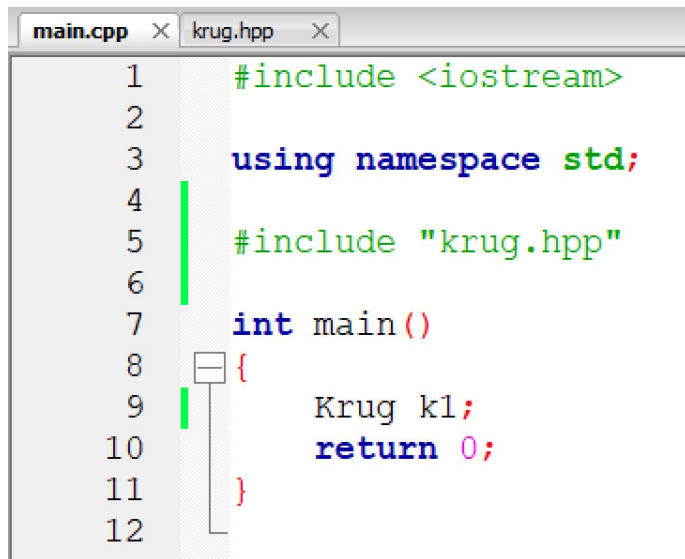
## Rešenje: 3.2. Kreirati klasu Krug - konstruktori

- Konstruktor bez parametara postavlja vrednosti polja na neke vrednosti koje predstavljaju početno stanje. U našem primeru, vrednost polja  $r$  je 1.
- Konstruktor u klasi krug

Kreiranje objekta pomoću konstruktora bez parametara



```
1  #ifndef KRUG_HPP_INCLUDED
2  #define KRUG_HPP_INCLUDED
3
4  class Krug{
5  private:
6      ///polja
7      double r;
8  public:
9      ///metode
10     Krug() {
11         r=1;
12     }
13 };
14
15 #endif // KRUG_HPP_INCLUDED
16
```

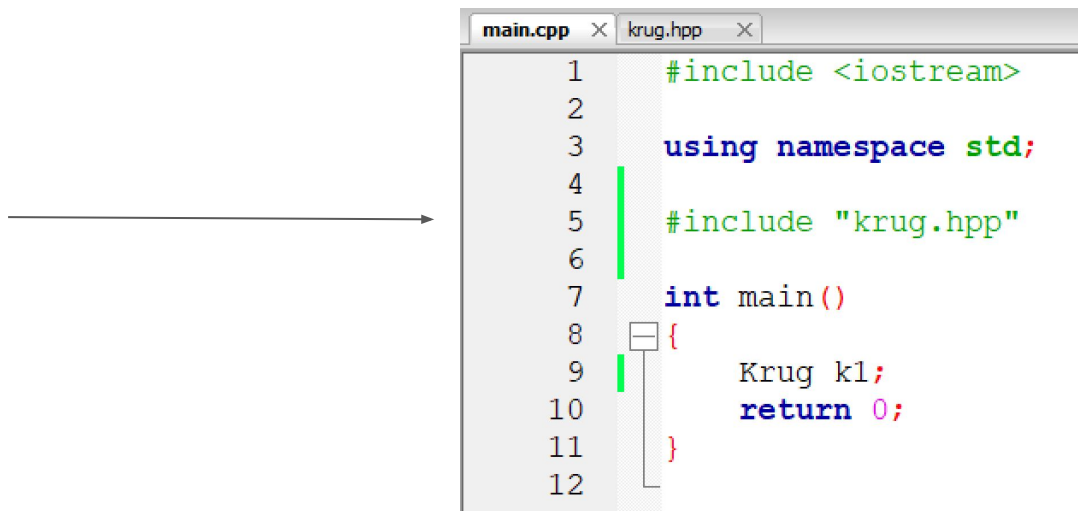


```
1  #include <iostream>
2
3  using namespace std;
4
5  #include "krug.hpp"
6
7  int main()
8  {
9      Krug k1;
10     return 0;
11 }
12
```

