

dr Dragan Milićev

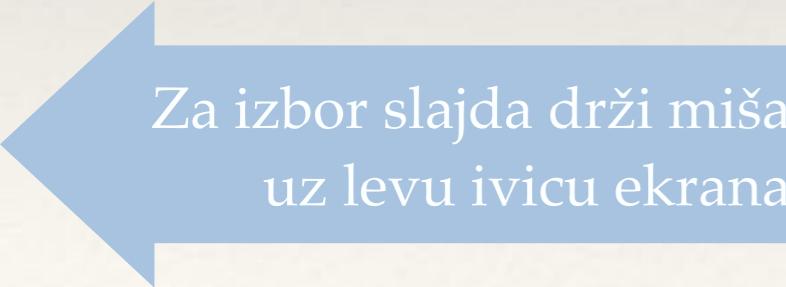
redovni profesor

Elektrotehnički fakultet u Beogradu

dmilicev@etf.rs, www.rcub.bg.ac.rs/~dmilicev

Objektno orijentisano programiranje

Slajdovi za predavanja



Za izbor slajda drži miša
uz levu ivicu ekrana

Sadržaj

- ❖ Uvod u OOP i C++
- ❖ Apstrakcija i objektna dekompozicija
- ❖ Detalji jezika C++

Deo I: Uvod u OOP i C++

- ❖ O predmetu
- ❖ Uvod u OOP
- ❖ Pregled osnovnih koncepata OOP
na jeziku C++

Glava 1: O predmetu

- ❖ Sadržaj, ciljevi i preduslovi
- ❖ Predispitne i ispitne obaveze
- ❖ Literatura
- ❖ Kontakti



Sadržaj, ciljevi i preduslovi

- ❖ Sadržaj:
 - Osnovni koncepti i principi objektne paradigme
 - Programski jezik C++
- ❖ Ciljevi:
 - Upoznati se sa osnovnim konceptima i principima objektne paradigme i njihovim pogodnostima
 - Osposobiti se za projektovanje i implementaciju složenih programa korišćenjem objektnih koncepata
 - Osposobiti se za programiranje na jeziku C++

Sadržaj, ciljevi i preduslovi

- ❖ Preduslovi:
 - Dobro poznavanje osnovnih principa i koncepata proceduralnog programiranja
 - Poznavanje rada na računaru i korišćenje osnovnih alatki za programiranje (editor, prevodilac, debager)
 - Dobro savladano gradivo predmeta Programiranje 1 i 2, Praktikum iz programiranja
 - Veština u programiranju na jeziku C
 - **Samostalan, praktičan i kontinuiran rad!**

Predispitne i ispitne obaveze

- ❖ Kolokvijumi (lab vežbe):
 - ❖ rade se pojedinačno i na računaru u laboratoriji
 - ❖ dva neobavezna kolokvijuma, svaki nosi po 20% ocene
- ❖ K1: analiza, razumevanje i izmena / proširenje jednog gotovog OO sistema
 - ❖ sistem se proučava na vežbama i samostalno
 - ❖ zadata modifikacija se radi i brani samostalno, na računaru u laboratoriji
 - ❖ peta nastavna nedelja (prva kolokvijumska nedelja na SI)
 - ❖ nadoknada alternativno sa K2
- ❖ K2: prvi deo (jezgro) projekta
 - ❖ priprema se kao domaći zadatak, a brani samostalno, na računaru u laboratoriji kao K2
 - ❖ 10. nastavna nedelja (druga kolokvijumska nedelja na SI)
 - ❖ može se nadoknaditi / zameniti projektom (poeni sa projekta se skaliraju)

Predispitne i ispitne obaveze

- ❖ Projekat (neobavezan):
 - priprema se kao domaći zadatak, brani samostalno, na računaru u laboratoriji
 - brani se u januarskom, februarskom ili avgustovskom ispitnom roku
 - nosi 30% ocene
- ❖ Ispit:
 - radi se pojedinačno i na računaru u laboratoriji
 - kratki praktični zadaci na licu mesta
 - nosi 30% ocene

Literatura

- ❖ Materijali za predavanja i vežbe: <http://oop.etf.rs>
- ❖ Neobavezna literatura:
 - D. Milićev, “Objektno orijentisano programiranje na jeziku C++, Skripta sa praktikumom”, Mikro knjiga, Beograd, 2001.
 - D. Milićev, “Objektno orijentisano modelovanje na jeziku UML, Skripta sa praktikumom”, Mikro knjiga, Beograd, 2001.
 - D. Milićev, “Objektno orijentisano programiranje na jeziku C++”, Mikro knjiga, Beograd, 1995.
 - L. Kraus, “Programski jezik C++ sa rešenim zadacima”, 10. izdanje, Akademска misao, Beograd, 2016.
 - L. Kraus, “Rešeni zadaci iz programskog jezika C++”, 5. izdanje, Akademска misao, Beograd, 2016.
 - Referentni priručnik: en.cppreference.com

Kontakti

- ❖ Sajt predmeta: <http://oop.etf.rs>
- ❖ Predavanja:
 - ❖ prof. dr Dragan Milićev, dmilicev@etf.rs, www.rcub.bg.ac.rs/~dmilicev
 - ❖ doc. dr Vladimir Jocović, jocke@etf.rs
 - ❖ doc. dr Uroš Radenković, uki@etf.rs
- ❖ Vežbe:
 - ❖ Adrian Milaković, aki@etf.bg.ac.rs
 - ❖ Kristijan Žiža, ziza@etf.rs
 - ❖ Miloš Obradović, miobra@etf.rs
 - ❖ Miloš Milošević, mm@etf.bg.ac.rs

Glava 2: Uvod u OOP

- ❖ Zašto OOP?
- ❖ Šta donosi OOP?
- ❖ Šta se menja prelaskom na OOP?



Zašto OOP?

- ❖ Problemi u razvoju softvera:
 - Zahtevi korisnika su složeni i stalno se povećavaju
 - Softverski sistemi su inherentno složeni
- ❖ Uvek je potrebno povećati produktivnost proizvodnje softvera. Kako? Povećanjem broja programera u timu? Problemi – interakcija između delova softvera!
- ❖ Način povećanja produktivnosti – ponovna upotreba softvera (*software reuse*). Kako obezbediti?
- ❖ Problemi održavanja softvera: ispravljanje grešaka, promena zahteva i dodavanje zahteva. Kako postići?
- ❖ Kako odgovoriti na izazove? Unapređenjem koncepata!

Šta donosi OOP?

OOP je deo objektne paradigme koja obuhvata osnovne objektne koncepte, od kojih su neki:

- ❖ apstraktni tipovi podataka (*abstract data types*): tip koji je definisao programer, za koji se mogu kreirati primerci (instance) i koji je predstavljen strukturom i ponašanjem, s tim da je za korisnike tipa bitno samo ponašanje, ne i interna struktura (implementacija)
- ❖ enkapsulacija (*encapsulation*): deo softvera ima jasno definisan interfejs i implementaciju; interfejs je svima dostupan, implementacija je nedostupna
- ❖ nasleđivanje (*inheritance*): jedan tip može da nasledi drugi, osnovni tip, sa značenjem da su njegove instance jedna vrsta instanci tog osnovnog tipa
- ❖ polimorfizam (*polymorphism*)

Šta se menja prelaskom na OOP?

- ❖ Prelazak sa proceduralnog programiranja na OOP je promena paradigme!
- ❖ OO paradigma sadrži koncepte višeg nivoa apstrakcije, izražajnije, bliže domenima problema
- ❖ Menja se način razmišljanja
- ❖ Težište se prebacuje sa implementacije na interfejse i veze između delova softvera – cilj je oslabiti veze između delova i učiniti ih lakšim za kontrolu i modifikaciju
- ❖ Umesto isključivo *algoritamske dekompozicije* koristi se *objektna dekompozicija*

Glava 3: Pregled osnovnih koncepata OOP na jeziku C++

- ❖ Klase i objekti
- ❖ Enkapsulacija
- ❖ Konstruktori i destruktori
- ❖ Nasleđivanje
- ❖ Polimorfizam
- ❖ Uvod u jezik C++



Klase i objekti

- ❖ Apstrakcija *časovnika* (*clock*):
 - meri vreme u toku dana (00:00:00 - 23:59:59)
 - otkucava sekunde (*tick*)
 - može da prikaže vreme (*getTime*)
 - može da mu se podesi vreme (*setTime*)
- ❖ *Klasa* (*class*) je koncept OOP podržan OO jezikom kojim se realizuje apstrakcija
- ❖ Klasa grupiše *strukturu* (podatke) i *ponašanje* (operacije nad strukturom) u jedinstvenu logičku celinu

Klase i objekti

Komentar: od // do kraja reda

Početak definicije klase *Clock*

```
// Clock: Measures time during the day  
class Clock {  
...  
int h, m, s; // Hour, minute, second  
void tick (); // Tick a second  
string getTime (); // Returns current time  
void setTime (int hour, int min, int sec);  
};
```

Podaci članovi klase *Clock*

Funkcije članice klase *Clock*

Kraj definicije klase *Clock*

Klase i objekti

- ❖ *Objekat (object)* je primerak (instanca) klase

```
Clock* pClk1 = new Clock;  
pClk1->setTime(10,12,0);
```

Pravi se novi objekat klase *Clock* na koga ukazuje pokazivač *pClk1*

```
Clock* pClk2 = new Clock;  
pClk2->setTime(23,25,36);
```

Poziv operacije *setTime* objekta na koga ukazuje pokazivač *pClk1*

```
pClk1->tick();  
pClk2->tick();
```

```
cout << pClk1->getTime() << endl;  
delete pClk1;
```

Ispis na standardni izlaz

```
cout << pClk2->getTime() << endl,  
delete pClk2;
```

Uništava se objekat klase *Clock* na koga ukazuje pokazivač *pClk1*

Klase i objekti

- ❖ Klasa je
 - deo programa
 - definicija tipa (korisničkog)
 - obrazac za kreiranje objekata
 - specifikacija svojstava i usluga svih svojih pripadnika (objekata)
- ❖ Objekti
 - žive (nastaju i nestaju) u vreme izvršavanja programa
 - postoje u memoriji računara
 - imaju sva svojstva svoje klase
 - jesu nezavisni primerci svoje klase, svaki objekat ima svoje vrednosti podataka članova

Klase i objekti

Definicije funkcija članica klase *Clock*

```
void Clock::tick () {
    s++;
    if (s==60) s=0, m++;
    if (m==60) m=0, h++;
    if (h==24) h=0;
}

string Clock::getTime () {
    return to_string(h) + ":" + to_string(m) + ":"
        + to_string(s);
}

void Clock::setTime (int hh, int mm, int ss) {
    h = (hh>=0 && hh<=23) ? hh : 0;
    m = (mm>=0 && mm<=59) ? mm : 0;
    s = (ss>=0 && ss<=59) ? ss : 0;
}
```

Pristup podatku članu *s*, *m*, *h* objekta čija je funkcija članica *tick* pozvana

Klase i objekti

Terminologija:

- ❖ *Članovi klase (class members)*: elementi deklarisani unutar definicije klase
- ❖ *Podatak član (data member)*: *svojstvo (property)*, *atribut (attribute)*, *polje (field)*
- ❖ *Funkcija članica (member function)*: *operacija (operation)*, *metoda (method)*

Stil pisanja identifikatora:

- ❖ C stil: `get_time`
- ❖ *Camel-case*: `getTime`
- ❖ Imena klasa velikim početnim slovom, ostalo malim
- ❖ Drugi proizvođači i jezici koriste drugačije stilove (npr. Microsoft .Net i C#)

Enkapsulacija

- ❖ Da li bi bilo dobro da neko uradi i ovako nešto?

```
Clock* pClk1 = new Clock;  
pClk1->h = 45;  
pClk1->m = -2;  
pClk1->s = 185;  
  
cout << pClk1->getTime() << endl;
```

- ❖ Očigledno ne. Potrebno je *sakriti* podatke članove, “učauriti” ih u “oklop” klase, sprečiti neposredan pristup spolja - *enkapsulacija (encapsulation)*

Enkapsulacija

- ❖ Specifikator *public*: govori prevodiocu da su samo članovi koji se nalaze iza njega dostupni spolja. Ovi članovi nazivaju se *javnim* i čine *interfejs* klase
- ❖ Članovi iza specifikatora *private*: su nedostupni korisnicima klase (ali ne i članovima klase) i nazivaju se *privatnim* i čine *implementaciju* klase

```
class Clock {  
public:  
  
    void tick (); // Tick a second  
    string getTime (); // Returns current time  
    void setTime (int hour, int min, int sec);  
  
private:  
  
    int h, m, s; // Hour, minute, second  
};
```

Enkapsulacija

- ❖ Sada ovo više nije moguće van funkcija članica klase (prevodilac prijavljuje grešku):

```
pClk1->h = 45;  
pClk1->m = -2;  
pClk1->s = 185;
```

Greška u prevodenju

- ❖ Umesto toga, sada je stanje objekta klase pod kontrolom, garantuje se njegova konzistentnost, jer može se menjati samo na način predviđen interfejsom:

```
pClk1->setTime(23,25,36);  
pClk2->setTime(45,-2,185); // (0, 0, 0)
```

Konstruktori i destruktori

- ❖ Kao i na jeziku C, inicijalne vrednosti promenljivih ugrađenih tipova nisu definisane. Zbog toga ovo daje neodređen rezultat:

```
Clock* pClk1 = new Clock;  
cout << pClk1->getTime() << endl;
```

Vrednosti podataka članova *s*, *m*, *h* objekta su potpuno neodređene

- ❖ Zato se od programera zahteva da odmah nakon kreiranja novog objekta, *obavezno* pozove operaciju *setTime*, kako bi postavio željene vrednosti:

```
Clock* pClk1 = new Clock;  
pClk1->setTime(23, 25, 36);
```

- ❖ Podložno greškama: bilo šta što se programer *obavezuje* da uradi, a ne proverava se automatski (nego zavisi od discipline), će *sigurno ponekad biti zaboravljen!*

Konstruktori i destruktori

- ❖ Konstruktor (*constructor*) je posebna funkcija članica klase sa istim imenom kao i klasa, koja obezbeđuje inicijalizaciju objekata te klase:

```
class Clock {  
public:  
    Clock (int hh, int mm, int ss);  
  
    void tick (); // Tick a second  
    string getTime (); // Returns current time  
    void setTime (int hh, int mm, int ss);  
  
private:  
    int h, m, s; // Hour, minute, second  
};  
  
Clock::Clock (int hh, int mm, int ss) {  
    setTime(hh,mm,ss);  
}
```

Deklaracija konstruktora klase

Definicija konstruktora klase

Konstruktori i destruktori

- ❖ Prilikom nastanka *svakog objekta* te klase (apsolutno bez izuzetka) poziva se konstruktor klase (jedan od više mogućih)
- ❖ Ovaj poziv obezbeđuje prevodilac implicitno (automatski):

```
Clock* pClk1 = new Clock(23,25,36); Implicitan poziv konstruktora
```

- ❖ Ako konstruktor klase zahteva argumente, onda se ovo *ne može zaboraviti* i izostaviti - prevodilac će prijaviti grešku:

```
Clock* pClk1 = new Clock; Greška u prevođenju
```

- ❖ Moguće je definisati i funkciju koja se *uvek* implicitno poziva kada objekat prestaje da živi (uništava se).

Ova funkcija naziva se destruktur (*destructor*):

```
delete pClk1; Implicitan poziv destruktora
```

Nasleđivanje

- ❖ Pravimo apstrakciju “hotelskog predvorja” u kom je više časovnika koji pokazuju trenutno vreme u različitim gradovima sveta:

```
class Lobby {  
public:  
  
    Lobby (unsigned numOfClocks, string cities[], int lags[]);  
  
    void tick (); // Tick a second  
    void print (); // Displays the clocks  
    void setTime (int h, int m, int s); // Sets the referential time  
  
private:  
  
    int num;  
    Clock* clocks[MaxNumOfClocks];  
    string cities[MaxNumOfClocks];  
    int lags[MaxNumOfClocks];  
};
```

Nasleđivanje

```
Lobby::Lobby (unsigned n, string ct[], int lg[]) {
    num = (n>MaxNumOfClocks) ? MaxNumOfClocks : n;
    for (int i=0; i<num; i++) {
        cities[i] = ct[i];
        lags[i] = lg[i] % 24;
        int h = 0 + lags[i];
        if (h<0) h = 24+h;
        clocks[i] = new Clock(h,0,0);
    }
}

void Lobby::tick () {
    for (int i=0; i<num; i++) clocks[i]->tick();
}

void Lobby::print () {
    for (int i=0; i<num; i++)
        cout<< cities[i] << ":" << clocks[i]->getTime() << endl;
}

void Lobby::setTime (int hh, int mm, int ss) {
    for (int i=0; i<num; i++) {
        int h = hh + lags[i];
        if (h>=24) h %= 24;
        if (h<0) h = 24+h;
        clocks[i]->setTime(h,mm,ss);
    }
}
```

Nasleđivanje

- ❖ Želimo novu apstrakciju, *časovnika sa datumom*
- ❖ Časovnik sa datumom je jedna vrsta časovnika, specijalizacija ili proširenje apstrakcije časovnika
- ❖ Klasa *ClockWithDate* nasleđuje (*inherits*) klasu *Clock*:

```
class ClockWithDate : public Clock {  
public:  
    ClockWithDate (int y, int m, int d, int h, int min, int s);  
    string getDate (); // Returns current date  
    void setDate (int yy, int mm, int dd); // Year, month, day  
  
private:  
    int y, m, d; // Year, month, day  
};
```

Nasleđivanje

- ❖ Ova relacija naziva se *nasleđivanje* (*inheritance*): klasa *ClockWithDate* nasleđuje sve osobine (atribute, operacije) osnovne klase *Clock*, ali je i *specijalizuje* (*specializes*) ili *proširuje* (*extends*), dodavanjem osobina (atributa, operacija), koje osnovna klasa nema
- ❖ Terminologija:
 - *Osnovna klasa* (*base class*), *generalizacija* (*generalization*), *roditelj* (*parent*)
 - *Izvedena klasa* (*derived class*), *specijalizacija* (*specialization*), *dete* (*child*)
- ❖ Sada se sa objektima izvedene klase *ClockWithDate* može raditi sve što i sa objektima osnovne klase *Clock*, jer su oni *jedna vrsta* (*kind of*) objekata osnovne klase, pošto poseduju (nasleđuju) sve osobine osnovne klase
- ❖ Sa objektima izvedene klase može se raditi i ono što je specifično za tu izvenedu klasu, a nije svojstveno osnovnoj:

```
Clock* simpleClock = new Clock(13,17,0);
ClockWithDate* smartClock = new ClockWithDate(2018,9,13,13,17,0);

simpleClock->setTime(13,20,0);
smartClock->setTime(13,20,0);

simpleClock->setDate(2018,9,14);
smartClock->setDate(2018,9,14);
```

Greška u prevodenju

Nasleđivanje

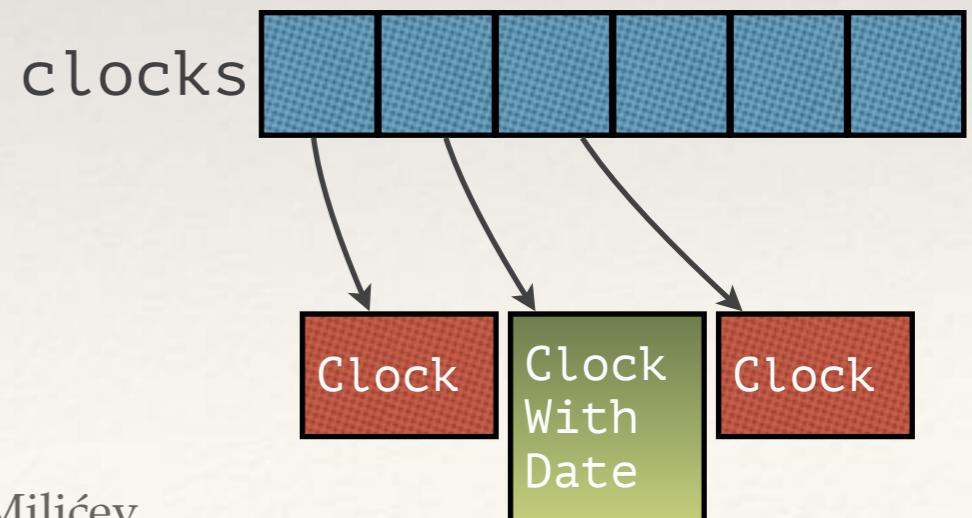
- ❖ Objekti izvedene klase su (indirektne) instance i osnovne klase!
- ❖ *Pravilo supstitucije (Substitution rule, B. Liskov)*: gde god i kad god se očekuje objekat osnovne klase, može se pojaviti i objekat izvedene klase, jer
 - je objekat izvedene klase takođe i primerak (instanca) osnovne klase, jer
 - se sa njim može raditi sve što i sa objektima osnovne klase, jer
 - objekat izvedene klase ima (nasleđuje) sve osobine osnovne klase.
- ❖ Konverzija koju dozvoljava C++ i to implicitno:

DerivedClass* → BaseClass*

ClockWithDate* → Clock*

Pravilo supstitucije i implicitna konverzija
*ClockWithDate** u *Clock**

```
Lobby::Lobby (unsigned n, string s[], int lg[], int dateYN[]) {  
    ...  
    for (int i=0; i<num; i++) {  
        ...  
        if (dateYN[i])  
            clocks[i] = new ClockWithDate(1970,1,1,h,0,0);  
        else  
            clocks[i] = new Clock(h,0,0);  
    }  
}
```



Nasleđivanje

- ❖ Objekat izvedene klase ima sva svojstva osnovne klase — u sebi ima ugrađen *podobjekat* osnovne klase
- ❖ Za inicijalizaciju podobjekta osnovne klase, odnosno nasleđenih svojstava, odgovoran je konstruktor te osnovne klase
- ❖ Kada se kreira objekat izvedene klase, poziva se konstruktor te (izvedene) klase. Taj konstruktor, pre izvršavanja svog tela, *uvek* poziva konstruktor osnovne klase:

```
ClockWithDate::ClockWithDate (int y, int m, int d,  
    int h, int min, int s)  
: Clock (h,min,s) {  
    setDate(y,m,d);
```

}

```
Clock* clk = new ClockWithDate(...);
```

Poziv konstruktora osnovne klase kome se prosleđuju neki dobijeni argumenti

Poziv konstruktora izvedene klase
(koji onda poziva konstruktor osnovne klase)

Polimorfizam

- ❖ Objekti klase *ClockWithDate* moraju drugačije da reaguju na
 - operaciju *tick*, jer moraju da paze i na ažuriranje datuma
 - operaciju *getTime*, jer želimo da vrate i datum i vreme
- ❖ Funkcija članica koja će u izvedenim klasama imati *drugačiju implementaciju* deklariše se u osnovnoj klasi kao *virtuelna funkcija (virtual)*. Izvedena klasa može da dâ svoju definiciju virtuelne funkcije, ali i ne mora:

```
class Clock {  
public:  
  
    Clock (int hh, int mm, int ss);  
  
    virtual void tick (); // Tick a second  
    virtual string getTime (); // Returns current time  
    void setTime (int hh, int mm, int ss);  
  
    int getHour() { return h; }  
    int getMinute() { return m; }  
    int getSecond() { return s; }  
    bool isMidnight () { return (h==0 && m==0 && s==0); }  
  
private:  
  
    int h, m, s; // Hour, minute, second  
};
```

Virtuelne funkcije članice

Polimorfizam

- ❖ Izvedena klasa može da dâ svoju definiciju virtuelne funkcije, tj. da je *redefiniše*, ali i ne mora (onda nasleđuje implementaciju):

```
class ClockWithDate : public Clock {  
public:  
    ClockWithDate (int y, int m, int d, int h, int min, int s);  
    virtual void tick ();  
    virtual string getTime ();  
    string getDate ();  
    void setDate (int yy, int mm, int dd);  
  
private:  
    int y, m, d;  
};  
void ClockWithDate::tick () {  
    Clock::tick();  
    if (isMidnight()) {  
        // increment the date  
    }  
}  
  
string ClockWithDate::getTime () {  
    return to_string(m) + "/" + to_string(d) + "/" +  
           to_string(y) + " " + Clock::getTime();  
}
```

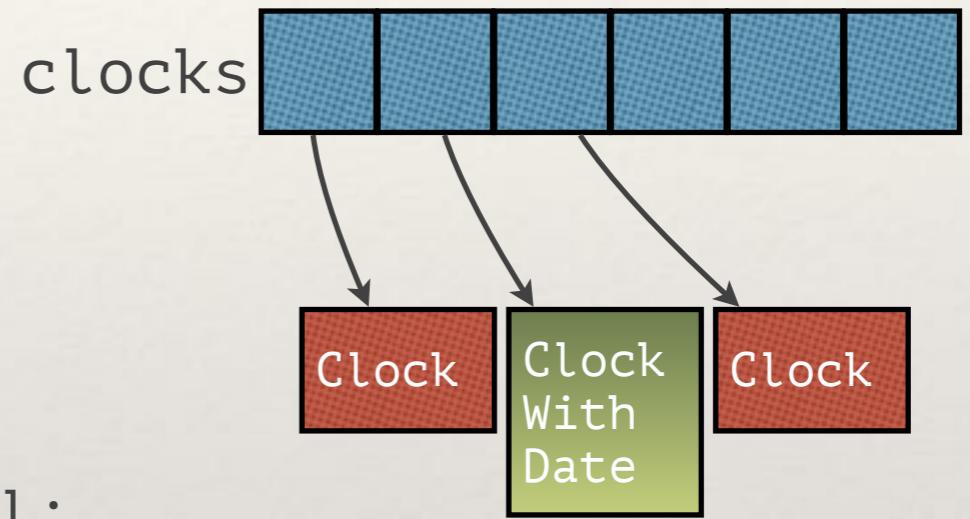
Redefinisane virtuelne funkcije

Poziv verzije implementacije iz osnovne klase

Polimorfizam

- ❖ Drugi delovi programa koji objektima pristupaju generalizovano, kao (direktnim ili indirektnim)instancama osnovne klase *Clock*, ne moraju da se menjaju:

```
void Lobby::tick () {  
    for (int i=0; i<num; i++)  
        clocks[i]->tick();  
}  
  
void Lobby::print () {  
    for (int i=0; i<num; i++)  
        cout<< cities[i] << ":"  
        << clocks[i]->getTime() << endl;  
}
```



Polimorfizam: pokazivač *clocks[i]* je tipa *Clock**

- ❖ Bez obzira na to što se objektu obraća kao instanci osnovne klase (pokazivač je tipa *Clock* *), odaziva se ona implementacija operacije koja odgovara klasi čija je objekat direktna instance; svaki objekat reaguje na način svojstven svojoj klasi
- ❖ Ovaj mehanizam naziva se *polimorfizam* (*polymorphism*, *poly* - više, μορφή - oblik, pojavljivanje u više oblika)

Uvod u jezik C++

- ❖ C++ je standardan, objektno orijentisan programski jezik opšte namene
- ❖ C++ je vertikalno kompatibilan (uz manje izuzetke) sa jezikom C: svi C programi se mogu prevoditi prevodiocem za C++
- ❖ Međutim, C++ je jezik nove generacije i različite paradigme nego što je C! Razlika između jezika C i C++ je veća (konceptualna i paradigmatska) nego što je između jezika C++ i drugih OO jezika, npr. jezika Java
- ❖ C++ podržava sve osnovne OO koncepte, ali dozvoljava i neobjektne delove programa ili cele programe
- ❖ Kao i svaki jezik, C++ je alat koji *pomaže* u pisanju objektnih programa i izražavaznju objektnih koncepata, ali *ne sprečava* pravljenje loših programa
- ❖ C++ je sredstvo za izražavanje objektne koncepcije programa

Uvod u jezik C++: Principi dizajna

- ❖ C++ je vertikalno kompatibilan sa jezikom C: svi C programi su prevodivi pomoću prevodilaca za jezik C++
- ❖ Efikasnost implementacije: prevedeni kod ima skoro istu efikasnost kao ekvivalentni rukom pisani asemblerski kod (a po pravilu i veću, zbog naprednih optimizacija)
- ❖ Obimna provera u vreme prevođenja
- ❖ Nikakva kontrola u vreme izvršavanja
- ❖ Statička, jaka, ali ne i isključiva tipizacija (dozvoljene su eksplisitne i implicitne konverzije)
- ❖ Jednak tretman i jednobrazan način upotrebe ugrađenih (primitivnih) i korisnički definisanih (klasa) tipova objekata
- ❖ Poravnanje veličine ugrađenih (primitivnih) tipova podataka prema ciljnoj arhitekturi računara; kao posledica, veličine ugrađenih tipova nisu standardne i svuda iste

Uvod u jezik C++: Istorijat razvoja

- ❖ 1972. C – D. Ritchie u Bell laboratorijama
- ❖ 1980. B. Stroustrup u AT&T Bell laboratorijama, "C sa klasama" za simulacije vođene događajima
- ❖ 1983. C++
- ❖ 1989. ANSI standard za C
- ❖ 1997. ANSI/ISO prihvaćen finalni nacrt standarda za C++
- ❖ 1998. publikovan C++ standard ISO/IEC 14882:1998 (C++98)
- ❖ 2003. revizija C++ standarda ISO/IEC 14882:2003 (C++03)
- ❖ 2011. novi C++ standard ISO/IEC 14882:2011 (C++11)
- ❖ 2014. novi C++ standard ISO/IEC 14882:2014 (C++14)
- ❖ 2017. novi C++ standard ISO/IEC 14882:2017 (C++17)
- ❖ C++20, C++23...
- ❖ Telo za standardizaciju: JTC1/SC22/WG21 - The C++ Standards Committee

Deo II:

Apstrakcija i objektna dekompozicija

- ❖ Apstrakcija
- ❖ Objektna dekompozicija

Glava 4: Apstrakcija

- ❖ Modularnost i enkapsulacija
- ❖ Sa proceduralnog na OOP jezik
- ❖ Klasa kao realizacija apstraktnog tipa podataka
- ❖ Klasa kao realizacija apstraktne strukture podataka
- ❖ Klasa kao realizacija apstrakcije
- ❖ Klasa kao realizacija softverske mašine



Modularnost i enkapsulacija

- ❖ Rešavamo sledeći problem: za dati niz znakova koji sadrži samo otvorene i zatvorene zagrade, potrebno je utvrditi da li su zagrade propisno uparene i ugnezđene i ako jesu, koja otvorena zagrada odgovara kojoj zatvorenoj zagradi
- ❖ Rešenje se zasniva na korišćenju *steka* (*stack*):
 - stek je inicijalno prazan; obrađujemo znakove u ulaznom nizu redom, jedan po jedan;
 - po nailasku na '(', na vrh steka smeštamo poziciju te otvorene zagrade (operacija *push*);
 - po nailasku na ')', sa vrha steka skidamo poziciju njoj odgovarajuće otvorene zagrade (operacija *pop*);
 - ako je stek prazan pri nailasku na zatvorenu zagrada ili ako je ostao neprazan nakon kraja ulaznog niza, zagrade nisu dobro uparene
- ❖ Potreban nam je stek:
 - linearna struktura elemenata (svaki element ima najviše po jednog prethodnika i sledbenika)
 - operacije *push* i *pop* sa LIFO (*last-in-first-out*) protokolom
- ❖ Složen program čiji je ovo samo mali deo *dekomponujemo* na logičke celine - *module*; u proceduralnom programiranju, ovakav stek bismo implementirali u jednom modulu
- ❖ *Dekompozicija* (*decomposition*) je jedno od osnovnih oruđa (pored apstrakcije) koje čovek koristi u rešavanju kompleksnosti (softvera): podela složenog problema / sistema na delove i odvojeno napadanje i rešavanje tih delova

Modularnost i enkapsulacija

- ❖ Implementacija na jeziku C: modul je jedan .c fajl

```
#define MaxStackSize 256
unsigned stack[MaxStackSize]; // Stack
unsigned sp = 0; // Stack pointer

int push (unsigned in) {
    if (sp==MaxStackSize) return -1; // Exception: stack full
    stack[sp++] = in;
    return 0;
}

int pop (unsigned* out) {
    if (sp==0) return -1; // Exception: stack empty
    *out = stack[--sp];
    return 0;
}
```

Modularnost i enkapsulacija

- ❖ Problem: interna implementacija steka (struktura, *stack* i *sp*) je dostupna ostalim delovima programa, pa stoga ti delovi programa mogu:
 - greškom da poremete tu strukturu i dovedu je u nekonzistentno stanje, npr:
sp = -5;
 - da se osalone na informaciju o načinu implementacije (postojanju niza *stack* i indeksa *sp*); ako iz bilo kog razloga imamo potrebu da promenimo tu implementaciju (npr. pređemo na neograničenu, dinamičku strukturu), promene će uticati na sve takve druge delove programa: oni se ili moraju menjati (teško i podložno greškama) ili neće raditi valjano
- ❖ Princip *sakrivanja informacija* (*information hiding*, David Parnas, 1972): svaka logička celina - modul programa, treba da ima jasno izdvojen
 - *interfejs* (*interface*): specifikaciju elemenata (struktura, tipova, operacija...) koje ostali delovi programa mogu da vide i prepostavki na koje smeju da se osalone, i
 - *implementaciju* (*implementation*): interne delove (strukturu, ponašanje) koje drugi delovi programa ne smeju da vide, niti da se oslanjaju na prepostavke o njоj
- ❖ Interfejsi treba da budu što opštiji, jednostavniji, kako bi sprege između delova softvera bile jednostavnije, labavije, lakše za kontrolu, a time ti delovi softvera nezavisniji i fleksibilniji
- ❖ *Enkapsulacija* (*encapsulation*) je programska tehnika koja podržava princip sakrivanja informacija (često se ova dva termina poistovećuju i koriste ravnopravno)

Modularnost i enkapsulacija

- ❖ Rudiment enkapsulacije u proceduralnom programiranju: razlika između
 - deklaracije (potpisa, *signature*) nekog potprograma:

```
int push (unsigned in);
```

- implementacije (potpune definicije tela) tog potprograma:

```
int push (unsigned in) {  
    if (sp==MaxStackSize) return -1; // Exception: stack full  
    stack[sp++] = in;  
    return 0;  
}
```

- ❖ Motivacija je izvorno banalna — nezavisno prevodenje modula: na mestu poziva, za generisanje koda pozivaoca, dovoljno je znati samo potpis potprograma, ne i celo telo; telo je potrebno na mestu gde se generiše kod za telo tog potprograma
- ❖ Opštije, ova ista rudimentarna tehnika predstavlja i podršku enkapsulaciji na nivou potprograma, kao osnovne gradivne jedinice proceduralnog programa
- ❖ Sličan princip potreban je i na krupnijem stepenu granularnosti - modulu

Modularnost i enkapsulacija

- ❖ Enkapsulacija na nivou modula u proceduralnom programiranju: razlika između
 - interfejsa modula; na jeziku C u fajlu zaglavlju (*header file, .h*):

```
/* File: stack.h */  
int push (unsigned in);  
int pop (unsigned* out);
```

- implementacije tog modula; na jeziku C u fajlu zaglavlju (*.c file*):

```
/* File stack.c */  
#include "stack.h"  
  
#define MaxStackSize 256  
unsigned stack[MaxStackSize]; // Stack  
int sp = 0; // Stack pointer  
  
int push (unsigned in) {  
    ...  
}  
  
int pop (unsigned* out) {  
    ...  
}
```

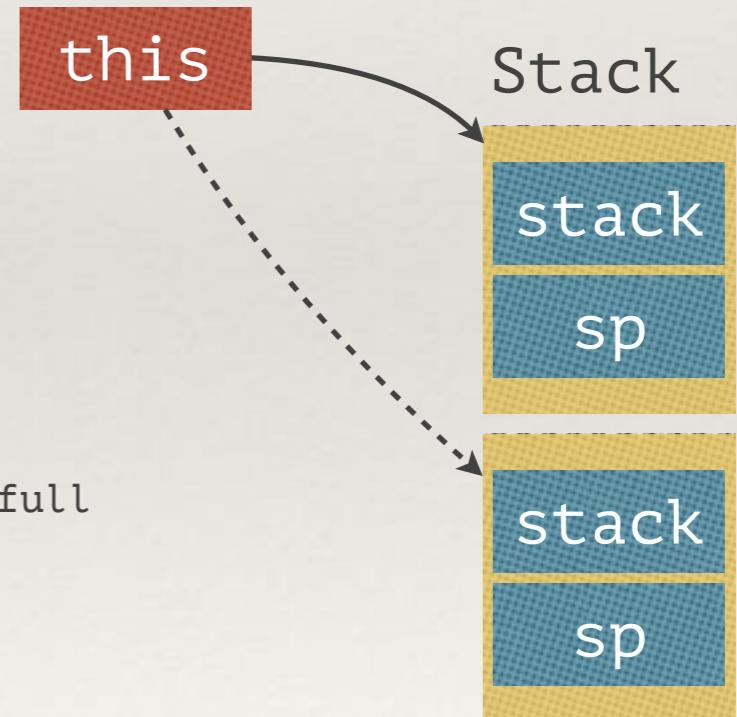
- ❖ Da globalni identifikatori koji ne pripadaju interfejsu *ne bi bili dostupni* u drugim modulima, tj. da bi bili sakriveni od drugih modula, moraju se deklarisati tako da imaju tzv. *interno vezivanje* (*internal linking*):

```
static unsigned stack[MaxStackSize]; // Stack  
static int sp = 0; // Stack pointer
```

Sa proceduralnog na OO programiranje: klase i objekti

- ❖ Nedostatak ovog rešenja: u programu imamo samo jedan ovakav stek, jednu instancu.
Šta ako nam je potrebno više instanci ovakve strukture? Organizacija koja bi ovo omogućila:

```
/* File: stack.h */  
  
#define MaxStackSize 256  
  
struct Stack {  
    unsigned stack[MaxStackSize]; // Stack  
    int sp; // Stack pointer  
};  
  
void stack_init (Stack* this);  
int stack_push (Stack* this, unsigned in);  
int stack_pop (Stack* this, unsigned* out);  
  
/* File stack.c */  
#include "stack.h"  
  
void stack_init (Stack* this) {  
    this->sp = 0;  
}  
  
int stack_push (Stack* this, unsigned in) {  
    if (this->sp==MaxStackSize) return -1; // Exception: stack full  
    this->stack[this->sp++] = in;  
    return 0;  
}  
  
int stack_pop (Stack* this, unsigned* out) {  
    if (this->sp==0) return -1; // Exception: stack empty  
    *out = this->stack[--this->sp];  
    return 0;  
}
```

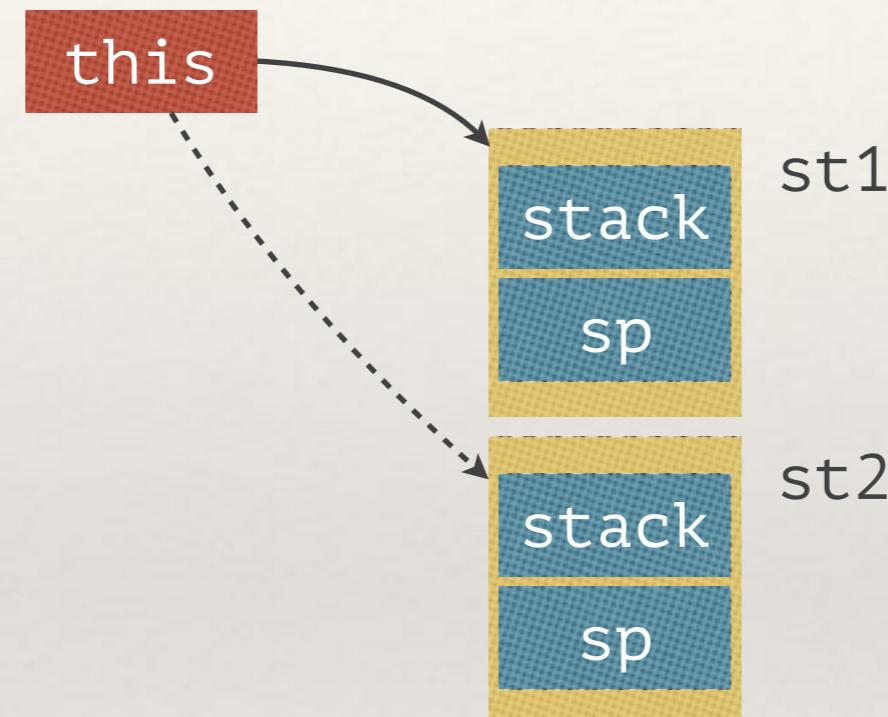


Sa proceduralnog na OO programiranje: klase i objekti

- ❖ Sada se ovo koristi ovako:

```
#include "stack.h"

Stack* pSt1 = ...;
stack_init(pSt1);
...
unsigned out;
...
stack_push(pSt1,in);
...
stack_pop(pSt1,&out);
...
Stack* pSt2 = ...;
stack_init(pSt2);
...
stack_push(pSt2,in);
...
stack_pop(pSt2,&out);
...
```



Sa proceduralnog na OO programiranje: klase i objekti

Upravo tako se na jeziku C++ implementiraju klase i objekti:

- ❖ Objekti se u vreme izvršavanja implementiraju kao instance obične C strukture koja sadrži samo vrednosti podataka članova date klase
- ❖ Za svaku (nestatičku) funkciju članicu klase, prevodilac generiše kod koji izgleda potpuno isto kao za "običnu" C funkciju, s tim što ta funkcija ima još jedan, prvi, skriveni argument *this* koji je pokazivač na ovu strukturu
- ❖ Svako neposredno obraćanje podatku članu unutar te funkcije članice:

```
sp = 0;
```

prevodilac prevodi u implicitan pristup preko pokazivača *this*:

