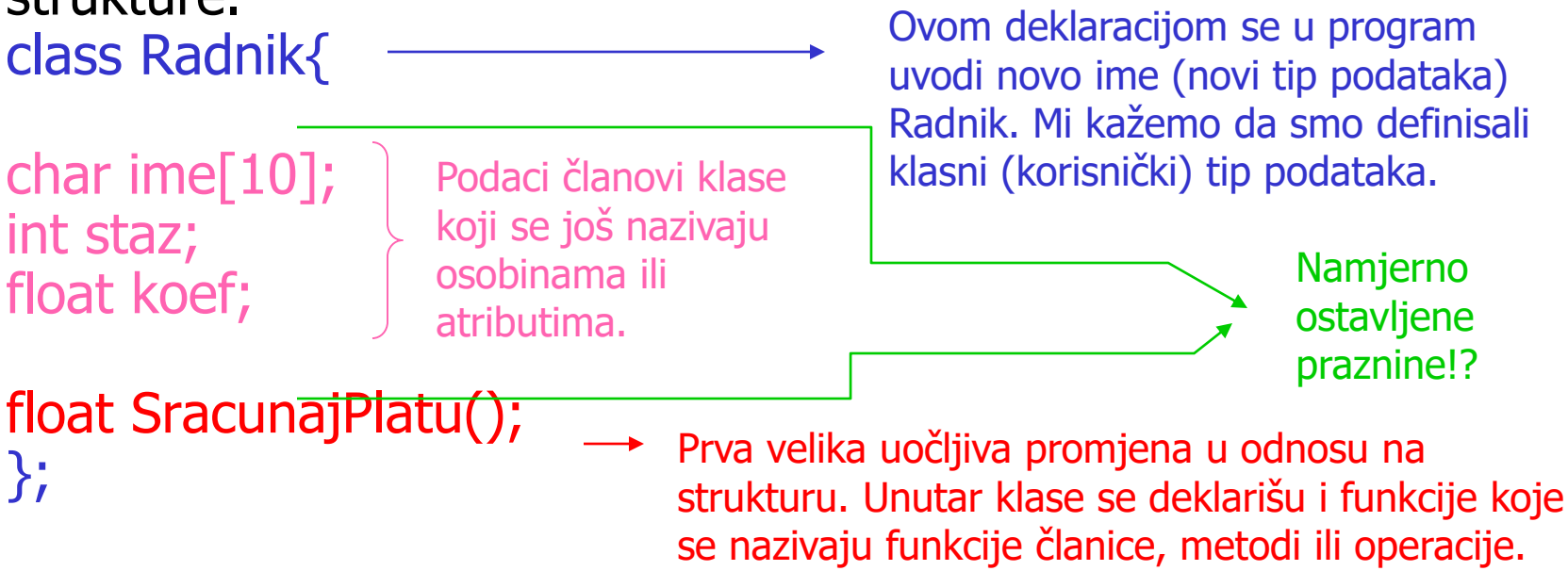
Programabilni uređaji i objektno orjentisano programiranje

Klasa i klasin interfejs.
Konstruktor, inspektori i mutatori.

Deklaracija klase

- Klasa se obično deklariše van svih funkcija i važi globalno (ako bi se definisala unutar neke funkcije samo bi ta funkcija znala da ta klasa postoji).
- Deklaracija klase zapravo podsjeća na deklaraciju strukture.

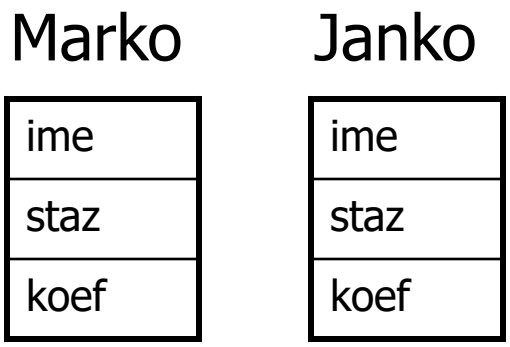


Podaci klasnog tipa

- Nakon deklaracije klase imamo pravo da uvedemo promjenljive klasnog tipa:
  Radnik Marko, Janko;
- Uočite da, za razliku od strukture, ne navodimo ključnu riječ class kada uvodimo promjenljive klasnog tipa **(vidjećemo kasnije da su promjenljive klasnog tipa pravi tip podataka za razliku od onih strukturnog tipa koje imaju neke nedostatke).**
- Podaci klasnog tipa se nazivaju **objekti** **ili instance** i odatle naziv objektno-orijentisano programiranje.
- Podaci zauzimaju memoriju koja je jednaka memoriji koju zauzimaju podaci članovi, dok sama klasa ne zauzima memoriju – klasa je mustra (obrazac) kako memoriju zauzimaju podaci klasnog tipa.