## Rešenje: 3.2. Kreirati klasu Krug - konstruktori

- Napomena:

Da bi u *main.cpp* mogli da pristupamo klasi iz *krug.hpp* potrebno je uključiti je pomoću `#include "krug.hpp"`



## Rešenje: 3.2. Kreirati klasu Krug - konstruktori

- Na isti način kreiramo konstruktor sa parametrima i konstruktor kopije i odmah ih pozivamo u *main* funkciji.
- *krug.hpp*:

```
public:
    ///metode
    Krug() {
        r=1;
    }
    Krug(double rr) {
        r=rr;
    }
    Krug(const Krug &k) {
        r=k.r;
    }
};
```

*main.cpp*:

```
int main()
{
    Krug k1, k2(4.5), k3(k2);
    return 0;
}
```

# Konstruktor sa parametrima sa podrazumevanim vrednostima

- Konstruktor sa parametrima sa podrazumevanim vrednostima spaja konstruktor bez parametara i konstruktor sa parametrima.

```
public:
    ///metode
    Krug(double rr=1) {
        r=rr;
    }
    Krug(const Krug &k) {
        r=k.r;
    }
};
```

Ukoliko je prilikom kreiranja “pozvan konstruktor bez parametara”, odnosno parametri nisu prosleđeni, vrednost polja *r* postaje 1.

Na primer: Krug k1;

Ukoliko je prilikom kreiranja “pozvan konstruktor sa parametrima”, odnosno parametri su prosleđeni, vrednost polja *r* postaje vrednost koja je prosleđena.

Na primer: Krug k2(4.5);

# Konstruktor sa parametrima sa podrazumevanim vrednostima

**\*\* Napomena:** Kada pišemo konstruktor sa parametrima sa podrazumevanim vrednostima, bitno je razumeti da onda ne smemo pisati i konstruktor bez parametara i sa parametrima jer će to dovesti do dvosmislenosti zbog čega će nastati greška.

```
public:
    ///metode
    Krug() {
        r=1;
    }
    Krug(double rr=1) {
        r=rr;
    }
    Krug(const Krug &k) {
        r=k.r;
    }
};
```

```
7 int main()
8 {
9     Krug k1, k2(4.5), k3(k2);
10    return 0;
11 }
```

Line	Message
=== Build: Debug in Valjak (compiler: GNU GCC Compiler) ===	
In function 'int main()':	
9	error: call of overloaded 'Krug()' is ambiguous
13	note: candidate: Krug::Krug(double)
10	note: candidate: Krug::Krug()

## Rešenje: 3.3. Kreirati klasu Krug - geter

- geteri se koriste za vraćanje vrednosti polja
- imaju isti tip povratne vrednosti kao i polje koje vraćaju
- nemaju parametre
- stavljamo *const* da osiguramo da u telu metode neće doći do promene vrednosti polja
- *krug.hpp* *main.cpp*

```
public:
    ///metode
    Krug(double rr=1) {
        r=rr;
    }
    Krug(const Krug &k) {
        r=k.r;
    }
    double getR() const{return r;}
```

```
int main()
{
    Krug k1, k2(4.5), k3(k2);
    k1.getR();
    cout<<"Poluprecnik kruga k1 je: "<<k1.getR();
    return 0;
}
```

## Rešenje: 3.4. Kreirati klasu Krug - seter

- seteri se koriste za menjanje vrednosti polja
- obično im je tip povratne vrednost *void* ili *bool*
- imaju parametar istog tipa kao što je polje
- *krug.hpp*

*main.cpp*

```
public:
    ///metode
    Krug(double rr=1) {
        r=rr;
    }
    Krug(const Krug &k) {
        r=k.r;
    }
    double getR() const{return r;}
    void setR(double rr){r=rr;}
```

```
int main()
{
    Krug k1, k2(4.5), k3(k2);
    k1.getR();
    cout<<"Poluprecnik kruga k1 je: "<<k1.getR();
    k1.setR(1.2);
    return 0;
}
```