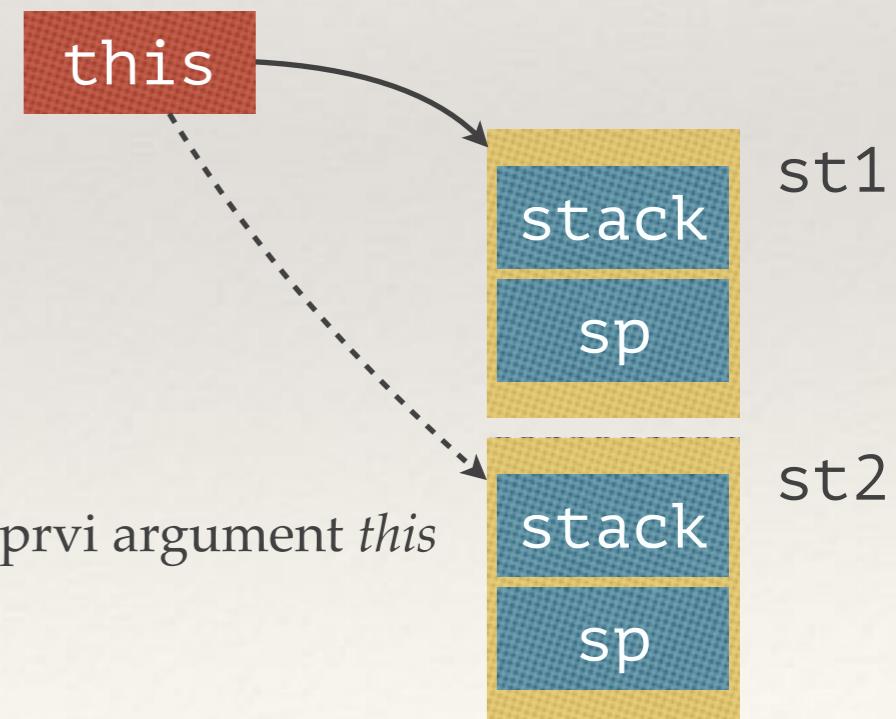
```
this->sp = 0;
```

- ❖ Svaki poziv funkcije članice za dati objekat:

```
pSt1->push(in);
```

prevodilac prevodi u poziv "obične" C funkcije, pri čemu joj kao taj prvi argument *this* prenosi pokazivač na taj objekat:

```
stack_push(pSt1, in);
```



Sa proceduralnog na OO programiranje: klase i objekti

- ❖ Na jeziku C++, svaka (nestatička) funkcija članica klase poseduje lokalni, implicitno deklarisan pokazivač *this*; ovaj pokazivač je konstantan (ne može mu se promeniti vrednost)
- ❖ Tokom izvršavanja tela ovakve funkcije članice pokazivač *this* pokazuje na objekat za koga je ta funkcija pozvana
- ❖ Svaki neposredan pristup članu klase unutar ovakve funkcije smatra se implicitnim pristupom članu onog objekta na koga pokazuje *this* (uvek se može vršiti i eksplicitan pristup, što ponekad može poboljšati čitljivost ili razrešiti više značnost):

`sp = 0;`

je isto što i:

`this->sp = 0;`

- ❖ Privid da svaki objekat “poseduje svoju” funkciju članicu je zapravo posledica toga što se funkcija članica poziva za dati objekat, a članovima u tekstu tela funkcije može pristupati neposredno, bez navođenja objekta (podrazumeva se pristup članu onog objekta “čija” se funkcija izvršava). Ovaj privid stvara se jednostavnim opisanim mehanizmom implicitnog pristupa preko pokazivača *this*

Sa proceduralnog na OO programiranje: klase i objekti

```
/* File: stack.h */
#define MaxStackSize 256

struct Stack;

void stack_init (Stack* this);
int stack_push (Stack* this, unsigned in);
int stack_pop (Stack* this, unsigned* out);

struct Stack {
    unsigned stack[MaxStackSize]; // Stack
    int sp; // Stack pointer
};

/* File stack.c */
#include "stack.h"

void stack_init (Stack* this) {
    this->sp = 0;
}

int stack_push (Stack* this, unsigned in) {
    if (this->sp==MaxStackSize) return -1;
    this->stack[this->sp++] = in;
    return 0;
}

int stack_pop (Stack* this, unsigned* out) {
    if (this->sp==0) return -1;
    *out = this->stack[--this->sp];
    return 0;
}
```

```
// File: stack.h

const int MaxStackSize = 256;

class Stack {
public:
    Stack ();
    int push (unsigned in);
    int pop (unsigned* out);

private:
    unsigned stack[MaxStackSize]; // Stack
    int sp; // Stack pointer
};

// File stack.cpp
#include "stack.h"

Stack::Stack () {
    this->sp = 0;
}

int Stack::push (unsigned in) {
    if (this->sp==MaxStackSize) return -1;
    this->stack[this->sp++] = in;
    return 0;
}

int Stack::pop (unsigned* out) {
    if (this->sp==0) return -1;
    *out = this->stack[--this->sp];
    return 0;
}
```

Sa proceduralnog na OO programiranje: klase i objekti

- ❖ Na ovaj način postiže se potpuna efikasnost — kod generisan za funkcije članice i njihov poziv, kao i pristup članovima objekata podjednako je efikasan kao i ekvivalentan C kod, jer nema nikakve dodatne režije (*overhead*)
- ❖ Sa druge strane, postiže se višestruka dobit u pisanju i strukturiranju programa, kao i u podršci prevodioca koji vrši odgovarajuće automatske provere (samo u vreme prevodenja), umesto da sve to zavisi samo od discipline i pedantnosti programera:
 - Koncept klase kao apstrakcije se jasno ističe i podstiče programera da je na ispravan način definiše, odnosno grupiše strukturu i ponašanje (operacije nad tom strukturom)
 - Struktura i operacije nad njom su “upakovani” u jedinstvenu, jasno istaknutu logičku celinu, i pripadaju *oblasti važenja klase* (*class scope*)
 - Podržana je enkapsulacija, jer prevodilac kontroliše i ne dozvoljava neovlašćen pristup do implementacije klase (privatnih članova)
 - Postoji koncept konstruktora (i destruktora), pa je inicijalizacija objekata lakša, očiglednija i manje podložna greškama

Sa proceduralnog na OO programiranje: klase i objekti

- ❖ Pokazivač *this* se može koristiti kao i svaki drugi pokazivač unutar funkcije članice, jer on prosto ukazuje na objekat (čija je funkcija pozvana), na primer, ako je drugom objektu potreba veza ka tom objektu
- ❖ Na primer, hoćemo da objekti klase *Clock* imaju vezu ka objektu klase *Lobby* koji ih sadrži:

```
class Clock {  
public:  
    Clock (Lobby* owner, int hh, int mm, int ss);  
    ...  
private:  
    Lobby* myOwner; // The Lobby that contains this Clock  
    ...  
};  
  
Clock::Clock (Lobby* owner, int hh, int mm, int ss) {  
    this->myOwner = owner;  
}  
  
Lobby::Lobby (unsigned n, string ct[], int lg[]) {  
    ...  
    for (int i=0; i<n; i++) {  
        ...  
        this->clocks[i] = new Clock(this,h,0,0);  
    }  
}
```

Sa proceduralnog na OO programiranje: polimorfizam

- ❖ Rudiment polimorfizma u proceduralnom programiranju — isti kao i za modularnost i enkapsulaciju, isključivo staticki, vezan za pisanje i prevođenje programa:

- deklaracija (potpis, *signature*) nekog potprograma:

```
int printf (char* format, ...);
```

- implementacije (potpune definicije tela) tog potprograma:

```
int printf (char* format, ...) {  
    ...  
}
```

- ❖ Iza istog interfejsa se može kriti različita implementacija i ona se može i promeniti, ali isključivo promenom koda implementacije i njegovim ponovnim prevođenjem.

Sa proceduralnog na OO programiranje: polimorfizam

- ❖ Dinamički polimorfizam, u vreme izvršavanja:

```
int fprintf (FILE* stream, char* format, ...);
```

- ❖ Apstrakcija *izlaznog znakovnog toka* (*output character stream*): ima operacije izbacivanja (“ispisa”) jednog po jednog znaka (sekvencijalno):

```
int putc (int ch, FILE* stream);
```

- ❖ Konkretna implementacija zavisi od toga šta je izlazni tok, odnosno šta se krije iza pokazivača *stream* tipa *FILE**: može biti “konzola”, fajl na lokalnom disku, fajl na udaljenom računaru itd. (ovaj polimorfizam implementiran je u sistemskom pozivu operativnog sistema, ne u samom jeziku C/C++ tj. njegovoj standardnoj biblioteci; ali je i tamo implementiran na jeziku C)
- ❖ Svaka konkretna vrsta izlaznog toka ima svoju implementaciju svake operacije (npr. *putc*)
- ❖ Iza pokazivača se može kriti različit izlazni tok, i to se može menjati *dinamički*, za vreme izvršavanja programa (u jednom pozivu operacije *fprintf* jedan, u drugom drugi izlazni tok)
- ❖ Potrebno je *dinamičko vezivanje* (*dynamic binding*): umesto u vreme prevođenja, adresa pozvanog potprograma određuje se dinamički, u vreme izvršavanja — potprogram koji se poziva se vezuje za poziv dinamički, u vreme izvršavanja, a ne statički u vreme prevođenja

Sa proceduralnog na OO programiranje: polimorfizam

- ❖ Drugi primer: pravimo program (na jeziku C) za igranje šaha na računaru. Skica delova:

```
enum FigureKind { pawn, bishop, knight, rook, queen, king };  
enum FigureColor { white, black };
```

```
struct Figure {  
    FigureKind kind;  
    FigureColor color;  
    unsigned posCol, posRow; // Current position  
    ...  
};
```

```
int canMoveTo (Figure* fig, unsigned col, unsigned row);
```

- ❖ Kako implementirati funkciju *canMoveTo* (da li data figura može da se pomeri na dato polje)?

Sa proceduralnog na OO programiranje: polimorfizam

- ❖ Jedan, očigledan i jednostavan način je sledeći:

```
int canMoveTo (Figure* fig, unsigned col, unsigned row) {
    switch (fig->kind) {

        case pawn:
            if (fig->color==white) ...
            else ...
            break;

        case bishop:
            ...
            break;
        ...
    }
}
```

- ❖ Problemi:

- nepregledno, jer je obrada svih slučajeva smeštena u jedan glomazan *switch*, pa je podložno greškama
- u nekom drugom slučaju, kada skup podvrsta objekata nije konačan i može se menjati i proširivati, nije lako dodavati nov slučaj, mora se menjati i ponovo prevoditi kod, i to na svim ovakvim mestima gde se radi grananje na osnovu vrste objekta

Sa proceduralnog na OO programiranje: polimorfizam

- ❖ Nešto pregledniji pristup — razdvojiti obradu svake situacije u zaseban potprogram:

```
int canMoveTo (Figure* fig, unsigned col, unsigned row) {
    switch (fig->kind) {

        case pawn: return canPawnMoveTo(fig,col,row);
        case bishop: return canBishopMoveTo(fig,col,row);
        ...
    }
}

int canPawnMoveTo (Figure* fig, unsigned col, unsigned row) {
    if (fig->color==white) ...
    else ...
}

int canBishopMoveTo (Figure* fig, unsigned col, unsigned row) {
    ...
}
...
...
```

- ❖ I dalje moramo menjati *switch* u slučaju da dodajemo novu vrstu objekata, i to na svim ovakvim mestima (polimorfnim operacijama)

Sa proceduralnog na OO programiranje: polimorfizam

- ❖ Ideja — osloniti se na strukture podataka i na (dinamičko) ulančavanje pokazivača i pokazivače na funkcije, a onda i na dinamičko vezivanje:

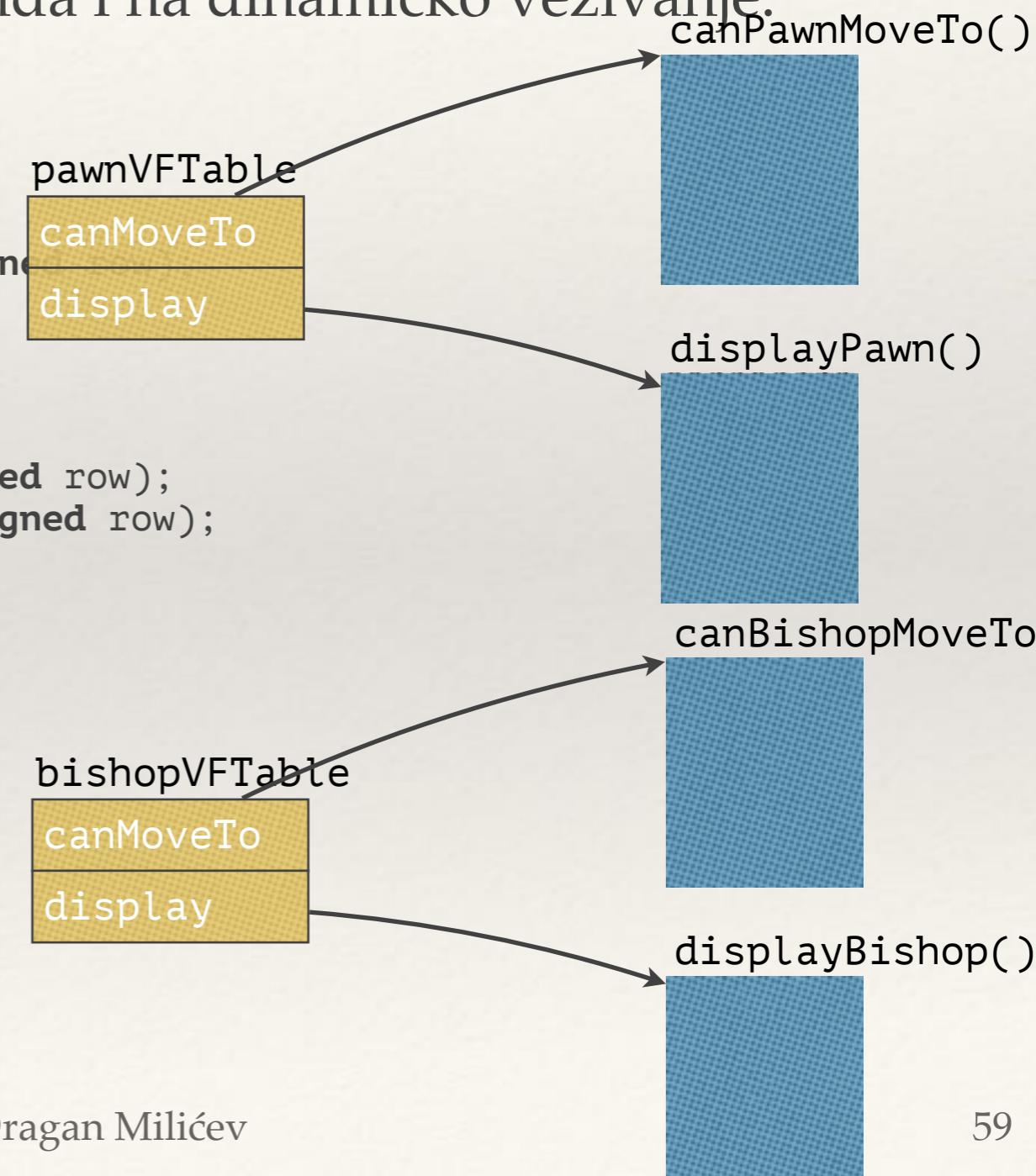
```
// Table of pointers to implementations
// of virtual functions ("virtual table")

struct Figure_VFTable {
    int (*canMoveTo) (Figure* fig, unsigned col, unsigned row);
    int (*display) (Figure* fig, ...);
...
};

int canPawnMoveTo (Figure* fig, unsigned col, unsigned row);
int canBishopMoveTo (Figure* fig, unsigned col, unsigned row);
...

Figure_VFTable pawnVFTable;
pawnVFTable.canMoveTo = &canPawnMoveTo;
pawnVFTable.display = &displayPawn;
...

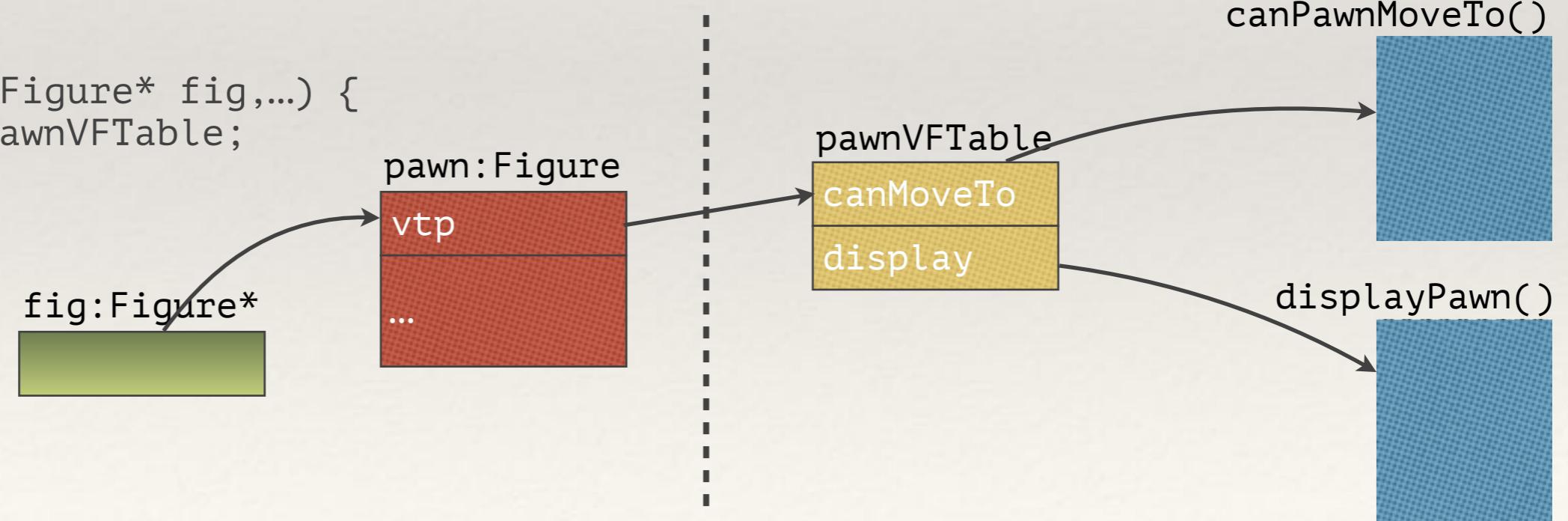
Figure_VFTable bishopVFTable;
bishopVFTable.canMoveTo = &canBishopMoveTo;
bishopVFTable.display = &displayBishop;
...
```



Sa proceduralnog na OO programiranje: polimorfizam

- ❖ Kod za funkcije generiše prevodilac, a prikazane strukture (tabele) postoje po jedna za svaku klasu (vrstu objekta) i inicijalizuju se statički, za vreme prevođenja
- ❖ Svaki objekat ima pokazivač na takvu tabelu pokazivača na implementacije virtuelnih funkcija koje odgovaraju svakoj pojedinoj klasi (vrsti objekta), tzv. *virtual table pointer*
- ❖ Ovaj pokazivač potrebno je inicijalizovati za svaki objekat, u zavisnosti od njegove vrste:

```
struct Figure {  
    Figure_VFTable* vtp; // Virtual table pointer  
    FigureColor color;  
    ...  
};  
  
void initPawn (Figure* fig,...) {  
    fig->vtp = &pawnVFTable;  
    ...  
}
```

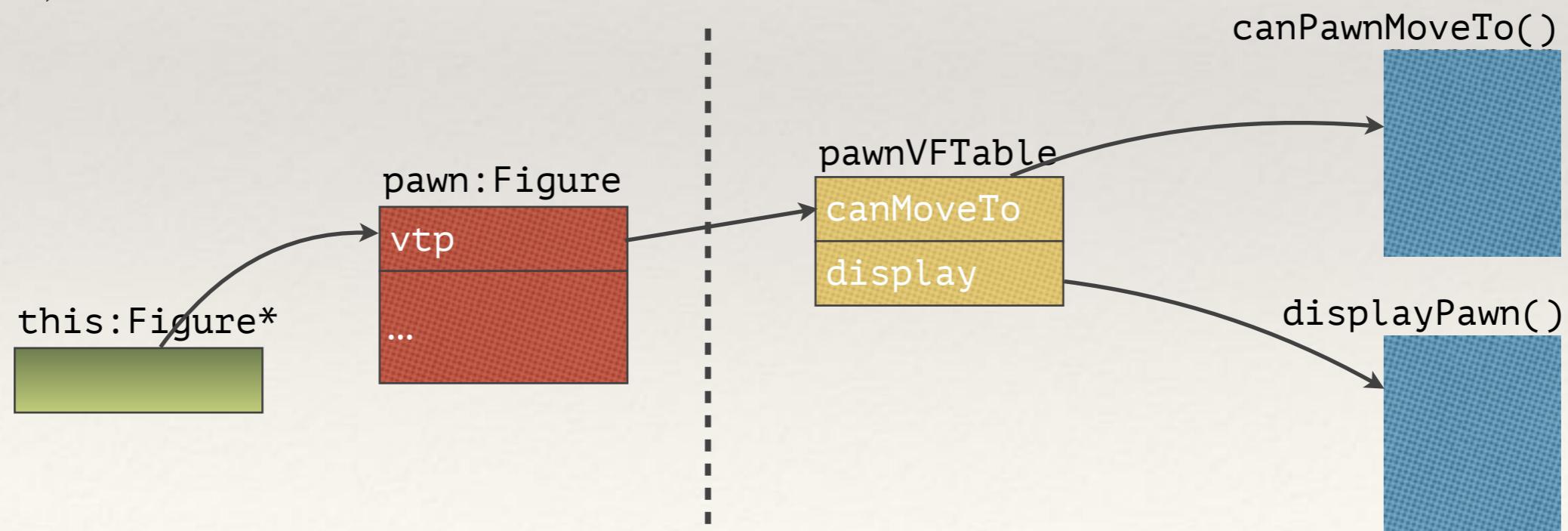


Sa proceduralnog na OO programiranje: polimorfizam

- ❖ Sada se *polimorfan* poziv realizuje jednostavno, *dinamičkim vezivanjem* (*dynamic binding*), preko lanca pokazivača, uvek istim kodom:

```
int canMoveTo (Figure* this, unsigned col, unsigned row) {  
    return this->vtp->canMoveTo(this,col,row);  
}  
  
int display (Figure* this, ...) {  
    return this->vtp->display(this,...);  
}
```

- ❖ Pogodnost: kod pozivaoca se nimalo ne menja proširenjem hijerarhije klasa (dodavanjem novih podvrsta)

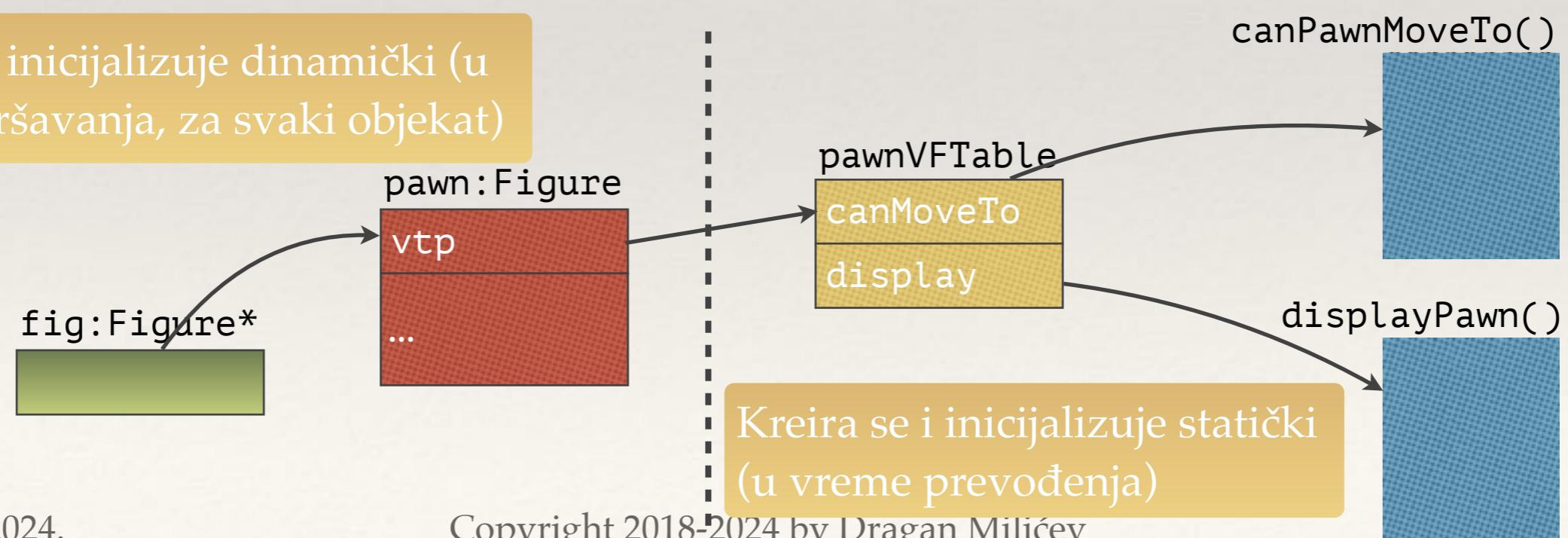


Sa proceduralnog na OO programiranje: polimorfizam

Upravo na ovakav način se na jeziku C++ implementira polimorfizam:

- ❖ za svaku klasu koja ima bar jednu virtuelnu funkciju članicu, prevodilac generiše *tabelu virtuelnih funkcija (virtual table)*, sa pokazivačima koje (statički, u vreme prevođenja) inicijalizuje tako da ukazuju na verzije implementacija funkcija koje odgovaraju toj klasi (nasleđena ili redefinisana)
- ❖ u svakom objektu ovakve klase postoji *pokazivač na tabelu virtuelnih funkcija (virtual table pointer)*; ovaj pokazivač inicijalizuje konstruktor; svaki konstruktor na tabelu koja odgovara baš toj klasi, tako da ga konstruktor osnovne klase najpre postavi na svoju tabelu (te osnovne klase), a onda konstruktor izvedene klase prepiše tako da ukazuje na tabelu te klase itd.
- ❖ za svaki poziv virtuelne funkcije objekta kome se pristupa preko pokazivača, npr. `fig->canMoveTo(...)`, prevodilac generiše kod koji taj poziv rešava dinamičkim vezivanjem, tj. indirektnim pristupom (memorijskim indirektnim adresiranjem) preko pokazivača na virtuelnu tabelu i pokazivača na funkciju u ulazu te tabele koji odgovara toj virtuelnoj funkciji u klasi

Kreira se i inicijalizuje dinamički (u vreme izvršavanja, za svaki objekat)



Sa proceduralnog na OO programiranje: polimorfizam

- ❖ Prema tome, efikasnost poziva virtuelne funkcije na jeziku C++ je ista kao i u ekvivalentnom C kodu sa dinamičkim vezivanjem, i tek nešto malo manja nego poziv statičkim vezivanjem (skok na adresu memorijski direktnim adresiranjem), ali je ceo mehanizam sakriven od programera i program je apstraktniji, kompaktniji i lakši za razumevanje i održavanje:

```
struct Figure_VTable {
    int (*canMoveTo) (Figure* fig,...);
    int (*display) (Figure* fig, ...);
    ...
};

int canPawnMoveTo (Figure* fig,...);
int canBishopMoveTo (Figure* fig,...);
...

Figure_VTable pawnVFTable;
pawnVFTable.canMoveTo = &canPawnMoveTo;
pawnVFTable.display = &displayPawn;
...

struct Figure {
    Figure_VTable* vtp;
    FigureKind kind;
    ...
};

void initPawn (Figure* fig,...) {
    fig->vtp = &pawnVFTable;
    fig->kind = pawn;
    ...
}

int canMoveTo (Figure* fig,...) {
    return fig->vtp->canMoveTo(fig,col,row);
}
```

```
class Figure {
public:
    Figure ();
    virtual int canMoveTo (...);
    virtual int display (...);
    ...
};

class Pawn : public Figure {
public:
    Pawn ();
    virtual int canMoveTo (...);
    virtual int display (...);
    ...
};

...aFig->canMoveTo(...)...
```

Klasa kao realizacija apstraktnog tipa podataka

- ❖ Želimo da realizujemo *apstraktni tip podataka (abstract data type)*, kompleksan broj: strukturu sa pridruženim operacijama koja predstavlja *korisnički definisani tip (user-defined type)* - tip koji ne postoji ugrađen u jezik (*built-in type*), već ga definiše programer (kao korisnik jezika)
- ❖ Možemo kao i ranije, na jeziku C:

```
struct _complex {  
    double re, im;  
};  
typedef struct _complex complex;  
  
void complex_init (complex* this, double real, double imag);  
complex complex_add (complex c1, complex c2);  
complex complex_sub (complex c1, complex c2);  
...  
  
void complex_init (complex* this, double real, double imag) {  
    this->re = real; this->im = imag;  
}  
  
complex complex_add (complex c1, complex c2) {  
    complex result;  
    result.re = c1.re + c2.re;  
    result.im = c1.im + c2.im;  
    return result;  
}  
...
```

- ❖ I onda to da koristimo:

```
complex c1, c2, c3, c4;  
complex_init(&c1, -2.5, 6.8);  
complex_init(&c2, 46.5, -34.45);  
c3 = complex_add(c1,c2);  
c4 = complex_sub(c1,c2);  
complex c5 = c3;  
c3 = c4;
```

Klase kao realizacija apstraktnog tipa podataka

- ❖ Primetimo sledeće: vrednosti tipa *complex* su za jezik C proste strukture i *imaju semantiku kreiranja i kopiranja po vrednosti* (*value semantics, copy semantics*):

- Kreiraju se instance (primeri, promenljive) svih vrsta životnog veka (automatski, statički, privremeni itd.):

```
complex result;  
complex c1, c2, c3, c4;
```

- Inicijalizuju se drugiminstancama istog tipa, ponovo po vrednosti, prostim kopiranjem:

```
complex c5 = c3;
```

- Prenose se kao argumenti poziva funkcija po vrednosti, prostim kopiranjem:

```
...complex_add(c1,c2)...
```

- Vraćaju se kao rezultati poziva funkcija po vrednosti, prostim kopiranjem:

```
return result;  
...  
c3 = complex_add(c1,c2);
```

- Dodeljuju se drugiminstancama istog tipa, ponovo po vrednosti, prostim kopiranjem:

```
c4 = complex_sub(c1,c2);  
c3 = c4;
```

- ❖ Sve ovo je zato što je na jeziku C ovako definisana semantika za strukture, na potpuno isti način kao i za sve ostale ugrađene tipove (celobrojne, racionalne, pokazivače itd.)
- ❖ Zašto nam ovo ovako odgovara: zato što različite instance ovog tipa, sa istim vrednostima (svih svojih atributa) predstavljaju *isti konceptualni entitet* - isti kompleksan broj; zato se mogu slobodno kopirati

Klase kao realizacija apstraktnog tipa podataka

- ❖ Kako bi to izgledalo na jeziku C++ korišćenjem samo onoga što smo do sada videli:

```
class complex {
public:
    complex (double real, double imag);
    complex (complex* other); // For copy-initialization

    void copy (complex* other); // For copy-assignment

    static complex* add (complex* c1, complex* c2);
    static complex* sub (complex* c1, complex* c2);

private:
    double re, im;
};

complex::complex (double real, double imag) {
    this->re = real; this->im = imag;
}

complex* complex::add (complex* c1, complex* c2) {
    complex* result = new complex(c1->re+c2->re,c1->im+c2->im);
    return result;
}

...
```

- ❖ *Staticke funkcije članice (static member functions)* su deklarisane uz ključnu reč *static* ispred:

- jesu članice klase, pripadaju oblasti važenja klase i mogu da pristupaju privatnim članovima te klase (bilo kog objekta)
- nemaju u sebi implicitno deklarisan pokazivač *this*
- stoga ne mogu da pristupaju članovima neposredno, bez navođenja objekta kome član pripada
- zato konceptualno “ne pripadaju” pojedinačnom objektu, već celoj klasi: pružaju uslugu koja se traži od klase, a ne pojedinačnog objekta

Klasa kao realizacija apstraktnog tipa podataka

- ❖ Nove operacije koje smo uveli:

```
complex::complex (complex* other) {  
    this->copy(other);  
}  
  
void complex::copy (complex* other) {  
    if (other!=nullptr && other!=this) {  
        this->re = other->re;  
        this->im = other->im;  
    }  
}
```

- ❖ I onda to koristimo:

```
complex* c1 = new complex(-2.5,6.8);  
complex* c2 = new complex(46.5,-34.45);  
complex* c3 = complex::add(c1,c2);  
complex* c4 = complex::sub(c1,c2);  
complex* c5 = new complex(c3);  
c3->copy(c4);  
...  
delete c1; delete c2; delete c3;  
delete c4; delete c5;
```

- ❖ Problemi:

- zamorno i nezgrapno (nečitko) manipulisanje objektima i pokazivačima
- neophodno uništavati objekte (operator *delete*)

Klase kao realizacija apstraktnog tipa podataka

- ❖ Jedna od osnovnih projektnih odluka u dizajniranju jezika C++: korisnički definisane tipove (*user-defined type*) i njihove instance koristiti *što približnije moguće*, ako ne i identično kao i ugrađene tipove (npr. primitivne tipove)
- ❖ To znači: *potrebna nam je i semantika kreiranja i kopiranja po vrednosti i za klase, odnosno njihove objekte*
- ❖ Tako se notacija i način upotrebe pojednostavljuje:

```
complex c1(-2.5,6.8), c2(246.5,-34.45), c3(0.0,0.0), c4(0.0,0.0);
c3 = complex::add(c1,c2);
c4 = complex::sub(c1,c2);
complex c5 = c3;
c3 = c4;
```

- ❖ Ali se semantika izuzetno komplikuje - jedan od najvećih izvora složenosti jezika C++ koju *nemaju* neki drugi, noviji OO jezici, upravo iz ovog razloga
- ❖ Potrebno je definisati način inicijalizacije drugim instancama istog tipa, ponovo po vrednosti, kopiranjem:

```
complex c5 = c3; // c3 is not a pointer, but an object
```
- ❖ Podrazumevano ponašanje može da bude prosto kopiranje svih podataka članova (što i jeste), ali šta ako je implementacija klase takva da zahteva drugačije, posebno ponašanje u slučaju ovakve inicijalizacije?
- ❖ Potreban je *konstruktor kopije (copy constructor)* - konstruktor koji se poziva kada se objekat date klase inicijalizuje objektom istog tipa; možda ovako:

```
complex::complex(complex other) {
    this->re = other.re;
    this->im = other.im;
}
```

Pristup do člana objekta koji je levi operand operatora `.`,
za razliku od pristupa članu objekta na koga ukazuje
pokazivač kao levi operand operatora `->`

Klase kao realizacija apstraktnog tipa podataka

- ❖ Prenos objekata kao argumenata poziva funkcija po vrednosti, kopiranjem:

```
complex complex::add (complex ca, complex cb);  
...complex::add(c3,c4)...
```

- ❖ Formalni argument, kao lokalni automatski objekat, inicijalizuje se stvarnim argumentom, na isti način - konstruktorom kopije:

```
complex::complex (complex other);
```

U trenutku poziva funkcije:

```
complex::add(c3,c4)
```

na mestu ulaska u funkciju, kreira se lokalni, automatski objekat - formalni argument *ca* (odnosno *cb*) i inicijalizuje stvarnim argumentom *c3* (odnosno *c4*); semantika te inicijalizacije ista je kao i semantika bilo koje druge inicijalizacije, kao da je izvedeno:

```
complex ca = c3;
```

- ❖ A kako se vrši ova inicijalizacija? Pozivom konstruktora kopije za objekat *ca* koji se inicijalizuje objektom *c3*:

```
complex::complex(c3)
```

- ❖ Međutim, i konstruktor kopije ima formalni argument *other*, koji se opet formira kao automatski objekat i inicijalizuje stvarnim argumentom (*c3*) u trenutku ovog poziva. Kako? Pozivom istog tog konstruktora kopije...

- ❖ Problem - beskonačna rekurzija.

Klasa kao realizacija apstraktnog tipa podataka

- ❖ Ideja rešenja: umesto prenosa po vrednosti (*pass by value*), prenositi argumente *po referenci* (*pass by reference*) - formalni argument je *referenca na objekat* (posrednik do objekta), slično kao pokazivač:

```
complex::complex (complex& other) {  
    this->re = other.re;  
    this->im = other.im;  
}
```

- ❖ Sada je rekurzija prekinuta, jer se prilikom poziva konstruktora kopije:

```
complex::complex(c3)
```

formalni argument *other*, kao referenca, *vezuje* tako da referencira (upućuje) na stvarni argument *c3*

- ❖ Svaka upotreba reference, nakon njene inicijalizacije, odnosi se na *referencirani objekat* (bez izuzetka); zato se kaže da je referenca *alias* (*alias*) za objekat za koji je vezan
- ❖ Nema načina da se izvrši operacija nad referencom, veza reference sa objektom na koji ona upućuje ne može se raskinuti ili preusmeriti kao za pokazivače, iako se u principu implementira i ponaša slično:

```
complex::complex (complex& other) {  
    this->re = other.re;  
    this->im = other.im;  
}
```

Pristup do člana objekta na koji upućuje referenca *other* - upotreba reference u izrazu uvek se odnosi na referencirani objekat - stvarni argument funkcije u ovom slučaju

Klase kao realizacija apstraktnog tipa podataka

- ❖ Prenos objekata kao argumenata poziva funkcija po vrednosti, kopiranjem, se rešava na isti način:

```
complex complex::add (complex ca, complex cb);  
...complex::add(c3,c4)...
```

- ❖ Ponovo se formalni argument *ca* (*cb*) inicijalizuje stvarnim argumentom *c3* (*c4*) pozivom konstruktora kopije, sa istom semantikom:

```
complex ca = c3;  
complex cb = c4;
```

Poziva se konstruktor kopije `complex::complex(complex& other)` za *ca*, pri čemu se referenca - formalni argument *other* vezuje za stvarni argument *c3*

- ❖ I povratna vrednost funkcije tj. rezultat poziva funkcije se vrši po vrednosti, takođe kopiranjem, na isti način
- ❖ Na mestu poziva funkcije, u trenutku povratka iz funkcije, kreira se *privremeni* (*temporary*) objekat koji se inicijalizuje izrazom iz naredbe *return*; semantika ove inicijalizacije je ista kao i svake druge inicijalizacije:

```
complex complex::add (complex ca, complex cb) {  
    ...  
    return result;  
}  
...complex::add(c1,c2)...
```

Poziv negde u izrazu; kao da je tu kreiran privremeni, bezimeni objekat *temp* tipa *complex* koji prihvata vraćenu vrednost i inicijalizovan sa *result*:
`complex temp = result;`

Rezultat poziva `complex::add` jeste jedan bezimeni, privremeni objekat koji se kreira na mestu poziva ove funkcije, u trenutku povratka iz nje, i inicijalizuje rezultatom izraza iza naredbe *return* - objektom *result*. Poziva se konstruktor kopije, sa sledećim značenjem:

```
complex temp = result;
```

Poziva se konstruktor kopije `complex::complex(complex& other)` za *temp*, pri čemu se referenca - formalni argument *other* vezuje za stvarni argument *result*

Klasa kao realizacija apstraktnog tipa podataka

- ❖ Sledeće pitanje: ako želimo da instance korisničkih tipova koristimo kao i instance ugrađenih tipova, zašto i notacija operacija, recimo za ovakve, matematičke tipove, ne bi bila ista?

- ❖ Umesto:

```
...complex::sub(complex::add(c3,c4),c5)...
```

zašto ne bismo pisali prosto ovako:

```
...c3+c4-c5...
```

- ❖ U skladu sa opredeljenjem na ovakvo korišćenje instanci korisničkih tipova, jezik C++ omogućava i ovo - *preklapanje operatora* (*operator overloading*)
- ❖ Umesto "klasičnih" identifikatora, funkcije mogu imati i posebno ime, *operator@*, gde je @ simbol nekog operatora ugrađenog u jezik:

```
class complex {  
public:  
    complex (double real, double imag);  
    complex (complex& other); // Copy constructor  
    friend complex operator+ (complex c1, complex c2);  
    friend complex operator- (complex c1, complex c2);  
  
private:  
    double re, im;  
};  
  
complex::complex (double real, double imag) {...}  
  
complex operator+ (complex c1, complex c2) {  
    return complex(c1.re+c2.re,c1.im+c2.im);  
}  
  
...
```

Prijateljske finkcije (*friend*) nisu članice date klase, ali imaju pravo pristupa do njenih privatnih

Klasa kao realizacija apstraktnog tipa podataka

- ❖ Osim uobičajenog načina poziva preko identifikatora, ovakve *operatorske funkcije (operator functions)* mogu se pozivati i implicitno, iz izraza, korišćenjem notacije operatora ugrađenih u jezik:

...c3+c4-c5...