Objekti

- Memoriju objekata ne čine funkcije članice, jer se jedna funkcija realizuje za sve objekte te klase.
- Memorija zauzeta za naša dva objekta je:



Dakle, svaki radnik ima svoje ime, staz i koef.

Kako pristupiti podacima članovima?

- Podacima članovima se može pristupiti kao i kod struktura: Marko.staz=6; odnosno Janko.koef=4.25;
- Ali ne!!! Kompajler prijavljuje grešku!!!!

public - private

- Kompajler je prijavio grešku zbog toga što podrazumijeva da sve što drugačije nije deklarisano unutar klasinog interfejsa je **privatno** – private, odnosno da se vidi samo iz klase, a ne i van klase.
- Po pravilu **(od kojeg postoje neki izuzeci)** svi podaci članovi su privatni i na taj način je unutrašnja organizacija klase **enkapsulirana** - sakrivena unutar klase.
- Privatne sekcije se najavljuju sa **private**: i sve što slijedi nakon ove ključne riječi je privatno (**vidljivo samo iz klase**).
- U jednoj klasi može biti više private djelova.
- Suprotno od private je **public:**, što predstavlja najavu javnog dijela.

private - public

- Po pravilu, sve funkcije su javne (vidljive spolja). Od ovog pravila također postoje odstupanja.

```
class Radnik{
private:
    char ime[10];
    int staz;
    float koef;
public:
    float SracunajPlatu();
};
```

Iako se private podrazumijeva obično se postavlja radi jasnoće koda. Kako se podacima članovima koji su privatni ne može pristupiti spolja (kompajler prijavljuje grešku), to se za čitanje i upisivanje podataka u objekte klasnog tipa koriste funkcije članice klase.

Funkcije članice klasa koje su namijenjene za čitanje i upis podataka nazivaju se **pristupni metodi**.

Pristupni metodi

- Postoje dvije grupe pristupnih metoda:
  - Metodi koji postavljaju vrijednosti podacima članovima (nazivaju se **mutatori** ili u engl. terminologiji setersi).
  - Metodi koji čitaju vrijednosti podataka članova (nazivaju se **inspektori** ili u engl. terminologiji getersi).
- Primjer metoda i klase:

```
class Student{
private:
    int gs,gr;
  public:
    void set_gs(int a);
    void set_gr(int a);
    int get_gs() const;
    int get_gr() const;
};
```

Očigledno mutatori, uzimaju cijeli broj kao argument i treba da postave godinu rođenja, odnosno godinu studija studentu.

Očigledno inspektori, ne uzimaju argument, a vraćaju rezultat koji je jednak godini studija, odnosno godini rođenja studenta.

Modifikator const nije obavezan i ovdje označava tzv. konstante metode, odnosno metode koji ne mjenjaju unutrašnja stanja objekta (podatke članove).

Konstantni metodi i realizacija metoda

- Ako bi konstantni metod incidentno pokušao da promijeni vrijednost podatka člana objekta kompajler bi nam prijavio grešku.
- Pošto se mi pripremamo da pišemo izuzetno komplikovane programe dobro je natjerati kompajler da nam otkriva grešku kao što je eventualno slučajno mijenjanje podataka članova.
- Postavlja se pitanje kako se realizuju metodi?
- Jedan metod se realizuje za čitavu klasu, a realizacija te funkcije se može obaviti u tijelu klase ili van tijela klase.
- U C++ se preporučuje druga varijanta, osim kod izuzetno jednostavnih metoda.
  - Funkcije koje su realizovane u tijelu klase su podrazumijevano inline!!!
  - Funkcije koje se realizuju van klase mogu biti inline ako se to eksplicitno naglasi.