## Rešenje: 3.5. Kreirati klasu Krug - metode

- potrebno je napisati metodu za izračunavanje površine i zapremine.
- metode vraćaju *double* vrednost
- koristimo *M\_PI* konstantu iz *cmath* biblioteke
- *krug.hpp*:

```
double getO() const{return 2*r*M_PI;}  
double getP() const{return r*r*M_PI;}
```

- *main.cpp*:

```
cout<<"Obim kruga je: "<<k1.getO()<<endl;  
cout<<"Povrsina kruga je: "<<k1.getP()<<endl;
```

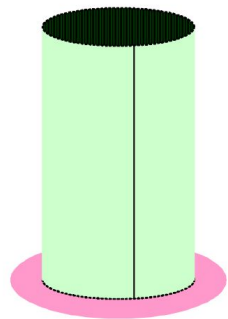
## Rešenje: 4. Kreirati klasu Pravougaonik

- Kreirati *pravougaonik.hpp* i u njemu pisati kod za klasu **Pravougaonik**.
- Uraditi korake kao za klasu **Krug**.
- Rešenje se nalazi u folderu 1, u fajlu *pravougaonik.hpp*.
- Uključiti *pravougaonik.hpp* u *main.cpp*-u i testirati ga kao što je urađeno za klasu **Krug**.

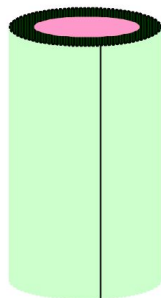
# Rešenje: 5. Kreirati klasu Valjak

**Valjak** se sastoji iz objekta klase **Krug**, koji predstavlja njegovu bazu, i objekta klase **Pravougaonik**, koji predstavlja njegov omotač.

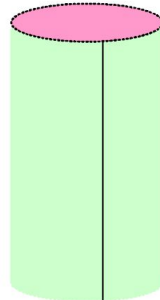
Treba paziti na naredne situacije:



Osnova je “veća” od  
savijenog omotaca



Osnova je “manja” od  
savijenog omotaca



Osnova “odgovara”  
savijenom omotacu

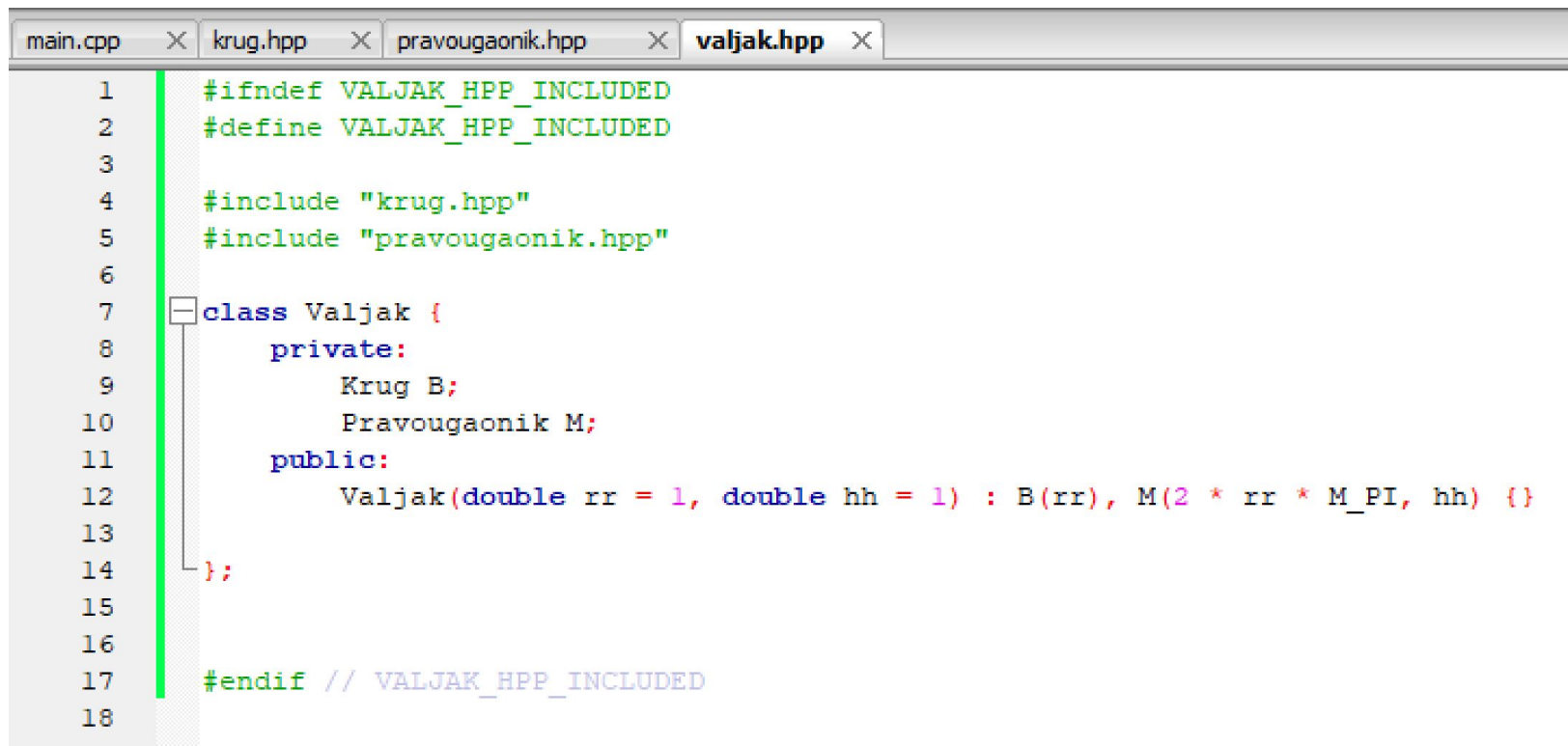
# Rešenje: 5. Kreirati klasu Valjak

- Dakle da bi valjak bio ispravan potrebno je da dužina stranice omotača koja se spaja sa bazom valjka bude jednaka obimu baze valjka.
- Da bismo ovo postigli u kompoziciji koristimo **konstruktor inicijalizator**.
- Najpre treba napraviti polja u klasi Valjak:
  - **Krug B;**
  - **Pravougaonik M;**

\*\*\* Potrebno je u klasi valjak uraditi include hpp fajlova za krug i pravougaonik

- A zatim i konstruktor inicijalizator:
  - **Valjak(double rr = 1, double hh = 1) : B(rr), M(2 \* rr \* M\_PI, hh) {}**
  - On kreira valjak tako što kreira odgovarajući krug i pravougaonik koji ga izgrađuju.

## Rešenje: 5.1. Kreirati Valjak - polja, konstruktori



```
1  #ifndef VALJAK_HPP_INCLUDED
2  #define VALJAK_HPP_INCLUDED
3
4  #include "krug.hpp"
5  #include "pravougaonik.hpp"
6
7  class Valjak {
8      private:
9          Krug B;
10         Pravougaonik M;
11     public:
12         Valjak(double rr = 1, double hh = 1) : B(rr), M(2 * rr * M_PI, hh) {}
13
14 };
15
16
17 #endif // VALJAK_HPP_INCLUDED
18
```

## Rešenje: 5.2. Kreirati klasu Valjak - geteri

- Da bismo dobili poluprečnik baze valjka koristimo geter za poluprečnik kruga iz valjka:

```
double getR() const{  
    return B.getR();  
}
```

- Da bismo dobili visinu valjka koristimo geter za stranicu b pravougaonika iz valjka:

```
double getH() const{  
    return M.getB();  
}
```