Poziv negde u izrazu. Poziva se:

operator-(operator+(c3,c4),c5)

- ❖ Prevodilac raščlanjuje (*parsira, parse*) izraz na isti način kao što je definisano pravilima jezika i to se ne može promeniti:

- broj operanada svakog od operadora (*n-arnost*)
- prioriteti i redosled izračunavanja
- asocijativnost (način grupisanja - sleva nadesno ili obratno)

- ❖ Međutim, ako je neki od operanada instanca korisničkog tipa, poziva se odgovarajuća operatorska funkcija definisana za taj tip operanda, umesto da se generiše uobičajeni mašinski kod za operacije nad ugrađenim tipovima (ponovo polimorfizam!)
- ❖ Većina operadora ugrađenih u jezik može se definisati za korisničke tipove (ali ne svi)
- ❖ Za mnoge operatore postoji podrazumevano značenje koje se može promeniti (redefinisati)
- ❖ Ne mogu se definisati novi operatori (samo redefinisati značenje onih koji su već ugrađeni u jezik)
- ❖ Ne može se promeniti (redefinisati) značenje za operatore koji rade (samo) nad ugrađenim tipovima (bar jedan operand operatorskih funkcija mora biti korisničkog tipa)
- ❖ Mnogo opštih i posebnih pravila za neke posebne operatore - detalji kasnije

Klasa kao realizacija apstraktnog tipa podataka

- ❖ Ostaje još operacija dodele vrednosti:

```
c3 = c5;
```

- ❖ Operacija dodele vrednosti je *različita od inicijalizacije*:

```
complex c3 = c5; // Initialization  
c3 = c5; // Assignment operation
```

Inicijalizacija; poziva se:
`complex::complex(c5)`

Operacija dodele; poziva se:
`c3.complex::operator=(c5)`

- inicijalizacija se vrši kada se objekat kreira; operacija dodele se vrši kao operacija u izrazu
 - inicijalizacija se vrši nad objektom koji tek nastaje i do tada nije postojao; operacija dodele se vrši nad objektom koji već postoji
 - kod inicijalizacije se poziva konstruktor; kod dodele vrednosti se poziva operator dodele kao operatorska funkcija
- ❖ U jeziku C nije bilo posebne potrebe da se razlikuju ove dve operacije, jer postoje samo ugrađeni tipovi, za koje je semantika ove dve operacije uvek ista - prosto kopiranje vrednosti
 - ❖ Operator dodele je nestatička operatorska funkcija klase:

```
complex complex::operator= (complex other) {  
    this->re = other.re; this->im = other.im;  
    return *this;  
}
```

- ❖ Zašto vraćamo `*this`? Da bi se redefinisana operatorska funkcija ponašala što sličnije onoj ugrađenoj, a ona vraća vrednost, i to baš onu dodeljenu, kako bi se moglo pisati:

```
c1 = c2 = c3; // Computed as: c1 = (c2=c3)
```

- ❖ Svaka klasa ima podrazumevani operator dodele, koji vrši podrazumevanu dodelu član po član, ali se on može i eksplisitno redefinisati

Klasa kao realizacija strukture podataka

- ❖ Želimo da realizujemo određenu strukturu podataka, npr. stek kakav smo već skicirali, sa elementima tipa *unsigned int* i kapaciteta *MaxStackSize*:

```
// File: stack.h

const int MaxStackSize = 256;

class Stack {
public:
    Stack ();
    int push (unsigned in);
    int pop (unsigned* out);

private:
    unsigned stack[MaxStackSize]; // Stack
    int sp; // Stack pointer
};
```

```
// File stack.cpp
#include "stack.h"

Stack::Stack () {
    this->sp = 0;
}

int Stack::push (unsigned in) {
    if (this->sp==MaxStackSize) return -1;
    this->stack[this->sp++] = in;
    return 0;
}

int Stack::pop (unsigned* out) {
    if (this->sp==0) return -1;
    *out = this->stack[--this->sp];
    return 0;
}
```

Konstanta tipa *int*, ne može se menjati nakon inicijalizacije

Klasa kao realizacija strukture podataka

- ❖ Šta ako nam treba stek koji će skladištiti elemente nekog drugog tipa T i/ili drugog kapaciteta?

```
// File: stack.h

const int MaxStackSize = 512;

class Stack {
public:
    Stack ();
    int push (T in);
    int pop (T* out);

private:
    T stack[MaxStackSize]; // Stack
    int sp; // Stack pointer
};

// File stack.cpp
#include "stack.h"

Stack::Stack () {
    this->sp = 0;
}

int Stack::push (T in) {
    if (this->sp==MaxStackSize) return -1;
    this->stack[this->sp++] = in;
    return 0;
}

int Stack::pop (T* out) {
    if (this->sp==0) return -1;
    *out = this->stack[--this->sp];
    return 0;
}
```

Klase kao realizacija strukture podataka

- ❖ Zaključujemo:
 - tip T može biti bilo koji tip, sve dok su za taj tip T definisane operacije koje se u ovoj klasi očekuju:
 - inicijalizacija kopiranjem, zbog prenosa argumenata u operaciju *push* i inicijalizacije niza *stack* (podrazumevano imaju svi ugrađeni tipovi i klase)
 - dodela vrednosti, zbog smeštanja u elemente niza *stack* u operaciji *push* i smeštanja povratne vrednosti u operaciji *pop* (podrazumevano imaju svi ugrađeni tipovi i klase)
 - kapacitet steka može biti bilo koja celobrojna pozitivna vrednost
 - ako želimo nešto od ovoga da promenimo, tj. da napravimo više *klasa* sa promenjenim vrednostima nekog od ovih parametara, moramo da radimo dosadan, pravolinijski, fizički posao proste zamene svih pojava određenog parametra, uz variranje naziva za svaku od tih klasa
- ❖ Očigledna potreba za *automatizacijom*: umesto programera, ovaj rutinski posao može da radi *prevodilac*
- ❖ Koncept *šablonske klase* (*template class*) ili *generičke klase* (*generic class*): obrazac klase, parametrizovan tipovima i / ili konstantama, po kome će prevodilac *generisati* kod zamenom svih parametara šablonu konkretnim, stvarnim parametrima
- ❖ Rezultat je isti kao da je ovaj posao urađen ručno: generisane klase su *različite klase*, nemaju nikakve posebne međusobne veze
- ❖ Dakle, klasa može realizovati i *apstraktну strukturu podataka* (*abstract data structure*): strukturu koja skladišti elemente proizvoljnog tipa, pri čemu zahteva od tog tipa samo određena svojstva i usluge, ne i obavezu da bude neki konkretni tip

Klasa kao realizacija strukture podataka

```
template <typename T, int MaxStackSize>
class Stack {
public:
    Stack ();
    int push (T in);
    int pop (T* out);

private:
    T stack[MaxStackSize]; // Stack
    int sp; // Stack pointer
};

template <typename T, int MaxStackSize>
Stack<T,MaxStackSize>::Stack () {
    this->sp = 0;
}

template <typename T, int MaxStackSize>
int Stack<T,MaxStackSize>::push (T in) {
    if (this->sp==MaxStackSize) return -1;
    this->stack[this->sp++] = in;
    return 0;
}

template <typename T, int MaxStackSize>
int Stack<T,MaxStackSize>::pop (T* out) {
    if (this->sp==0) return -1;
    *out = this->stack[--this->sp];
    return 0;
}
```

- ❖ Konkretne klase se generišu na zahtev, kada se prvi put upotrebi konkretan tip sa stvarnim parametrima šablonu:

```
Stack<unsigned,256> parPositions;
Stack<Figure*,256> moves;
...
parPositions.push(pos);
```

- ❖ Klasu generiše prevodilac i ona ima sve karakteristike obične klase, s tim što joj naziv sadrži sve parametre:

```
Stack<unsigned,256>
```

Klase kao realizacija strukture podataka

Standardna biblioteka jezika C++ (tzv. *Standard Template Library*, STL) sadrži mnogo definisanih šablonskih klasa, funkcija i drugih elemenata; na primer:

- ❖ niske elemenata i znakova:

```
template<typename Z> class basic_string;  
typedef basic_string<char> string;
```

Generisana bibliotečna klasa

- ❖ uređen par:

```
template<typename A, typename B> struct pair;
```

- ❖ kompleksan broj čije su komponente datog tipa:

```
template<typename T> class complex;
```

- ❖ vektor (niz dinamički proširivog kapaciteta):

```
template<typename E> class vector;
```

- ❖ red sa pristupom elementima sa oba kraja:

```
template<typename E> class deque;
```

- ❖ lista:

```
template<typename E> class list;
```

- ❖ red:

```
template<typename E, typename Z=deque<E>> class queue;
```

- ❖ prioritetni red:

```
template<typename E, typename Z=vector<E>, typename U=less<E>> class priority_queue;
```

- ❖ stek:

```
template<typename E, typename Z=deque<E>> class stack;
```

- ❖ skupovi, multiskupovi, mape

Parametarski tip koji se koristi kao implementacija interne strukture.

Parametarski tip koji se koristi za implementaciju operacije poređenja.

Podrazumevana vrednost parametra šablonu:
ako se prilikom generisanja klase ne navede ovaj parametar, uzeće se ova njegova vrednost.

Klasa kao realizacija apstrakcije

- ❖ *Apstrakcija (abstraction)* je misaoni (mentalni) postupak kojim se:
 - problem, pojava, sistem ili entitet od interesa posmatra pojednostavljenom, zanemarujući aspekte, efekte i detalje koji nisu od značaja za dato rešenje, kako bi se ono što preostane učinilo dovoljno pojednostavljenim da bi se moglo rešiti ili realizovati
 - zanemaruju različitosti pojedinačnih entiteta zarad isticanja njihovih zajedničkih osobina, kako bi se ti entiteti posmatrali generalizovano (uopšteno), grupisani u određene kategorije po svojoj sličnosti, opet zarad pojednostavljenja (posmatrati individualne entitete na isti način, umesto svaki posebno)
— *klasifikacija (classification), generalizacija (generalization)*
 - ovakvim pojednostavljenjem osmišljava pojam (koncept) koji poseduje istaknute zajedničke osobine, a bez zanemarenih detalja
- ❖ Apstrakcijom se naziva i rezultat ovog procesa, apstrahovani pojam, koncept osmišljen tim postupkom
- ❖ Apstrakcija je, pored dekompozicije, osnovno oruđe koje čovek primenjuje u borbi protiv *kompleksnosti domena problema (problem domain)* koji se rešava (softverom u našem interesu)
- ❖ Apstrakcija je osnovni postupak u inženjerstvu, jer se inženjerske discipline bave pojednostavljenim (apstrahovanim) *modelima* realnih pojava, sistema i entiteta, kako bi se mogli projektovati, ispitati i konstruisati (npr. matematički modeli, numerički modeli, simulacioni modeli, makete itd.)

Klasa kao realizacija apstrakcije

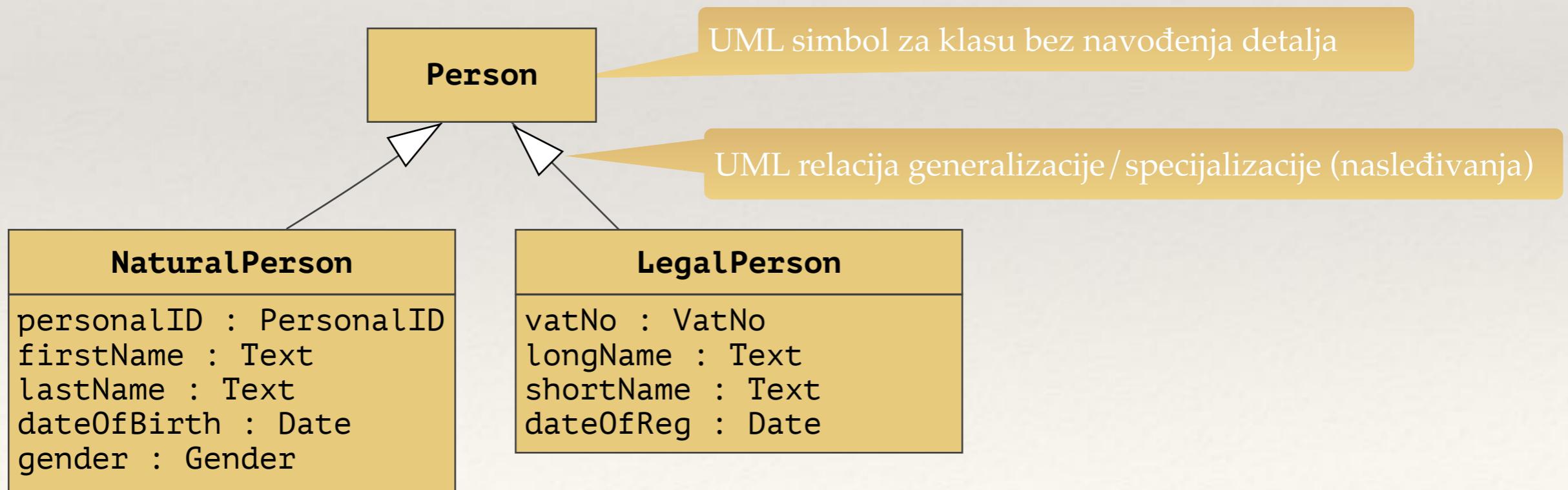
- ❖ Klasa može predstavljati *realan (fizički)* koncept (apstrakciju) iz stvarnog sveta i *domena problema*, ali i njeno *uopštenje (generalizaciju /generalization/)* ili *specijalizaciju (specialization)*
- ❖ Na primer, pojam *osobe* u realnom svetu je veoma složen koncept, praktično nedokučiv: svaka osoba u realnom svetu je individua za sebe i osobe se razlikuju po bezbroj svojstava, počev od svog DNK, preko bioloških i antropoloških veličina, karaktera, prošlosti itd.
- ❖ Međutim, za potrebe određenog softverskog sistema, uvodimo apstrakciju *Osobe (Person)* koja predstavlja (drastično) pojednostavljenje ovog realnog koncepta - apstrakciju: instance klase *Person* će imati samo svojstva od interesa za dati sistem koji se implementira (npr. matični broj, ime i prezime, datum rođenja, pol)



Klasa kao realizacija apstrakcije

Još nekoliko primera:

- ❖ *Lice (Person)*, koje može biti *Fizičko lice (Natural person)* ili *Pravno lice (Legal person)*
- ❖ *Šahovska tabla (Chess board)*, *Figura (Figure)*, *Kralj (King)*...
- ❖ *Katastarska parcela (Parcel)*, *Građevinski objekat (Building)*
- ❖ *Lični identifikacioni dokument (Personal ID Document)*, *Pasoš (Passport)*, *Lična karta (ID Card)*
- ❖ *Vozilo (Vehicle)*, *Prevoznik (Transporter)*
- ❖ i mnogo, mnogo drugih



Klasa kao realizacija apstrakcije

- ❖ Klasa može predstavljati *logički* koncept (apstrakciju) iz domena problema (uključujući i generalizaciju ili specijalizaciju)

Nekoliko primera:

- ❖ *Telefonski poziv (Phone call), Pozivalac (Caller), Pozvani (Callee)*
- ❖ *Grafička kontrola (GUI Control), Dugme (Button), Tekstualno polje (Textbox), ...*
- ❖ *Porudžbina (Order), Isporuka (Delivery), Račun (Invoice), Plaćanje (Payment)*
- ❖ *Zahtev za izdavanje ličnog dokumenta (Application for ID Document), Odobrenje (Approval), Izdavanje (Issuance)*
- ❖ i mnogo, mnogo drugih

Klasa kao realizacija apstrakcije

- ❖ Klasa može realizovati i apstrakciju *osmišljenu unutar samog rešenja* datog problema
- ❖ Takav koncept može da pomogne u rešavanju problema, pojednostavi sistem i njegovo korišćenje, uvede red ili uopštenje u domen problema
- ❖ Stoga takav koncept postaje “vidljiv” za korisnike sistema, jer softver “otkriva” taj koncept ili neposredno, nudeći ga korisnicima ekslicitno kao pojam, ili posredno, kroz ponašanje softvera
- ❖ Na neki način, tako osmišljen koncept može da postane i sastavni deo domena problema, jer je posredno “otkriven”, iako nije inicijalno postojao (ili nije bio eksplicitno prepozнат) u domenu problema
- ❖ Primeri:
 - Šablon (*Template*) slajda, tekstualnog dokumenta i slično
 - Prečica (*Shortcut*) do neke složenije operacije ili kompozicija operacija koje se često ponavljaju
- ❖ Sa druge strane, tako osmišljen koncept može biti isključivo deo rešenja, odnosno implementacije, bez ikakve vidljivosti spolja, jer se je nastao, na primer:
 - kao deo implementacije neke složene strukture, protokola ili mehanizma sistema
 - kao deo implementacije neke druge apstrakcije
 - kao posledica sprovođenja nekog projektnog obrasca (*design patter*)
 - kao posledica sprovođenja neke tehnike implementacije, optimizacije i slično

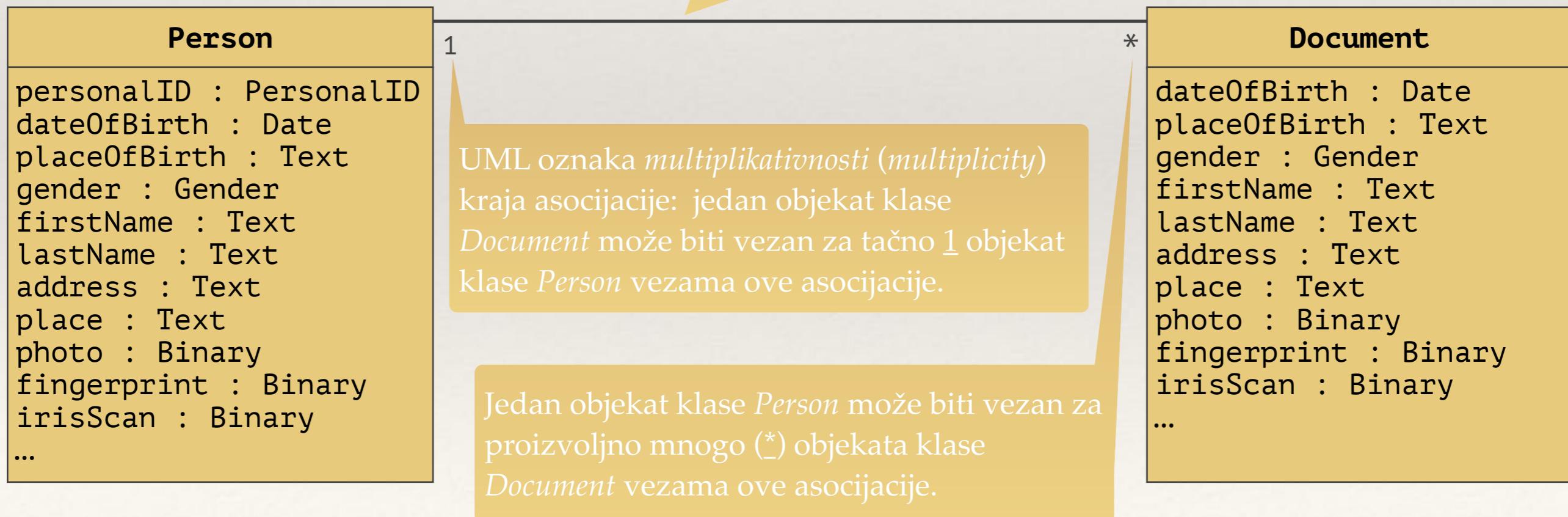
Klasa kao realizacija apstrakcije

Primer: sistem za izdavanje ličnih identifikacionih dokumenata

- ❖ Za *Osobe* se čuvaju lični (ime i prezime, datum rođenja, pol, adresa itd.) i biometrijski podaci (fotografija, otisak prsta, sken dužice oka itd.)
- ❖ Lični *Dokumenti* (pasoš, lična karta, vozačka dozvola itd.) sadrže deo tih podataka (na papiru i unutar čipa)
- ❖ Podaci osoba se povremeno menjaju, a dokumenti imaju podatke koji važe u trenutku podnošenja zahteva za izdavanje dokumenta

Klasa kao realizacija apstrakcije

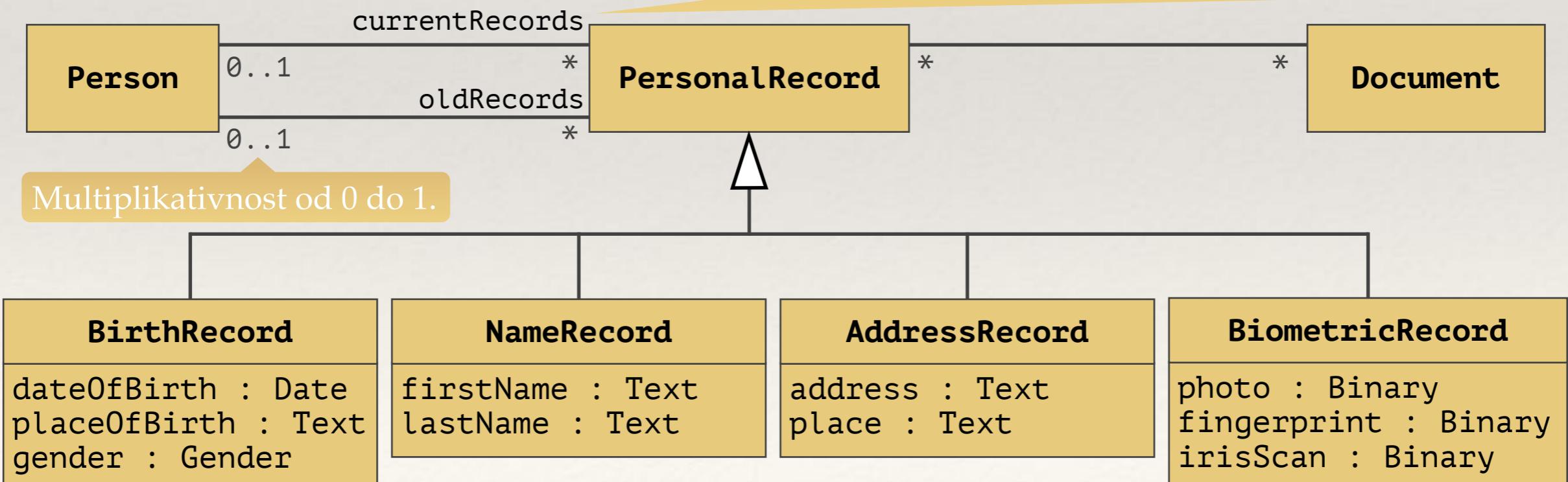
- ❖ Banalno rešenje: preslikati dati opis u klase i raspored njihovih svojstava
 - ❖ Problemi:
 - nepotrebno mnogostruko kopiranje podataka, jer jedna osoba ima mnogo izdatih dokumenata
 - ako je potrebno čuvati istoriju promena podataka, prave se kopije svih podataka, iako se menja samo manji deo njih; gde čuvati te promene?
- UML simbol za *asocijaciju (association)*: relacija između klasa koja definiše skup *veza (link)* koje mogu postojati između objekata tih klasa.



Klasa kao realizacija apstrakcije

- ❖ Ideja boljeg rešenja: uvesti koncept *Personalnog zapisa* (*Personal Record*) koji sadrži određene lične podatke:
 - umesto da sadrže kopije, *Osoba* i *Dokument* referenciraju (vezani su za) *Personalne zapise*: sve dok se podatak ne menja, ovi objekti vezani su za iste zapise
 - prilikom promene nekog podatka, pravi se nova kopija zapisa u kom se taj podatak menja
 - kako bi se dodatno smanjilo kopiranje, podaci su grupisani u različite specijalizacije zapisa, po tome koliko se često i da li se zajedno ili odvojeno menjaju
 - za *Osobu* se mogu čuvati i raniji zapisi, sa svim podacima koji su ikada bili registrovani za tu osobu (nezavisno od *Dokumenata* u kojima su upotrebljeni)

UML oznaka *uloge* (*role*) kraja asocijacije: objekti na ovom kraju veza igraju ovu ulogu u relaciji.



Klasa kao realizacija softverske mašine

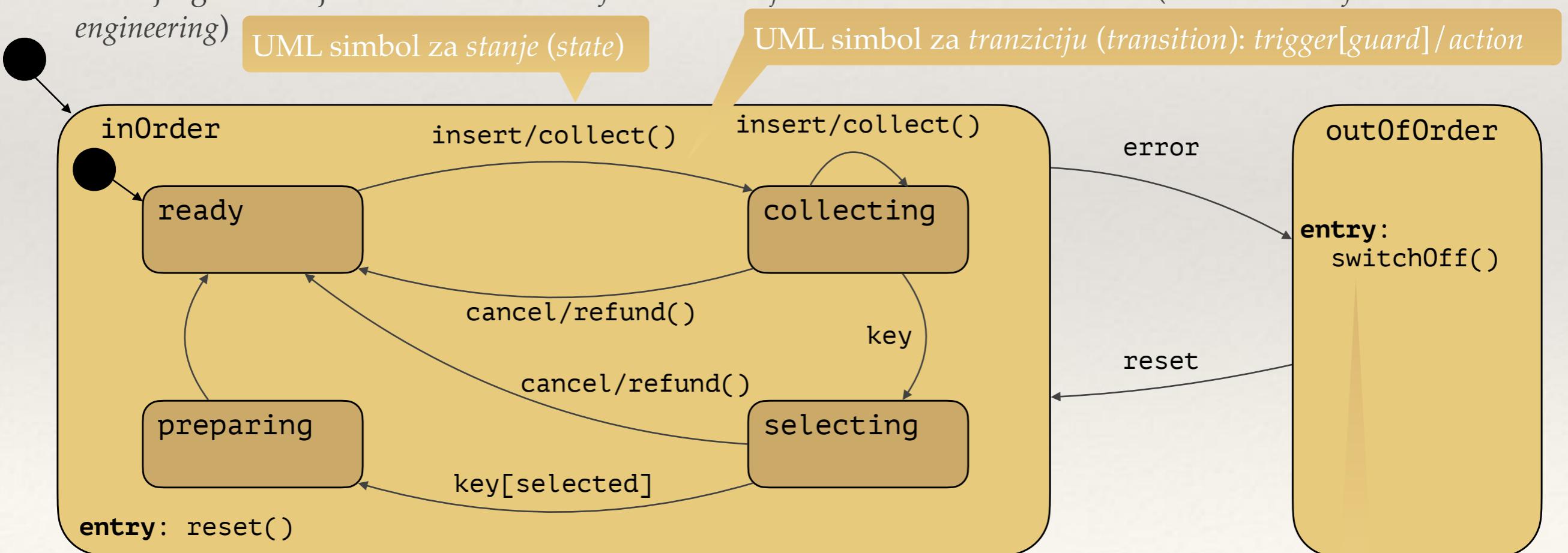
- ❖ U nekim slučajevima, reakcija objekta zavisi ne samo od *pobude* (pozvane operacije), nego i od *predistorije* pobuda, odnosno od trenutnog *stanja* tog objekta
- ❖ Primer: softver koji upravlja automatom za prodaju napitaka - reakcija na pritisak tastera za izbor artikla zavisi od toga šta se prethodno dogodilo
- ❖ Automat može da bude *ispravan* ili u stanju *van upotrebe* (npr. zbog problema ili punog kontejnera za novac) itd.
- ❖ Kako bi se ovakvo ponašanje implementiralo? Na primer:

```
void VendingMachine::insertCoin (int value) {  
    if (this->inOrder)  
        this->amount += value;  
    else  
        if (!this->isRefundBarrierOpen)  
            this->openRefundBarrier();  
}  
  
void VendingMachine::selectArticle (int code) {  
    if (this->inOrder)  
        if (this->articles[code]>0 && this->prices[code]<=this->amount) {  
            this->amount -= this->prices[code];  
            this->deliverArticle(code);  
        }  
}  
  
...
```

- ❖ Problem: kod složenijeg ponašanja, kod postaje nepregledan, težak za razumevanje, težak za održavanje (izmene, proširenja) i podložan greškama

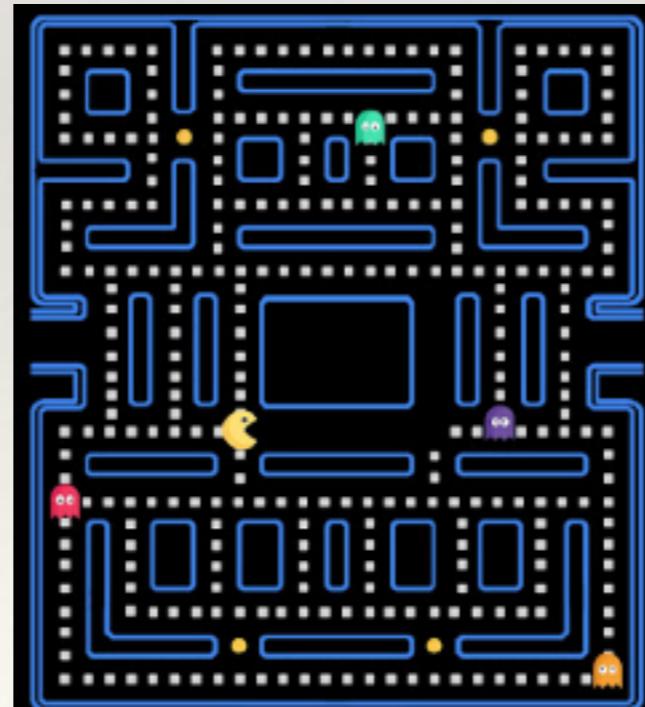
Klasa kao realizacija softverske mašine

- ❖ U ovakvim situacijama, za modelovanje ponašanja objekata neke klase koristi se *mašina stanja (state machine)* — koncept višeg nivoa apstrakcije koji modeluje ponašanje, uz mogućnost predstavljanja modela dijagramom
- ❖ Mašine stanja su napredan softverski koncept koji potiče od konačnih automata, ali dodaje mnoge druge, naprednije koncepte, kao što su hijerarhijska (ugnežđena) stanja, ulazne i izlazne akcije i mnoge druge
- ❖ Nažalost, nijedan klasičan OO programski jezik ne podržava mašine stanja: ovakvo ponašanje se mora implementirati ručno, što je naporno i podložno greškama
- ❖ Rešenje: generisanje koda iz modela - *softversko inženjerstvo zasnovano na modelima (model-based software engineering)*



Klasa kao realizacija softverske mašine

- ❖ Prepostavimo da pravimo softver za neku arkadnu igricu, u kojoj se različiti objekti, tzv. likovi, kreću uporedno po igračkom polju. Kako da implementiramo to nezavisno i uporedno kretanje koje se (makar prividno) dešava istovremeno?
- ❖ Ideja:
 - svaki lik (*character*) predstavimo objektom, po potrebi određene izvedene klase
 - svaki lik implementira operaciju koja izvodi jedan korak (*step*) svog kretanja; ova operacija treba da bude kratka i da izvrši jedan elementaran pomeraj, po što je moguće kraći
 - jedna glavna petlja “proziva” sve likove na igračkom polju i svakome daje da se pomeri za po jedan korak
- ❖ Na ovaj način možemo da stvorimo privid uporednog kretanja objekata kao *aktivnih* entiteta



Klasa kao realizacija softverske mašine

```
class Character {  
public:  
    ...  
    virtual bool step ();  
    ...  
protected:  
    virtual void calcMove (int& dx, int& dy);  
    int myX, myY;  
};  
  
class Pacman : public Character {  
public:  
    ...  
    virtual bool step ();  
    ...  
};  
  
class Ghost : public Character {  
public:  
    ...  
    virtual bool step ();  
    ...  
};  
  
class Engine {  
public:  
    ...  
    void run ();  
    ...  
private:  
    list<Character*> myChars;  
    ...  
};
```

Zaštićeni (*protected*) članovi su dostupni u izvedenim klasama, ali ne i drugde van klase

```
void Egine::run () {  
    while (!gameOver) {  
        for (Character* c : this->myChars) {  
            c->step();  
            ...  
        }  
    }  
  
    bool Ghost::step () {  
        int dx, dy;  
        this->calcMove(dx,dy);  
        if (dx!=0 || dy!=0) {  
            theField->clearGhostSprite(myX,myY);  
            this->myX += dx;  
            this->myY += dy;  
            theField->drawGhostSprite(myX,myY);  
        }  
        return true;  
    }  
  
    bool Pacman::step () {  
        ...  
    }  
}
```

Klasa kao realizacija softverske mašine

- ❖ Problem: ako ponašanje nekog aktivnog objekta postane suviše složeno i dugotrajno, da bi se postigao privid paralelizma, to ponašanje se mora deliti na manje intervale; tada se kontekst (stanje) mora čuvati i prenositi između različitih poziva operacije *step*, pa programiranje postaje teško, a kod nepregledan
- ❖ Osim toga, šta ako aktivni objekat ne može ili ne treba da nastavi svoju aktivnost, odnosno treba da je suspenduje dok se ne ispuni neki uslov?
- ❖ Na primer: program za obradu teksta; korisnik pokreće operaciju snimanja dokumenta u fajl (*save*); da li čekati da se ova operacija završi i ne dozvoliti korisniku da bilo šta radi sve dok ona traje? Šta ako to potraje previše? Bespotrebno!
- ❖ Rešenje: ponašanje aktivnih objekata realizovati kao *niti (thread)*
- ❖ *Nit (thread)* je sekvenca izvršenih akcija (naredbi) koja teče *uporedo* sa drugim takvim sekvencama (nitima)
- ❖ Nit se kreira nad pozivom neke funkcije, a kontrola niti dalje ide kako diktira kod te funkcije i onoga što ona dalje poziva
- ❖ Svaka nit ima svoj *tok kontrole (control flow)*, koji uključuje i automatske promenljive — svaka nit ima *svoj stek poziva*

Klasa kao realizacija softverske mašine

```
class DocumentSaver {  
public:  
    ...  
    DocumentSaver (Document*);  
    ~DocumentSaver ();  
    ...  
private:  
    static void run (DocumentSaver*);  
    void save () // Performs document saving  
    Document* myDocument;  
    thread* myThread;  
    ...  
};  
  
void DocumentSaver::run (DocumentSaver* ds) {  
    if (ds) ds->save();  
    delete ds;  
}  
  
DocumentSaver::DocumentSaver (Document* d) {  
    this->myDocument = d;  
    if (d) myThread = new thread(run, this);  
}  
  
DocumentSaver::~DocumentSaver () {  
    delete myThread;  
}
```

- ❖ Kada treba pokrenuti akciju snimanja dokumenta, samo treba kreirati jedan objekat klase *Document Saver*:
new DocumentSaver(theDocument);
- ❖ Sada su objekti klase *DocumentSaver aktivni*: svaki objekat ove klase ima jednu nit u kojoj se izvršava operacije *save*
- ❖ Kada se ova operacija završi, objekat ove klase se uništava
- ❖ Ova nit, pa time i operacija *save*, izvršava se uporedo sa svim drugim nitima u programu
- ❖ Ova nit, odnosno operacija *save* koju ona izvršava, može da pristupa *deljenim podacima*: pristupa strukturi podataka koja predstavlja dokument (*Document*), kako bi ga snimila u fajl
- ❖ Uporedo sa tim, ostale niti rade svoj posao: na primer, "glavna" nit može da obrađuje akcije korisnika, pa korisnik ne mora više da čeka na završetak operacije *save*
- ❖ Operacija *save* ne mora da se deli na korake, već se programira kao jedinstvena, sekvensijalna operacija, što olakšava programiranje

Glava 5: Objektna dekompozicija

- ❖ Karakteristike lošeg i dobrog softvera
- ❖ Raspodela odgovornosti
- ❖ Algoritamska dekompozicija
- ❖ Relacije i zavisnosti između klasa
- ❖ Enkapsulacija
- ❖ Hijerarhijska dekompozicija



Karakteristike lošeg i dobrog softvera

Karakteristike lošeg softvera:

- ❖ *Rigidnost (rigidity)* — teško ga je promeniti, jer svaka promena jednog utiče na suviše drugih delova softvera:
 - delovi softvera su međusobno jako zavisni, pa svaka promena u jednom zahteva promene u drugim delovima
 - zato je efekte i troškove promene teško predvideti, programeri ne mogu da ih procene, pa rukovodioci teško odobravaju promene
- ❖ *Krhkost (lomljivost, fragility)* — tendencija da promena u softveru izaziva neočekivane probleme i otkaze u mnogim delovima tog softvera:
 - zbog jake isprepletanosti i zamršenosti, svaka promena u jednom delu izaziva problem u drugom - *efekat domina (domino effect)*
 - problemi su često u delovima koji nemaju konceptualne veze sa onim na koje se promena odnosi
 - degradira se kredibilitet softvera i poverenje u njegov kvalitet
- ❖ *Neprenosivost (immobility)* — delove jedne aplikacije teško je ponovo upotrebiti u drugoj, jer se ne mogu rasplesti od ostatka koji nije potreban, opet zbog prevelikih zavisnosti:
 - napor potreban za takvo raspetljavanje je prevelik, pa se lakše odlučuje za pravljenje istih stvari iznova

Karakteristike lošeg i dobrog softvera

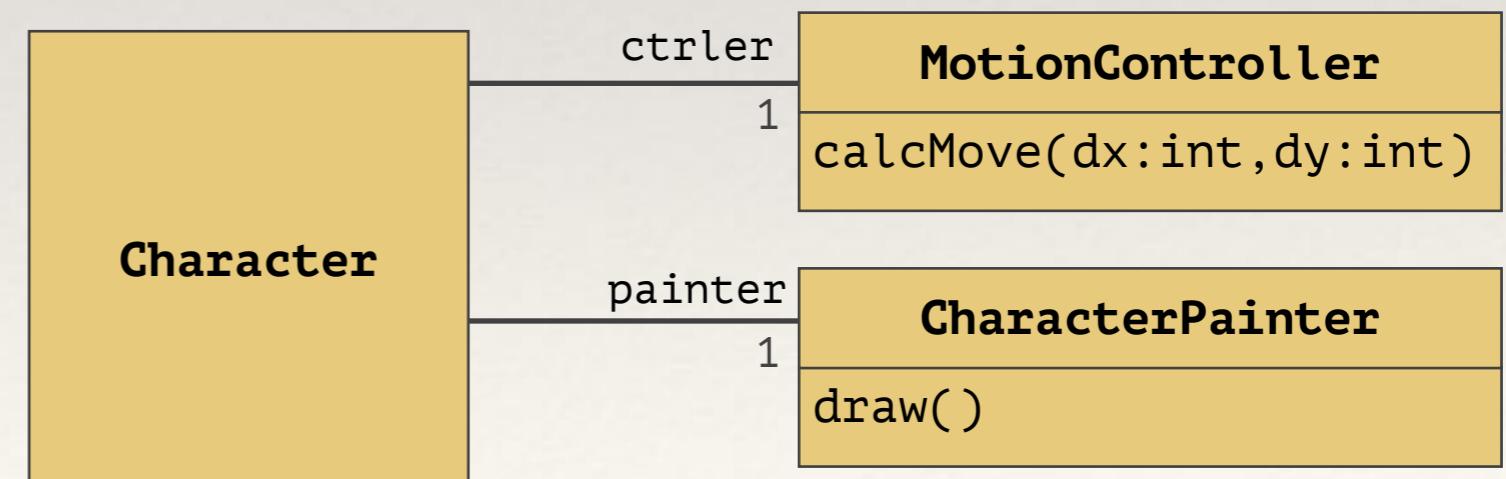
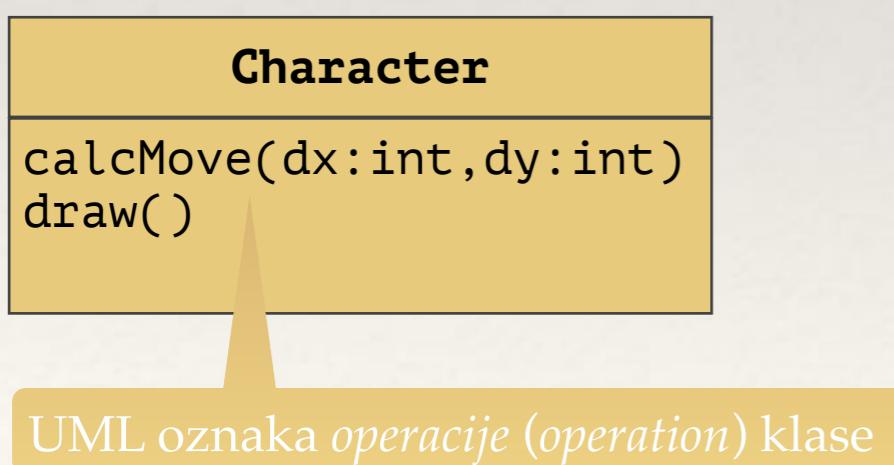
- ❖ Dobar softver ne poseduje ove karakteristike, što znači da je:
 - *fleksibilan (flexible)*: lako ga je održavati (menjati i proširivati)
 - *robustan (robust)*: otporan na promene, tj. promene ne utiču na njegovu ispravnost
 - *prenosiv (portable)* i *ponovno upotrebljiv (reusable)*: delovi se mogu iskoristiti u drugim aplikacijama bez mnogo ulaganja
- ❖ Naravno, ne postoji idealan softver i nijedan softver ne može biti neograničeno fleksibilan, robustan i prenosiv, ali je cilj da bude takav u opsegu *predvidivih* promena uslova i načina korišćenja u određenom domenu primene. Dobar i iskusan arhitekta softvera i poznavalac njegovog domena mogu ovo da predvide i procene
- ❖ Osnovni preduslovi za kvalitetan softver sa ovakvim karakteristikama:
 - jasna, jednostavna, pravilna, razumljiva, fleksibilna *arhitektura*
 - prave apstrakcije, dobra raspodela odgovornosti
 - modularnost, enkapsulacija
 - slabe zavisnosti između delova (apstrakcija, modula), jednostavni interfejsi
- ❖ Sve ovo ima svoju cenu, pa je u dobru arhitekturu i dizajn softvera potrebno uložiti više napora i vremena u početku, da bi se kasnije efekti videli sve većom brzinom; kod lošeg softvera je obrnuto
- ❖ Karakteristika nepreporučljivog ponašanja: *žurba ka kodovanju (rush to code syndrom)*