Realizacija metoda

```
class Student{
private:
    int gs,gr;
public:
    void set_gs(int a) {gs=a;};
    void set_gr(int a);
    int get_gs() const {return gs;};
    int get_gr() const;
};
void Student::set_gr(int a){gr=a;}
int Student::get_gr() {return gr;}
```

Kada se funkcija realizuje van klase mora se najaviti kojoj klasi funkcija pripada praćeno operatorom :: (operatorom dosega) prije naziva funkcije, a nakon navođenja tipa rezultata.

U glavnom programu bismo imali naredbe tipa:

```
Student Janko;//kreiramo objekat klase Student
Janko.set_gs(1); Marko.set_gs(5);//sada zna za koga
   //mijenja gs
```

- Primijetite da podacima članovima objekta za koji se poziva funkcija članica njegove klase, funkcija članica pristupa direktno, bez navođenja objekta kojem pripada i oni se ne prosleđuju kao argumenti. Izvršene izmjene nad podacima ovog objekta važe i nakon izvršavanja funkcije.
- O ovome će biti više riječi u nastavku.

Pozivanje funkcija

- Pogledajmo sljedeći primjer (izvod iz glavnog programa):

```
Student Marko,Janko;
Janko.set_gs(2);
Janko.set_gr(1997);
cout<<Janko.get_gs()<<Janko.get_gr();
Marko = Janko;
Marko.gr = Janko.gr;
```

Dakle, metodi se moraju pozvati navodeći ime objekta na koji se odnose. Prvi pozvani metod postavlja gs za objekat Janko na 2.

Metodi vide podatke članove za objekat za koji se pozivaju kao neku vrstu podrazumijevanih argumenata koji se ne moraju navoditi.

- Čak i u slučaju da se funkcija poziva bez argumenata, zagrade se moraju pisati.
- Peta linija ovog primjera (Marko=Janko) je dozvoljena u C++-u. Objekti klasnog tipa se mogu pridruživati jedan drugom (ako su istog tipa uvijek, a ponekad čak i kada su različitog pod nekim uslovima).

Nije dozvoljeno jer podrazumijeva pristup privatnim podacima članovima.

Primjer metoda za klasu complex

- Veoma ilustrativan primjer je sabiranje dva kompleksna broja. Neka je data klasa complex:

```
class complex{
      private:
            float re,im;
      public:
complex sabcomp(complex);
            ...
};
// u glavnom programu bismo imali poziv
// complex a,b,c; .. a=b.sabcom(c)

complex complex::sabcomp(complex y){
      complex temp;
      temp.re=re+y.re;
      temp.im=im+y.im;
      return temp;}
```

Izostavili smo sve ostale metode radi jednostavnosti. Predmetni metod treba da sabere dva kompleksna broja (rezultat koji vraća je tipa complex, baš kao i jedan argument, što je dozvoljeno). Na prvi pogled nedostaje jedan argument, ali nije tako jer je drugi argument (tzv. implicitni) kompleksni broj (objekat) za koji se funkcija poziva.

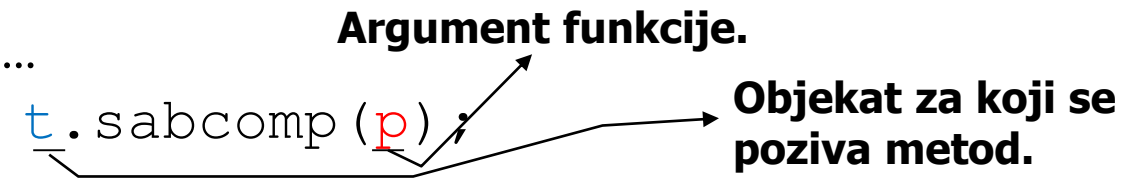
Realni dio objekta za koji se funkcija poziva (b u gornjem pozivu).

Imaginarni dio objekta koji je argument funkcije (c u gornjem pozivu).