## Rešenje: 5.3. Kreirati klasu Valjak - druge metode

- Površina valjka je jednaka zbiru dvostruke površine njegove baze i njegovog omotača. Odnosno treba sabrati dvostruku površinu kruga i pravougaonika koji se nalaze u valjku.

```
double getP() const{  
    return 2 * B.getP() + M.getP();  
}
```

- Zapremina valjka je jednaka površini baze pomnoženoj sa visinom valjka.

```
double getV() const{  
    return B.getP() * getH();  
}
```

## Rešenje: 5.4. Testiranje valjka u main-u

```
#include <iostream>

#include "pravougaonik.hpp"
#include "krug.hpp"

using namespace std;

int main()
{
    Valjak v1;
    Valjak v2(2,4);
    cout << "v1: P = " << v1.getP() << ", V = " << v1.getV() << endl;
    cout << "v2: P = " << v2.getP() << ", V = " << v2.getV() << endl << endl;

    return 0;
}
```



# Zadatak 1-1.

Napisati klase **Automobil**, **Skoljka** i **Menjac** upotrebom kompozicije.

Automobil je kompozicija klasa **Skoljka** i **Menjac**.

(videti tekst zadatka u folderu 1-1)

# Rešenje: Klasa Menjac

- Kao što je rečeno u tekstu zadatka, klasa **Menjac** sadrži polja za tip menjača (nabrojivog tipa) i za broj brzina (int).

```
enum TipMenjaca { AUTOMATIK, MANUELNI };
```

```
class Menjac{  
    private:  
        TipMenjaca tip;  
        int brzina;
```

```
|
```

```
};
```

# Rešenje: Klasa Menjac

- Sada je potrebno napisati metode, najpre konstruktore, a potom i getere i setere.

```
enum TipMenjaca { AUTOMATIK, MANUELNI };

class Menjac{
private:
    TipMenjaca tip;
    int brzina;

public:
    Menjac(){ tip = AUTOMATIK; brzina = 1;}
    Menjac(TipMenjaca t, int b) { tip = t; brzina = b;}
    Menjac(const Menjac& m) { tip = m.tip; brzina = m.brzina;}

    bool setBrzina(int b){
        if (b >= 1 && b <= 6){
            brzina = b;
            return true;
        }
        return false;
    }

    void setTip(TipMenjaca t) { tip = t; }
    int getBrzina() const { return brzina;}
    TipMenjaca getTip() const { return tip;}
};
```

# Rešenje: Klasa Skoljka

- Kao što je rečeno u tekstu zadatka, klasa **Skoljka** ima jedno polje za boja školjke (nabrojivog tipa). Pored polja treba napisati konstruktore i getere i setere.

```
enum BojaSkoljke { PLAVA, CRVENA, ZELENA};

class Skoljka {

private:
    BojaSkoljke boja;

public:
    Skoljka() { boja = PLAVA; }
    Skoljka(BojaSkoljke b) { boja = b; }
    Skoljka(const Skoljka& s) { boja = s.boja; }

    void setBoja(BojaSkoljke b){ boja = b; }
    BojaSkoljke getBoja() const { return boja; }

};
```

# Rešenje: Klasa Automobil

- Klasa **Automobil** ima dva polja, polje *m* koje je objekat klase **Menjac** i polje *s* koje je objekat klase **Skoljka**.

```
#include "menjac.hpp"
#include "skoljka.hpp"
```

```
class Automobil {
```

```
private:
```

```
    Menjac m;
```

```
    Skoljka s;
```

```
public:
```

```
    Automobil() : m(), s() {}
```

```
    Automobil(TipMenjaca tm, int br, BojaSkoljke bs) : m(tm,br), s(bs) {}
```

```
};
```