Raspodela odgovornosti

- ❖ Preduslov za postizanje opisanih vrlina jeste *dobro razdvajanje brige (separation of concerns)* i *raspodela odgovornosti (distribution of responsibility)* po apstrakcijama, klasama, modulima
- ❖ Elemente softvera (podatke, funkcionalnosti) treba grupisati u klase ili module po principu *jake kohezije i slabe sprege (strong cohesion/loose coupling)*:
 - elementi unutar iste klase ili modula treba da budu *jako i tesno međusobno povezani (kohezija, cohesion)*
 - elementi iz različitih klasa i modula treba da budu *slabo spregnuti (loose coupling)*
- ❖ Ponekad se ovo izražava i kao *princip jedinstvene odgovornosti klase (single responsibility principle)*: klasa treba da ima ograničenu odgovornost, tako da postoji samo jedan razlog za njenu izmenu
- ❖ Ako se različite odgovornosti mogu nezavisno menjati, postoji ozbiljan razlog za razdvajanje tih odgovornosti u različite klase
- ❖ Klasa svakako ne treba da ima previše odgovornosti, posebno ako su one slabo povezane ili nepovezane - raspodela odgovornosti je jedan element *objektne dekompozicije*

Raspodela odgovornosti

- ❖ Na primer, logično je da u odgovornosti klase za lika u arkadnoj igri bude izračunavanje njegovih sledećih koordinata tokom kretanja po polju, kao i iscrtavanje tog karaktera na polju; međutim, ako je neka od ove dve odgovornosti suviše složena (npr. zavisi od složenih stanja, struktura ili algoritama), ili se mogu nezavisno kombinovati, treba razmisliti o njihovom razdvajanju u različite klase: lik, kao apstrakcija, može imati, kao deo svoje implementacije, objekte klasa zaduženih za izračunavanje pomeraja i za iscrtavanje
- ❖ Objekat klase *Character* delegira (*delegates*) odgovornost za ove obaveze tako što poziva odgovarajuće operacije tih pridruženih objekata kada mu je potrebno izračunavanje pomeraja ili iscrtavanje
- ❖ Na taj način, način izračunavanja pomeraja i/ili iscrtavanja može se nezavisno menjati specijalizacijama ovih klasa, jer navedeni pozivi mogu biti polimorfni
- ❖ Naravno, sve ovo ima smisla samo ako su ove promene predvidive: nema potrebe raditi dodatan posao i činiti softver složenijim ako takve promene nisu očekivane. Čak i ako nisu predviđene, ukoliko je ispoštovana enkapsulacija apstrakcije lika, softver se može *refaktorisati* (*refactoring*) sa ciljem ovakve dekompozicije



Raspodela odgovornosti

- ❖ Jedan od osnovnih principa softverskog inženjerstva, srođan sa navedenim, jeste princip *lokalizacije projektnih odluka* (*localization of design decisions*): manifestacija neke projektne odluke u programu treba da bude *lokализована* na jednom mestu, a ne rasuta na više mesta
- ❖ Ovo stoga što bi promena te odluke bila teška za sprovođenje ukoliko su njene manifestacije rasute po programu: izmena je teška i podložna greškama, a rizik domino efekta veliki; lokalizacija projektne odluke olakšava njenu promenu i povećava šansu da ta promena prođe bez problema
- ❖ Projektna odluka može da bude različite vrste, složenosti i nivoa apstrakcije: od najprostije odluke da je neka promenljiva nekog tipa ili neka konstanta ima neku vrednost, pa sve do odluke o načinu sprovođenja nekog scenarija u programu ili o arhitekturi datog softvera
- ❖ Najjednostavniji primer jeste korišćenje konstanti u programu:
 - korišćenje neposrednog literala (npr. broja 100) za neku konstantu (npr. dimenziju nekog niza) na više mesta u kodu (npr. prilikom deklarisanja niza i prilikom iteriranja kroz niz do njegove granice) predstavlja prestup ovog principa: promenu odluke da niz bude baš te veličine na neku drugu: a) zahteva prolazak kod i sva mesta korišćenja tog literala i pažljivu zamenu drugom vrednošću, b) podložno je greškama, jer se na nekom mestu ista ta vrednost (npr. 100) može koristiti sa potpuno drugim značenjem, i ovakovom zamenom pogrešno promeniti
 - umesto toga, uvođenje simboličke konstante koja ima željenu vrednost predstavlja doslednu primenu ovog principa: odluka da je data dimenzija baš određena je lokalizovana na jednom mestu - u deklaraciji te konstante
- ❖ Naravno, i ovo ima ograničene domete i važi samo za odluke koje imaju perspektivu da se promene: celokupnu arhitekturu datog softvera nije lako promeniti, dok se neke činjenice verovatno neće promeniti za života softvera (npr. činjenica da sat ima 60 minuta ili sedmica 7 dana itd.)
- ❖ Klasa može da bude način i mesto za lokalizaciju projektnih odluka određenog nivoa granularnosti i apstrakcije: raspodela neke odgovornosti u neku klasu jeste jedna pojava lokalizacije projektne odluke

Algoritamska dekompozicija

- ❖ Fundament proceduralnog programiranja jeste *algoritamska (proceduralna) dekompozicija (algorithmic, procedural decomposition)*:
 - u centru pažnje jeste zadatak ili posao koji treba uraditi, za koji se definiše *postupak (procedura, algoritam)*
 - taj postupak se inicijalno deli na *korake*, najpre visokog nivoa apstrakcije i velike granularnosti (“da bi se to uradilo, potrebno je najpre uraditi a , pa onda b , potom, ako je ispunjen uslov c , treba uraditi d , inače e itd); ovi koraci definišu se kao *procedure* (potprogrami)
 - dalje se svaki krupniji korak deli istim postupkom na manje korake (rekurzivna primena istog postupka dekompozicije), sve dok se ne dođe do najsitnijeg nivoa granularnosti, onog koji se može izraziti elementarnim konstruktima programskog jezika koji se koristi
- ❖ Procedure su takođe proizvod apstrakcije: određeni (krupniji) korak se apstrahuje kao celina, pri čemu se njegova razrada i detalji odlažu za kasnije, jer za korišćenje nisu bitni (bitan je samo interfejs)
- ❖ Primer: Program koji održava n stek-mašina koje treba da izvršavaju komande (operacije *add*, *sub*, *push*, *pop* itd). Na ulazu se nalaze komande u tekstualnom obliku, svaka u po jednom redu. Svaka komanda je u formatu $i:command$, gde je i redni broj stek-maštine na koju se odnosi komanda u nastavku. Sa ulaza se učitava jedan po jedan znak operacijom *getChar*. Zapis komande sa ulaza treba prevesti pre nego što se izvrši.

Algoritamska dekompozicija

```
enum OpCode { add, sub, ...};  
  
int main () {  
  
    int out;  
    string cmdin;  
    OpCode cmdout;  
  
    while (readCommand(out,cmdin)) {  
        translate(cmdin,cmdout);  
        performCmd(out,cmdout);  
    }  
}
```

Prenos argumenta po referenci (by reference)

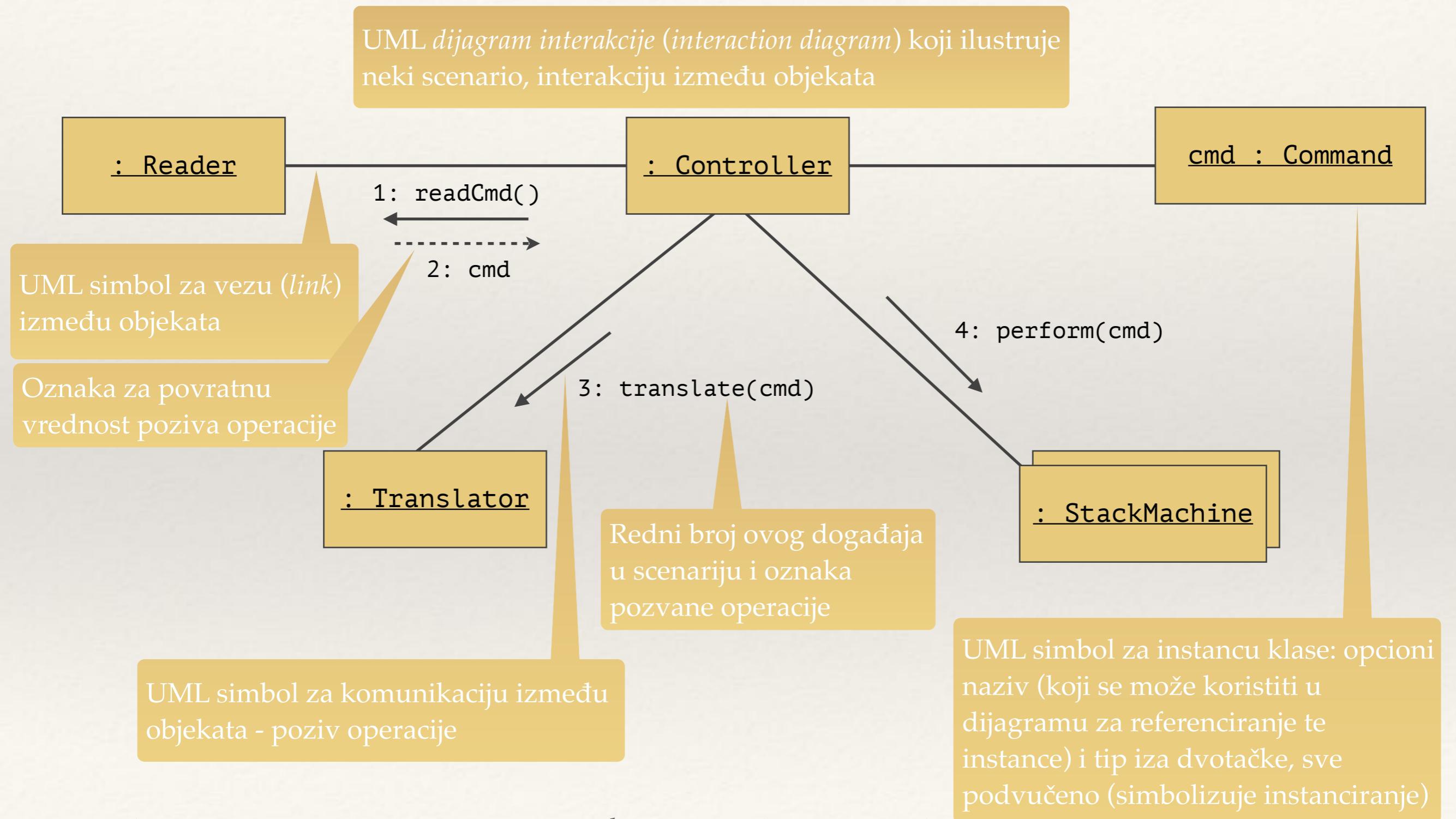
```
bool readCommand (int& out, string& cmd) {  
  
    bool ret = readOut(out);  
    if (!ret) return false;  
  
    return readCmd(cmd);  
}  
  
bool readOut (int& out) {  
  
    out = 0;  
  
    char c = getChar();  
    while (isDigit(c)) {  
        out = out*10 + (c-'0');  
        c = getChar();  
    }  
    return (c!=EOF);  
}  
  
bool readCmd (string& cmd) {  
    ...  
}
```

```
struct Stack {  
    int stack[MaxStackSize];  
    unsigned sp;  
};  
  
Stack sm[N];  
...  
  
void translate (string cmdin, OpCode& cmdout) {  
    if (cmdin=="ADD") cmdout = add;  
    if (cmdin=="SUB") cmdout = sub;  
    ...  
}  
  
void performCmd (int out, OpCode cmd) {  
    switch (cmd) {  
  
        case add: {  
            int op1 = pop(out);  
            int op2 = pop(out);  
            push(out,op1+op2);  
            break;  
        }  
  
        case sub: ...  
        ...  
    }  
}  
  
int pop (int out) {  
    if (sm[out].sp==0) return 0;  
    else return sm[out].stack[--sm[out].sp];  
}  
  
...
```

Algoritamska dekompozicija

- ❖ U OOP algoritamska dekompozicija ima podjednako važnu ulogu, mada ne više centralnu i nije jedini element, već sastavni element *objektne dekompozicije*
- ❖ Kao i u proceduralnom programiranju, algoritamska dekompozicija se koristi za razlaganje određenog postupka, ali uz jednu bitnu razliku: u okviru razlaganja postupka, za svaki korak definiše se *činilac* koji je *odgovoran* za izvršavanje tog koraka
- ❖ Koraci se raspodeljuju činiocima (apstrakcijama, klasama) na osnovu njihovih ranije definisanih opsega odgovornosti, ali je moguće i obratno: dekompozicijom na korake se raspodeljuju odgovornosti po apstrakcijama, ali se mogu i uvoditi ili identifikovati nove apstrakcije
- ❖ Zbog toga se postupci dekomponovani na korake i raspoređeni kao odgovornosti po klasama nazivaju češće *scenarijima* (*scenario*) ili *mehanizmima* (*mechanism*)

Algoritamska dekompozicija



Algoritamska dekompozicija

```
enum OpCode { add, sub, ... };

class Command {
public:
    Command (int out, string cmd);
    OpCode getOpCode ();
    int    getOut ();
    ...
private:
    string in;
    OpCode code;
    int out;
};

class Controller {
public:
    void main ();
    ...
private:
    Reader* myReader;
    Translator* myTranslator;
    StackMachine* mySms[...];
};

void Controller::main () {
    Command* cmd = myReader->read();
    while (cmd!=nullptr) {
        myTranslator->translate(cmd);
        int out = cmd->getOut();
        mySms[out]->perform(cmd);
    }
}

class Reader {
public:
    Command* read ();
    ...
};

class Translator {
public:
    void translate (Command* );
    ...
};

class StackMachine {
public:
    void perform (Command* );
    ...
protected:
    void push (int);
    int pop ();
    ...
private:
    int stack[MaxStackSize];
    unsigned sp;
    ...
};

void StackMachine::perform (Command* cmd) {
    switch (cmd->getOpCode()) {
        case add: {
            int op1 = this->pop();
            int op2 = this->pop();
            this->push(op1+op2);
            break;
        }
        case sub: ...
        ...
    }
}
```

Algoritamska dekompozicija

- ❖ Niži nivo granularnosti primene ovakve dekompozicije jeste u implementaciji pojedinačnih operacija neke klase. Kada se takva operacija dekomponuje na korake (potprograme), ti koraci mogu biti delegirani drugim učesnicima (klasama) u scenariju, ali mogu biti i dalje u opsegu odgovornosti iste klase
- ❖ Ove poslednje pomenute operacije nazivaju se *pomoćne (helper)*, jer se koriste u implementaciji ostalih operacija iste klase
- ❖ Neki saveti za to kada dekomponovati metodu (implementaciju operacije):
 - Kada implementacija ima previše koda, postaje teška za praćenje i razumevanje; po pravilu, metoda sa preko 15-20 linija koda predstavlja signal da treba razmisliti o dekomponovanju na potprograme - krupnije potkorake
 - Kada postoje dublje ugnezđene uslovne strukture (*if-then-else*) ili petlje: po pravilu, nije dobro imati više od dva nivoa ugnezđivanja, jer to otežava čitanje i razumevanje; tada treba razmisliti o izdvajanju posla koji rade ugnezđene strukture u potprograme; to onda olakšava i samo programiranje
 - Kada se delovi implementacije jedne operacije koriste i u drugim - veoma jak razlog za izdvajanje u potprogram, radi izbegavanja dupliranja koda; *copy-paste* je veoma korisna, ali može biti i veoma loša praksa u programiranju; svako kopiranje koda treba da bude signal za razmišljanje da li je propuštena neka apstrakcija (uključujući i algoritamsku dekompoziciju - izdvajanje u potprogram)
- ❖ Ovakve pomoćne operacije najčešće nisu deo interfejsa klase, već deo njene implementacije, pa *ne treba* da budu javne (*public*), već privatne (*private*) ili češće *zaštićene (protected)*, kako bi bile dostupne izvedenim klasama; ovo stoga što ako su već korišćene za implementaciju drugih operacija u osnovnoj klasi, velika je šansa da budu korisne i potrebne u implementaciji (npr. redefinisanih) operacija u izvedenim klasama
- ❖ Često su ovakvi koraci - potprogrami u implementaciji operacija osnovne klase *polimorfni*, tj. imaju svoje podrazumevano ponašanje, ali se to ponašanje može redefinisati u izvedenim klasama; ovakve operacije nazivaju se *kukice (hook methods)*, jer se na njih može "okačiti" drugačija implementacija i time promeniti ponašanje operacije osnovne klase u nekom delu

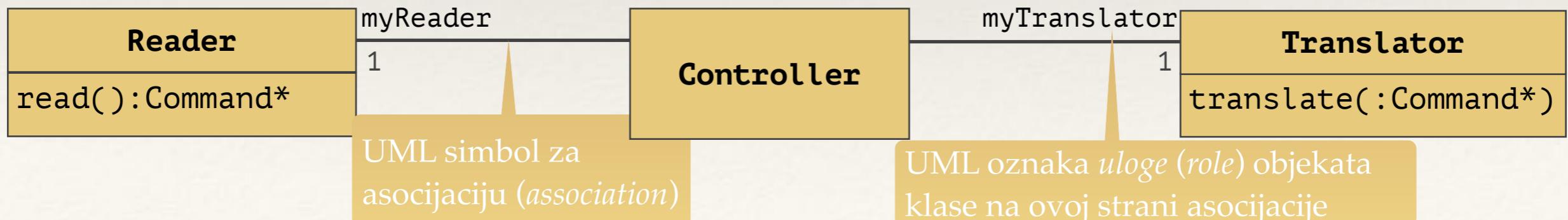
Algoritamska dekompozicija

```
class Reader {  
public:  
    Command* read ();  
    ...  
  
protected:  
    bool readOut(int& out);  
    bool readCmd(string& cmd);  
  
private:    Pomoćne (helper) operacije  
    ...  
}  
  
Command* Reader::read () {  
    int out;  
    string cmd;  
  
    bool ret = readOut(out);  
    if (!ret) return nullptr;  
  
    ret = readCmd(cmd);  
    if (!ret) return nullptr;  
  
    return new Command(out,cmd);  
}
```

```
class StackMachine {  
public:  
    void perform (Command*);  
    ...  
  
protected:  
    void push (int);  
    int pop ();  
    ...  
};  
  
void StackMachine::perform (Command* cmd) {  
    switch (cmd->getOpCode()) {  
  
        case add: {  
            int op1 = this->pop();  
            int op2 = this->pop();  
            this->push(op1+op2);  
            break;  
        }  
  
        case sub: ...  
        ...  
    }  
  
    int StackMachine::pop () {  
        if (this->sp==0) return 0;  
        else return this->stack[--this->sp];  
    }  
    ...  
}
```

Relacije i zavisnosti između klasa

- ❖ Da bi ispunila svoje odgovornosti, klasa (npr. *Controller*) najčešće sarađuje sa drugim klasama (npr. *Reader*, *Translator*); preciznije, objekat te klase mora da sarađuje sa objektima drugih klasa, recimo tako što od njih traži usluge, tj. poziva njihove operacije
- ❖ Da bi jedan objekat (klijent) sarađivao sa drugim (serverom), odnosno pozivao njegove operacije, mora da ima struktturnu vezu (*link*) sa njim — u OOP jeziku, da poseduje pokazivač/ referencu ka njemu
- ❖ Relacije između klasa koje predstavljaju strukturne veze između objekata tih klasa nazivaju se *asocijacije* (*association*); veze su instance asocijacija i povezuju objekte kao instance klasa
- ❖ Tokom izvršavanja programa, objekti nastaju i nestaju, a veze između objekata se uspostavljaju i raskidaju; klase i asocijacije su deo *programa* (ili modela softvera) kao statičkog zapisa, objekti i veze su deo *objektnog prostora* (*object space*) koji postoji u vreme izvršavanja i poseduje dinamiku
- ❖ Veze asocijacija ne moraju služiti samo za “prenos” poziva operacija između objekata; one mogu i samo predstavljati prostu informaciju da su određeni objekti povezani nekom konceptualnom relacijom; na primer, činjenicu da neki *učenik* (*Student*) *pohađa* (*attend*) određenu *školu* (*School*)



Relacije i zavisnosti između klasa

- ❖ Veze između objekata mogu biti kratkotrajne, tj. takve da ne nadživljavaju trajanje metode u kojoj se koriste, tako što posmatrana operacija:
 - kao argument prima pokazivač na objekat-server:

```
void Controller::main (Reader* myReader, Translator* myTranslator) {  
    Command* cmd = myReader->read();  
    while (cmd!=nullptr) {  
        myTranslator->translate(cmd);  
        int out = cmd->getOut();  
        mySMs[out]->perform(cmd);  
    }  
}
```

- dobija pristup do datog objekta putem neke druge operacije (npr. kao njenu povratnu vrednost), ili traži takav objekat na neki drugi način:

```
void Controller::main () {  
    Command* cmd = myReader->read();  
    while (cmd!=nullptr) {  
        myTranslator->translate(cmd);  
        int out = cmd->getOut();  
        mySMs[out]->perform(cmd);  
    }  
}
```

Relacije i zavisnosti između klasa

- ❖ Sa druge strane, veze mogu da nadžive izvršavanje operacije, pa se moraju skladištiti unutar objekta:

```
class Controller {  
    ...  
  
private:  
    Reader* myReader;  
    Translator* myTranslator;  
    StackMachine* mySMs[...];  
};
```

- ❖ Ukoliko je gornja granica multiplikativnosti odgovarajućeg kraja asocijације veća od 1, za implementaciju se mora koristiti neka kolekcija:

```
class Teacher {  
public:  
    ...  
    void addCourse(Course*);  
    void removeCourse(Course*);  
  
private:  
    list<Course*> myCourses;  
    ...  
};
```

Operacije koje uspostavljaju i raskidaju veze između objekata klasa *Teacher* i *Course*

Relacije i zavisnosti između klasa

- ❖ Umesto da objekat-klijent sam kreira potrebne objekte-servere sa kojima sarađuje, ili sam traži pristup do njih i identificuje ih (npr. pozivom nekih usluga drugih klasa), praksa pokazuje da je bolji pristup da neko drugi, i to onaj ko koristi posmatranu klasu, spolja, *injektira* te veze, odnosno definiše zavisnosti od servera
- ❖ Ovo se može uraditi na sledeće načine:
 - kroz konstruktor posmatrane klase, ali samo ako su te veze obavezne (minimalna multiplikativnost je veća od 0, odnosno objekat *mora* biti vezan):

```
class Controller {  
public:  
    Controller (Reader* reader, Translator* translator);  
    ...  
};  
  
Controller::Controller (Reader* reader, Translator* translator) {  
    this->myReader = reader; this->myTranslator = translator;  
}
```

- kroz posebnu operaciju klase kojom se ova veza uspostavlja, ako se takva veza može menjati tokom života objekta:

```
class Controller {  
public:  
    void setReader (Reader* reader);  
    void setTranslator (Translator* translator);  
    ...  
};  
  
void Controller::setReader (Reader* reader) {  
    this->myReader = reader;  
}
```

- ili oba, ako važi i jedno i drugo:

```
Controller::Controller (Reader* reader, Translator* translator) {  
    this->setReader(reader); this->setTranslator(translator);  
}
```

Relacije i zavisnosti između klasa

- ❖ Ovaj princip naziva se *injekcija zavisnosti* (*dependency injection*)
- ❖ Na ovaj način se postiže bolja fleksibilnost, jer se objekti posmatrane klase mogu upotrebljavati na različite načine i u različitim kontekstima, odnosno mogu se menjati konfiguracije njihovih veza sa drugim objektima, bez izmena njihovih klasa
- ❖ Drugim rečima, odgovornost za povezivanje objekata u *kolaboraciju* (*collaboration*), dodeljuje se nekoj drugoj klasi — opet je važno dobro identifikovati i raspodeliti odgovornosti
- ❖ Na primer, u zavisnosti od situacije, dati objekat klase *Controller* može se povezati sa jednim ili drugim objektom klase *Reader* i *Translator*, ili čak njihovim specijalizacijama; on neće trpeti nikakvu izmenu i ne zavisi od svega toga, pošto nije ni svestan tih različitih konfiguracija u kojima je upotrebljen:

```
class FileReader : public Reader {...};

class ExtendedTranslator : public Translator {...};

...
Reader* simpleReader = new Reader(...);
FileReader* fReader = new FileReader(...);
ExtendedTranslator* translator = new ExtendedTranslator(...);

Controller* ctrlr1 = new Controller(simpleReader,translator);
Controller* ctrlr2 = new Controller(fReader,translator);
```

Enkapsulacija

- ❖ Svi elementi deklarisani unutar klase, a to mogu biti podaci članovi, funkcije članice, ali i tipovi (enumeracije, strukture, pa i klase) nazivaju se njenim *članovima (member)*
- ❖ Specifikatori *public*: *protected*: i *private*: se mogu navoditi u proizvoljnom redosledu, pa i više puta ponavljati (pa čak i navoditi ispred svakog člana); ako se ne navede neki od ovih specifikatora od početka definicije klase, podrazumeva se da su ti članovi *private*

```
class Controller {  
    ...  
    protected:  
    ...  
  
    public:  
    ...  
    private:  
    ...  
  
    public:  
    ...  
};
```

Podrazumeva se *private*

Loš stil (nepregledno),
mada je dozvoljeno jezikom

Preporučen stil

```
class Controller {  
    public:  
    ...  
    protected:  
    ...  
    private:  
    ...  
};
```

- ❖ Međutim, radi povećanja čitljivosti, preporučuje se sledeći stil i redosled:
 - *public*, jer je to interfejs klase i smatra se da je najviše onih koji su zainteresovani za interfejs, jer koriste tu klasu
 - *protected*, jer je to specifičan interfejs klase, za povlašćene korisnike - one koji prave izvedene klase
 - *private*, jer je to implementacija klase i ona treba da bude sakrivena, a za nju je najmanje zainteresovanih - oni koji se bave implementacijom klase

Enkapsulacija

- ❖ Što se tiče pravila jezika C++, bilo koji član može biti bilo koje vrste dostupnosti: javan (*public*), zaštićen (*protected*) ili privatан (*private*) ; međutim, iskustvena preporuka jeste ta da se klasa pravi tako da (osim ako postoje jaki razlozi i opravdanje za drugačije odluke):
 - podaci članovi budu isključivo privatni, kako bi pristup do njih mogao lakše da se kontroliše i menja; ako je potrebno pristupati podacima članovima, napraviti operacije za:
 - čitanje, tzv. *getter* operacije koje čitaju i vraćaju vrednost podatka (npr. *getName*)
 - upis, tzv. *setter* operacije koje postavljaju vrednost podatka (npr. *setName*)
 - javne treba da budu samo operacije koje čine interfejs date klase
 - zaštićene mogu da budu operacije:
 - *getter / setter* operacije, ukoliko ne predstavljaju deo javnog interfejsa klase, a izvedenim klasama je potreban pristup do podataka članova
 - pomoćne (*helper*) operacije, jer se one koriste za implementaciju metoda operacija iz interfejsa, pa je velika šansa da takve iste budu potrebne i u redefinisanim metodama izvedenih klasa; tu se uključuju i “kukice” (*hook*), jer su one svakako predviđene za redefinisanje u izvedenim klasama

Enkapsulacija

- ❖ U nekim situacijama deo interfejsa (tj. neki članovi) jedne klase ne treba da bude dostupan svim ostalim delovima programa, bez razlike, već samo nekoj “povlašćenoj” operaciji neke druge klase ili mnogim / svim operacijama te druge klase
- ❖ U osnovi, takvi članovi morali bi biti javni, kako bi bili dostupni toj drugoj klasi, ali su oni onda dostupni i ostalim delovima programa kojima je ta klasa dostupna, bez izuzetka; na taj način može biti kompromitovana enkapsulacija posmatrane klase
- ❖ Jezik C++ pruža delimičnu podršku za rešavanje ovakvih situacija u vidu tzv. *prijateljskih funkcija* (*friend function*) ili *prijateljskih klasa* (*friend class*):
 - prijateljska funkcija *f* nekoj klasi *X* je funkcija koja nije članica te klase *X*, već je članica neke druge klase ili je globalna (nije članica nijedne klase), a ima pravo pristupa do privatnih i zaštićenih članova klase *X*
 - prijateljska klasa *F* nekoj klasi *X* je klasa čije sve funkcije članice imaju pravo pristupa do privatnih i zaštićenih članova klase *X*
- ❖ Prijateljske funkcije ili klase klasi *X* specifikuju se navođenjem njihovih deklaracija u bilo kom delu definicije klase *X* (svejedno u kom delu, tj. iza bilo kog specifikatora *public*, *protected*, *private*)

Enkapsulacija

- ❖ Na primer, želimo da samo operacija *configure* klase *Configurator* može da kreira objekte klase *Controller*:

```
class Controller {  
public:  
    ...  
protected:  
    friend void Configurator::configure();  
    Controller (Reader*, Translator*);  
    ...  
private:  
    ...  
};  
void Configurator::configure () {  
    ...  
    ...new Controller(...)...  
    ...  
}
```

Funkcija *configure* članica klase *Configurator* je prijatelj klasi *Controller*, pa ima pristup do njenih privatnih i zaštićenih članova

Nijedan konstruktor nije javan, pa se van ove klase ne mogu kreirati objekti te klase (osim u izvedenim klasama, jer je njima konstruktor dostupan, pošto je zaštićen)

- ❖ Ako je cela klasa prijatelj nekoj drugoj klasi, onda su sve njene funkcije članice prijatelji toj drugoj klasi:

```
friend class Configurator;
```

- ❖ Za “prijateljstvo” važi sledeće:

- ono se ne može “preoteti”, jer ne može bilo ko da sebe proglaši prijateljem neke klase, pošto bi to narušilo enkapsulaciju te klase; “prijateljstvo” se *odobrava*: prijatelji se deklarišu u samoj klasi koja im odobrava pristup
- ono se ne nasleđuje: ako je klasa *B* prijatelj klasi *X*, a klasa *D* nasleđuje klasu *B*, klasa *D* nije implicitno prijatelj klasi *X*
- ono nije ni tranzitivna relacija: ako je klasa *B* prijatelj klasi *A*, a klasa *C* prijatelj klasi *B*, klasa *C* nije implicitno prijatelj klasi *A*

Enkapsulacija

- ❖ U mnogim slučajevima postoji potreba za čuvanje informacija (podataka) koji nisu svojstva svakog pojedinačnog objekta, već cele klase, odnosno zajednički su za sve objekte te klase
- ❖ Na proceduralnom jeziku, poput jezika C, ovakve podatke moramo definisati kao globalno dostupne (po oblasti važenja), što narušava enkapsulaciju, jer su oni onda dostupni svim delovima programa
- ❖ U OOP i na jeziku C++, kao i na mnogim drugim jezicima, na raspolaganju su *statički podaci članovi* (*static data members*): postoji samo po jedna instanca za svaki definisan statički podatak član, on je jedna instanca, deljena između svih objekata te klase
- ❖ Na primer, želimo da brojimo koliko je objekata kalse *Clock* ukupno kreirano - to je informacija bitna za celu klasu:

```
class Clock {  
public:  
    Clock ();  
    ...  
  
private:  
    static int count;           ← Statički podatak član  
    ...  
};  
  
void Clock::Clock () {  
    ...  
    count++;                  ← Pristup statičkom podatku članu ne zahteva objekat kome pripada  
}  
  
int Clock::count = 0;          ← Statički podatak član mora se definisati i inicijalizovati
```

- ❖ Pristup statičkom podatku članu ne zahteva objekat (kao levi operand operatora `.`) ili pokazivač na objekat (kao levi operand operatora `->`), jer on pripada klasi, a ne pojedinačnom objektu; taj objekat/pokazivač se ipak može i zadati

Enkapsulacija

- ❖ Ako je statički podatak član instance neke klase, njegova inicijalizacija zahteva poziv odgovarajućeg konstruktora, a kod za taj poziv se izvršava u vreme izvršavanja programa i prevodilac treba negde da ga generiše; jedino što se od prevodioca zahteva i garantuje jeste to da se ta inicijalizacija sigurno vrši pre bilo kog pristupa tom objektu ili poziva funkcije članice te klase koji se nalazi u istom fajlu u kom je taj statički podatak član definisan; ovo ne mora biti *pre* početka izvršavanja funkcije *main*
- ❖ Zbog toga je korišćenje statičkih podataka članova kao instanci klasa nepouzdano, jer ne moraju obavezno biti propisno inicijalizovani pre svakog korišćenja; zato je umesto njih bolje koristiti lokalne statičke objekte (detalji kasnije)
- ❖ Statički podaci članovi klase imaju isti životni vek i skladište se na isti način u memoriji kao i globalni statički objekti, ali je korišćenje statičkih podataka članova bolje u mnogim slučajevima, jer je statički podatak član:
 - deo klase kao logičke celine, logički je “upakovani” u nju, pa je jasnija njegova upotreba i namena - program je čitljiviji i lakši za razumevanje
 - u oblasti važenja klase, a nije globalan, pa se ne sukobljava po imenu (*name clashing*) sa ostalim globalnim imenima (može da se zove isto)
 - član klase, kao i svaki drugi, pa se može (i po pravilu treba) enkapsulirati: on može da bude zaštićen ili privatan, pa tako i nedostupan ostalim delovima programa (osim izvedenim klasama, ako je zaštićen)
- ❖ Zbog toga, neki noviji jezici (npr. Java) i ne omogućavaju globalne objekte (tačnije reference na njih), a umesto njih podržavaju statičke podatke članove: svaka referenca na objekat mora biti članica neke klase (statička ili nestatička)

Enkapsulacija

- ❖ Slično, postoje potrebe za usluge koje se ne traže od svakog pojedinačnog objekta, već od cele klase, odnosno predstavljaju uslugu te klase
- ❖ U OOP i na jeziku C++, kao i na mnogim drugim jezicima, na raspolaganju su *statičke operacije*, tj. *funkcije članice (static member functions)*
- ❖ Za prethodni primer, usluga dobijanja informacije koliko je objekata klase *Clock* ukupno napravljeno jeste usluga cele te klase:

```
class Clock {  
public:  
    Clock ();  
    static int getCount ();  
    ...  
  
private:  
    static int count;  
    ...  
};  
  
int Clock::getCount () {  
    return count;  
}
```

Statička funkcija članica

- ❖ Statička funkcija članica nema pokazivač *this*, pa ne može pristupati nestatičkim članovima svoje klase bez eksplicitnog navođenja objekta kome ti članovi pripadaju

Enkapsulacija

- ❖ Statička funkcija članica može da se pozove bez navođenja objekta za koji se poziva (mada taj objekat može i da se navede):

```
int num = Clock::getCount();
```

- ❖ Statičke funkcije članice implementiraju se isto kao i globalne funkcije nečlanice, jer nemaju pokazivač *this*; zbog toga se umesto statičke funkcije članice može upotrebiti i globalna funkcija, ali je korišćenje statičkih funkcija članica bolje u mnogim slučajevima, jer je statička funkcija članica:

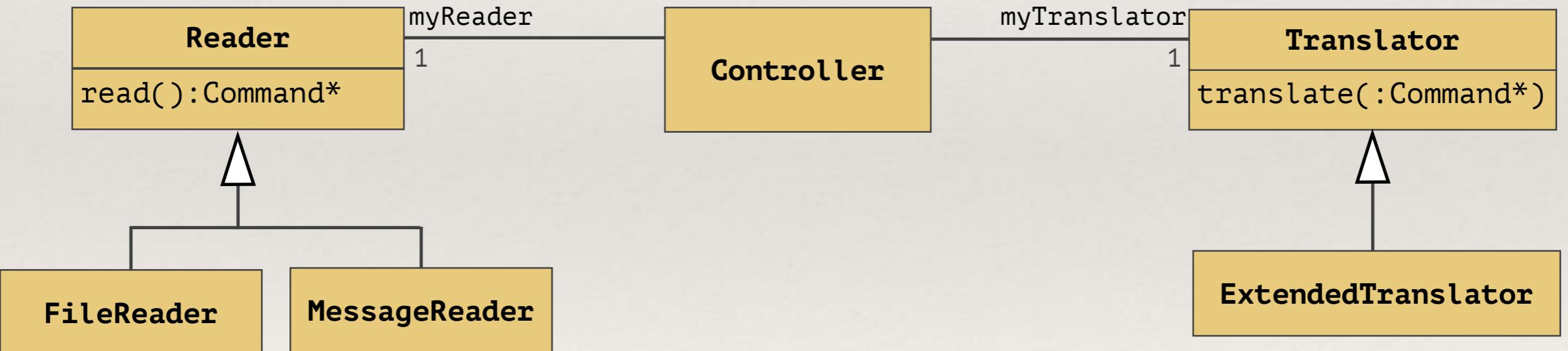
- deo klase kao logičke celine, logički je “upakovana” u nju, pa je jasnija njena upotreba i namena
- program je čitljiviji i lakši za razumevanje
- u oblasti važenja klase, a nije globalna, pa se ne sukobljava po imenu (*name clashing*) sa ostalim globalnim imenima (može da se zove isto)
- članica klase, kao i svaki drugi član, pa se može enkapsulirati: ona može da bude zaštićena ili privatna
- članica klase, pa ima pravo pristupa do privatnih i zaštićenih članova te klase (globalna bi morala da bude prijatelj toj klasi)
- ❖ Zbog toga, neki noviji jezici (npr. Java) i ne omogućavaju globalne operacije, a umesto njih podržavaju statičke operacije: svaka operacija mora biti članica neke klase (statička ili nestatička)

Hijerarhijska dekompozicija

- ❖ *Hijerarhijska dekompozicija (hierarchical decomposition)* je još jedan element objektne dekompozicije i podrazumeva kreiranje hijerarhija klasa povezanih relacijama *nasleđivanja (inheritance)*
- ❖ Relacija nasleđivanja označava da izvedena klasa *nasleđuje* sve osobine osnovne klase i to:
 - *semantiku* (značenje): svaka tvrdnja ili ograničenje koje važi za instance osnovne klase, važi i za instance izvedene klase (obrnuto ne mora)
 - *intefejs*: sa objektima izvedene klase može se raditi sve što i sa objektima osnovne klase (obrnuto ne mora)
 - *svojstva i ponašanje*: objekti izvedene klase poseduju i strukturu i ponašanje definisano u implementaciji osnovne klase, s tim što ih mogu redefinisati i proširiti
- ❖ Ovakve relacije mogu se ravnopravno otkrivati ili osmišljavati u oba smera:
 - *specijalizacija (specialization)*: izvedena klasa predstavlja konkretizaciju, posebnu ili pojedinačnu potkategoriju (podskup) instanci osnovne klase koje imaju neke specifičnosti, redefinišu, variraju i/ili specijalizuju ponašanje
 - *generalizacija (generalization)*: osnovna klasa je apstrakcija, generalizacija, uopštenje više izvedenih klasa, unija njihovih instanci i generalizuje (uopštava) njihove interfejse i prikuplja zajednička svojstva
- ❖ Prema tome, ista relacija između dve klase može se posmatrati u oba smera, pa se na jeziku UML ona i naziva relacija *generalizacije/specijalizacije*

Hijerarhijska dekompozicija

- ❖ Specijalizacijom se uvode izvedene klase koje imaju neke specifičnosti, proširuju, redefinišu, variraju i/ili specijalizuju ponašanje
- ❖ Na primer, u već postojeću konstrukciju primera sa interpretacijom komandi, mogu se uvesti specijalizacije klase *Reader* i/ili *Translator*. Ove klase mogu redefinisati polimorfne operacije interfejsa osnovnih klasa *read* i *translate*, čime se postiže *promena ponašanja sistema bez izmene ostalih postojećih delova*



Na jeziku UML sve operacije su podrazumevano polimorfne

Hijerarhijska dekompozicija

- ❖ Ovo je jedan od osnovnih doprinosa OO programiranja uopšte, a polimorfizma posebno, jer se *izmene* ponašanja softvera mogu postizati proširivanjem tj. dodavanjem (specijalizacija i redefinisanih metoda), a ne *izmenama* postojećih delova softvera; izmene po pravilu nose veći rizik od "lomljivosti" softvera i domino efekta
- ❖ Ovaj princip se ponekad naziva i *princip otvoreno/zatvoreno (open-closed principle)*: softverski entitet (klasa, modul) treba da bude *zatvoren za promene*, ali *otvoren za proširenja*
- ❖ Za klasu, to znači sledeće:
 - implementacija ponašanja klase treba da bude enkapsulirana i nedostupna za izmene: ako je potrebno promeniti nešto, to ne treba da utiče na implementaciju klase;
 - zahtevana promena ponašanja može da se postigne izvođenjem klasa i redefinisanjem ponašanja, odnosno proširenjem klase
- ❖ Naravno, sve ovo ima svoja ograničenja i odnosi se samo na strateške promene i proširenja, one koja se mogu predvideti: nijedan softver ne može biti potpuno zatvoren za promene, posebno one nepredviđene

Hijerarhijska dekompozicija

- ❖ Zbog svega ovoga se pri projektovanju klasa po pravilu računa na polimorfizam i operacije podrazumevano treba da budu polimorfne, što i jesu u praktično svim OO jezicima novijim od jezika C++
- ❖ Neki dinamički OO jezici (npr. JavaScript) omogućavaju redefinisanje operacija na nivou svakog pojedinačnog objekta, a ne samo cele klase, kako je uobičajeno u statički orijentisanim jezicima
- ❖ Na jeziku C++, polimorfnu operaciju treba označiti posebno, ključnom reči *virtual*
- ❖ U izvedenim klasama reč *virtual* ne mora (ali može) da se piše: funkcija sa identičnim potpisom kao i virtuelna funkcija članica osnovne klase je takođe virtuelna
- ❖ Da bi se poboljšala čitljivost i smanjila mogućnost greške (zbog nepoklapanja potpisa), iza redefinisane virtuelne funkcije može se pisati reč *override*: ona označava da je ovo redefinisana virtuelna funkcija; ukoliko potpis funkcije nije identičan, prevodilac će generisati grešku:

```
class Reader {  
public:  
    virtual Command* read ();  
    ...  
};  
  
class FileReader : public Reader {  
public:  
    virtual Command* read () override;  
    ...  
};
```

Reč *override* je identifikator sa posebnim značenjem samo na ovim mestima, a nije rezervisana ključna reč (može se koristiti kao identifikator na drugim mestima)

Hijerarhijska dekompozicija

- ❖ Samo u izuzetnim situacijama i sa posebnim razlogom, neka operacija ne treba da bude polimorfna, tj. potrebno je sprečiti njeno redefinisanje; takve operacije nazivaju se u mnogim jezicima *završnim (final)* i mogu se tako označiti
- ❖ Primer su operacije koje fiksiraju neki postupak (algoritam), tako da se on ne može promeniti (jer bi to moglo da uzrokuje propuste), ali dozvoljavaju da neke korake tog postupka izvedene klase definišu ili redefinišu (“kukice”)
- ❖ I na jeziku C++ virtualna funkcija može da se označi kao *final*; tada se ona ne može redefinisati u izvedenim klasama (prevodilac će prijaviti grešku u suprotnom):

```
class Controller {  
public:  
    virtual void main () final;  
    ...  
};
```

Reč *final* je identifikator sa posebnim značenjem samo na ovim mestima, a nije rezervisana ključna reč (može se koristiti kao identifikator na drugim mestima)

- ❖ I klasa može biti *završna*, čime se sprečava njena dalja specijalizacija:

```
class ExtendedTranslator final : public Translator {  
    ...  
};
```

Hijerarhijska dekompozicija

- ❖ Generalizacijom se uvode osnovne klase koje uopštavaju, apstrahuju, grupišu zajednička svojstva i ponašanje posebnih klasa
- ❖ Sama činjenica da neke klase imaju zajednička svojstva ili operacije može, ali ne mora biti dobar razlog za uvođenje njihove generalizacije
- ❖ Osnovni motiv za generalizaciju jeste to što neka druga apstrakcija ili drugi deo softvera (klijent) ima potrebu da instance datih različitih pojedinačnih klasa *posmatra i koristi* na isti način, kroz isti uopšteni *interfejs*, ne praveći razliku između njih
- ❖ Na primer, u programu za crtanje dijagrama, na *crtežu* (*Drawing*) se nalaze figure različitih vrsta: pravougaonici (*Rectangle*), poligonalne linije (*Polyline*), krugovi (*Circle*) itd. Kako ih iscrtati?