Sabcomp - detalji

- Pozivanje ovog metoda se obavlja:

```
complex t, p, r;
...
r = t.sabcomp(p);
```



- Očigledno ovo nije najljepši način da se obavi sabiranje kompleksnih brojeva, ali nam završava posao.
- Još jedno pitanje koje vrijedi tumačiti je kako to da funkcija koja se poziva za jedan objekat pristupa podacima članovima drugog objekta istog kompleksnog tipa?
- Ovo je moguće zato što su metodima dostupni atributi svih objekata klasnog tipa kojem pripadaju **(prethodni primjer je dobra upotreba ove osobine koju ne treba zloupotrebljavati).**

Klasin interfejs – Manje važni detalji

- Kao i kod strukure, kod klase ne može podatak član biti istog klasnog tipa **class C{C c;};** jer bi to dovelo do beskonačne alokacije memorije.
- Međutim, dozvoljeno je da podatak član bude pokazivač, uključujući pokazivač na klasu istog tipa: **class C{C *c;};**. Sva ostala pravila važe kao i kod struktura.
- Prilikom definisanja klase mogu se deklarisati i objekti klasnog tipa: class X {…} a, b, x[50];.
- Moguće je kreirati i neimenovanu klasu, ali se tada sa deklaracijom klase moraju deklarisati objekti klasnog tipa, jer ih ne možemo naknadno deklarisati, jer ne postoji uvedeno ime klase: class {…} a, b, x[50];. **Ovo je, naravno, više atrakcija nego što je nešto što ćete koristiti.**

Vidljivost klase

- Klasa se po pravilu definiše van svih funkcija kako bi bila dostupna globalno, odnosno, kako bi se objekti klasnog tipa mogli deklarisati i koristiti iz svih funkcija.
- Ovo, međutim, nije obavezno. Klasa se može definisati i unutar funkcije, ali je tada vidljiva samo iz te funkcije, odnosno samo ta funkcija može da deklariše i koristi objekte klasnog tipa.
- Ovo se, ipak, relativno rijetko koristi!

Konstruktor i destruktor

- Pored opisanih, klasa obično posjeduje i specijalne metode koji kontrolišu kreiranje **(konstrukciju),** odnosno brisanje **(destrukciju)** objekata klasnog tipa.
- Ovi metodi se nazivaju konstruktor i destruktor.
- Ovi metodi nijesu obavezni i kreira ih kompajler umjesto nas ako mi to ne uradimo.
- Kompajler kreira tzv. podrazumijevani **(default)** konstruktor i destruktor **(bez argumenata).**
- Podrazumijevani konstruktor zauzima memoriju za podatke članove, ali ne postavlja početne vrijednosti podacima članovima, već podaci članovi bivaju inicijalizovani onim što zateknu na posmatranim memorijskim lokacijama.

Konstruktor i destruktor

- Grubo pravilo: ako klase imaju podatke članove koji su pokazivači moramo realizovati konstruktor i destruktor. Često se i kod klasa koje nemaju pokazivačke promjenljive kao članove realizuju konstruktor i destruktor.
- Drugo pravilo (sada obavezno) je da **ako se napravi bilo kakav konstruktor, moramo napraviti i svoju verziju podrazumijevanog konstruktora (onog konstruktora bez argumenata), inače ga nećemo moći koristiti.**
- Ne bismo mogli imati deklaraciju tipa `Complex t;!!!`
- Konstruktor je funkcija koja ima ime kao i sama klasa, ali ne vraća nikakav rezultat. Konstruktor može imati argumente (može biti preklopljen).
- Destruktor je funkcija koja ima ime kao i sama klasa sa karakterom **~** ispred, nema rezultata niti argumente, niti može biti preklopljena.

Rezultat ove dvije funkcije nije čak ni void!!!

Konstruktor i destruktor - primjer

```
class Student{
private:
    int gs;
    int gr; //kao godina studija i godina rođenja
public:
    Student();
    Student(int);
    Student(int,int);
//tri verzije konstruktora
// - preklopljeni konstruktori
    ~Student(); //Destruktor
//Još hrpa metoda
};
```

Konstruktor i destruktor - primjer

- Kako su konstruktor i destruktor često veoma jednostavne funkcije to se one kao inline realizuju unutar tijela klase. Ovdje ćemo ih realizovati vani:
-

```
Student::Student(){} //zauzima memoriju za
//podatke. Prazno tijelo funkcije, ali se moraju
//pisati {}ništa ne radi
Student::Student(int a){
    gs=a;}
Student::Student(int a,int b){
    gs=a;
    gr=b;}
 Student::~Student(){}
```

Nema return-a u konstruktoru i destruktoru, ni tipa rezultata, čak se ni void ne stavlja !!!