← konstruktori  
← inicijalizatori

# Rešenje: Klasa Automobil - geteri i seteri

- geteri i seteri u klasi **Automobil** rade tako što pozivaju odgovarajuće getere i setere iz objekta članova **Skoljke** i **Menjaca**.

```
int getBrzina() const {  
    return m.getBrzina();  
}
```

```
TipMenjaca getTipMenjaca() {  
    return m.getTip();  
}
```

```
BojaSkoljke getBoja() const {  
    return s.getBoja();  
}
```

```
bool setBrzina(int br) {  
    return m.setBrzina(br);  
}
```

```
void setBoja(BojaSkoljke bs) {  
    s.setBoja(bs);  
}
```

```
void setTipMenjaca(TipMenjaca tm) {  
    m.setTip(tm);  
}
```

# Rešenje: Funkcija za ispis polja

- Za svaku od klasa moguće je napraviti slobodnu funkciju koja ispisuje sve vrednosti polja iz klase.
- Primer:

```
void ispisiMenjac(const Menjac& m){  
  
    cout << "Menjac: " << endl;  
    cout << "\tBrzina = " << m.getBrzina() << endl;  
    cout << "\tTip = " << (m.getTip() == AUTOMATIK ? "Automatik" : "Manuelni") << endl;  
}
```

- Vidimo da je potrebno napraviti getere i setere i pozvati ih u funkciji kako bi ispisali sva polja menjača.

# Prijateljske funkcije

- Da ne bismo morali da koristimo getere u funkcijama za ispis (ili drugim funkcijama koje ne menjaju polja objekta) koristimo prijateljske funkcije.
- Prijateljska funkcija može da pristupa private delovima klase.
- Da bismo proglasili funkciju prijateljskom, ispred njene deklaracije dodajemo ključnu reč *friend*.
- Funkcija se proglašava za prijateljsku u klasi čijim poljima želimo da pristupamo bez korišćenja getera.



# Prijateljske funkcije

- Primer:
- *menjac.hpp*:

```
class Menjac{
private:
    TipMenjaca tip;
    int brzina;

public:
    Menjac(){ tip = AUTOMATIK; brzina = 1;}
    Menjac(TipMenjaca t, int b) { tip = t; brzina = b;}
    Menjac(const Menjac& m) { tip = m.tip; brzina = m.brzina;}

    bool setBrzina(int b){
        if (b >= 1 && b <= 6){
            brzina = b;
            return true;
        }
        return false;
    }

    void setTip(TipMenjaca t) { tip = t; }
    int getBrzina() const { return brzina;}
    TipMenjaca getTip() const { return tip;}

    friend void ispisiMenjac(const Menjac&); // prijateljska funkcija
};
```

- *main.cpp*

```
void ispisiMenjac(const Menjac& m){

    cout << "Menjac: " << endl;
    cout << "\tBrzina = " << m.brzina << endl;
    cout << "\tTip = " << (m.tip == AUTOMATIK ? "Automatik" : "Manuelni") << endl;
}
```

## Zadatak 2-1.

Napisati klase **Automobil**, **Skoljka** i **Menjac** upotrebom kompozicije.

**Automobil** je kompozicija klasa **Skoljka** i **Menjac**. Pored ovih polja koja su objekti članovi, automobil sadrži i polje trenutna brzina (int) i polje koje označava stanje automobila (nabrojivog tipa).

(videti tekst zadatka u folderu 2-1)

# Rešenje - Menjač i Školjka

- Uraditi kao u zadatku 1-1

# Rešenje - Automobil - polja

```
#include "menjac.hpp"
#include "skoljka.hpp"

enum StanjeAutomobila {UGASEN, U_VOZNJI, POKVAREN};

class Automobil {

private:
    Menjac menjac;
    Skoljka skoljka;
    StanjeAutomobila stanje;
    int trenutnaBrzina;

};|
```