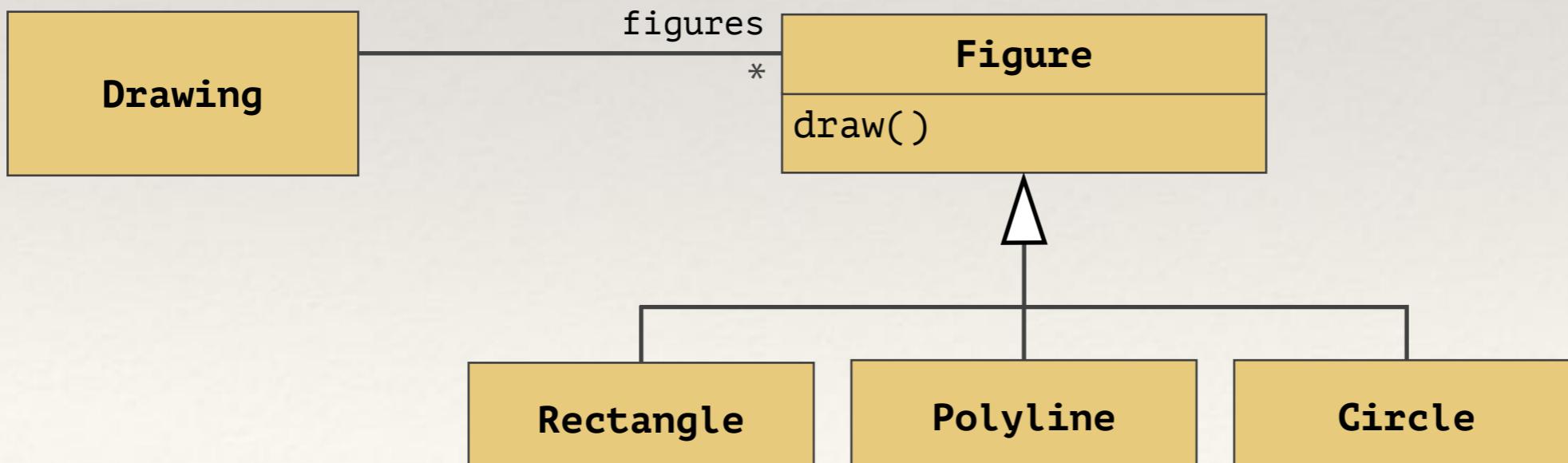
```
class Rectangle {  
public:  
    void draw (Viewport*);  
    ...  
};  
  
class Polyline {  
public:  
    void draw (Viewport*);  
    ...  
};  
  
class Circle {  
public:  
    void draw (Viewport*);  
    ...  
};  
  
void Drawing::draw (Viewport* vp)  
{  
    // How to draw figures  
    // of different kind?  
}  
...
```

Hijerarhijska dekompozicija

- ❖ U nekom proceduralnom jeziku bismo se oslonili na već pokazano, klasično rešenje:

```
switch (fig->kind) {  
    case rectangle: drawRectangle(fig, vp);  
    case polyline: drawPolyline(fig, vp);  
    case circle: drawCircle(fig, vp);  
    ...  
}
```

- ❖ Pošto je *Crtež* (*Drawing*) klijent koji različite figure posmatra na isti način, jer sve želi da ih *nacrtá* (*draw*), potrebna nam je generalizacija različitih vrsta figura
- ❖ Ova generalizacija onda grupiše i zajedničke osobine i zajednički interfejs posebnih klasa:
 - činjenicu da se mogu smeštati na crtež
 - uslugu da se iscrtaju

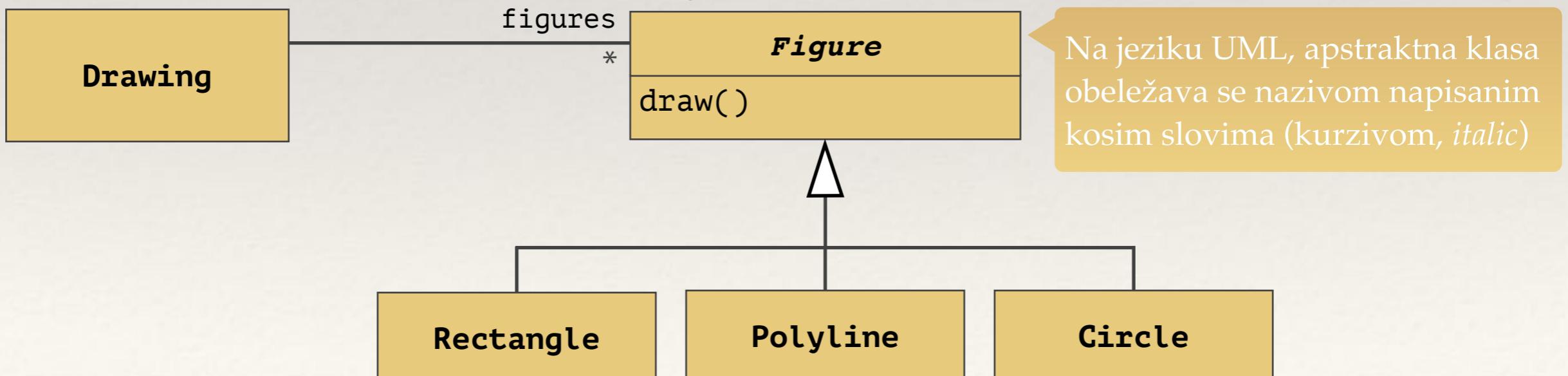


Hijerarhijska dekompozicija

- ❖ Jedna od osnovnih tendencija u OO programiranju jeste upravo generalizacija: apstrahovanje zajedničkih svojstava i formiranje uopštenih interfejsa za klijente, kako bi oni bili što manje zavisni od specifičnosti pojedinačnih slučajeva
- ❖ Specifičnosti se onda sakrivaju iza polimorfnih operacija i ugrađuju u njihove različite implementacije (metode) u izvedenim klasama
- ❖ Na ovaj način se sprege između delova softvera čine labavijim, jednostavnijim za kontrolu, a klijenti čine nezavisnijim i time fleksibilnijim (“*the less you know, the better*”)
- ❖ Ovo se ponekad naziva *principom inverzije zavisnosti* (*dependency inversion principle*):
 - u proceduralnom programiranju, u algoritamskoj dekompoziciji, potprogram (apstrakcija) na višem nivou i krupnije granularnosti zavisi od potprograma (apstrakcije) na nižem nivou, jer je poziva kao deo svoje implementacije
 - u OOP, tendencija je da jedna apstrakcija, klijent (*Drawing*) zavisi od druge uopštene apstrakcije na višem nivou (*Figure*), a da specijalizacije (*Rectangle*, *Circle*, ...), kao pojedinačni slučajevi, zavise od opštije apstrakcije (*Figure*) - obrnuta zavisnost
- ❖ Generalizacijom i korišćenjem polimorfizma se drastično smanjuje količina uslovnih grananja na osnovu tipa objekta kojim se rukuje

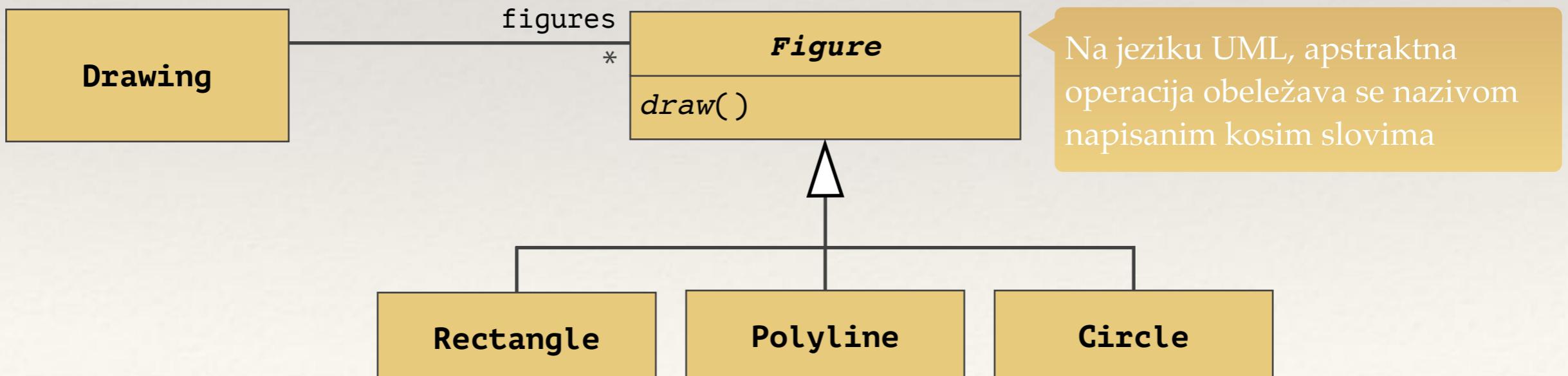
Hijerarhijska dekompozicija

- ❖ Klasa *Figure* neće imati svoje direktne instance, već samo indirektne instance (koje su direktne instance izvedenih klasa)
- ❖ Ovakva klasa se naziva *apstraktnom klasom* (*abstract class*) ili apstraktnom generalizacijom
- ❖ Neki jezici omogućavaju da se apstraktna klasa posebno označi
- ❖ Na jeziku C++ nema posebne oznake za apstraktnu klasu, ali se pravljenje objekata te klase (direktnih instanci) može sprečiti deklarisanjem *svih* postojećih konstruktora zaštićenim (*protected*). Zašto baš zaštićenim?
 - ne mogu se kreirati nezavisni objekti te klase, jer konstruktor nije javan (*public*), pa bi prevodilac generisao grešku pri svakom pokušaju kreiranja tog objekta van te klase
 - kada se pravi objekat izvedene klase, poziva se konstruktor te izvedene klase; ali svaki konstruktor izvedene klase obavezno poziva neki konstruktor osnovne klase; ako bi on bio privatan (*private*), ne bi bio dostupan ni u izvedenim klasama, pa se data klasa ne bi mogla nasleđivati



Hijerarhijska dekompozicija

- ❖ Kako treba da izgleda implementacija operacije (metoda) *draw* klase *Figure*? Može li se nacrtati figura koja “nije ništa posebno” (a takva i ne postoji, jer je ova klasa apstraktna)?
- ❖ U nekim slučajevima, polimorfna operacija apstraktne klase se može implementirati u toj klasi:
 - kada ova operacija ima neko logično podrazumevano ponašanje, makar i prazno
 - ona je pomoćna (*helper*) metoda koja se može, ali ne mora redefinisati; recimo, opcionalno proširenje podrazumevano praznog koraka neke složenije metode osnovne klase
- ❖ Ako to nije slučaj, operacija je *apstraktna (abstract operation)*: predstavlja samo *specifikaciju usluge* koja se može tražiti od objekata, ali ne daje *implementaciju* te usluge (metodu); tu implementaciju daće izvedene klase



Hijerarhijska dekompozicija

- ❖ Na jeziku C++ apstraktne operacije nazivaju se *čisto virtuelnim funkcijama* (*pure virtual functions*) i označavaju specifikatorom `= 0` u potpisu funkcije; takva funkcija neće imati definiciju:

```
class Figure {  
public:  
    virtual void draw (Viewport*) = 0;  
    ...  
};  
  
class Circle : public Figure {  
public:  
    virtual void draw (Viewport*) override;  
    ...  
};  
  
void Circle::draw (Viewport* vp) {  
    ...  
}
```

Čisto virtuelna funkcija (*pure virtual function*)

Izvedena klasa definiše metodu za ovu operaciju

- ❖ Na jeziku C++, ako klasa ima bar jednu čisto virtuelnu funkciju, onda je ona za prevodilac apstraktna - neće dozvoliti kreiranje direktnih instanci te klase (bez obzira na dostupnost konstruktora)
- ❖ Konceptualno, obrnuto ne važi: klasa može biti apstraktna i ako nema nijednu apstraktну operaciju, mada prevodilac za jezik C++ ne tretira posebno tu klasu kao apstraktну (u užem smislu, prema pravilima jezika)
- ❖ Izvedena klasa može, a ne mora da definiše metodu za apstraktnu operaciju koju nasleđuje; ako je ne definiše, ta operacija i dalje ostaje apstraktna, a klasa je takođe apstraktna

Hijerarhijska dekompozicija

- ❖ Klase imaju interpretaciju i u skupovnoj matematičkoj logici:
 - klasa je skup, objekat je element; objekat x je instanca klase $X \Leftrightarrow x$ je element skupa X
 - instanca izvedene klase je uvek (indirektno) i instanca osnovne klase: ako je objekat x instanca izvedene klase D (x je element D), iz toga sledi da je x i instanca osnovne klase B (x je element B)
 - prema tome: osnovna klasa je nadskup, izvedena klasa je podskup
- ❖ Iz svega toga sledi jedan od fundamentalnih principa objektnog programiranja, *princip supstitucije* (*Liskov substitution principle*, Barbara Liskov, 1994): instance izvedene klase D mogu se pojaviti i upotrebiti gde god i kad god se očekuju instance osnovne klase B - instance izvedene klase mogu biti supstituti (zamene) za instance osnovne klase, bez ugrožavanja bilo kog željenog ponašanja programa
- ❖ Ovaj princip je posledica semantike nasleđivanja: kako objekti izvedene klase nasleđuju sve osobine objekata osnovne klase, i za njih važe sve tvrdnje koje važe za objekte osnovne klase, sa njima se može raditi sve što i sa objektima osnovne klase; zapravo, oni su zato instance te osnovne klase

Hijerarhijska dekompozicija

- ❖ Svi savremeni OO jezici, pa i C++, podržavaju ovaj princip sledećim pravilom *implicitne konverzije*: pokazivač/referenca na izvedenu klasu može se konvertovati (implicitno, bez eksplisitnog zahteva) u pokazivač/referencu na osnovnu klasu - tzv. "kalupljenje nagore" (*upcast*):

```
DerivedClass* → BaseClass*
```

```
DerivedClass& → BaseClass&
```

- ❖ Upravo ova konverzija omogućava da se objektima konkretnih, izvedenih klasa pristupa kaoinstancama osnovnih, generalizovanih klasa
- ❖ Informacija o konkretnom tipu objekta (klasi čija je on direktna instanca) treba da bude što manje bitna i poznata ostatku softvera; ta informacija se može zanemariti odmah nakon kreiranja objekta:

```
Figure* fig = new Circle(...);
```

- ❖ Suprotna konverzija, nadole (*downcast*), nije uvek bezbedna, jer objekat osnovne klase ne mora biti i instanca neke izvedene klase; pošto prevodilac ne može da proveri tu činjenicu, ovakva konverzija ne može se raditi implicitno:

```
Figure* fig = new Circle(...);  
Circle* crc = fig;
```

Greška u prevođenju: konverzija *Figure** u *Circle** ne može se raditi implicitno

ali može eksplisitno:

```
Circle* crc = (Circle*)fig;
```

- ❖ Prevodilac generiše kod za pristup objektu te izvedene preko tog pokazivača, bez ikakvih dodatnih provera. Na ovaj način, programer preuzima odgovornost da se iza pokazivača na osnovnu klasu zaista krije objekat tražene izvedene klase. Ako ovo nije zadovoljeno, program će se ponašati potpuno nepredvidivo u vreme izvršavanja (nepredvidive posledice: greška u logici, poremećaj podataka ili izuzetak na nivou hardvera ili operativnog sistema zbog neovlašćenog pristupa delu memorije)

Hijerarhijska dekompozicija

- ❖ Za *polimorfne klase (polymorphic class)*, a to su klase sa bar jednom (makar i nasleđenom) virtuelnom funkcijom, odnosno klase čiji objekti imaju u sebi VTP, ovakav *downcast* se može izvršiti i bezbednije, *dinamičkom konverzijom (dynamic cast)*:

```
Circle* crc = dynamic_cast<Circle*>(fig);
```

- ❖ Ukoliko se iza pokazivača kojim rezultuje izraz unutar zagrada ovog operatora krije zaista objekat koji jeste (direktna ili indirektna) instanca tražene ciljne klase, rezultat će biti validan pokazivač na objekat te klase; u suprotnom, rezultat će biti nula pokazivač (*null*) (ako je odredišni tip referenca, u ovom slučaju biće bačen izuzetak)
- ❖ Upotreba dinamičke konverzije u ovakve svrhe je opravdana u situacijama kada se zna ili se očekuje da je iza pokazivača instanca potrebne specijalizacije; međutim, pogrešna upotreba može da signalizira propuštanje neke apstrakcije i polimorfizma:

```
Circle* crc = dynamic_cast<Circle*>(fig);
if (crc) drawCircle(crc);
```

Očigledno polimorfizam!

```
Rectangle* rct = dynamic_cast<Rectangle*>(fig);
if (rct) drawRectangle(rct);
...
```

Hijerarhijska dekompozicija

❖ Posmatrajmo sledeći zahtev:

- realizujemo apstraktnu strukturu podataka *lista* (*list*), koja će imati operacije smeštanja novog elementa na proizvoljnu poziciju u listi, iza nekog drugog elementa u toj listi, i uzimanja elementa sa proizvoljnog mesta u listi
- želimo da obe operacije budu kompleksnosti $O(1)$, pa ćemo koristiti dvostruko ulančanu listu
- ne želimo da za strukture pokazivača koje koristimo dinamički alociramo potreban prostor, već želimo da te pokazivače ugradimo u same objekte koji će biti ulančavani, koji god da su; zato nećemo koristiti šablone, pa ni one bibliotečne
- pravimo *spisak zadataka* (*task list*), u koju ćemo smeštati *zadatke* (*task*)
- *zadatak* ima i mnoge druge osobine, ponašanje, koristi se u mnogim drugim, različitim kontekstima aplikacije itd.

```
class ListElem {
public:
    void insert (ListElem* prev, ListElem* next);

protected:
    ListElem () { prev = next = nullptr; };

private:
    friend class List;
    ListElem *prev, *next;
};

void ListElem::insert (ListElem* p, ListElem* n) {
    if (p) p->next = this;
    if (n) n->prev = this;
    this->prev = p;
    this->next = n;
}

class List {
public:
    List () { head = tail = nullptr; }

    void addAtHead (ListElem* e);
    void addAtTail (ListElem* e);
    void addAfter (ListElem* e, ListElem* prev);

private:
    ListElem *head, *tail;
};

void List::addAfter (ListElem* e, ListElem* p) {
    if (!e) return;
    if (!p) insertAtHead(e);
    else
        if (!p->next) insertAtTail(e);
        else e->insert(p,p->next);
}
```

Hijerarhijska dekompozicija

- ❖ Da bi *zadatak* (*Task*) bio umetan u listu, mora da “umeša” (*mix in*) strukturu i ponašanje klase *ListElem*:

```
class Task : public ListElem {...};
```

- ❖ Dakle, klasa *ListElem* jeste apstraktna klasa, koja ima i strukturu i definisane (barem neke) metode i koja predstavlja potreban *interfejs* koji neka druga klasa treba da zadovolji, da bi učestvovala u nekom mehanizmu (ovde, da bi bila umetana u listu)

- ❖ Ovakve klase nazivaju se *mixin* klase

- ❖ Pored ovog interfejsa, klasa *Task* može imati druge takve interfejse za potrebe drugih konteksta (mehanizama, scenarija) u kojima učestvuje, pa će biti *izvedena* iz više takvih klasa:

```
class Task : public ListElem, public Runnable, public Drawable {...};
```

- ❖ U potpuno agresivnom i doslednom sprovođenju principa apstrakcije, mogu se praviti ovakvi interfejsi za *svaki* pojedinačan kontekst u kome klasa učestvuje, odnosno za svaku vrstu klijenta te klase - svaki takav interfejs biće predstavljen posebnom apstraktnom klasom

- ❖ Ovakav pristup naziva se *princip segregacije interfejsa* (*interface segregation*)

- ❖ Klasa koja *zadovoljava* (*implementira*, *implement*) sve te interfejse je onda izvedena iz svih tih klasa

- ❖ U suprotnom, ako se ne bi radilo tako, sva svojstva i ponašanje potrebni za različite kontekste bili bi ugrađeni neposrednu u tu klasu i pomešani u njoj, pa bi ona mogla da postane glomazna i teža za razumevanje i održavanje

Hijerarhijska dekompozicija

- ❖ Ovakav pristup dosta se koristi i u složenim programima na jeziku C (npr. implementaciji operativnih sistema) na sledeći način:

```
struct ListElem {...};  
struct Runnable {...};  
struct Drawable {...};  
  
struct Task {  
    ListElem listElem;  
    Runnable runnable;  
    Drawable drawable;  
    ...  
};
```

- ❖ Dakle, instanca strukture *Task* u sebi sadrži podstrukture koje predstavljaju odgovarajuće interfejse. Kada instancu strukture *Task* treba koristiti u nekom od ovih konteksta, dostavlja se pokazivač na odgovarajuću ugrađenu podstrukturu:

```
Task* aTask = ...;  
  
List* taskList = ...;  
  
addAtTail(taskList, &aTask->listElem);
```

- ❖ Sa idejom da podrži ovakve načine korišćenja, ali i da implementacija u njima bude podjednaka (i podjednako efikasna) kao ova na jeziku C, jezik C++ zapravo ima koncept *izvedenih* klasa (*derived class*), sa sledećim značenjem:

- klasa može biti *izvedena* iz više osnovnih klasa koje su navedene u zaglavlju definicije klase, iza dvotačke
- svaki objekat izvedene klase u sebi sadrži po jedan podobjekat svake od tih osnovnih klasa
- specifikator pristupa (*public*, *protected*, *private*) označava dostupnost dog podobjekta, na isti način kao i za članove te klase

Hijerarhijska dekompozicija

- ❖ Na taj način, ako je osnovna klasa u zaglavlju definicije izvedene klase označena kao *javna (public)* osnovna klasa, ovaj podobjekat osnovne klase dostupan je (implicitnom) konverzijom pokazivača (ili reference) na tu izvedenu klasu u pokazivač (ili referencu) na tu osnovnu klasu na svim mestima gde su dostupne i te klase:

```
class ListElem {...};  
  
class Task : public ListElem {...};  
  
Task* aTask = ...;  
  
List* taskList = ...;  
  
taskList->addAtTail(aTask);
```

Implicitna konverzija pokazivača *aTask* tipa *Task** u pokazivač tipa *ListElem** na javnu osnovnu klasu

- ❖ Osim toga, svi javni članovi javne osnovne klase jesu i javni članovi izvedene klase, pa se može raditi npr:

```
aTask->insert(...)
```

- ❖ Zaštićeni članovi osnovne klase dostupni su u izvedenoj klasi, ali ne i van nje i ostaju zaštićeni i dalje; privatni članovi osnovne klase nisu dostupni u izvedenoj klasi
- ❖ Dakle, važi pravilo supstitucije i sve što se može uraditi sa objektom osnovne klase može se uraditi i sa objektom izvedene klase - *javno izvođenje klasa* na jeziku C++ jeste jezički koncept koji realizuje *nasleđivanje* kao objektni koncept

Hijerarhijska dekompozicija

- ❖ Ako pak osnovna klasa nije javna, već *zaštićena* (*protected*) ili *privatna* (*private*), objekat izvedene klase i dalje će u sebi imati ugrađen podobjekat te osnovne klase, ali on neće biti dostupan van te klase, osim u izvedenim klasama ako je zaštićen:
 - implicitna konverzija pokazivača (ili reference) na objekat izvedene klase u pokazivač (ili referencu) na osnovnu klasu dozvoljena je samo u toj klasi (i u izvedenoj klasi, za zaštićeno izvođenje)
 - javni i zaštićeni članovi osnovne klase dostupni su samo u toj izvedenoj klasi, ali ne i van nje (osim u izvedenim klasama za zaštićeno izvođenje):

```
class Task : protected ListElem {...};  
Task* aTask = ...;  
List* taskList = ...;  
taskList->addAtTail(aTask);
```

Ovo više nije moguće van klase *Task* ili njene izvedene klase

- ❖ Ali unutar te izvedene klase (*Task*), može se vršiti ova implicitna konverzija, čime klasa zapravo nudi svoj odgovarajući interfejs potrebnom kontekstu:
- ❖ Prema tome, kod zaštićenog i privatnog izvođenja, ne važi pravilo supstitucije u celom programu i sa objektima izvedene klase ne može se uvek i svuda uraditi sve što i sa objektima osnovne klase, pa ovakvo izvođenje klasa na jeziku C++ *ne realizuje nasleđivanje* kao objektni koncept, već:

- predstavlja relaciju ugrađivanja objekta jedne klase u objekte druge klase
- može se koristiti za kontrolisanu i skrivenu ugradnju *mixin* klasa, uz otkrivanje tih interfejsa samo u određenim ograničenim delovima programa radi strožije enkapsulacije

Implicitna konverzija pokazivača *this* tipa *Task** u pokazivač tipa *ListElem** na zaštićenu (ili privatnu) osnovnu klasu dozvoljena je u toj izvedenoj klasi

Hijerarhijska dekompozicija

- ❖ Objekat izvedene klase u sebi sadrži podobjekat svake osnovne klase; i tako rekurzivno, ako ta osnovna klasa ima svoje osnovne klase, taj podobjekat imaće u sebi po jedan podobjekat svake od tih daljih osnovnih klasa itd. Kako je operativna memorija linearna, da bi se objekat klase izvedene iz više osnovnih klasa smestio u nju, ovi podobjekti se moraju poređati, i to po redosledu navođenja osnovnih klasa u definiciji izvedene klase:

```
class D : public B1, public B2 {...};
```

- ❖ Kada se pravi objekat izvedene klase (*D*), poziva se konstruktor te klase. Ali svaki konstruktor izvedene klase uvek poziva (pre izvršavanja svog tela) konstruktor osnovne klase; taj poziv obezbeđuje prevodilac u generisanom kodu za konstruktor izvedene klase. Analogno, konstruktor osnovne klase u sebi ima poziv konstruktora svoje osnovne klase itd.
- ❖ U slučaju višestrukog izvođenja, konstruktori osnovnih klasa pozivaju se po redosledu navođenja tih klasa u definiciji izvedene klase, bez obzira na to kako su ti pozivi navedeni:

```
D::D (...) : B1(...), B2(...) {  
    ...  
}
```

Redosled posiva je uvek *B1()* pa *B2()*,
čak i ako su ovde navedeni drugačije



Hijerarhijska dekompozicija

- Treba primetiti da konverzija pokazivača na izvedenu klasu u pokazivač na osnovnu klasu daje kao rezultat pokazivač iste vrednosti u slučaju jednostrukog izvođenja klase, ali može da daje promenjenu vrednost u slučaju višestrukog izvođenja; analogno važi i za konverziju nadole:

```
class D1 : public B2 {...};
```

```
class D2 : public B1, public B2 {...};
```

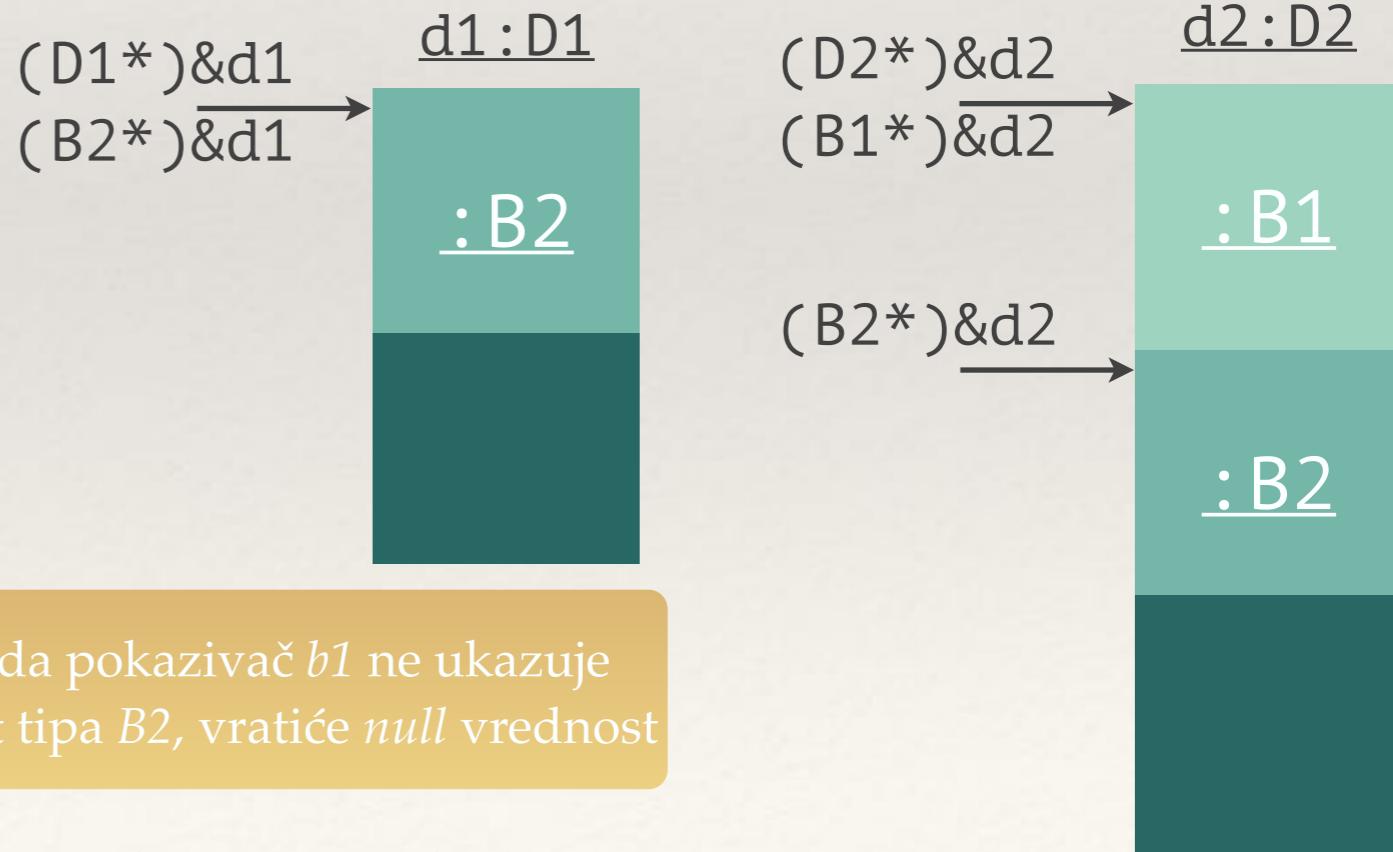
- Kao i uvek, eksplicitna statička konverzija nadole ne proverava činjenicu da li se iza pokazivača zaista krije objekat tražene klase, pa stoga nije bezbedna iz ugla prevodioca, osim ako se to ne obezbedi nekim drugim načinom, odnosno logikom programa

- Dinamička konverzija je bezbednija, jer dinamički proverava ovo, i vraća *null* vrednost ako objekat nije tražene klase. Ona se može koristiti za konverzije nadole, nagore, ili bočno:

```
D2* d = new D2;
```

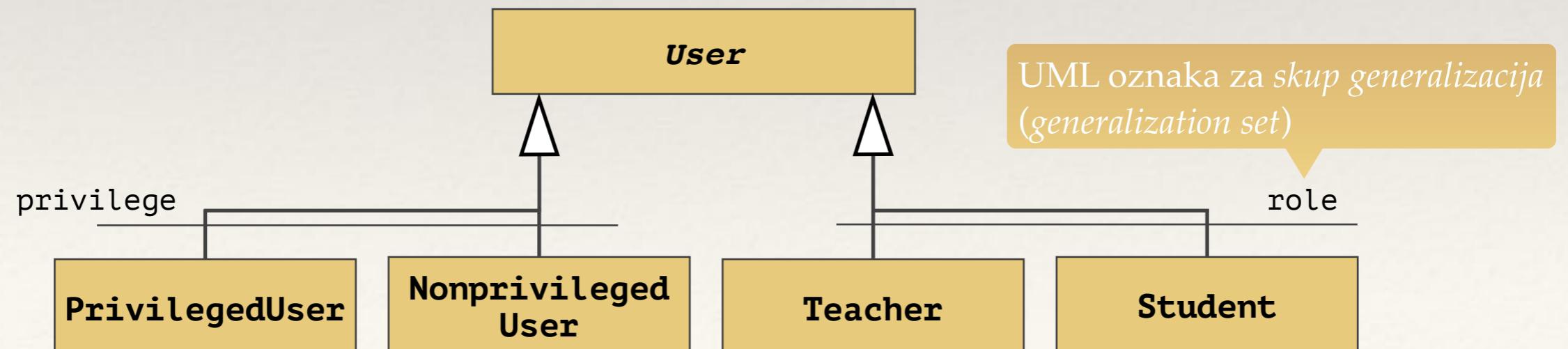
```
B1* b1 = d;
```

```
B2* b2 = dynamic_cast<B2*>(b1);
```



Hijerarhijska dekompozicija

- ❖ U skupovnoj logici, osnovna klasa (generalizacija) predstavlja nadskup skupa objekata njene izvedene klase (specijalizacije). Ali šta ako je osnovna klasa apstraktna?
- ❖ Pošto ona tada nema svoje direktnе instance, ne postoje elementi tog skupa koji nisu ujedno i elementi skupa predstavljenog nekom od izvedenih klasa. Prema tome, apstraktna klasa predstavlja *uniju* skupova predstavljenih izvedenim klasama
- ❖ A šta ako izvedene klase, posmatrane kao skupovi, imaju presek, odnosno zajedničke elemente?
- ❖ Ovo se obično dešava ako se generalizacije/specijalizacije prave po različitim, ortogonalnim kriterijumima
- ❖ Na primer, u nekom školskom sistemu, *korisnici (User)* mogu biti klasifikovani po pravima pristupa na *privilegovane (Privileged User)* i *neprivilegovane (NonprivilegedUser)*; sa druge strane, mogu se klasifikovati prema svojoj ulozi (*role*) na *nastavnike (Teacher)* i *učenike (Student)*
- ❖ Na jeziku UML, ovakve različite grupe relacija generalizacija/specijalizacija nazivaju se *skupovima generalizacija (generalization set)*
- ❖ Ako su izvedene klase kao skupovi disjunktni, tj. nemaju presek (zajedničke instance), skup generalizacija naziva se *isključiv (exclusive)*; ako je takva osnovna klasa apstraktna, ona predstavlja uniju disjunktnih podskupova - *particiju (partition)*



Hijerarhijska dekompozicija

- ❖ Međutim, izvedene klase iz različitih skupova generalizacija često imaju zajedničke instance, odnosno presek. Na primer, jedan nastavnik može biti privilegovan korisnik
- ❖ U nekim jezicima, kao što je UML, objekat može biti instanca više klasa (koje nisu u relaciji generalizacije/specijalizacije). Štaviše, objekat se može *dinamički reklasifikovati (reclassify)* tokom svog životnog veka: mogu mu se dodavati i oduzimati klase kojima pripada (time se dodaju ili oduzimaju sva svojstva tih klasa)
- ❖ U tradicionalnim, statički tipiziranim OO programskim jezicima, kakav je i C++, ovo nije podržano, pa objekat uvek mora biti direktna instanca jedne i samo jedne klase. Ta klasa se uvek mora odrediti u trenutku kreiranja tog objekta i objekat se ne može reklassifikovati tokom svog životnog veka
- ❖ Kako onda rešiti ovaku situaciju? Višestrukim izvođenjem nove klase koja predstavlja presek dve osnovne:

```
class User {...};  
  
class PrivilegedUser : public User {...};  
class NonprivilegedUser : public User {...};  
  
class Teacher : public User {...};  
class Student : public User {...};  
  
class PrivilegedTeacher : public PrivilegedUser, public Teacher {...};
```