Konstruktor i destruktor - komentari

- Uočimo da u našoj implementaciji konstruktor i destruktor bez argumenata imaju prazno tijelo, ali ovo ne mora biti pravilo.
- Implicitno konstruktor zauzima memoriju, ali u slučaju našeg bez argumenata, to je sve što radi. On ne inicijalizuje vrijednosti promjenljivih članica.
- Destruktor i bez naše kontrole oslobađa memoriju koja je bila statički dodijeljenja promjenljivim. Ne zadaje mu se to u tijelu.
- Sljedeće deklaracije:
  Student Marko, Janko(2), Pero(2,3);
  zauzimaju memoriju redom: za Marka, za koga se poziva konstruktor bez argumenata, Janka, za koga se poziva konstruktor sa jednim argumentom (nakon čega Janko.gs postaje 2) i Pera, kod koga se poziva konstruktor sa 2 argumenta.

Konstruktor – razrješenje poziva

- Konstruktor se može pozvati i eksplicitno:
  `Student Sale; Sale.Student();`
  ali se ovo rijetko radi jer je objekat Sale već konstruisan u samoj deklaraciji.
- Za razrješenje poziva kod konstruktora važe sva pravila kao i kod ostalih funkcija – poziva se onaj čiji argumenti najbolje odgovaraju argumentima za koje je funkcija pozvana. **Programer mora da vodi računa o tome da ne dođe do dvosmislenosti u pozivu prije svega dizajnom preklopljenih konstruktora.**
- Konstruktor (pa čak i destruktor) mogu biti privatne funkcije! Prilikom određivanja koji će konstruktor biti pozvan bira se onaj koji se najbolje slaže po argumentima, pa ako se on ne može pozvati zbog toga što je privatan dolazi do greške u programu. Stoga oprez!!!!!!!

Konstruktor – Inicijalizacija podataka članova

- Jedna od funkcija konstruktora **(demonstrirana u našem primjeru)** je inicijalizacija podataka članova.
- Za ovako jednostavnu aktivnost mi plaćamo skupu cijenu jer obavljamo operacije u tijelu funkcije.
- Stoga je u OOP dozvoljeno da se ova operacije obavljaju u sekciji za inicijalizaciju prije samog tijela funkcije:

```
Student::Student(int a):gs(a){}
Student::Student(int a, int b):gs(a),gr(b){}
```

Početak sekcije za inicijalizaciju

gs inicijalizovano na a, a gr inicijalizovano na b

Tijelo funkcije je prazno. Ovo omogućava efikasnije (brže) izvršavanje konstruktora.

Kada se konstruktor poziva?

- Situacije kada se konstruktor poziva su uglavnom slične sa situacijama kada se kreira objekat standardnih tipova podataka:
  - Prilikom deklaracije statičkih objekata klasnog tipa (ključna riječ static ili kao globalna promjenljiva, podsjetite se ovih pojmova). Ovi objekti podrazumijevaju i inicijalizaciju;
  - Prilikom deklaracije promjenljivih i kreiranja svih dinamičkih objekata;
  - Prilikom kreiranja dinamičkog objekta "naredbom" new;
  - Prilikom vraćanja rezultata funkcije koji je klasnog tipa (kreira se privremeni objekat tog klasnog tipa i poziva konstruktor);
  - Konstruktor se može i eksplicitno pozvati.
- Postoje još neke situacije kada se konstruktor poziva, a uglavnom se tiču nasljeđivanja i izvedenih klasa što ćemo učiti za mjesec i po dana.

Kada se poziva destruktor?

- Opet veoma slično kao i kod destrukcije (brisanja iz memorije) objekata standardnih tipova:
  - Na kraju programa kada se dealociraju statički objekti;
  - Na kraju programskih blokova u kojima su deklarisani dinamički objekti;
  - Prilikom poziva "funkcije" delete;
  - U situacijama kada se dealociraju privremeni objekti;
  - Destruktor se može eksplicitno pozvati.

Što se može uraditi sa objektom klasnog tipa?