# Rešenje - Automobil - konstruktori

```
public:
    Automobil() : menjac(), skoljka(), stanje(UGASEN), trenutnaBrzina(0) {}

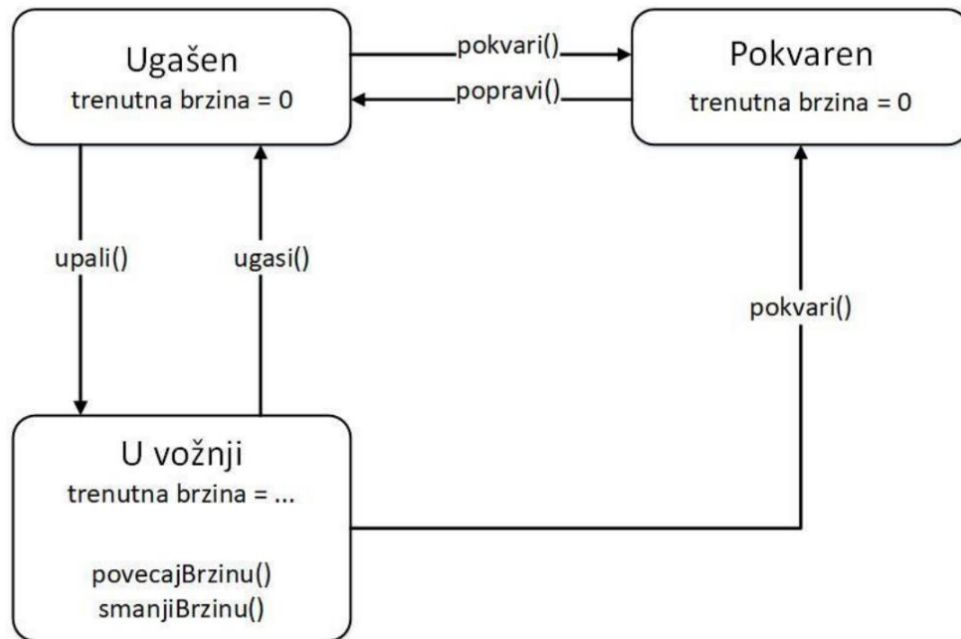
    Automobil(TipMenjaca tip, int brojBrzina, BojaSkoljke boja, StanjeAutomobila s, int b) :
        menjac(tip, brojBrzina), skoljka(boja), stanje(s), trenutnaBrzina(b) {}

    Automobil(const Automobil& a) : menjac(a.menjac), skoljka(a.skoljka), stanje(a.stanje), trenutnaBrzina(a.trenutnaBrzina) {}
```

Kreiramo automobil pomoću konstruktora inicijalizatora koji kreira menjač i školjku i inicijalizuje ostala polja u klasi **Automobil**.

# Rešenje - Automobil

Dijagram sa slike opisuje ponašanje automobila:



# Rešenje - Automobil - metoda upali

Metoda *upali* menja stanje automobila u *U\_VOZNJI* ako je stanje bilo *UGASEN*.  
Vraća *true* ili *false* u zavisnosti od toga da li je operacija uspešno izvršena ili ne.

```
bool upali() {  
    if (stanje != UGASEN)  
        return false;  
  
    stanje = U_VOZNJI;  
    return true;  
}
```

# Rešenje - Automobil - metoda *povećaj brzinu*

Metoda *povećaj brzinu* povećava trenutnu brzinu automobila ukoliko su zadovoljeni uslovi: automobil je *U\_VOZNI* i trenutna brzina nije maksimalna brzina menjača. Drugim rečima ukoliko je stanje automobila različito od stanja *U\_VOZNI* ili je trenutna brzina jednaka maksimalnoj brzini menjača, metoda neće biti uspešno izvršena (*false*). U suprotnom povećavamo brzinu i vraćamo *true*.

```
bool povecajBrzinu() {  
    if ((stanje != U_VOZNI) || (trenutnaBrzina == menjac.getBrojBrzina()))  
        return false;  
  
    trenutnaBrzina++;  
    return true;  
}
```



# Literatura

1. Kupusinac A. : Zbirka rešenih zadataka iz programskog jezika C++. Novi Sad: FTN, 2011.