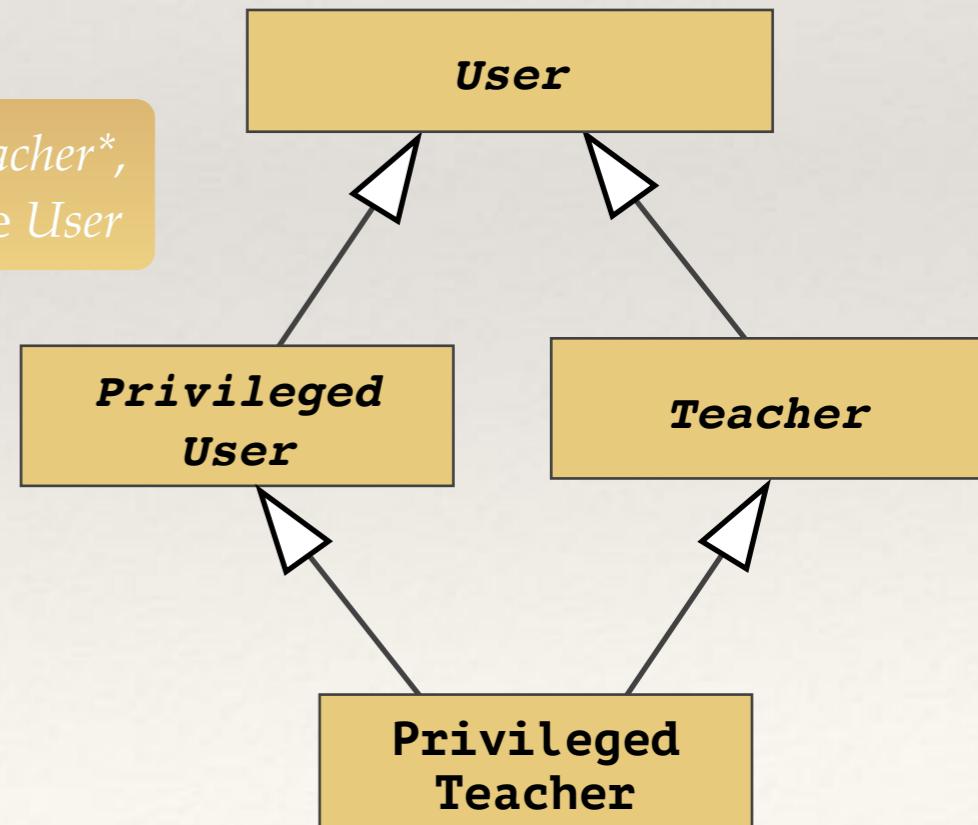
Hijerarhijska dekompozicija

- ❖ Međutim, kako C++ uvek implementira objekte izvedene klase tako da u njih ugrađuje po jedan objekat svake osnovne klase (i tako rekursivno), sledi:
 - objekat klase *PrivilegedTeacher* u sebi ima podobjekat klase *PrivilegedUser*, a ovaj u sebi podobjekat klase *User*
 - objekat klase *PrivilegedTeacher* u sebi ima podobjekat klase *Teacher*, a ovaj u sebi podobjekat klase *User*
- ❖ Tako će objekat klase *PrivilegedTeacher* u sebi imati dva podobjekta osnovne klase *User*, što znači dva kompleta svih svojstava, što nije poželjno. Ova pojava naziva se “problem dijamanta” (*diamond problem*), zbog grafa nasleđivanja oblika romba (liči na dijamant i često se tako naziva na engleskom)
- ❖ Ovim podobjektima može se pristupiti navođenjem kvalifikovanog imena (staze) i operadora ::

```
this->PrivilegedUser::username = ...;
```

```
this->Teacher::username = ...;
```

this je tipa PrivilegedTeacher,
a username je član klase User*

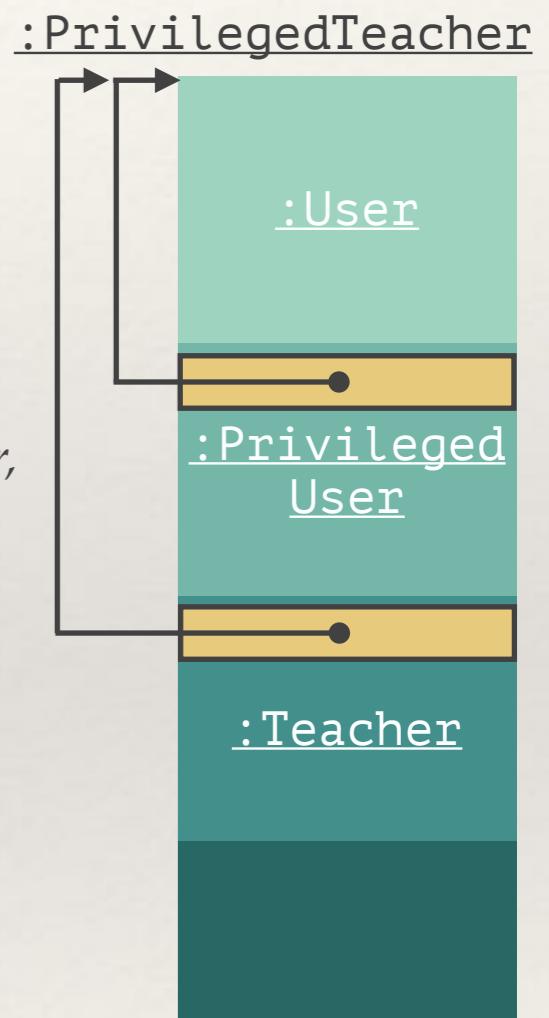


Hijerarhijska dekompozicija

- ❖ Da bi se rešio ovaj problem, osnovna klasa *User* treba da se deklariše kao tzv. *virtuelna osnovna klasa*:

```
class User {...};  
  
class PrivilegedUser : virtual public User {...};  
class NonprivilegedUser : virtual public User {...};  
  
class Teacher : virtual public User {...};  
class Student : virtual public User {...};  
  
class PrivilegedTeacher : public PrivilegedUser, public Teacher {...};
```

- ❖ Značenje virtuelne osnovne klase je sledeće: ako se objekat neke klase (*PrivilegedUser*, *Teacher*, ...) nađe kao podobjekat unutar objekta neke izvedene klase (*PrivilegedTeacher*), u njega ugrađen podobjekat virtuelne osnovne klase (*User*) će biti zajednički (deljen) za sve druge podobjekte unutar tog objekta koji takođe imaju tu osnovnu klasu kao virtuelnu
- ❖ Ovo se implementira na sledeći način: unutar objekta izvedene klase postoji pokazivač na podobjekat virtuelne osnovne klase, jer on ne mora uvek biti na unapred poznatom mestu unutar tog objekta (na njegovom početku), pošto se može nalaziti unutar nekog drugog objekta sa kojim se deli
- ❖ Zbog toga prevodilac ne može generisati kod za pristup članovima virtuelne osnovne klase statičkim vezivanjem (adresiranjem preko adresa, tj. pomeraja u odnosu na adresu početka objekta poznatim u vreme prevodenja), već dinamičkim vezivanjem, preko ovog pokazivača



Hijerarhijska dekompozicija

- ❖ Kod višestrukog izvođenja, stvari se dosta komplikuju:
 - Konstruktori osnovnih klasa se svakako pozivaju unutar konstruktora izvedenih klasa, ali kojim redom? - Po redosledu navođenja tih osnovnih klasa u definiciji izvedene klase; prema tome, zbog ugnezđenih poziva, redosled poziva konstruktora je uvek po dubini grafa, sleva nadesno (posmatrano po redosledu navođenja osnovnih klasa)
 - Međutim, konstruktori virtualnih osnovnih klasa se pozivaju pre konstruktora ostalih (nevirtualnih) klasa, opet po određenom redosledu, ali je to pravilo sada već teško za pamćenje i razumevanje programa, pa ako logika programa zavisi od tog redosleda, takav program postaje težak za razumevanje
 - Konverzije pokazivača imaju komplikovaniju implementaciju, rezultati tih konverzija mogu davati različite vrednosti pokazivača, ali je dinamička konverzija (pomoću *dynamic_cast*) uvek ispravna i bezbedna i može se raditi nagore, nadole i bočno po grafu izvođenja
- ❖ Ovakve situacije u kojima je potrebno raditi višestruko nasleđivanje klasa su u praksi ipak retke. Čak i ako se na njih nađe, mogu se rešiti nekim drugim načinom, npr. atributima ili vezama sa objektima različitih klasa
- ❖ Zbog toga mnogi drugi jezici ne podržavaju višestruko nasleđivanje / izvođenje
- ❖ Međutim, višestruko izvođenje jeste korisno kada se upotrebljava za implementaciju različitih interfejsa i *mixin* klase
- ❖ Kako bi podržali ovu važnu upotrebu, a ipak pojednostavili implementaciju i logiku jezika, mnogi drugi jezici podržavaju ograničen koncept *interfejsa*, koji može imati samo polimorfne operacije (ali ne i strukturu, attribute), sa metodama ili bez njih (apstraktne operacije)
- ❖ U takvim jezicima (npr. Java) klasa može *naslediti* (*proširiti, extend*) samo jednu osnovnu klasu (nasleđujući njenu strukturu i ugrađujući samo jedan podobjekat osnovne klase), ali može *implementirati* (*implement*) više interfejsa; pošto interfejsi tu imaju samo polimorfne operacije, implementacija se svodi na jednostavno dinamičko vezivanje preko VTP

Obrada izuzetaka

- ❖ Važno svojstvo robusnih složenih softverskih sistema jeste *tolerancija otkaza (fault tolerance)*: sposobnost da softver nastavi svoje izvršavanje u prisustvu *otkaza (faults)*
- ❖ Otkazi su odstupanja softvera ili njegovog okruženja od specifikovanog, predviđenog ili očekivanog ponašanja
- ❖ Softver koji nema mehanizme tolerancije otkaza će u slučaju otkaza ili u potpunosti prekinuti svoj rad (“pad”, *failure*) ili nastaviti izvršavanje sa nepredvidivim ili neželjenim ponašanjem
- ❖ Softver koji ima ugrađene mehanizme tolerancije otkaza će u slučaju otkaza nastaviti svoje izvršavanje uz zadovljavajuće i predviđeno ponašanje, eventualno uz degradirane neke funkcionalnosti i/ili performanse, ili preuzeti kontrolisane procedure za sopstveno gašenje i gašenje sistema kojim upravlja
- ❖ Jedan od osnovnih mehanizama u hardveru i softveru za toleranciju otkaza jeste mehanizam *obrade izuzetaka (exception handling)*
- ❖ *Izuzetak (exception)* je nastanak nenormalnih, neregularnih ili neočekivanih uslova koji zahtevaju poseban tretman, različit od onog predviđenog normalnim, regularnim tokom programa

Obrada izuzetaka

- ❖ Izuzetak može biti detektovan u *okruženju* programa
- ❖ Na primer, sledeće izuzetne situacije detektuje hardver (procesor), a obrađuje operativni sistem, podrazumevano tako što gasi program koji je izazavao tu situaciju:
 - nelegalan kod operacije, način adresiranja i slično: ako se radi o programu koji je preveden sa višeg programskog jezika, može da znači "skok" na adresu na kojoj se ne nalazi regularan mašinski kod, već slučajan sadržaj; na primer, skok u potprogram dinamičkim vezivanjem: preko "oštećenog" pokazivača na funkciju, ili poziv virtuelne funkcije oštećenog pokazivača na objekat ili oštećenog objekta
 - neregularan pristup memorijskoj lokaciji: adresiranje memorijske lokacije koja nije dozvoljena, ili pristup toj lokaciji koji nije dozvoljen (npr. upis u lokaciju za koju je dozvoljeno samo čitanje); slično, ovo može biti posledica oštećenih pokazivača na objekte
- ❖ Kako C/C++ ne proverava pokazivače, već prevodilac prosto generiše kod za pristup do objekta ili za poziv funkcije kao u regularnim uslovima, ponašanje programa u slučaju oštećenih pokazivača postaje nedefinisano i pre ili kasnije može da rezultuje ovakvim izuzecima
- ❖ Izuzetak može da detektuje i signalizira i operativni sistem, kada program zahteva neku njegovu uslugu tzv. *sistemskim pozivom*, npr. zahteva ulazno/izlaznu operaciju, alokaciju dela memorije i slično; u ovom slučaju, operativni sistem će vratiti grešku, a implementacija bibliotečnog potprograma u kom je sistemski poziv može da signalizira izuzetak
- ❖ Bibliotečni potprogrami ili konstrukti programskog jezika takođe mogu da signaliziraju izuzetak jer ne mogu da obave svoj zadatak iz određenog razloga
- ❖ Slično, i potprogrami ili moduli samog programa mogu da signaliziraju izuzetak jer iz nekog razloga ne mogu da obave zahtevanu operaciju kako je to predviđeno

Obrada izuzetaka

- ❖ Kako se mogu tretirati izuzeci u slučaju da na programskom jeziku nemamo posebnu podršku za njihovu obradu?
Svaki potprogram (koji može signalizirati izuzetak) može da vrati status svog izvršavanja kao povratnu vrednost:

```
enum DeviceStatus {ok, deviceFaulty, communicationFailed};

DeviceStatus readTemperature (Temperature* value);
DeviceStatus readPressure (Pressure* value);
DeviceStatus readHumidity (Humidity* value);
...

void readMeteo () {
    Temperature temp;
    Pressure press;
    Humidity hum;

    DeviceStatus status = readTemperature(&temp);
    if (status==ok) {

        status = readPressure(&press);
        if (status==ok) {

            status = readHumidity(&hum);
            if (status==ok) {

                // Finally, do the job with temp, press, and hum
            } else {
                // Handle readHumidity exception
            }
        } else {
            // Handle readPressure exception
        }
    } else {
        // Handle readTemperature exception
    }
}
```

Obrada izuzetaka

❖ Problem:

- kod postaje nepregledan čim postoji nekoliko koraka koji mogu da rezultuju izuzetkom (duboko i nepregledno ugnezđivanje naredbi *if-then-else*)
- funkcije koje vraćaju status ne mogu da vraćaju svoje “prirodne” rezultate, koji se zato moraju prenositi kao izlazni argumenti po referenci / preko pokazivača
- pozivalac funkcija koje vraćaju kod greške može greškom ili namerno da ignoriše te greške: da ih ne proverava i ne obrađuje, ili da ih ne propagira svom pozivaocu
- ispitivanje povratnih statusa, tj. postojanja grešaka, troši vreme i resurse najčešće bespotrebno

❖ Uzrok problema: obrada izuzetaka učešljana je u regularan tok, isprepletana je sa osnovnim tokom, pa ga čini nepreglednim i težim za programiranje

❖ Bolji pristup:

- regularni tok programira se kao da je sve u redu, tj. da nema izuzetaka i nije opterećen njihovom obradom,
- s tim da određeni koraci regularnog toka mogu da *bace* (*throw*) izuzetak
- obrada izuzetaka se piše kao *obrađivač izuzetka* (*exception handler*) pridružen bloku sa regularnim tokom, koji *hvata* izuzetak (*catch*), ali nije izmešana sa njim

❖ Mehanizam obrade izuzetaka nije direktno vezan za objektno orijentisani paradigmu, ali je podržan u svim današnjim popularnim OO programskim jezicima na vrlo sličan način, po istom principu

Obrada izuzetaka

```
enum DeviceException {deviceFaulty, communicationFailed};
```

```
Temperature readTemperature();
```

```
Pressure readPressure();
```

```
Humidity readHumidity();
```

```
...
```

```
void readMeteo () {
```

```
try {
```

try blok uokviruje regularan tok u kom se može baciti izuzetak

```
    Temperature temp = readTemperature();
```

```
    Pressure press = readPressure();
```

```
    Humidity hum = readHumidity();
```

```
    // Finally, do the job with temp, press, and hum
```

```
}
```

Ukoliko se try blok završi bez izuzetka, završava se
ceo konstrukt *try-catch* (*catch* blokovi se preskaču)

```
catch (DeviceException& de) {
```

```
    // Handle exceptions
```

```
}
```

Ukoliko se tokom izvršavanja try bloka baci
izuzetak, prekida se izvršavanje tog *try* bloka i
kontrola prebacuje na *catch* blok (*exception handler*)
koji prihvata izuzetak tog tipa kao svoj argument

Obrada izuzetaka

Bacanje izuzetka:

- ❖ Operatorom *throw*:

```
template<typename T>
T& vector<T>::at (int pos) {
    if (pos<0 || pos>=this->size) throw out_of_range;
    return this->array[pos];
}
```

Unutar *catch* bloka, tekući izuzetak koji se obraduje može biti ponovo bačen operatorom *throw* bez operanda:

```
try {
    ...
}
catch (...) {
    device->reset();
    throw;
}
```

- ❖ Mnoge funkcije iz standardne biblioteke jezika C++ mogu da bace izuzetke, npr. `vector::at`, `string::substr` itd, čime signaliziraju greške (nemogućnost da urade zahtevano npr. zbog nekorektnosti argumenta)
- ❖ Neki operatori ugrađeni u jezik mogu da bace izuzetke, npr. `dynamic_cast` (ako rezultat izraza koji se konvertuje nije zahtevanog odredišnog tipa za reference) ili `new` (ako ne može da alocira prostor u memoriji za objekat koji se kreira), opet signalizirajući grešku

Obrada izuzetaka

- ❖ Izuzetak na jeziku C++ može biti objekat bilo kog tipa, ugrađenog tipa ili klase
- ❖ Funkcije iz standardne biblioteke bacaju izuzetke koji su bezimeni objekti klasa izvedenih iz osnovne klase *exception*; sve ove izvedene klase imaju sledeća osnovna svojstva:
 - objekti ovih klasa mogu se kreirati pozivom konstruktora sa argumentom koji predstavlja niz znakova koji daje objašnjenje za izuzetak:

```
...throw out_of_range("Argument pos in function vector::at out of range.");
```
 - objekti ovih klasa mogu se kopirati (prilikom inicijalizacije ili operatorom dodele =)
 - polimorfnu operaciju *what* koja vraća niz znakova koja predstavlja objašnjenje (zadat inicijalizacijom)
- ❖ Po pravilu, i korisnički definisani izuzeci treba da slede isti obrazac, odnosno da budu objekti klasa (izvedenih iz klase) iz ove hijerarhije

Obrada izuzetaka

Obrada (hvatanje) izuzetka:

- ❖ Kada se izuzetak baci, prekida se izvršavanje prvog dinamički okružujućeg *try* bloka koji je započet, a nije završen, i u njemu traži *catch* blok koji može priхватiti tip bačenog izuzetka; pravila uparivanja (skoro) su ista kao za argumente funkcija (uz sprovođenje implicitnih konverzija)
- ❖ *catch* blokovi priduženi istom *try* bloku pretražuju se redom kako su navedeni (može ih biti više); bira se prvi koji po tipu argumenta odgovara bačenom izuzetku (može obraditi taj izuzetak) i započinje se njegovo izvršavanje
- ❖ ukoliko u datom *try* bloku nijedan *catch* blok ne može priхватiti bačeni izuzetak, traži se sledeći dinamički okrućujući *try* blok (mogu se ugnezđivati), koji može biti i van funkcije koja se izvršava - funkcija tako baca izuzetak (prosleđuje ga svom pozivaocu), i tako redom po steku ugnezđenih poziva funkcija u tekućoj niti
- ❖ *catch* blok može baciti isti ili neki novi izuzetak, npr. tako što funkcija koja se unutar njega poziva baca izuzetak
- ❖ ako se *try* blok završi bez bačenog izuzetka, ili se *catch* blok koji je aktiviran završi bez bačenog izuzetka, izvršavanje nastavlja iza *try-catch* konstrukta

```
void readMeteo () {  
    try {  
        ...  
    }  
    catch (ThermometerException& e) {  
        ...  
    }  
    catch (ManometerException& e) {  
        ...  
    }  
}  
  
void calcMeteo () {  
    try {  
        ...  
        ...readMeteo()...  
        ...  
    }  
    catch (DeviceException& e) {  
        ...  
    }  
}
```

Ukoliko je bačen izuzetak nekog trećeg tipa, biće prosleđen pozivaocu ove funkcije, tj. bačen iz nje

Ova funkcija može baciti izuzetak koji se može uhvatiti u nekom od *catch* blokova dole, ili dalje proslediti pozivaocu, ukoliko se tu ne uhvati

Obrada izuzetaka

- ❖ Zbog ovog redosleda, *catch* blok koji prima objekat izvedene klase (po vrednosti ili preko reference ili pokazivača na izvedenu klasu) mora da bude ispred onog koji hvata objekat osnovne klase (po vrednosti ili preko reference ili pokazivača na osnovnu klasu), jer u suprotnom nikada neće biti aktiviran:

```
try {  
    ...  
}  
catch (Derived& e) {  
    ...  
}  
catch (Base& e) {  
    ...  
}
```

Ovaj hvatač biće aktiviran za izuzetke tipa *Derived*...

... a ovaj za ostale koji su tipa *Base*, ali nisu *Derived*.
Da je ovaj ispred prvog, uvek bi on bio aktiviran (jer je svaki *Derived* ujedno i *Base*)

- ❖ *catch(...)* hvata izuzetke bilo kog tipa i, shodno tome, mora uvek biti poslednji u nizu (inače ostali nikada ne bi bili aktivirani); ovakav hvatač može poslužiti kao obezbeđenje za to da nijedan izuzetak ne bude bačen iz date funkcije:

```
try {  
    ...  
}  
catch (ThermometerException& e) {  
    ...  
}  
catch (ManometerException& e) {  
    ...  
}  
catch (...) {  
    ...  
}
```

Hvata izuzetke bilo kog tipa

Obrada izuzetaka

- ❖ *try-catch* konstrukt može da uokviri celo telo funkcije, uključujući i inicijalizatore podobjekta osnovne klase i članova klase za konstruktore te klase (u kojima se ovo tipično i koristi); svaki *catch* za ovakav *try* treba da se završi bacanjem izuzetka; ukoliko toga nema, isti izuzetak koji se obrađuje se implicitno baca na kraju *catch* bloka (kao sa *throw*;

```
template <typename T>
class huge_vector : public vector<T> {
    ...
}

template <typename T>
huge_vector<T>::huge_vector<T> () try : vector<T>(0x100000000) {
}
catch (bad_alloc& e) {
    cerr<<"Cannot allocate a huge vector: "<< e.what() << "\n";
}
```

Pokušaj alokacije niza ove veličine;
može baciti izuzetak ako alokacija ne uspe

Ovde je implicitan *throw*;

- ❖ Ako se ne pronađe odgovarajući hvatač za bačeni izuzetak sve do dna steka poziva za tekuću nit, izvršava se bibliotečna funkcija *terminate* koja podrazumevano prekida izvršavanje programa (ali se to može promeniti)
- ❖ Ako se funkcija označi kao *noexcept*, onda ona implicitno hvata sve izuzetke koji eventualno nisu obrađeni unutar nje i poziva funkciju *terminate*; takva funkcija tako nikada ne baca (ne prosleđuje) izuzetke:

```
void transaction () noexcept {
    ...
}
```

Ova funkcija ne baca izuzetke,
čak i ako ih ne obrađuje

isto ovo se može postići i eksplisitno:

```
void transaction () noexcept {
    try {
        ...
    }
    catch (...) {
        std::terminate();
    }
}
```

Obrada izuzetaka

- ❖ Šta je uopšte izuzetak? Šta je izuzetna situacija, a šta nije? Kada koristiti izuzetke, a kada prosto ispitivati stanje i rezultate?
- ❖ Primer: procedura koja učitava znakove iz nekog ulaznog znakovnog toka (npr. fajla) i obrađuje ih. Da li je nailazak na kraj fajla izuzetak?

- ❖ Ako se ne tretira kao izuzetak:

```
ifstream f("input.txt");
while (!f.eof()) {
    char c = f.getc();
    // ...
}
```

ifstream je bibliotečna klasa za ulazne tekstualne fajlove kao znakovne tokove

- ❖ Ako se tretira kao izuzetak:

```
ifstream f("input.txt");
while (true) {
    try {
        char c = f.getc();
        // ...
    } catch (...) {
        break;
}}
```

- ❖ Naravno da je ovo drugo potpuno besmisleno, jer kraj fajla nije neregularna, nego sasvim očekivana i ispravna situacija. Osim toga, kod koji bi tako tretirao kraj fajla bio bi potpuno nepregledan i nelogičan
- ❖ Međutim, šta ako procedura treba da učitava složenije strukture, recimo zapise koji imaju odgovarajući format i strukturu; šta ako se tada nađe na kraj fajla u sledećim slučajevima: a) tačno nakon završetka celog zapisa; b) pre završetka zapisa?
- ❖ Sledeći primer: procedura pretražuje neku kolekciju elemenata i traži zadati element: a) koji može, a ne mora da bude u njoj; b) koji bi morao da bude u njoj, jer je to regularno stanje programa? Kako se tretira situacija ako ta procedura ne nađe traženi element?

Obrada izuzetaka

- ❖ Paradigma programiranja i projektovanja softvera pod nazivom *programiranje po ugovoru* (*programming by contract/design by contract*, Bertrand Meyer, 1986, programski jezik *Eiffel*) prepostavlja preciznu, formalnu specifikaciju interfejsa softverskih komponenata (npr. potprograma ili klase) za koje se definišu logički uslovi (kao logički izrazi) sledećih vrsta:
 - *preduslovi (precondition)*: logički uslovi koji moraju da budu zadovoljeni prilikom poziva potprograma da bi taj potprogram mogao da obavi svoj zadatak; ove uslove mora da zadovolji pozivalac pri pozivu potprograma; na primer, uslovi u pogledu stanja globalnih promenljivih ili argumenata poziva potprograma (npr. indeks mora biti unutar granica niza)
 - *postuslovi (postcondition)*: logički uslovi koji moraju da budu zadovoljeni na kraju izvršavanja potprograma; ove uslove mora da zadovolji pozvani potprogram; na primer, uslovi u pogledu stanja globalnih podataka ili validnosti povratne vrednosti funkcije
 - *invarijante (invariant)*: logički uslovi vezani za neku komponentu (npr. instancu apstraktnog tipa podataka, objekat klase, modul, strukturu podataka) koji moraju da budu zadovoljeni uvek, osim u tranzijentnim periodima kada se ta komponenta prevodi iz jednog validnog stanja (kada invarijanta važi) u drugo validno stanje (kada invarijanta ponovo važi); na primer, metode klase koje prevode objekat iz jednog regularnog u drugo regularno stanje, tokom koje invarijanta privremeno ne mora da važi
- ❖ Prema tome, preduslovi su obaveze koje mora da izvrši pozivalac prema pozvanom potprogramu, da bi taj potprogram mogao da izvrši svoju obavezu; postuslovi su obaveze koje potprogram mora da ispuni ukoliko je pozivalac ispunio svoje obaveze; zato oni predstavljaju *ugovor (contract)* između dve strane

Obrada izuzetaka

- ❖ Preporučeni stil tretiranja izuzetaka jeste taj da se, prema ovoj paradigmi, izuzetkom tretira nemogućnost zadovoljenja nekog od ovih logičkih uslova, i *samo* to (sve drugo nisu izuzeci):
 - preduslovi (*precondition*): pozvani potprogram, ukoliko nije ispunjen njegov preduslov (npr. neregularan argument) treba da podigne izuzetak; na primer, neispravan parametar
 - postuslovi (*postcondition*): ukoliko potprogram nije u mogućnosti da ispuni svoje postuslove, treba da signalizira izuzetak svom pozivaocu; na primer, funkcija ne može da kreira povratnu vrednost ili ne može da uspostavi njenu invarijantu, ne može da pronađe ono što bi morala da pronađe i vrati, i slično
 - invarijante (*invariant*): ukoliko metoda klase ne može da očuva invarijantu, treba da baci izuzetak; invarijante su zapravo preduslovi i postuslovi koji važe za objekat na ulazu i izlazu svake metode klase
- ❖ Jedna tehnika za apstrahovano ispitivanje uslova - tzv. *tvrdnje (assertion)*:

```
assert(i>=0 && i<this->size);
```
- ❖ U standardnoj biblioteci za C++, *assert* je definisan kao makro koji zavisi od drugog definisanog makroa *NDEBUG* koji nije definisan u biblioteci, već se može definisati (uključiti ili isključiti) u okruženju prevodioca:
 - ako je *NDEBUG* definisan (uključen), makro *assert* nema nikavog efekta (zamenjuje se sa ((void)0))
 - u suprotnom, zamenjuje se implementacijom koja ispituje vrednost datog skalarnog izraza; ako je ta vrednost nula, na standardni izlaz za greške ispisuje informacije o mestu u programu i završava program
- ❖ Ovakvi slični makroi ili funkcije mogu da se koriste za ispitivanje preduslova, postuslova i invarijanti, ali tako da podižu izuzetke ukoliko uslov nije ispunjen

Obrada izuzetaka

- ❖ Tretman izuzetaka, odnosno izuzetnih situacija koje su detektovane, može da bude na jednom od sledećih nivoa sigurnosti (*safety*), od najstrožijeg ka najlabavijem:
 - *Nothrow* (ili *nofail*) garancija: potprogram nikada ne baca izuzetak (funkcije označene kao *noexcept*); to znači da nema preduslove i uvek će sigurno očuvati sve invarijante i ispuniti svoje postuslove, ili pak greške sakriva ili tretira na drugi način; ovakvo ponašanje očekuje se od destruktora (oni su implicitno *noexcept*) i *move* konstruktora i operatora dodele, kao i drugih funkcija koje se implicitno pozivaju tokom prosleđivanja bačenog izuzetka do hvatača
 - *Jaka garancija sigurnosti od izuzetaka* (*strong exception safety guarantee*): ako potprogram baci izuzetak, stanje programa biće vraćeno na ono pre poziva tog potprograma (tzv. *rollback*) - sam potprogram je napravljen tako da to stanje povrati ili očuva
 - *Osnovna garancija sigurnosti od izuzetaka* (*basic exception safety guarantee*): ako potprogram baci izuzetak, stanje programa biće validno (iako može biti različito od onog pre poziva tog potprograma), tj. sve invarijante biće očuvane
 - *Nikakva garancija*: ukoliko se baci izuzetak, program može ostati u nekorektnom stanju; ovakve pojave predstavljaju *bagove* (*bug*), odnosno nepravilnosti u programu koje treba rešavati
- ❖ Preporuka je da svaka funkcija podrži najstrožiji od ovih redom iznesenih nivoa koji je moguće ispuniti

Deo III: Detalji jezika C++

- ❖ Elementi i principi prevođenja i povezivanja
- ❖ Tipovi i konverzije
- ❖ Deklaracije i opseg važenja
- ❖ Izrazi
- ❖ Funkcije
- ❖ Životni vek varijabli
- ❖ Konstruktori, destruktori i operator dodele
- ❖ Objekti sa zauzetim resursima
- ❖ Inicijalizacija
- ❖ Preklapanje operatora

Glava 6: Elementi i principi prevođenja i povezivanja

- ❖ Neki osnovni pojmovi jezika
- ❖ Prevodenje
- ❖ Povezivanje
- ❖ Preprocesor
- ❖ Leksički elementi



Neki osnovni pojmovi jezika

- ❖ U skladu sa opredeljenjem za to da objekte klase tretira što sličnije, ako ne i isto kao i primerke ugrađenih tipova, C++ uvodi uopštenje pojma *objekta*, pošto mnoga pravila važe podjednako za primerke ugrađenih tipova i klase
- ❖ Na jeziku C++, *objekat* jeste *oblast skladišta* (*region of storage*), odnosno deo memorije, koji ima:
 - veličinu, koja se može odrediti operatorom *sizeof* (u jedinicama *sizeof(char)*)
 - poravnanje (*alignment*) u memoriji, koje se može odrediti operatorom *alignof*
 - trajanje skladištenja (*storage duration*), koje može biti automatsko, statičko, dinamičko ili lokalno za nit (*thread-local*)
 - životni vek (*lifetime*), koji je određen trajanjem skladištenja ili je privremen (*temporary*): vreme tokom izvršavanja za koje objekat postoji
 - tip, koji može biti *klasni tip* (klasa, struktura ili unija) ili neklasni tip
 - vrednost, koja može biti i neodređena
 - opcionalno ime: identifikator u programu koji se odnosi na taj objekat
- ❖ Prema tome, termin *objekat* može imati donekle različito značenje, u zavisnosti od konteksta:
 - u širem kontekstu objektno orijentisanog programiranja, objekat je primerak (instanca) klase
 - u užem kontekstu pravila jezika C++, objekat može biti primerak klase ili neklasnog (ugađenog) tipa
- ❖ Najveći broj pravila i najveći deo složenosti semantike i pravila jezika C++ jeste posledica upravo ove odluke!

Neki osnovni pojmovi jezika

- ❖ Sledeće stvari nisu objekti:
 - vrednost: rezultat izraza (operacije nad operandom), osim kada se zahteva pravljenje privremenog objekta
 - referenca
 - funkcija
 - enumerator: član enumeracije (simbolička konstanta)
 - tip
 - nestatički član klase
 - šablon
 - *this*
 - i još neke stvari
- ❖ *Varijabla (variable)* je objekat ili referenca koji nije nestatički podatak član i koji je uveden deklaracijom
- ❖ Objekti se prave:
 - definicijama: posebne vrste deklaracija koje imaju efekat pravljenja objekta
 - operatorom *new*
 - operatorom *throw*
 - kada se zahteva pravljenje privremenog objekta
- ❖ *Deklaracija (declaration)* je iskaz koji uvodi identifikator u program: svaki identifikator mora se deklarisati pre upotrebe

Prevodenje

- ❖ Program na jeziku C++ sastoji se od jednog ili više modula, pri čemu se modulom smatra sadržaj jednog fajla za izvornim kodom (tipična ekstenzija imena je *.cpp*)
- ❖ Svaki izvorni fajl je *odvojena jedinica prevodenja (compilation unit)*: svaki fajl se prevodi odvojeno i nezavisno: kada prevodilac (*compiler*) prevodi jedan fajl, ne izlazi iz granica tog fajla, odnosno ne tretira druge fajlove programa
- ❖ Kada prevodi jedan fajl sa izvornim kodom (*.cpp*), prevodilac će generisati jedan fajl sa prevedenim kodom, fajl sa tzv. *objektnim kodom (object file*, tipična ekstenzija *.obj* ili *.o*; termin nema veze sa objektno orijentisanim programiranjem)
- ❖ Bilo koja greška u prevodenju uzrokuje da prevodilac ne proizvede izlazni *obj* fajl; bez obzira na to, prevodilac po pravilu nastavlja prevodenje, pokušavajući da prevaziđe svaku grešku i prijavljuje sve eventualne druge greške na koje nađe
- ❖ Fajl sa izvornim kodom sastoji se isključivo od deklaracija: tipova (uključujući i klase), funkcija, objekata i drugog. *Deklaracija (declaration)* je iskaz koji uvodi identifikator u program
- ❖ Svako ime (identifikator) koje se koristi u programu mora najpre biti *deklarisano*, u suprotnom će prevodilac prijaviti grešku u prevodenju

Prevodenje

- ❖ Prevodilac učitava znak po znak iz ulaznog fajla sa izvornim kodom programa; iako tekst programa ljudi doživaljavaju dvodimenzionalno (u ravni), pri čemu im te dve dimenzije pomažu u čitanju i razumevanju (prelom redova, proredi i uvlačenje, odnosno “nazubljivanje” koda), prevodilac kod tumači isključivo sekvencijalno
- ❖ Prevodilac najpre učitane znakove grupiše u veće celine, tzv. *leksičke elemente* ili *lekseme* (*lexical element, lexem*), ili *žetone* (*token*), u skladu sa pravilima jezika; ova faza prevodenja naziva se *leksička analiza* (*lexical analysis*); na primer, u sledećem delu koda, različitim bojama označene su različite lekseme:

```
if (i++ +j>=0 && i<this->size())
```

- ❖ U daljem postupku prevodilac tretira lekseme kao integralne celine, odnosno kao elemente od kojih su izgrađeni krupniji jezički iskazi, tj. rečenice
- ❖ Prevodilac tokom prevodenja prepoznaje te veće jezičke celine (rečenice) na osnovu *gramatike* (*grammar*) jezika; ova faza prevodenja naziva se *parsiranje* (*parsing*); u slučaju prestupa nekog pravila gramatike, prevodilac prijavljuje grešku u prevodenju
- ❖ Za prepoznate rečenice i elemente u njima, prevodilac proverava ostala pravila jezika, tzv. semantička pravila (*semantic rules*), i opet prijavljuje greške u slučaju prestupa
- ❖ Konačno, za one elemente rečenica za koje je to definisano semantikom jezika, prevodilac generiše sadržaj u prevedenom objektnom fajlu u kome se principijelno nalazi:
 - binarni mašinski kod za mašinske (procesorske) instrukcije
 - alociran prostor za određene kategorije objekata sa tzv. *statičkim trajanjem skladištenja* (*static storage duration*)

Prevodenje

- ❖ Kada nađe na novu deklaraciju, prevodilac dodaje deklarisani identifikator u strukturu podataka koju izgrađuje tokom prevodenja i koja se tradicionalno naziva *tabela simbola* (*symbol table*); u ovoj strukturi prevodilac čuva informacije o svakom deklarisanim identifikatoru: o tome kojoj jezičkoj kategoriji pripada (tip, objekat, funkcija itd.), kog je tipa, kao i sva ostala svojstva deklarisanih entiteta definisana pravilima jezika
- ❖ Kada nađe na neki upotrebljen identifikator, prevodilac:
 - proverava da li je taj identifikator deklarisan i da li je dostupan, po pravilima jezika; ako nije, prijavljuje grešku;
 - proverava da li je identifikator upotrebljen u skladu sa pravilima jezika i ako nije, prijavljuje grešku; na primer, ne može se vršiti operacija $f++$ ako je f funkcija, ili operacija $a()$ ako je a objekat tipa *int* i slično;
 - ako je to definisano semantikom jezika, zna kako da generiše kod za upotrebu tog identifikatora u odgovarajućem kontekstu

Prevodenje

- ❖ Na primer, deklaracija globalnog statičkog objekta jeste i *definicija*, koja ima efekat pravljenja objekta:
`int n = -16;`
- ❖ Ova definicija ima:
 - deklarativni deo (pre znaka `=`), koji deklariše ime (prevodilac uvodi ime u tabelu simbola)
 - inicijalizator (izraz iza znaka `=`), kojim se inicijalizuje objekat
- ❖ Po pravilima jezika, ovakav objekat ima *statičko trajanje skladištenja* (*static storage duration*) i *statički životni vek*; prema semantičkim pravilima jezika C++, za ovakve objekte važi to da postoji jedna instanca objekta za svaku definiciju (za razliku od npr. definicija automatskih objekata, za koje se kreira nova instanca svaki put kada izvršavanje dođe do takve definicije)
- ❖ Zbog toga, za ovakve objekte prevodilac može (i to po pravilu radi) da alocira prostor *statički*, u vreme prevođenja; taj prostor odvaja se u prevedenom objektnom fajlu (obično u posebnom segmentu za podatke, odvojenom od segmenta za instrukcije): za svaki takav objekat odvoji se prostor u prevedenom zapisu za smeštanje tog objekta
- ❖ U navedenom primeru, inicijalizator objekta je *konstantan izraz* (*constant expression*) — izraz čiji se rezultat može izračunati u vreme prevođenja
- ❖ Za ovakve statičke objekte fundamentalnog tipa, inicijalizovane konstantnim izrazom, prevodilac po pravilu inicijalizuje statički alocirani prostor vrednošću izraza još u vreme prevođenja (ovde je konstantni izraz trivijalan - celobrojni literal `-16`, čiji se binarni zapis upisuje u alocirani prostor)
- ❖ Za funkcije, definicija je ona deklaracija koja daje i telo funkcije; za definiciju funkcije, prevodilac generiše binarni mašinski kod za instrukcije koje predstavljaju prevod naredbi iz tela funkcije

A.cpp

```
int n = -16;  
void f () {  
    n++;  
}
```

A.obj

```
n: ff ff ff f0  
f: ld r1,n  
    inc r1  
    st r1,n  
    ret
```

Ovo je izmišljen, pojednostavljen format *obj* fajla. Za mašinske instrukcije generisani kod je binaran.

Prevodenje

- ❖ Međutim, osim toga, prevodilac u prevedenom fajlu ostavlja i informacije o svim imenima (simbolima) koji su definisani u datom fajlu, a mogu se koristiti u drugim fajlovima; ovakva imena nazivaju se imena sa *spoljašnjim vezivanjem (external linking)*
- ❖ U posebnom delu objektnog fajla, tipično u zaglavlju, prevodilac pravi tabelu takvih, *izvezenih simbola*, ostavljajući samo informaciju o:
 - imenu (simbolu, prost niz znakova), bez ikakvih informacija o tome kakav je entitet predstavljalo to ime u programu (šta je, kog je tipa itd.)
 - adresi, relativnoj u odnosu na početak binarnog prevoda (ili segmenta) u koji se to ime preslikava
- ❖ Na primer, po pravilima jezika C++, globalni objekat ili funkcija ima spoljašnje vezivanje, osim ako je eksplicitno deklarisana kao *static* (tada ima interno vezivanje)
- ❖ Imena koja imaju *interno vezivanje (internal linking)* ne mogu se koristiti u drugim fajlovima; prevodilac za ovakva imena ne ostavlja ovakve informacije u generisanoj tabeli simbola u objektnom fajlu

A.cpp

```
int n = -16;  
void f () {  
    n++;  
}
```

A.obj

```
symbols  
    ↑n: data:0  
    ↑f: code:0  
end symbols  
  
seg data  
n: ff ff ff f0  
end seg  
  
seg code  
f: ld r1,n  
    inc r1  
    st r1,n  
    ret  
end seg
```

Prevodenje

- ❖ Sada definisane entitete želimo da koristimo u drugom fajlu *B.cpp*, ali tako da se odnose na entitete već definisane u *A.cpp*:

```
// B.cpp

void g () {
    n++;
    f();
}
```

- ❖ Ako se u ovom fajlu ne navede deklaracija objekta *n* i funkcije *f*, prevodilac će prijaviti grešku jer identifikator nije deklarisan
- ❖ Ako se u *B.cpp* navede sledeća deklaracija:

```
int n;
```

onda će prevodilac nju i dalje smatrati *definicijom*, i ponovo će alocirati prostor za taj objekat, iako nije inicijalizovan; osim toga, mogao bi da operacije sa *n* u potpunosti prevede, koristeći adresiranje lokacije tog alociranog prostora

- ❖ Ovo nije željeno ponašanje, već želimo da se ove operacije odnose na *n* i *f* definisane u drugom fajlu *A.cpp*, a ne da se definišu novi entiteti

B.cpp

```
int n;

void g () {
    n++;
    f();
}
```

B.obj

```
symbols
    ↑n: data:0
    ↑g: code:0
end symbols

seg data
n: 00 00 00 00
end seg

seg code
g: ld r1,n
    inc r1
    st r1,n
    call f
    ret
end seg
```

Prevodenje

- ❖ Da bismo napravili deklaraciju koja nije i definicija, za ovakav statički objekat potrebno je navesti ključnu reč *extern*:

```
extern int n;
```

Sada prevodilac neće alocirati prostor za ovaj objekat

- ❖ Za funkciju je dovoljna deklaracija bez tela funkcije, ona nije definicija (reč *extern* može da se piše, ali ne mora):

```
extern void f();
```

- ❖ Međutim, pošto *n* i *f* nisu definisani, prevodilac ne može u potpunosti prevesti operacije sa njima; umesto toga, on će generisati binarni kod za mašinske instrukcije koje implementiraju potrebne operacije, ali sa adresnim poljima u kojima je potrebno definisati adrese tih entiteta postavljenim na proizvoljnu, nedefinisanu vrednost (npr. sve nule); ovakva polja tako ostaju *nerazrešena* (*unresolved*) u vreme prevođenja
- ❖ Zbog toga prevodilac u zaglavlju objektnog fajla, u spisku simbola, ostavlja informacije o svim takvim *uvezenim* simbolima, tj. simbolima koji se koriste, a nisu definisani u datom fajlu, kao i o mestima u binarnom kodu na kojima se nalaze nerazrešena adresna polja mašinskih instrukcija u koja treba naknadno upisati adrese
- ❖ Time prevodilac završava svoj posao; prema tome, objektni fajl nije izvršiv, između ostalog, zbog toga što mašinske instrukcije mogu da imaju nerazrešena adresna polja

B.cpp

```
extern int n;  
void f();  
  
void g () {  
    n++;  
    f();  
}
```

B.obj

```
symbols  
↓n: ...  
↓f: ...  
↑g: code:0  
end symbols  
  
seg code  
g: ld r1,?  
   inc r1  
   st r1,?  
   call ?  
   ret  
end seg
```

Povezivanje

- ❖ Zadatak da od skupa objektnih fajlova napravi program, tj. izvršiv fajl, ima *povezivač*, tj. *linker* (*linker*); linkeru se zadaje spisak ulaznih *obj* fajlova i zadatak da napravi određeni *exe* fajl kao svoj izlaz
- ❖ Linker taj zadatak obavlja u dva prolaza:
 - u prvom prolazu analizira ulazne fajlove, veličinu njihovog binarnog sadržaja (prevoda), i pravi mapu *exe* fajla; osim toga, sakuplja infomacije iz tabela simbola *obj* fajlova i izgrađuje svoju tabelu simbola; u tu tabelu simbola unosi izvezene simbole iz *obj* fajlova, za koje odmah može da izračuna adresu u odnosu na ceo *exe* fajl
 - u drugom prolazu generiše binarni kod, i ujedno razrešava nerazrešena adresna polja mašinskih instrukcija na osnovu informacija o adresama u koje se preslikavaju simbola iz svoje tabele simbola

A.obj

```
symbols
↑n: data:0
↑f: code:0
end symbols

seg data
n: ff ff ff f0
end seg

seg code
f: ...
end seg
```

B.obj

```
symbols
↑g: code:0
↓a: ...
↓f: ...
end symbols

seg code
g: ld r1,?
    inc r1
    st r1,?
    call ?
    ret
end seg
```

C.obj

```
...
```

P.exe

```
seg data
n: ff ff ff f0
end seg

seg code
f: ...
end seg

seg code
g: ld r1,n
    inc r1
    st r1,n
    call f
    ret
end seg
```

...