- Opet jednostavan odgovor. Sve što i sa objektima standardnog tipa (a možda, još ponešto):
  - Kreirati objekti i nizovi objekata klasnog tipa;
  - Definisati reference i pokazivači na objekte klasnog tipa;
  - Dodijeliti vrijednost jednog objekta drugom (operator =; za objekte je dozvoljeno napisati A=B; kada se, ako programer posebnom tehnikom to drugačije ne specificira, standardno vrši kopiranje podataka iz objekta na desnoj strani u objekat na lijevoj strani);
  - Uzeti adresa objekta & i izvršiti posredan pristup preko pokazivača;
  - Pristupati podacima članovima klase (javnim) direktno (operator .) ili indirektno – preko pokazivača (operator ->);
  - Objekti, pokazivači na objekte i reference na objekte se mogu koristiti kao argumenti funkcije;
  - Objekti, pokazivači na objekte i reference na objekte mogu biti i rezultati funkcije.
- Napomena: Nekim operatorima se može promijeniti predefinisano značenje, ali će o tome više riječi biti kasnije.

Članovi klase koji su pokazivači – konstruktor i destruktor

- Pretpostavimo da klasa ima pokazivač kao podatak član:

```
class X{private:
        int *p;
        public: /**/};
```

- Postavlja se pitanje što će se dogoditi prilikom destrukcije objekta ovog klasnog tipa. Odgovor: Biće dealociran podatak član, u ovom slučaju pokazivač p.
- Problem je taj što smo mi preko tog pokazivača mogli da zauzmemo memoriju za cijeli broj (ili niz cijelih brojeva!!!) koji i dalje zauzima memoriju. Stoga je neophodno da programer oslobodi memoriju koja je bila zauzeta preko datog pokazivača.

Destruktor - pokazivač

- Jedna moguća realizacija konstruktora i destruktora za klasu X:

U p-u je vrijednost koja je upisana u memoriji koja je zauzeta za pokazivač, mi još nismo zauzeli memoriju za ono na šta će p da ukazuje, pa p postavljamo na 0..

Veoma neobičan ali efektivan konstruktor. Naime, vrši se inicijalizacija pokazivača p na vrijednosti pokazivača koji je alociran naredbom new int, a (i) znači da je objekat na koji pokazivač pokazuje zapravo cijeli broj **i**.

```
class X{
private:
      int *p;
public: X(){p=0;}
      X(int i):p(new int(i)){}
// U tijelu funkcije bi bilo p = new int; *p=i;
      ~X() {delete p; p=0;}
};
```

Destruktor naredbom (operatorom) delete oslobađa memoriju koja je bila zauzeta za objekat na koji pokazuje p. Memoriju za sam pokazivač dealocira destruktor bez naše kontrole. Podrazumijevani destruktor oslobađa memoriju za pokazivač, ali ne i za objekat na koji pokazivač pokazuje.

Neoslobađanje memorije može da nam donese mnoge nevolje uključujući usporavanje i pad računarskog sistema!!!

p=0; signalizira da p u datom trenutku ne pokazuje ni na šta. Naš primjer bi preživio i bez ovoga, ali je u praksi ipak obavezno naglašavati da pokazivač u datom trenutku ne pokazuje ni na jedan memorijski objekat!

Dodajmo inspektor i main

```
#include <iostream>
class X{
private:
      int *p;
public:
     X(){p=0;}
     X(int i):p(new int(i)){}
       ~X() {delete p; p=0;}
       int get_int(){return *p;}
       };
int main(){
     X pom; // poziva default konstruktor
     //cout<<pom.get_int()<<endl;
     X a(3);
     cout<<a.get_int()<<endl;//ispisuje 3
     return 0;}
```

Ne bi bila greška i dalo bi nelogičnu vrijednost, ako u podrazumijevanom konstruktoru ne postavimo p=0. U tom slučaju se ne može čitati vrijednost cijelog broja na koji ukazuje p, jer p za pom ne ukazuje ni na šta.

Pokazivač na podatak klasnog tipa

- Uvijek se može deklarisati promjenljiva koja predstavlja pokazivač na podatak klasnog tipa:

```
Student *st; //klasa mora biti unaprijed
//deklarisana

st=new Student; //uz eventualnu provjeru da li je
// alokacija uspjela
```

- Deklaracija pokazivača i alokacija objekta se mogu odraditi zajedno, "o jednom trošku":

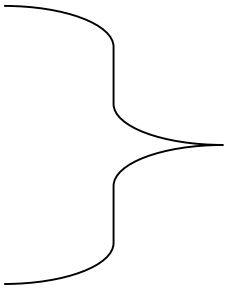
```
Student *st=new Student;
```

- Preko pokazivača se može pristupati podacima članovima objekta (ako su javni), kao i funkcijama članicama na jedan od dva načina:

```
st->a=5;

st->fun();
(*st).a=5;

(*st).fun();
```



Opet napominjemo, sve čemu želimo pristupiti van klase mora biti javno, inače će doći do greške u programu.

Pokazivač this

- Funkcije članice klase se realizuju kao i standardne funkcije – jedna za čitavu klasu.
- Stoga funkcije članice ne čine dio objekta klasnog tipa, već memoriju objekta klasnog tipa čine samo podaci članovi!!!!
- Jedna specifičnost funkcija članica je da one vide podatke članove objekta za koji se pozivaju; nešto kao globalne promjenljive koje ne deklarišemo unutar funkcije niti ih navodimo kao argumente.
- Kako funkcije članice klase znaju kome objektu klasnog tipa pristupaju i čije podatke članove da koriste?

Pokazivač this

- Odgovor je na osnovu pokazivača this.
- Naime, prilikom poziva funkcije: Marko.fun(a); funkciji fun se, pored argumenta a, automatski prosljeđuje i pokazivač na objekat za koji je ova funkcija pozvana (Marko). Ovaj pokazivač naziva se this **(na engleskom znači ovaj).** Ovaj pokazivač se sasvim slobodno može koristiti unutar funkcije:

```
void Student::UpisGodine(int a=1)
//funkcija sa default argumentom
{this->gs += a;}  //Ovo nijesmo morali uraditi na
//ovaj način već je bilo dovoljno gs+=a;
```

- Pretpostavimo da želimo uvećati kompleksni broj (za koji se funkcija poziva) sa nekim drugim kompleksnim brojem i ujedno vratiti dobijeni zbir. Ovdje je već this neprevaziđen.

Pokazivač this - Primjer

```
complex complex::sabcomp(complex &y){
                re+=y.re;
                im+=y.im;
                return *this;}
```

Ovdje smo uštedili vrijeme i memoriju jer nijesmo deklarisali promjenljivu temp koja je klasnog tipa (i samo zauzimanje memorije za promjenljive klasnog tipa traje neko vrijeme).

this u punoj veličini. Ako je this pokazivač na objekat za koji se funkcija poziva, onda je *this sam objekat, tj. vraćamo vrijednost kompleksnog broja za koji je funkcija pozvana!

Štedimo i prilikom prijema argumenta, jer referenca znači da nećemo kreirati novi (privremeni) objekat prilikom prenosa argumenata, već da ćemo koristiti objekat koji već postoji u memoriji. Ovo nije baš najkorektnije u nekim situacijama.

U ovom slučaju smo mogli bez ikakvih problema da rezultat funkcije vratimo kao referencu. Da li je ovo tačno?

```cpp
#include <iostream>

using namespace std;
class complex{
    private:
        float re, im;
    public:
        complex(){}
        complex(float a,float b):re(a),im(b){}
        float Real(){return re;}
        float Imag(){return im;}
        complex sabcomp(complex &);
        };

complex complex::sabcomp(complex &y){
    re+=y.re;
    im+=y.im;
    return *this;}


int main()
{
    complex a(2,3),b(1.2,-0.5),c;
    c=a.sabcomp(b);//mijenja se i a i c
    cout<<"nova vrijednost za a je a = ";
    cout<<'('<<a.Real()<<","<<a.Imag()<<')'<<endl;
    cout<<"Vrijednost objekta a je dodijeljena objektu c. c= ";
    cout<<'('<<a.Real()<<","<<a.Imag()<<')'<<endl;
    return 0;
}
```

```
nova vrijednost za a je a = (3.2,2.5)
Vrijednost objekta a je dodijeljena objektu c. c= (3.2,2.5)
```

Zaključak

- Prošli smo kroz značajan dio klasinog interfejsa.
- Preostalo nam je da objasnimo:
  - Konstruktor kopije i još neke druge “tajne” karakteristike konstruktora;
  - Statičke podatke i funkcije članice;
  - Pokazivače na članove klase.
- Kada to završimo daćemo jedan relativno kompleksan primjer klase i pripremiti se da pređemo na napredne opcije kod klasa.