Povezivanje

- ❖ Biblioteka (*library*) je fajl sa tipičnom ekstenzijom *.lib*, koja ima principijelno isti format kao i objektni fajl; kada povezuje fajlove, linker tretira ulazne *lib* i *obj* fajlove na isti način (zadaje mu se spisak ulaznih *obj* i *lib* fajlove koje treba povezati)
- ❖ Razlika je u tome što je *obj* fajl nastao prevodenjem jednog izvornog fajla, dok je *lib* nastao *povezivanjem* više *obj* (i moguće drugih *lib*) fajlova u jedan *lib* fajl
- ❖ Motiv je praktičan: kod za neku biblioteku (potprograma, struktura, tipova) za određenu namenu može da ima mnogo (na stotine) izvornih fajlova; nepraktično bi bilo koristiti to mnoštvo fajlova i sve pojedinačno ih davati na povezivanje u programe koji ih koriste; “pakovanjem” u biblioteku koja se isporučuje i povezuje kao jedan *lib* fajl olakšava se rukovanje
- ❖ Biblioteka svakako izvozi simbole koje definiše (upravo one koji predstavljaju njene usluge, ono za šta se ona i koristi), ali može i da ih uvozi, ukoliko zavisi od neke druge biblioteke (koristi simbole iz druge biblioteke)

Povezivanje

- ❖ Posao pravljenja biblioteke obavlja isti linker, koji može raditi u dva režima, za pravljenje *exe* i za pravljenje *lib* fajla (režim mu se zadaje prilikom pokretanja)
- ❖ Naravno, posao pravljenja *lib* fajla se razlikuje od posla pravljenja *exe* fajla iz nekoliko razloga:
 - *exe* može imati drugačiji format od *lib* (i *obj*) fajla
 - *exe* sadrži i informacije koje *lib* (i *obj*) fajlovi ne sadrže, a koje su potrebne operativnom sistemu za pokretanje programa; na primer, barem početnu adresu prve instrukcije od koje operativni sistem treba da počne izvršavanje programa
 - kada pravi *exe*, linker mora da završi posao bez nerazrešenih simbola; *lib* može da ima simbole koje uvozi i koji ostaju nerazrešeni
- ❖ Globalna funkcija *main* na jeziku C++ prevodi se kao najobičnija funkcija koju poziva kod koji se najpre izvršava prilikom pokretanja programa i koji onda poziva funkciju *main* kao bilo koju drugu funkciju; nakon povratka iz nje, poziva sistemski poziv za gašenje programa; ovaj okružujući kod može biti u nekom *obj* fajlu koji se uvek podrazumevano povezuje

Povezivanje

- ❖ U principu, prema tome, linker može da prijavi samo dve vrste grešaka:

 1. *Simbol nije definisan*: kada napravi evidenciju (u svojoj tabeli simbola) o tome koji fajlovi izvoze (definišu) koje simbole, a koji fajlovi uvoze koje simbole, linker može da zaključi da je neki fajl tražio (uvezao) neki simbol koji nijedan fajl nije definisao (izvezao); u informaciji o ovoj grešci linker ne može da kaže ništa o tome šta je simbol bio u izvornom programu niti gde je u izvornom kodu tražen (jer te informacije po pravilu i nema); tipični uzroci jesu:
 - zaboravljena definicija neke deklarisane funkcije ili objekta (retko, čista omaška)
 - zaboravljena neki *obj* ili *lib* fajl na spisku za linkovanje (omaška)
 - neka povezana biblioteka zavisi od (uvozi simbole iz) neke druge biblioteke, koja nije povezana
 2. *Simbol višestruko definisan*: kada nađe na simbol koji neki fajl izvozi, a isti taj simbol već postoji u tabeli simbola jer ga je neki drugi fajl već izvezao, linker prijavljuje grešku; ovo je situacija tzv. *sukoba imena* (*name clash*): dva izvorna fajla definisala su ista globalna imena sa eksternim vezivanjem (primetiti to da imena sa internim vezivanjem nisu ni vidljiva linkeru i ne mogu da naprave sukob); tipični uzroci jesu:
 - sukob imena u korisničkom programu
 - sukob imena iz korisničkog programa sa imenom koje je izvezla biblioteka

Ovakve pojave, tj. sukobe imena, značajno smanjuje korišćenje koncepta prostora imena (*namespace*) na jezik C++, čime se izbegava potreba za globalnim imenima i obeshrabruje njihova upotreba

Preprocesiranje

- ❖ Posledica: za svaki entitet definisan u fajlu *A* koji treba da se koristi u drugim fajlovima *B* moraju postojati deklaracije u svim tim fajlovima *B*
- ❖ Ali šta se dešava ako te deklaracije nisu identične, npr. kao posledica greške? Na primer, promenjena je projektna odluka da promenljiva *x* ne bude tipa *int* nego tipa *double*, ali izmena nije urađena svuda:

```
// A.cpp                                // B.cpp

double x = 2.3;                      extern int x;
                                         void g () {
                                         ...x++...
                                         }
```

- ❖ Prevodilac neće prijaviti grešku, jer je za njega kod u fajlu *B.cpp* ispravan; još gore, on će prevesti operacije sa *x* kao operacije sa celobrojnom promenljivom, a ne kao sa racionalnim brojem - potpuno drugačija implementacija i tretman binarnog sadržaja (može biti i različite veličine)
- ❖ Grešku neće prijaviti ni linker, jer je simbol *x* korektno izvezen i uvezan, osim ako prevodilac u generisani simbol (kao niz znakova) ne enkoduje i informaciju o tipu (što se često radi), npr. ako za ovaj *x* u zaglavlju *obj* fajla napravi simbol poput *x@int*
- ❖ Problem je nastao zbog prekršaja principa lokalizacije projektne odluke: manifestacije odluke da je *x* određenog tipa raštrkane su po kodu i redundantno kopirane kao deklaracije tog *x*; u slučaju potrebe za promenom, može se dogoditi ovakva greška zbog nekonzistentne izmene

Preprocesiranje

- ❖ Pre same faze prevodenja u užem smislu, prevodilac sprovodi fazu tzv. *preprocesiranja*
- ❖ Preprocesor je deo prevodioca koji transformiše tekst izvornog koda i rezultujući tekst prosleđuje dalje na prevodenje
- ❖ Preprocesorom se upravlja pomoću *direktiva (directive)*; svaka direktiva zauzima jedan red i ima sledeći format:
 - znak #
 - preprocesorska instrukcija: *define*, *undef*, *include*, *if*, *ifdef*, *ifndef*, *else*, *elif*, *endif*, *line*, *error*, *pragma*; svaki prevodilac može reagovati i na druge instrukcije koje nisu deo standarda i koje drugi prevodioci prosto ignorišu (npr. *#warning* za poruke upozorenja); u svakom slučaju, rezultujući tekst izvornog koda očišćen je od svih direktiva
 - opcionalni argumenti, u zavisnosti od vrste direktive
 - znak za novi red, kojim se završava direktiva
- ❖ Preprocesor ima sledeće mogućnosti transformacije teksta izvornog koda:
 - uslovno prevodenje delova izvornog koda (direktive *#if*, *#ifdef*, *#ifndef*, *#else*, *#elif* i *#endif*)
 - zamena tekstualnih makroa uz moguću konkatenaciju ili uokviravanje identifikatora u navodnike (direktive *#define* i *#undef* i operatori *#* i *##*)
 - uključivanje drugih fajlova (direktive *#include* i provera postojanja fajla pomoću *_has_include*)
 - izazivaju grešku u prevodenju (direktiva *#error*)

Preprocesiranje

- ❖ Direktiva `#include` ima za posledicu to da se sadržaj imenovanog fajla umetne u tekući izvorni kod umesto ove direktive (formalno, odmah nakon linije sa ovom direktivom); takav uključen kod se dalje preprocera, potencijalno rekurzivno (jer uključeni fajl može imati druge `#include` direktive)
- ❖ Ova direktiva ima dva oblika:
 - `#include <filename>` preprocesor traži imenovani fajl počev od predefinsanog mesta; uobičajeno je da se ovo koristi za fajlove koji sadrže deklaracije iz standardnih biblioteka koje dolaze uz prevodilac, pa se oni traže počev od direktorijuma u kome se nalaze ti bibliotečni fajlovi koji dolaze sa instalacijom prevodioca
 - `#include "filename"` preprocesor traži imenovani fajl počev od nekog drugog predefinisanog mesta, npr. od direktorijuma u kome je i tekući fajl ili koreni direktorijum podešen prevodiocu; koristi se za korisničke fajlove koji čine sam program
- ❖ Ovaj princip može se iskoristiti za rešavanje opisanog problema:
 - deklaracije elemenata programskog modula koji je dat u jednom fajlu, a čine *interfejs* tog modula (dakle ne *svih* elemenata koji su u tom modulu definisani, već samo onih koje drugi moduli treba da koriste), izdvajaju se u *fajlovu-zaglavlju (header file)*, sa tipičnom ekstenzijom `.h`
 - u fajlovima u kojima je potrebno koristiti te elemente, ali i u fajlu u kome je sama definicija (istim modul), deklaracije se uvoze uključivanjem fajla zaglavljva:

```
// A.h
extern int x;
// A.cpp
#include "A.h"
int x = 1;

// B.cpp
#include "A.h"
void g () {
    ...x++...
}
```

Ako deklaracija iz *A.h* ne odgovara definiciji u *A.cpp*, prevodilac će prijaviti grešku kada bude prevodio *A.cpp*, jer je *x* dva puta deklarisano različito

Preprocesiranje

- ❖ Rezultat je to da prevodilac “vidi” isti kod kao i kada su deklaracije u svim fajlovima u kojima se koriste bile pisane eksplisitno, ručno, kao i ranije, ali je razlika za programera značajna: deklaracije su lokalizovane na jednom mestu, u jednom modulu (tj. u njegovom *.h* i *.cpp* fajlu, koji moraju biti konzistentni), pa su modifikacije lakše i manje podložne greškama
- ❖ Posledično, kada se promeni neki *.h* fajl, svi *.cpp* fajlovi koji uključuju taj *.h* fajl moraju ponovo da se prevedu (ovo uključuje i tranzitivne zavisnosti od drugih uključenih *.h* fajlova), što podrazumeva da će prevodilac prevoditi i sve deklaracije u svim direktno ili indirektno uključenim *.h* fajlovima (barem da bi kerirao svoju tabelu simbola)
- ❖ Razvojno okruženje za programiranje (*integrated programming environment, IDE*) može da pomogne i smanji količinu fajlova koje treba prevoditi selektivnim prevođenjem samo onih *.cpp* fajlova koji su promenjeni, kao i onih koji zavise od promenjenih *.h* fajlova (odnosno uključuju te fajlove, uzimajući u obzir i tranzitivne zavisnosti)
- ❖ Ovakav postupak uslovnog prevođenja svih potrebnih fajlova naziva se *make* postupak (ponegde i *build* postupak), a konfiguracioni fajl koji definiše zavisnosti između fajlova *make* fajl
- ❖ Bez ozbira na tu mogućnost, programer treba da se trudi da smanji zavisnosti između fajlova, tako što ne uključuje nepotrebne *.h* fajlove u *.cpp* a posebno u druge *.h* fajlove, ali i ne unosi nepotrebne deklaracije, posebno definicije klasa u *.h* fajlove; ovo je posebno važno u velikim projektima koji mogu da imaju na stotine modula

Preprocesiranje

- ❖ Direktivom `#define` uvodi se nov simbol (identifikator) u preprocesiranje (u tabelu simbola preprocesora definisanih ovom direktivom)
- ❖ Opcionalo se može definisati i *makrozamena*: svaka pojava datog simbola zamenjuje se datim tekstrom u nastavku direkture; mogući su i argumenti makrozamene:

```
#define _a_h
```

Jednostavno definiše simbol `_a_h` u tabelu simbola preprocesora

```
#define N 10
```

Svaka pojava simbola `N` zamenjuje se tekstrom (literalom 10)

```
#define max(a,b) (((a)>=(b))?(a):(b))
```

Makrozamena sa argumentima

```
int a[N];
```

Prevodilac će videti deklaraciju: `int a[10];`

```
void f (int a, int b){
```

```
...
```

```
int c = max(x/3,y+1);
```

```
...
```

```
}
```

Prevodilac će videti deklaraciju:

```
int c = (((x/3)>=(y+1))?(x/3):(y+1));
```

Preprocesiranje

- ❖ Direktive `#if`, `#elif` (znači “*else-if*”), `#ifdef` (znači “*if-defined*”) i `#ifndef` (znači “*if-not-defined*”) omogućuju uslovno prevodenje: tekst uokviren odgovarajućim direktivama biće prevoden samo ako je odgovarajući uslov ispunjen; za direktive `#ifdef`/`#ifndef`, uslov je ispunjen ako je dati simbol definisan/nije definisan preprocessoru (pomoću prethodne direktive `#define`):

```
#ifdef _a_h  
...  
#endif
```

Ovaj kod biće prevoden samo ako je simbol `_a_h` definisan direktivom `#define` preprocessora

- ❖ Ove direktive mogu da se koriste npr. za prevodenje delova koda koji treba da postoji samo ako se program prevodi za odgovarajuću platformu, konfiguraciju i slično; prevodioci dozvoljavaju da im se zada simbol preprocessora koji se smatra generalno (globalno) definisanim za sve fajlove, iako nije eksplicitno definisan direktivom `#define`:

```
#ifdef Windows  
... // Windows-dependent code  
#elif defined(Linux)  
... // Linux-dependent code  
#endif
```

Ovaj kod biće prevoden samo ako je simbol `Windows` definisan, što se tipično radi podešavanjem prevodioca, tako da se dati simbol smatra definisanim pri preprocesiranju svih fajlova

Preprocesiranje

- ❖ Kod uključivanja *.h* fajlova, moguće su sledeće situacije:
 - isti fajl, npr. *x.h*, uključuje se (tipično indirektno) više puta u isti *.cpp* fajl, recimo tako što taj *.cpp* fajl uključuje npr. i *a.h* i *b.h*, a i jedan i drugi uključuju *x.h* (direktno ili indirektno); fajl *x.h* može sadržati definiciju nekog entiteta, tipično klase, jer se u *.h* fajlove po pravilu stavlja definicije klasa; tada će prevodilac dobiti na prevođenje višestruku definiciju istog entiteta, što nije dozvoljeno čak i ako su definicije identične
 - neki *.h* fajl uključuje se rekursivno (tipično indirektno)
- ❖ Da bi se ovakve pojave sprečile, potrebno je (i to se po pravilu radi, bez mnogo obaziranja na to da li je to neophodno u svakom konkretnom slučaju ili ne, jer svakako ne smeta) ceo sadržaj svakog *.h* fajla uokviriti u sledeću kombinaciju direktiva, što obezbeđuje da će prevodilac dobiti na prevođenje uokvirenih kod samo jednom, čak i ako je taj *.h* fajl uključen više puta:

```
#ifndef _figure_h
#define _figure_h
...
// The code of the header file;
#endif
```

Bilo koji simbol koji se jednoznačno može pridružiti datom fajlu; na primer, naziv tog fajla

Samo kada preprocesor prvi put nađe na ovaj sadržaj, ovaj simbol neće biti definisan i ceo kod će biti preveden, ali i simbol odmah definisan, što će sprečiti rekursivna uključivanja istog sadržaja ili ponovno uključivanje istog sadržaja na drugom mestu

Leksički elementi

- ❖ Komentar može biti u jednom od dva oblika:

- počinje znakovima /* i završava znakovima */; može se prostirati na više redova:

```
/*
  File: clock.h
  Author: dmilicev
*/
*****Clock: an abstraction that measures time and ticks seconds*****
*****
```

- počinje znakovima // i završava znakom za novi red; najčešći i koncizniji oblik komentara (lakši za pisanje), jer komentar najčešće zauzima ceo red ili kraj reda (iza koda koji komentariše):

```
// File: clock.h
// Author: dmilicev
//
// Clock: an abstraction that measures time and ticks seconds
//
int sum=0; // The sum of all items
```

- ❖ Komentari se ne mogu ugnezđivati:

- unutar komentara između znakova /* i */, znaci /* i // nemaju nikakvo posebno značenje
- unutar komentara koji počinje znakovima //, znaci /*, */ i // nemaju nikakvo posebno značenje

- ❖ Pre same faze preprocesiranja, svaki komentar zamenjuje se jednim znakom-belinom, što znači da samo razdvaja lekseme i nema nikakvo drugo značenje dalje tokom preprocesiranja i prevođenja (ne postoje tokom faza preprocesiranja i prevođenja)

Leksički elementi

- ❖ *Identifikator (identifier)* je proizvoljno duga sekvenca znakova koji mogu biti cifre, donja crta (_), mala i velika slova engleske abecede i neki od većine *Unicode* znakova (ali ne svi), s tim da prvi znak ne može biti cifra; na primer:

```
int α = 10;  
int бројЖаба = 0;
```

- ❖ Velika i mala slova se razlikuju i svi znaci su značajni
- ❖ Što se tiče pravila jezika, identifikatori se mogu pisati bilo kojim slovima, ali je preporučeno poštovati sledeći stil:
 - koristiti samo jedan od stilova dosledno i jednobrazno: C stil (`get_time`) ili tzv. *camel-case* (`getTime`)
 - imena klase pisati velikim početnim slovom, ostalo malim
- ❖ Identifikator se može koristiti da bi imenovao objekte, reference, funkcije, enumeratore, tipove (uključujući i klase), članove klase, šablove i druge elemente programa za koje je to dozvoljeno, sa sledećim izuzecima:
 - identifikatori koji predstavljaju ključne reči jezika ne mogu se koristiti za druge svrhe;
 - identifikatori koji počinju dvostrukom donjom crtom (_) su rezervisani
 - identifikatori koji počinju donjom crtom (_) iza koje je veliko slovo su rezervisani
 - identifikatori koji počinju donjom crtom (_) su rezervisani u globalnom opsegu
- ❖ “Rezervisani” znači da zaglavla iz standardne biblioteke mogu da definišu (direktivom `#define`) ili deklarišu takve identifikatore za svoje interne potrebe, prevodilac može predefinisati takve identifikatore i slično; ako programer definiše takve identifikatore, ponašanje može biti nedefinisano

Glava 7: Tipovi i konverzije

- ❖ Tipovi
- ❖ Konverzije
- ❖ Fundamentalni tipovi
- ❖ Nabrajanja
- ❖ Pokazivači
- ❖ Nizovi
- ❖ Reference
- ❖ Klasni tipovi
- ❖ Konstanti tipovi i funkcije članice
- ❖ Nestalni tipovi
- ❖ Korisnički definisane konverzije
- ❖ Korisnički definisani literali



Tipovi

- ❖ Objekti, reference, funkcije i izrazi imaju svojstvo koje se naziva *tip (type)*; ovo svojstvo ograničava skup operacija koje su dozvoljene za ove entitete i daje semantiku (značenje) sekvencama bita kojima se ti entiteti predstavljaju
- ❖ Fundamentalni tipovi (*fundamental types*):

- tip *void*
- tip *std::nullptr_t*
- aritmetički tipovi (*arithmetic types*):
 - tipovi racionalnih brojeva u pokretnom zarezu (*floating-point types*): *float*, *double*, *long double*
 - integralni tipovi (*integral types*):
 - tip *bool*
 - znakovni tipovi (*character types*):
 - uski znakovni tipovi (*narrow character types*): *char*, *signed char*, *unsigned char*
 - široki znakovni tipovi (*wide character types*): *char16_t*, *char32_t*, *wchar_t*
 - tipovi označenih celih brojeva (*signed integer types*): *short int*, *int*, *long int*, *long long int*
 - tipovi neoznačenih celih brojeva (*unsigned integer types*): *unsigned short int*, *unsigned int*, *unsigned long int*, *unsigned long long int*

Tipovi

❖ Složeni tipovi (*compound types*):

- tipovi referenci (*reference types*):
 - tipovi referenci na l-vrednosti (*lvalue reference types*):
 - tipovi *lvalue* referenci na objekte (*lvalue reference to object types*)
 - tipovi *lvalue* referenci na funkcije (*lvalue reference to function types*)
 - tipovi referenci na d-vrednosti (*rvalue reference types*):
 - tipovi *rvalue* referenci na objekte (*rvalue reference to object types*)
 - tipovi *rvalue* referenci na funkcije (*rvalue reference to function types*)
- tipovi pokazivača (*pointer types*):
 - tipovi pokazivača na objekte (*pointer-to-object types*)
 - tipovi pokazivača na funkcije (*pointer-to-function types*)
- tipovi pokazivača na članove (*pointer-to-member types*):
 - tipovi pokazivača na podatke članove (*pointer-to-data-member types*)
 - tipovi pokazivača na funkcije članice (*pointer-to-member-function types*)
- tipovi nizova (*array types*)
- tipovi funkcija (*function types*)
- tipovi nabrajanja (*enumeration types*)
- klasni tipovi (*class types*):
 - ne-unije (*non-union types*): strukture (*struct*) i klase (*class*)
 - tipovi unija (*union types*)

Tipovi

- ❖ Za svaki od tipova, osim tipova funkcija ili referenci, postoje tzv. *cv-kvalifikovane (cv-qualified)* varijacije: *const* (konstantni), *volatile* (nestalni) i *const volatile*
- ❖ Tipovi su grupisani i u različite kategorije po nekim svojstvima:
 - *objektni tipovi (object types)* su (potencijano cv-kvalifikovani) tipovi koji nisu tipovi funkcija, referenci i tip *void*; objekti mogu biti nekog od ovih tipova
 - *skalarni tipovi (scalar types)* su (potencijano cv-kvalifikovani) tipovi koji nisu tipovi nizova ili klasni tipovi; entiteti ovih tipova nemaju u sebi ugrađene objekte
 - ❖ *trivijalni tipovi (trivial types)*, tzv. *POD (Plain Old Data)* tipovi, tipovi literala (*literal types*) i drugi
- ❖ Za svaku od navedenih kategorija i potkategorija, u standardnoj biblioteci postoji odgovarajuća šablonska struktura *is_...* sa podatkom članom *value* koji u vreme prevođenja rezultuje vrednostima *true* ili *false*, u zavosnosti od toga da li je tip koji je argument šablonu pripadnik date kategorije; na primer:

```
bool b = std::is_integral<float>::value; // b==false  
b = std::is_reference<float&>::value; // b==true
```

- ❖ Specifikator *decltype* određuje tip datog entiteta ili tipa izraza datog u zagradama, na osnovu deklaracija, i to u vreme prevođenja (izraz kao operand se ne izračunava za vreme izvršavanja, već se njegov tip određuje za vreme prevođenja); ovo se može koristiti za definisanje tipova koje je teško odrediti, ili koji zavise od tipova drugih entiteta, tipično u šablonima; na primer:

```
b = is_reference<decltype(x+y)>::value;
```

Biće *true* akko operator + nad operandima *x* i *y* vraća tip reference

Tipovi

- ❖ Neko ime (identifikator) se može deklarisati kao ime tipa na jedan od sledećih načina:
 - deklaracijom klase
 - deklaracijom enumeracije (*enum*)
 - *typedef* deklaracijom
 - deklaracijom sinonima za definisani tip
- ❖ *typedef* deklaracija ima isti oblik kao kada se imenom deklariše entitet datog tipa, samo što ime ne deklariše takav entitet, već novo ime (sinonim) za taj isti tip (ne definiše se novi tip); tipično se koristi za konciznije pisanje složenih tipova:

```
typedef X* (*PXTransFun)(X*);  
PXTransFun vtable[N];
```

Ovo je složen tip pokazivača na funkciju koja prima X^* i vraća X^* ; *PXTransFun* je sinonim za taj tip; *vtable* je niz takvih pokazivača

- ❖ Potpuno isto se postiže sledećom deklaracijom sinonima za tip:

```
using PXTransFun = X*(*)(X*);
```

- ❖ Na mnogim mestima može se navesti neki tip koji nema ime, na primer kao odredišni tip operatora eksplicitne konverzije (*cast*); takav bezimeni tip piše se isto kao i tip u deklaraciji *typedef*, samo bez imena, odnosno kao u deklaraciji sinonima i naziva se *type-id*

Konverzije

- ❖ Konverzija je pretvaranje vrednosti (*value*) jednog tipa u vrednost drugog tipa
- ❖ Konverzija može, a ne mora imati bilo kakve efekte u vreme izvršavanja
- ❖ Konverzija može imati efekta samo na prevođenje, bez ikakvog efekta (same konverzije) za vreme izvršavanja (prevodilac ne generiše nikakav poseban kod koji vrši tu konverziju); prevodilac u vreme prevođenja može prosto smatrati da nakon konverzije ista vrednost ima drugi tip, i tu vrednost nadalje drugačije tumačiti; na primer: konverzija objekta nekonstantnog tipa u isti takav konstantni tip, konverzija pokazivača na izvedenu klasu u pokazivač na jedinu (i dostupnu) osnovnu klasu (i dalje ukazuje na isti objekat), konverzija iz *false* u 0, konverzija iz *char* u *int* i slično
- ❖ Konverzija može imati efekat u vreme izvršavanja, pa prevodilac generiše odgovarajući kod koji za vreme izvršavanja vrši tu konverziju, na primer tako što:
 - vrednost jednog tipa mora da dobije drugačiju binarnu predstavu za vreme izvršavanja, a možda i konceptualno drugačiju vrednost: na primer, konverzija racionalnog broja tipa *double* u ceo broj *int* promeniće svakako svoju binarnu predstavu (iz pokretnog zareza u drugi komplement), ali možda i vrednost (bez decimala)
 - konverzijom može nastati potpuno novi objekat, pa i objekat klase

Konverzije

- ❖ Konverzija može biti definisana (u smislu da je propisan način kako se vrši, za koje tipove i kada se može vršiti i šta joj je semantika):
 - samim pravilima jezika: tzv. *ugrađena (built-in)*; na primer, konverzija *char* u *int*, ili konverzija pokazivača / reference na izvedenu klasu u pokazivač / referencu na dostupnu osnovnu klasu
 - u programu: tzv. *korisnički definisane konverzije (user-defined conversion)*, za korisničke tipove, odnosno klase
- ❖ Konverzija se može raditi:
 - *implicitno*, što znači da je radi prevodilac bez ikakvg posebnog eksplicitnog zahteva programera; ako je na datom mestu potrebna konverzija, i ako je ona definisana tako da se može raditi implicitno, prevodilac će je izvršiti; ovakve konverzije su samo one koje se smatraju "bezbednim", u smislu da na neki način čuvaju korektnost semantike programa
 - eksplicitno zahtevane u programu jednim od sledećih operatora:
 - *const_cast*
 - *static_cast*
 - *dynamic_cast*
 - *reinterpret_cast*
 - operator *cast* nasleđen iz jezika C: *(type)expression*

Konverzije

❖ Na primer:

```
int i = 'a';
```

Implicitna, ugrađena konverzija iz *char* u *int*: literal 'a' je tipa *char*, a *i* je tipa *int*

```
float f = 5.6;
```

Implicitna, ugrađena konverzija iz *double* u *float*: literal 5.6 je tipa *double*, a *f* je tipa *float*

```
float g = (float)5.6;
```

Eksplisitna, ugrađena konverzija iz *double* u *float*: literal 5.6 je tipa *double*

```
int a = 5;
```

Redosled izvršavanja operacija i konverzija (sve su implicitne, ugrađene):

```
double d = 1.5e-3;
```

1. *a+d*: pošto je *a* tipa *int*, a *d* tipa *double*, *a* se konvertuje u *double*; rezultat je tipa *double*

```
float h = a+d-1;
```

2. ...-1: pošto je levi operand tipa *double*, a 1 tipa *int*, 1 se konvertuje u *double*; rezultat je tipa *double*

```
int b = a+'0';
```

3. *h* je tipa *float*, a inicijalizuje se izrazom tipa *double*; *double* se konvertuje u *float*

a+'0': pošto je *a* tipa *int*, a '0' tipa *char*, '0' se konvertuje u *int*; rezultat je tipa *int* (implicitna, standardna konverzija); *b* tipa *int* i inicijalizuje se izrazom tipa *int*, pa nema konverzije

```
Base* pB = new Derived;
```

Implicitna, standardna konverzija pokazivača na izvedenu klasu u pokazivač na osnovnu klasu (podrška pravilu supstitucije)

```
Derived* pD = (Derived*)pB;
```

Standardna konverzija pokazivača na osnovnu klasu u pokazivač na izvedenu klasu ne može implicitno, jer nije pouzdana

Konverzije

- ❖ Implicitna konverzija se vrši kad god se izraz jednog tipa T_1 koristi u kontekstu u kom se ne prihvata taj tip, nego se prihvata neki drugi tip T_2 , a to su sledeće situacije:
 - kada se izraz tipa T_1 koristi kao stvarni argument poziva funkcije čiji je odgovarajući formalni parametar deklarisan sa tipom T_2
 - kada u funkciji koja ima deklarisan povratni tip T_2 naredba *return* vraća izraz tipa T_1
 - kada se izraz tipa T_1 koristi kao operand operatora koji očekuje taj operand tipa T_2
 - kada se inicijalizuje novi objekat tipa T_2 izrazom tipa T_1
 - kada je izraz upotrebljen u narebi *switch* tipa T_1 koji nije integralan tip (T_2 je integralan tip)
 - kada je izraz upotrebljen u narebi *if* ili u naredbi petlje (naredbe *for*, *do*, *while*) tipa T_1 koji nije tip *bool* (T_2 je tip *bool*)
- ❖ Implicitna konverzija može se vršiti i tranzitivno, tj. prevodilac može izvršiti i čitav niz sukcesivnih implicitnih konverzija, ali taj niz ne može sadržati više od jedne korisnički definisane konverzije — sve ostale moraju biti tzv. *standardne konverzije* (*standard conversion*)

Konverzije

- ❖ Eksplisitna konverzija operatorom *const_cast* dozvoljena je samo za ukidanje cv-kvalifikacije (ili dodavanje takve kvalifikacije, ali se to može raditi i implicitno), najčešće za pokazivače i reference:

```
const Clock* pcc = new Clock;  
Clock* pc = const_cast<Clock*>cc;
```

Ovde implicitna konverzija nije moguća

- ❖ Eksplisitna konverzija operatorom *static_cast* dozvoljena je za skoro sve ugrađene konverzije, svakako za one koje se mogu raditi implicitno, kao i za one koje se smatraju nebezbednim (ali imaju bilo kakvog smisla), s tim da *ne može* da ukida cv-kvalifikaciju; u svakom slučaju, za vreme izvršavanja ne vrši se nikakva provera korektnosti, npr. u smislu da li je objekat koji se krije iza pokazivača / reference zaista odredišnog tipa (npr. za *downcast* pokazivača i referenci):

```
Base* pb = new Derived;  
Derived* pd = static_cast<Derived*>pb;
```

Ovde implicitna konverzija nije moguća

- ❖ Eksplisitna konverzija operatorom *dynamic_cast* namenjena je za konverzije između tipova u hijerarhiji klasa, i to nagore, nadole ili bočno, s tim da ona u vreme izvršavanja proverava tip polimorfног objekta (objekta klase sa bar jednom virtuelnom funkcijom), i u slučaju da izvořna vrednost nije odredišnog tipa, vraća *null* vrednost za pokazivače ili baca izuzetak za reference:

```
Base* pb = ...;  
Derived* pd = dynamic_cast<Derived*>pb;
```

Statička konverzija ne bi garantovala da *pd* zaista ukazuje na objekat klase *Derived*

- ❖ Eksplisitna konverzija operatorom *reinterpret_cast* može da vrši konverzije skoro svih tipova, osim da ukida cv-kvalifikacije, s tim da se za nju ne generišu nikakve mašinske instrukcije; jednostavno se binarna vrednost operanda tumači na drugačiji način, u skladu sa odredišnim tipom; veoma je osetljiva i retko potrebna

Konverzije

- ❖ Kod eksplisitne konverzije operatorom *cast* nasleđenim iz jezika C
(new_type)expression prevodilac pokušava redom sledeće konverzije izvorišnog u odredišni tip, i primenjuje prvu koja odgovara datim tipovima, čak i ako ona nije dozvoljena (u kom slučaju prijavljuje grešku):
 - a) *const_cast<new_type>(expression)*
 - b) *static_cast<new_type>(expression)*, sa tim proširenjem što pokazivač/referenca na izvedenu klasu može da se konvertuje u pokazivač/referencu na osnovnu klasu, čak i ako ta osnovna klasa nije dostupna (to znači, ne uzima se u obzir to što je osnovna klasa izvedena kao *private* ili *protected*)
 - c) *static_cast* (sa proširenjem), iza koje sledi *const_cast*
 - d) *reinterpret_cast<new_type>(expression)*
 - e) *reinterpret_cast*, iza koje sledi *const_cast*
- ❖ Prema tome, ovim operatorom mogu se vršiti sve konverzije navedenih operatora specifičnih konverzija, osim operatora *dynamic_cast*, ali se ipak preporučuje korišćenje upravo nekog od ovih operatora umesto operatora *cast* iz sledećih razloga:
 - upotrebom nekog od navedenih specifičnih operatora naglašava se namera date konverzije, tj. njena svrha i cilj, pa program može biti razumljiviji
 - ukoliko ta namera nije korektna, u smislu da zahtevana konverzija nije dozvoljena tim operatom za date tipove, prevodilac će programeru ukazati na grešku (dok bi operator *cast* “ćutke” možda primenio neku drugu, nenameravanu konverziju)

Fundamentalni tipovi *void*, *nullptr_t* i *bool*

- ❖ Tip *void* označava prazan skup vrednosti; on je uvek nekompletan tip: ne postoje objekti ovog tipa, nizovi objekata ovog tipa, niti reference na ovaj tip; postoje pokazivači na ovaj tip i funkcije koje imaju ovaj tip kao povratni za potprograme koji ne vraćaju rezultat (procedure)
- ❖ Tip *nullptr_t* iz standardne biblioteke je tip pokazivačkog literala *nullptr* koji ne ukazuje ni na šta; on je poseban tip, nije pokazivački tip i različit je od bilo kog drugog pokazivačkog tipa:

```
typedef decltype(nullptr) nullptr_t;
```

- ❖ Ovaj tip se može konvertovati u bilo koji pokazivački tip; rezultat je *null* pokazivač - pokazivač koji ne ukazuje ni na šta:

```
Clock* pc = nullptr;
```

- ❖ Tip *bool* je Bulov (logički) tip koji obuhvata samo dve vrednosti: *true* ili *false*; vrednost *sizeof(bool)* zavisi od implementacije i može biti različita od 1; *true* i *false* su literalni ovog tipa
- ❖ Vrednost aritmetičkog tipa, enumeracije ili pokazivača može se konvertovati u tip *bool*: vrednost nula (za aritmetičke tipove), *null* vrednost pokazivača i *nullptr* se konvertuju u *false*; sve druge vrednosti se konvertuju u *true*:

```
if (pc) ...
```

Isto što i *pc!=nullptr*

```
if (!pc) ...
```

Isto što i *pc==nullptr*

Fundamentalni znakovni tipovi

Znakovni tipovi:

- ❖ *signed char*: tip znakova sa reprezentacijom koja uključuje i aritmetički znak
- ❖ *unsigned char*: tip znakova sa reprezentacijom bez aritmetičkog znaka; koristi se i za inspekciju reprezentacije objekata, odnosno predstavljanje “presne memorije”
- ❖ *char*: tip znakova sa reprezentacijom koja se na ciljnom sistemu može najefikasnije procesirati; ima istu reprezentaciju i poravnanje ili kao *signed char* ili kao *unsigned char* (zavisi od sistema), ali je uvek različit tip; stringovi znakova sa višebajtnom reprezentacijom koriste ovaj tip za predstavu jedinice kodovanja; ovi znakovni tipovi su dovoljno veliki da predstave bilo koji jedinicu kodovanja UTF-8
- ❖ *wchar_t*: tip za “široku” reprezentaciju znakova; dovoljno velik da prihvati bilo koji podržanu kodnu tačku (*code point*) - 32 bita na sistemima koji podržavaju *Unicode* (sa izuzetkom sistema Windows, gde je *wchar_t* 16-bitni i sadrži UTF-16 kodne jedinice); ima istu veličinu, označenost i poravnanje kao jedan od celobrojnih tipova, ali je različit tip
- ❖ *char16_t*: tip za reprezentaciju UTF-16 znakova (16-bitni)
- ❖ *char32_t*: tip za reprezentaciju UTF-32 znakova (32-bitni)

Fundamentalni znakovni tipovi

❖ Znakovni literali:

- `'a'` — običan, “uski” znak tipa *char*; ako ne može da stane u jedan bajt, tip je *int* i vrednost koja zavisi od implementacije
- `u8'a'` — UTF-8 znak tipa *char*, ako se može predstaviti kao jedna kodna jedinica u UTF-8
- `u'貓'` (ali ne `u'🐈'`) — UCS-2 znak tipa *char16_t*, sa vrednošću datog *Unicode* znaka
- `U'貓'` ili `U'🐈'` — UCS-4 znak tipa *char32_t*, sa vrednošću datog *Unicode* znaka
- `L'貓'` ili `L'β'` — “široki” znak tipa *wchar_t*
- `'AB'` — kada literal sadrži dva ili više znakova unutar apostrofa, predstavlja literal tipa *int* sa vrednošću koja zavisi od implementacije

Fundamentalni znakovni tipovi

- ❖ Znak unutar apostrofa ne može biti ' (apostrof), \ (obrnuta kosa crta, *backslash*) ili znak za novi red (*newline*), ali može biti znak definisan tzv. *escape sekvencom*, koja predstavlja sekvencu koja počinje znakom \; neke češće korišćene *escape sekvence*:
 - \' — apostrof (*qoute*, ASCII kod 27h)
 - \" — navodnik (*double quote*, ASCII kod 22h)
 - \\ — obrnuta kosa crta (*backslash*, ASCII kod 5Ch)
 - \n — znak za novi red (*newline*, ASCII kod 0Ah); na znakovnim izlaznim uređajima interpretira se kao novi red
 - \t — horizontalni tabulator (*horizontal tab*, ASCII kod 09h)
 - \nnn — znak sa navedenim jednobajtnim kodom datim u oktalnom zapisu sa do tri oktalne cifre nnn;
najčešći je znak \0 — znak kojim se, po konvenciji iz jezika C, završavaju nizovi znakova kakve očekuju sve funkcije iz standardne C biblioteke (*null-terminated strings*), kao i string-literali
 - \xnn — znak sa datim jednobajtnim kodom datim u heksadecimalnim zapisu sa do dve heksadecimalne cifre nn

Fundamentalni celobrojni tipovi

- ❖ Osnovni celobrojni tip je tip *int*. Ako nema modifikatora veličine, garantovano je da *int* ima širinu od najmanje 16 bita. Na današnjim 32-bitnim i 64-bitnim sistemima skoro je sigurno da *int* ima širinu od najmanje 32 bita
- ❖ Modifikatori označenosti:
 - *signed* — tip ima označenu reprezentaciju; podrazumeva se ako nema modifikatora označenosti; prilikom proširenja se proširuje znakom
 - *unsigned* — tip ima neoznačenu reprezentaciju; prilikom proširenja proširuje se nulama
- ❖ Modifikatori veličine:
 - *short* — tip je optimizovane veličine i ima širinu od najmanje 16 bita
 - *long* — tip ima širinu od najmanje 32 bita
 - *long long* — tip ima širinu od najmanje 64 bita

Fundamentalni celobrojni tipovi

- ❖ Modifikatori za označenost i veličinu mogu da se kombinuju u bilo kom poretku, s tim da je dozvoljen samo po jedan modifikator svake vrste. Ključna reč *int* može da se izostavi ako se koristi bilo koji od modifikatora. Kao i za druge specifikatore tipa, dozvoljeni su svi redosledi: *unsigned long long int*, *long int unsigned long* i *long unsigned long* predstavljaju isti tip
- ❖ Treba primetiti da širine tipova u bitima nisu fiksno određene standardom (osim minimalnih) i mogu se razlikovati od sistema do sistema. Tip *int* je takav da se na svakoj mašini celobrojne operacije izvršavaju najefikasnije sa tim tipom. Osim toga, garantovano je i sledeće:

```
1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)
```

- ❖ Termin *bajta* (*byte*) na jeziku C++ ima značenje jedne jedinice prostora (*sizeof(char)*) i može biti različit od 8 bita; može biti čak i 64 bita u ekstremnom slučaju, dozvoljenom standardom, kada svi celobrojni i znakovni tipovi imaju širinu od 64 bita
- ❖ Program koji zavisi od konkretnih veličina tipova svakako nije prenosiv!

Fundamentalni celobrojni tipovi

- ❖ Celobrojni literali:
 - niz cifara koji ne počinje cifrom 0: decimalni zapis
 - niz cifara od 0 do 7 koji počinje cifrom 0: oktalni zapis
 - niz cifara i slova od *a* do *f* ili od *A* do *H* ispred koga su znaci *0x* ili *0X*: heksadecimalni zapis
 - niz cifara 0 ili 1 ispred koga su znaci *0b* ili *0B*: binarni zapis
- ❖ Literal može imati jedan od sledećih sufiksa ili oba, koji utiču na njegov tip:
 - znak *u* ili *U*: neoznačen tip
 - *l* ili *L* za *long*, odnosno *ll* ili *LL* za *long long*
- ❖ Unutar literala mogu da se nađu apostrofi ', koje prevodilac potpuno ignoriše, a koji mogu da pomognu u čitanju (npr. razdvajanje hiljada)

Fundamentalni celobrojni tipovi

- ❖ Tip celobrojnog literalna je prvi tip koji može da prihvati datu vrednost, počev od tipa *int*, ili od tipa *long* ako je dat sufiks *l* ili *L*, odnosno od tipa *long long* ako je dat sufiks *ll* ili *LL*, i neoznačen ako je dat sufiks *u* ili *U*
- ❖ Na primer:

```
int d = 42;  
int o = 052;  
int x = 0x2a;  
int X = 0X2A;  
int b = 0b101010;
```

```
unsigned long long l1 = 18446744073709550592ull;  
unsigned long long l2 = 18'446'744'073'709'550'592LLU;
```

Svi imaju istu vrednost

Oba imaju istu vrednost

Fundamentalni celobrojni tipovi

- ❖ Aritmetički operatori primaju operande najmanje tipa *int*; ukoliko je neki operand manjeg tipa, implicitno se konvertuje u tip *int*; ako je jedan operand binarnog operatora većeg tipa od drugog, onaj manji se implicitno konvertuje u tip većeg. Ove konverzije čuvaju vrednost. Ovo se naziva *integralna promocija* (*integral promotion*):
 - *signed char* ili *signed short* može da se konvertuje u *int*
 - *unsigned char* ili *unsigned short* može se konvertovati u *int* ako ovaj može da prihvati ceo opseg vrednosti, odnosno u *unsigned int* u suprotnom
 - *char* se može konvertovati u *int* ili *unsigned int* u zavisnosti od njegove interpretacije (*signed char* ili *unsigned char*)
 - *wchar_t*, *char16_t* i *char32_t* mogu da se konvertuju u prvi od sledećih tipova koji može da prihvati ceo opseg vrednosti: *int*, *unsigned int*, *long*, *unsigned long*, *long long*, *unsigned long long*
 - tip *bool* može da se konvertuje u tip *int*, pri čemu vrednosti *false* postaje 0, a vrednost *true* postaje 1

Fundamentalni celobrojni tipovi

- ❖ Osim integralne promocije, postoje i implicitne konverzije iz bilo kog integralnog tipa u bilo koji drugi integralni tip. Ako odredišni tip može da prihvati ceo opseg vrednosti izvorišnog tipa, vrednost se čuva; u suprotnom, vrednost se može izgubiti:
 - ako je odredišni tip neoznačen: u zavisnosti od toga da li je odredišni tip širi ili uži od izvorišnog, označeni brojevi se proširuju znakom ili odsecaju, a neoznačeni brojevi se proširuju nulama ili odsecaju, respektivno
 - ako je odredišni tip označen, vrednost se ne menja ako se izvorišni broj može predstaviti odredišnim tipom, u suprotnom je vrednost zavisna od implementacije
- ❖ Na primer:

```
char c1 = '0';
for (int i=0; i<10; i++) {
    char c2 = c1+i;
    ...
}
```

1. U operaciji `+ vrši se integralna promocija vrednosti c1 tipa char u int;` rezultat operacije `+ je tipa int`
2. Pri inicijalizaciji `c2` vrši se implicitna konverzija iz `int` u `char` (može izgubiti vrednost)
Pošto u svim sistemima kodovanja znakovi za cifre imaju susedne kodove, `c2` će uzimati vrednost znaka za cifre od 0 do 9

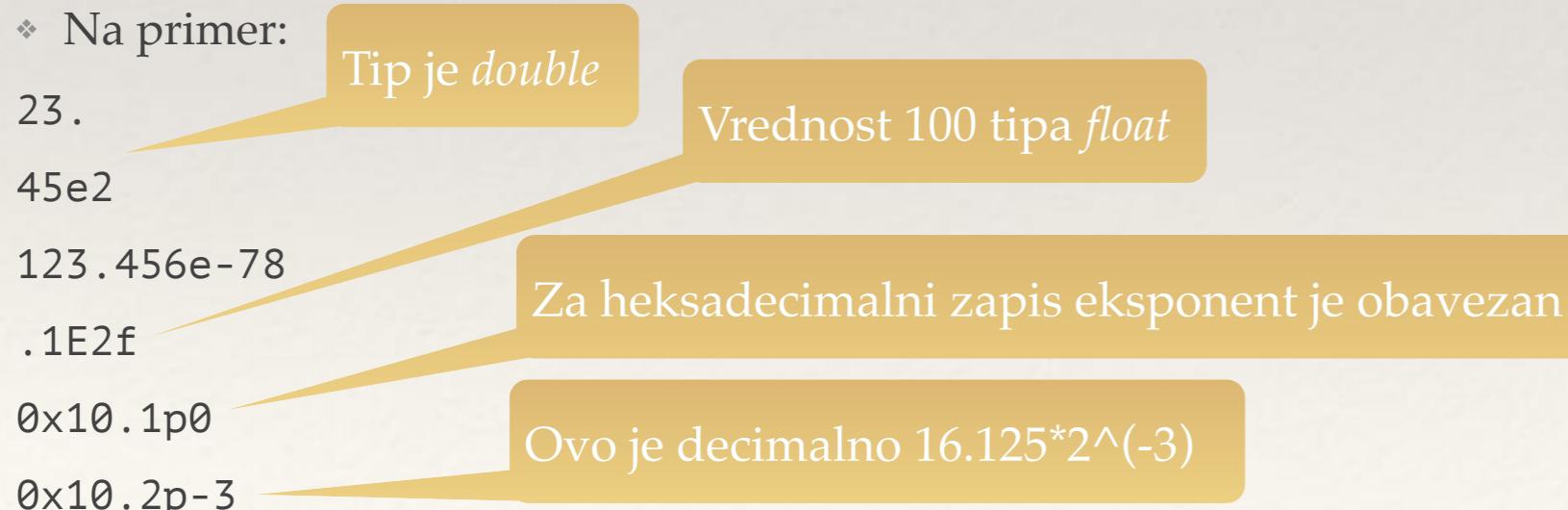
Fundamentalni tipovi pokretnog zareza

- ❖ Tipovi racionalnih brojeva u pokretnom zarezu:
 - *float* — racionalni broj u pokretnom zarezu jednostrukke tačnosti; obično 32-bitni broj u pokretnom zarezu prema standardu IEEE-754
 - *double* — racionalni broj u pokretnom zarezu dvostrukke tačnosti; obično 64-bitni broj u pokretnom zarezu prema standardu IEEE-754
 - *long double* — racionalni broj u pokretnom zarezu proširene preciznosti; ne mora da odgovara tipovima iz standarda IEEE-754; obično 80-bitni broj u pokretnom zarezu
- ❖ Ako se razlikuju, veći tip ima veći opseg i veću preciznost; uvek važi:
`sizeof(float) <= sizeof(double) <= sizeof(long double)`

Fundamentalni tipovi pokretnog zareza

- ❖ Literali mogu da sadrže:
 - mantisu i eksponent zapisane decimalno ili heksadecimalno (počinju znacima $0x$ ili $0X$)
 - decimalnu tačku
 - eksponent iza znaka e ili E za decimalne, odnosno p ili P za heksadecimalne zapise (da bi se razlikovao od e i E koji su heksadecimalne cifre); eksponent za heksadecimalne zapise je obavezan i predstavlja stepen dvojke, ne stepen 10
 - znak mantise i znak eksponenta
 - sufiks za tip: f ili F za *float* i l ili L za *long double*
 - apostrofe $',$ koje prevodilac ignoriše (odvajaju hiljade)
- ❖ Tip ovakvog literala je podrazumevano *double*, osim ako ima sufiks za tip

- ❖ Na primer:



Fundamentalni tipovi pokretnog zareza

- ❖ Postoje sledeće implicitne konverzije definisane jezikom:
 - *float* se konvertuje u *double* promocijom (*floating-point promotion*) koja se vrši u aritmetičkim operacijama, slično kao za celobrojne tipove, uz očuvanje vrednosti
 - bilo koji tip racionalnog broja u pokretnom zarezu može se konvertovati implicitno u bilo koji drugi takav tip, sa potencijalnim gubitkom vrednosti ili tačnosti
 - bilo koji tip racionalnog broja u pokretnom zarezu može se konvertovati implicitno u bilo koji celobrojni tip, uz potencijalno odsecanje
 - bilo koji celobrojni tip može se konvertovati u bilo koji tip racionalnog broja u pokretnom zarezu, uz eventualni gubitak vrednosti ako se ta vrednost ne može tačno predstaviti u pokretnom zarezu

Nabranja (enumeracije)

- ❖ *Nabranje* (enumeracija, *enumeration*) je poseban tip čije su vrednosti ograničene na konačan skup vrednosti, a koje predstavljaju simboličke imenovane konstante i koje se nazivaju *enumeratori* (*enumerators*)

- ❖ Na primer:

```
enum Bool { FALSE, TRUE };  
enum Reply { YES, NO, CANCEL };  
enum Status { initiated, suspended, committed, canceled, failed };  
Status s1 = initiated;  
...  
if (s1==committed) ...
```

Tip enumeracije

Enumerator

- ❖ Za enumeraciju se može i eksplicitno definisati “potporni tip”, tip koji “leži ispod” njega (*undelying type*); to je tip se koristi kao tip kojim se skladište vrednosti ovog tipa (podrazumeva se *int*), pod uslovom da može da prihvati sve nabrojane enumeratore:

```
enum Status : unsigned short { initiated, suspended, committed, canceled,  
failed };  
Status s1 = initiated;  
...  
if (s1==committed) ...
```

Potporni tip

Nabranja (enumeracije)

- ❖ Enumeracije *nisu* fundamentalan, celobrojan tip (nisu isto što i *int*), iako postoji implicitna konverzija iz tipa enumeracije u celobrojni tip
- ❖ Enumeracije treba koristiti kad god postoji potreba za tipom podataka koji apstrahuje koncept iz domena problema (ili rešenja) i čije instance uzimaju vrednosti iz konačnog, diskretnog skupa vrednosti koje se simbolički imenuju
- ❖ Nije dobro koristiti fundamentalan, celobrojni tip umesto enumeracije u ovakvim slučajevima, zato što:
 - je program tada teži za razumevanje, jer je teško razumeti značenje celobrojnih konstanti i nije ih lako razlikovati od drugog značenja istih konstanti (npr. kao zaista instanci matematičkog koncepta celog broja)
 - za vrednosti apstraktnog tipa enumeracije po pravilu nisu definisane (konceptualno) aritmetičke operacije (sabiranje, oduzimanje, množenje, deljenje itd)
 - program je manje fleksibilan: ako je potrebno promeniti odluku o implementaciji apstraktnog tipa, ili promeniti preslikavanje na celobrojne vrednosti, potrebno je zameniti sve pojave tih celobrojnih konstanti i pažljivo ih razlikovati od upotrebe istih vrednosti sa potpuno različitim značenjem
- ❖ Neke tipične upotrebe:
 - statusi operacija (funkcija)
 - stanja objekata
 - komande, signali i slično
 - statusi ili komande hardverskih uređaja
 - tip sa malim, konačnim i konstantnim, predefinisanim skupom instanci (npr. dani u nedelji, meseci u godini)

Nabranja (enumeracije)

- ❖ Postoji implicitna konverzija iz tipa enumeracije u celobrojni tip, koja spada u skup celobrojnih promocija. Vrednost tipa enumeracije konvertuje se u:
 - prvi od sledećih tipova koji može da prihvati ceo opseg vrednosti te enumeracije, ukoliko nije eksplicitno zadat potporni tip: *int*, *unsigned int*, *long*, *unsigned long*, *long long*, or *unsigned long long*
 - u potporni tip enumeracije, kao i u tip u koji se on može promovisati, ako je potporni tip eksplicitno zadat
- ❖ Međutim, takva konverzija najčešće nije adekvatna, jer meša tipove i suprotna je navedenim principima. Samo u posebnim slučajevima koje treba dobro lokalizovati i enkapsulirati, ovo ima smisla. Na primer:

```
enum Status { initiated, suspended, committed, canceled, failed };  
int i = suspended; // i gets the value 1  
Status s = canceled;  
...  
if (s==4) ... // conversion Status->int evaluates to true if s==failed
```

Ovo nije dobar način programiranja!

Nabranja (enumeracije)

- ❖ Vrednost koja se dobija ovom konverzijom definisana je na sledeći način:
 - ona koja je eksplicitno zadata inicijalizatorom (koji mora biti konstantni izraz)
 - ako inicijalizator nije zadat, jednaka je vrednosti prethodnog enumeratora plus 1; vrednost prvog enumeratora je podrazumevano 0, ako nije eksplicitno zadat inicijalizator:

```
enum Status { initiated = 1, suspended, //suspended == 2  
committed, canceled = 5,  
failed = committed+4 }; // failed == 7
```

- ❖ Suprotna konverzija iz celobrojnog tipa u tip enumeracije može se raditi samo eksplicitno, ne i implicitno; ukoliko data celobrojna vrednost ne spada u skup vrednosti enumeratora, ponašanje je nedefinisano; ovakve konverzije svakako nisu preporučljive:

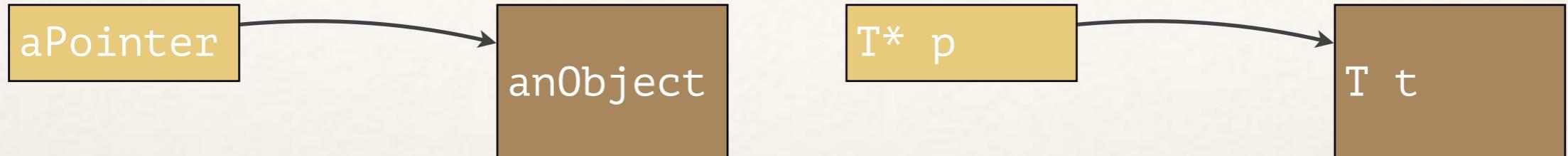
```
Status s = 3;
```

Greška u prevodenju

```
Status s = static_cast<Status>(3);
```

Ovo nije dobar način programiranja!

Pokazivači na objekte



- ❖ Pokazivač (*pointer*) na objekat je objekat koji *ukazuje* (*points to*) na neki drugi objekat; konceptualno, ovo predstavlja *jednosmernu* (*unidirectional*, *unidirectional*) vezu pokazivača ka objektu
- ❖ Ako je T neki objektni tip, tip pokazivača na tip T je tip T^* ; pokazivač ovog tipa može se inicijalizovati rezultatom, ili mu se može dodeliti rezultat operadora *uzimanja adrese* (*adress-of*) čiji je operand objekat tipa T ; tada pokazivač ukazuje na taj objekat:

$T \ t;$
 $T^* \ p \ = \ \&t;$ p ukazuje na t

- ❖ Rezultat operadora *indirekcije* (operator $*$) čiji je operand pokazivač na T odnosi se na objekat na koga taj pokazivač ukazuje:

$\dots *p \dots$ *p predstavlja objekat na koji p ukazuje

Pokazivači na objekte

- ❖ Na primer:

```
int i=0, j=0;  
int* pi = &i;  
*pi = 1;  
pi = &j;  
*pi = 2;
```

pi ukazuje na i

i dobija vrednost 1

pi sada ukazuje na j

j dobija vrednost 2

- ❖ Pošto su i pokazivači i nizovi objekti, pokazivač može ukazivati i na njih:

```
int i=0, j=0;  
int* pi=&i;  
int** ppi;  
ppi=&pi;  
*pi=1;  
**ppi=2;  
*ppi=&j;  
ppi=&i;  
  
int a[10];  
int (*pa)[10] = &a;
```

ppi je tipa pokazivač na - pokazivač na - int

ppi ukazuje na pi

i dobija vrednost 1

Izračunava se: $\ast(\ast\text{ppi})$; i dobija vrednost 2

pi sada ukazuje na j

pa je tipa pokazivač na - niz od 10 elemenata tipa - int i ukazuje na niz a koji je baš takvog tipa

Greška u prevodenju: ppi je tipa `int**`, a `&i` je tipa `int*`; ne postoji ova implicitna konv.

- ❖ Ako pokazivač ukazuje na objekat klasnog tipa, može se pristupiti članu tog objekta preko operatora `->`; ukoliko se radi o objektu polimorfne klase (sa bar jednom virtuelnom funkcijom), aktivira se polimorfizam:

```
Clock* pc = new ClockWithDates;  
pc->tick();  
(*pc).tick();
```

Ovo je isto

Pokazivači na objekte

- ❖ Pokazivač tipa *void** može da ukazuje na bilo koji objekat: postoji implicitna konverzija iz bilo kog pokazivačkog tipa u tip *void**; sa objektom na koji ukazuje ovaj pokazivač ne može se raditi ništa
- ❖ Ovakav pokazivač se vrlo ograničeno upotrebljava, tipično u starim C interfejsima, gde se prihvata pokazivač na bilo šta; na primer, nit (*thread*) u biblioteci POSIX kreira se zadavanjem argumenta funkcije koji je pokazivač tipa *void**
- ❖ Da bi se preko tog pokazivača uradilo nešto sa onim na šta on ukazuje, mora se izvršiti eksplicitna konverzija na odredišni tip, s tim da odgovornost za ispravnost te konverzije nosi programer:

```
void DocumentSaver::run (void* p) {  
    DocumentSaver* ds = static_cast<DocumentSaver*>(p);  
    if (ds) ds->save();  
    delete ds;  
}
```

Pokazivači na objekte

- ❖ Pokazivač može da ne ukazuje ni na šta; vrednost takvog pokazivača naziva se *null* vrednost
- ❖ Simbolička vrednost *null* označava se kao *nullptr*; pre njegovog uvođenja, kao simbolička vrednost *null* koristio se:
 - literal *0*, sa simboličkom vrednošću *null* za pokazivače
 - makro *NULL* defnisan u standardnom zaglavlju jezika C
- ❖ Ove stare simboličke vrednosti mogu i dalje da se upotrebljavaju, ali one mogu da dovedu do drugačijeg tumačenja tipa u različitim kontekstima (kao celobrojni literal *0*), dok je *nullptr* različitog tipa
- ❖ Pokazivač se može:
 - inicijalizovati vrednošću *null* ili mu se dodeliti ta vrednost:

```
Clock* pc1 = nullptr;
```

- porediti na jednakost/nejednakost sa tom vrednošću
- ```
if (pc1==nullptr)...
```
- ```
if (pc1!=nullptr)...
```

Pokazivači na objekte

- ❖ Na jeziku C++, kao i na jeziku C, vrednosti pokazivača predstavljaju adrese u memoriji. Međutim, ovo je stvar implementacije i konkretna vrednost pokazivača ne treba da bude od interesa, osim u prilikama kada se program debaguje i prati ta vrednost (u smislu jednakosti ili promena)
- ❖ Svaka vrednost tipa pokazivača na objekat može biti jedna od sledećih:
 - takva da ukazuje na neki objekat, ili
 - takva da ukazuje iza kraja nekog objekta, ili
 - *null* vrednost tog tipa, ili
 - invalidna (nekorektna) vrednost pokazivača
- ❖ Vrednost pokazivača koji ukazuje na neki objekat predstavlja adresu prvog bajta u memoriji koji zauzima taj objekat. Vrednost pokazivača koji ukazuje iza nekog objekta predstavlja adresu prvog bajta iza dela memorije koju zauzima taj objekat
- ❖ Dva pokazivača koji predstavljaju istu adresu mogu imati različite binarne vrednosti kod nekih arhitektura, ali njihovo poređenje na jednakost daje *true*
- ❖ Prva tri slučaja su korektne situacije, odnosno vrednost pokazivača je validna, iako se samo u prvom slučaju ta vrednost može dereferencirati (pristupiti objektu na koji ta vrednost ukazuje) sa definisanim značenjem; u svim drugim situacijama pristup do objekta ima nepredvidivo ponašanje: može izazvati hardverski izuzetak (npr. zbog pristupa delu memorije koji nije dozvoljen) ili "tiho" nepredviđeno ponašanje (bag)
- ❖ Pokazivač ne sadrži nikakvu dodatnu informaciju osim adrese. Za vreme izvršavanja ne vrše se nikakve dodatne provere validnosti vrednosti pokazivača. Zbog toga su pokazivači veoma osetljivi na ozbiljne probleme u programu i programer je odgovoran da obezbedi ispravnost pokazivača i njihove upotrebe. Ovo je jedan od najosetljivijih delova jezika C i C++

Pokazivači na objekte

Prema tome, pokazivači na jezicima C i C++ su veoma nebezbedni i mogu da uzrokuju veoma neprijatne greške. Evo nekih tipičnih problema sa pokazivačima:

- ❖ Neispravan (invalidan) pokazivač:

- Uzrok: pristup do objekta preko pokazivača koji ima invalidnu vrednost (predstavlja adresu na kojoj se ne nalazi objekat datog tipa); ni pokazivač ni objekat na koga on ukazuje se ne proveravaju za vreme izvršavanja:

```
Clock* p;  
p->tick();
```

Ovaj pokazivač ima nedefinisanu početnu vrednost

Vrlo moguća greška za vreme izvršavanja

- Efekti:

- moguća greška za vreme izvršavanja, npr. izuzetak koji podiže hardver i prosleđuje operativni sistem, zbog nelegalnog pristupa delu memorije, tipično završava gašenjem programa
 - nedefinisano ponašanje (“tiha” greška): program nastavlja dalje sa potpuno nedefinisanim ponašanjem i na kraju možda negde kasnije izaziva izuzetak
- Tipične situacije u kojima nastaje:
 - neinicijalizovan pokazivač: gornji primer

Pokazivači na objekte

- korumpiran pokazivač:

```
int a[5]; // array of 5 integers in range [0..4]
```

```
Clock* p = new Clock(...);
```

p je verovatno alociran u memoriji odmah iza niza a

U jednom kontekstu:

```
for (int i = 0; i<=5; i++) // i:=0..5  
    a[i]=i;
```

Za poslednju vrednost $i==5$ pregaziće pokazivač p

A onda u nekom potpuno drugom kontekstu:

```
p->tick();
```

p najverovatnije ima invalidnu vrednost

- Zaštita — pažljivo programiranje:
 - obavezna inicijalizacija, posebno pokazivača
 - provera granica prilikom pristupa nizovima, posebno baferima u koje se učitavaju podaci sa potencijalno neograničenim ili prevelikim sekvencama (provera na prekoračenje niza ili bafera, *buffer overflow*)

Pokazivači na objekte

- ❖ Dereferenciranje *null* vrednosti pokazivača:

- Uzrok: pristup do objekta preko pokazivača koji ima *null* vrednost; vrednost se ne proverava za vreme izvršavanja prilikom dereferenciranja:

```
Clock* pc = nullptr;  
pc->tick();
```

Greška za vreme izvršavanja

- Efekti:
 - po pravilu greška za vreme izvršavanja, npr. izuzetak koji podiže hardver i prosleđuje operativni sistem (zbog nelegalnog pristupa delu memorije, tipično završava gašenjem programa)
- Tipične situacije u kojima nastaje:
 - neka funkcija koja treba da vrati pokazivač na objekat koji se zahteva, može i da ne vrati to, jer ne može da uradi to što se od nje traži, što je regularna situacija, ili je neregularna, ali funkcija ne podiže izuzetak (a trebalo bi):

```
Clock* pc = ClockFactory::getClock();  
...  
pc->tick();
```

Ova funkcija možda vraća *null*

Pokazivači na objekte

- Zaštita:
 - dobro razlikovati situacije u kojima funkcije mogu da vrate *null* vrednost pokazivača; ako to nije regularna situacija, funkcija tada treba da baci izuzetak
 - ako to jeste regularna situacija, po povratku iz funkcije obavezno proveravati vrednost na *null*:

```
Clock* pc = ClockFactory::getClock();
```

```
...
```

```
if (pc) pc->tick();
```

- krajnje konzervativan stil programiranja (u žargonu se naziva i “paranoičan”) pretpostavlja da se nikada ne veruje pokazivačima i da se oni uvek proveravaju pre dereferenciranja na bilo koji način; ovo nije potrebno ako se dobro tretiraju izuzeci:

```
if (p) ...p->...
```

```
if (p) ...*p...
```

Pokazivači na objekte

❖ Viseći (ili "landaravi") pokazivač (*dangling pointer, dangling reference*):

- Uzrok: pokazivač koji je bio vezan za objekat, ali je objekat prestao da postoji; takav pokazivač ima invalidnu vrednost, iako je ona bila validna; kada objekat prestane da postoji, to nema nikakve efekte na pokazivače koji na njega ukazuju:

```
Clock* pc = new Clock(...);  
delete pc;  
pc->tick();
```

p je sada viseći pokazivač

- Efekti:

- nedefinisano ponašanje ("tiha" greška): program nastavlja dalje sa potpuno nedefinisanim ponašanjem i na kraju možda negde kasnije izaziva izuzetak
- korupcija drugih podataka koji su zauzeli mesto tog objekta
- Tipične situacije u kojima nastaje:
 - nekorektno postupanje sa pokazivačima:

```
Clock* p = new Clock(...);
```

U nekom potpuno drugom kontekstu:

```
Clock* q = p;
```

U nekom potpuno trećem kontekstu:

```
delete p;
```

p i q ne menjaju vrednost, čak ni na null

I potom, opet u nekom potpuno različitom kontekstu...:

```
...*q... ili ...q->...
```

q je viseći pokazivač

Pokazivači na objekte

- vezivanje pokazivača za automatski objekat koji prestaje da postoji po povratku iz funkcije:

```
int* f() {  
    int x = 5; x je automatski objekat  
  
    ...  
  
    return &x; x prestaje da postoji po izlasku iz funkcije  
  
}
```



```
...  
  
int* p = f(); p je sada viseći pokazivač
```

- Zaštita:
 - pažljivo programiranje
 - prevodioci mogu da upozore na pojave visećih pokazivača u nekim situacijama, posebno kod povratka iz funkcije i vezivanja za automatske objekte

Nizovi

- ❖ Niz (*array*) je objekat koji predstavlja uređenu, ograničenu kolekciju objekata, složenih u memoriju jedan iza drugog, deklarisane veličine 
- ❖ Elementi niza mogu biti bilo kog objektnog tipa: fundamentalnog tipa (osim tipa *void*), pokazivačkog tipa, klasnog tipa, nabranja, ili niza (tako se pravi višedimenzionalni nizovi). Ne postoje nizovi referenci ili funkcija
- ❖ Elementi niza veličine N označeni su indeksima (pozicijama u nizu) počev od 0 zaključno sa $N-1$ i može im se pristupati operatorom *indeksiranja niza* []:

`int a[100];`

a je niz od 100 elemenata tipa *int*

...
`a[0] = a[0] + a[99];`

Pristup prvom i poslednjem elementu niza *a*

- ❖ Elementi niza mogu biti drugi nizovi — tako se prave multidimenzionalni nizovi:

`int m[5][7];` *m* niz od 5 elemenata, gde je svaki element tipa niza od 3 elementa tipa *int* — matrica 5x7

`m[3][5] = 2;` Pristupa se 4. elementu niza *m*; on je opet niz; pristupa se 6. elementu tog niza

- ❖ Za vreme izvršavanja, ne proverava se ispravnost indeksa za pristup nizu u smislu da li su u opsegu dozvoljenih vrednosti (unutar granica niza); svaki prestup može da dovede do nekorektnog ponašanja, kao sa pokazivačima

Nizovi

- ❖ Niz može da se inicijalizuje tzv. *agregatnim inicijalizatorom* (*aggregate initializer*) između velikih zagrada; tada ne mora da se navede dimenzija niza, ona se određuje na osnovu broja elemenata u inicijalizatoru i smatra se definisanom:

```
int a[3] = {1, 2, 3};
```

b je niz od 3 elementa tipa int, tj. tipa int[3]

```
int b[] = {4, 5, 6};
```

c je niz od 3 elementa tipa int, inicijalizovanih redom na 1, 0 i 0

- ❖ Niz može da se deklariše i bez zadavanja (prve) dimenzije (ako je u pitanju niz nizova, sve ostale dimenzije moraju da se zadaju, jer one definišu tip elementa niza); ovaj niz je nekompletnog tipa (osim ako je inicijalizovan agregatnim inicijalizatorom). Za ovakve nizove mogu se praviti pokazivači i reference na njih, ali se ti pokazivači i reference ne mogu inicijalizovati tako da ukazuju na nizove sa poznatim dimenzijama. Takvi pokazivači se mogu dereferencirati i prenositi kao parametri funkcija, ali ne mogu učestvovati u pokazivačkoj aritmetici:

```
extern int a[];
```

p je pokazivač na niz nekompletnog tipa (nepoznate veličine)

```
int (*p)[] = &a;
```

Greška u prevodenju (ovo je ispravno na jeziku C)

```
int (*q)[2] = &a;
```

- ❖ Nizovi ne mogu da se kopiraju operacijom dodele, tj. ne mogu biti levi operandi operatora dodele. Međutim, ako je niz podatak član nekog klasnog tipa (strukture ili klase), on se može kopirati kao deo objekta kome pripada:

```
int a[10], b[10];
```

```
class X { public: int a[10]; };
```

```
X x1, x2;
```

```
a = b;
```

Greška u prevodenju

```
x1 = x2;
```

Niz x2.a kopira se u niz x1.a

Nizovi

- ❖ Zbog svoje izvorne orijentacije na efikasnost generisanog koda, operacije sa nizovima na jezicima C i C++ rade se sa najmanje moguće potrebnih informacija; da bi se izvršila operacija pristupa elementu niza, potrebno je, pored izračunavanja izraza koji određuje vrednost indeksa, znati samo sledeće:
 - početnu adresu (prvog elementa) niza i
 - veličinu (svakog) elementa niza, koja je poznata za vreme prevođenja iz tipa elementa tog niza

Pomoću ovih informacija, generisani kod za pristup elementu niza tipa T izračunava adresu indeksiranog elementa na sledeći način:

$$\text{adresa_elementa} := \text{adresa_početka_niza} + \text{vrednost_indeksa} * \text{sizeof}(T)$$

- ❖ Informacija o adresi i tipu elementa sadržana je i u pokazivaču tipa T^*
- ❖ U saglasnosti sa ovim, nizovi se ne prenose kao parametri funkcija, niti se mogu vraćati kao vrednosti funkcija (kao kompletni paketi elemenata); umesto toga, prenose se i vraćaju pokazivači na početak (prvi element) niza (ili reference na nizove, potpuno slično)
- ❖ Prema tome, pozvana funkcija nema implicitnu informaciju o stvarnoj veličini niza, nego se ona mora preneti funkciji na neki drugi način: ili preko posebnog parametra funkcije, ili niz mora imati neku posebnu vrednost koja označava njegov poslednji element (terminiran niz)
- ❖ Zbog svega ovoga, nizovi i pokazivači povezani su sledećim pravilima jezika

Nizovi

- ❖ Prvo pravilo: postoji implicitna konverzija niza elemenata tipa T u pokazivač na tip T , koja se vrši na svakom mestu gde se očekuje pokazivač, a pojavljuje se niz; vrednost ovog pokazivača ukazuje na prvi element datog niza (tzv. "rastakanje" niza u pokazivač, *array-to-pointer decay*):

$$T[] \Rightarrow T^*$$

Na primer:

```
void f (int a[]) { cout<<a[0]<<endl; }

void g (int* p) { cout<<*p<<endl; }

int main () {
    int a[3] = {1, 2, 3};
    int* p = a;
    f(a);
    f(p);
    g(a);
    g(p);
}
```

f zapravo prima pokazivač na *int*

Implicitna konverzija niza *a* tipa *int[3]* u pokazivač na njegov prvi element, tipa *int**; isto bi bilo za: *int* p = &a[0];*

Implicitna konverzija niza *a* tipa *int[3]* u pokazivač na njegov prvi element, tipa *int**

f zapravo prima pokazivač na *int*; potpuno isti efekat kao i *f(a)*

Implicitna konverzija niza *a* tipa *int[3]* u pokazivač na njegov prvi element, tipa *int**

Potpuno isti efekat kao i *g(a)*

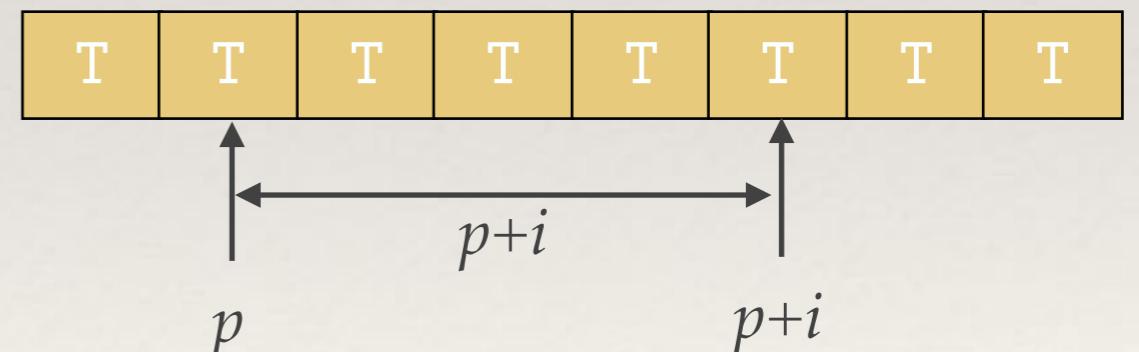
Nizovi

- ❖ Drugo pravilo (tzv. pokazivačka aritmetika, *pointer arithmetics*): ako vrednost p tipa T^* ukazuje na element nekog niza elemenata tipa T , i ako je i vrednost celobrojnog tipa, definisana je operacija sabiranja i oduzimanja ove dve vrednosti; zbir je vrednost koja ukazuje na element istog niza koji je za i elemenata dalje od elementa na koju kazuje p (analogno za oduzimanje)
- ❖ Rezultat je validna vrednost pokazivača samo ukoliko p ukazuje na element niza i ako ovakvom aritmetikom vrednost rezultata ukazuje na element istog niza ili iza poslednjeg elementa niza; inače je vrednost invalidna
- ❖ Na primer:

```
int a[3] = {1, 2, 3};  
int* p = &a[0];  
int i;  
  
for (i=0; i<3; i++, p++)  
    -- *p;  
  
++ *(p-1);
```

p ukazuje na $a[i]$

Po izlasku iz petlje p ukazuje iza $a[2]$; $p-1$ ukazuje na $a[2]$



Nizovi

- ❖ Treće pravilo: ugrađeni operator indeksiranja $x[y]$, gde su x i y izrazi, izračunava se uvek kao $*(x+y)$
- ❖ Na primer:

```
int a[10];
```

p ukazuje na $a[0]$

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

$a[2]$ se izračunava kao $*(a+2)$; pošto je a niz, konvertuje se u pokazivač na svoj prvi element; zbir tog pokazivača i celog broja 2 je vrednost koja ukazuje na $a[2]$

```
a[2]=1;
```

$p[3]$ se izračunava kao $*(p+3)$; pošto je p ukazuje na $a[0]$, zbir tog pokazivača i celog broja 3 je vrednost koja ukazuje na $a[3]$

```
p=p+1;
```

p sada ukazuje na $a[1]$

```
*(p+2)=1;
```

$p+2$ ukazuje na $a[3]$ koji dobija vrednost 1

```
p[-1]=0;
```

Isto što i $*(p+(-1)) = 0$

```
2[p-1] = 0;
```

Isto što i $*(2+(p-1)) = 0$

Nizovi

- ❖ *String literali (string literals)* predstavljaju literale tipa “niz (konstantnih) znakova”
- ❖ Navode se između navodnika, uz mogućnost definisanja tipa znakova i načina kodovanja slično kao i za znakovne literale; između navodnika mogu da se nađu odgovarajući znakovi i *escape* sekvence, kao za znakovne literale; na primer:

"I said: \"Hello!\""

String literal tipa *const char[]*

L"Hello"

String literal tipa *const wchar_t[]*

u8"Hello"

UTF-8 enkodovan string literal tipa *const char[]*

u"Hello"

UTF-16 enkodovan string literal tipa *const char16_t[]*

U"Hello"

UTF-32 enkodovan string literal tipa *const char32_t[]*

- ❖ Dva susedna string literalala, razdvojena samo belinama (uključujući i znak za novi red) spajaju se u jedan nakon preprocesiranja; ovako se dugački string literali mogu nastavljati u novi red:

"This is a very, very long string literal that cannot fit in "
"one single line of code, but has to be split in two lines."

- ❖ Isto se može koristiti za specijalne potrebe, kada bi se dati niz susednih znakova između navodnika tumačio kao *escape* sekvenca:

"\0" "1"

U literalu "\01" bi se sadržaj između navodnika tumačio kao jedan znak sa oktalnim kodom 1, ovako su to dva znaka: znak za kodom 0 i znak za cifru 1

Nizovi

- ❖ Kao podrška konvenciji nasleđenoj iz jezika C da se nizovi znakova završavaju terminalnim znakom '\0', a na koji se oslanjaju sve bibliotečne funkcije standardnih bilioteka koje rade sa nizovima znakova, svaki string literal ima još jedan, implicitan znak - taj terminalni znak '\0':

"Hello"

String literal tipa *const char[6]* sa 6 znakova: 'H', 'e', 'l', 'l', 'o', '\0'

- ❖ String literali imaju statičko trajanje skladištenja: postoje za sve vreme izvršavanja programa (alociraju se i inicijalizuju za vreme prevodenja)
- ❖ Pokušaj upisa u neki znak string literala, kao u element niza konstantnih znakova, ima nedefinsan ishod (taj niz znakova može biti smešten u deo memorije sa zabranjenim upisom, pa upis može izazvati hardverski izuzetak)
- ❖ String literali mogu da se koriste za inicijalizaciju nizova znakova; tada dati niz znakova ima poznatu dimenziju određenu dužinom znakovnog literala (uključujući i terminalni znak) i sadrži kopiju datog stringa:

char str[] = "Hello";

str je tipa *char[6]* čiji su elementi redom znakovi: 'H', 'e', 'l', 'l', 'o', '\0'

Nizovi

- ❖ Dve potpuno nezavisne implicitne konverzije mogu da dovedu do problema ukoliko se nizovi objekata ne koriste na korektan način:
 - konverzija $T[]$ u T^* , koja potiče iz jezika C sa ciljem efikasnosti
 - konverzija pokazivača na izvedenu u pokazivač na dostupnu osnovnu klasu ($Derived^*$ u $Base^*$), koja je uvedena u jezik C++ kao podrška principu supstitucije
- ❖ Ove dve konverzije se mogu raditi i povezano, obe, u lancu implicitnih konverzija, i mogu da dovedu do sledećeg problema: funkcija koja kao parametar ima niz objekata osnovne klase ($Base[]$, odnosno $Base^*$), može se pozvati sa argumentom koji je niz objekata izvedene klase ($Derived[]$):

$Derived[] \Rightarrow Derived^* \Rightarrow Base^*$

```
class Base {  
public: int bi;  
};
```

Objekti klase *Base* imaju samo jedan podatak član tipa *int*, pa su veličine jednog *int*

```
class Derived : public Base {  
public: int di;  
};
```

Objekti klase *Derived* su veličine dva *int*

```
void f (Base b[]) { cout<<b[2].bi; }
```

Funkcija *f* očekuje niz objekata klase *Base*, veličine jednog *int*:
b[2] je $*(b+2)$, a pošto je *b* tipa *B[]*, pokazivačka aritmetika dodaje
vrednost $2 * sizeof(B)$

```
int main () {  
Derived d[5];  
d[2].bi=77;
```

f(d);

}

d se konvertuje iz *Derived[]* u *Derived^**, a potom u *Base^**, pa je poziv ispravan

Nizovi

- ❖ Osim toga, kada se objekti, pa i nizovi objekata, koriste po vrednosti, nema nikakve supstitucije:

```
Base b[5];  
Derived d;  
b[0]=d;
```

b[0] je objekat tipa Base i sadrži kopiju podobjekta osnovne klase iz objekta d

- ❖ Zbog ovoga, ukoliko se računa sa nasleđivanjem i supstitucijom, ne treba praviti nizove objekata (po vrednosti), već nizove pokazivača na objekte:

```
void f (Base* b[], int i) { cout<<b[i]->bi; }  
...
```

```
Base b1,b2; Derived d1,d2,d3; d2.bi=77;
```

```
Base* b[5]; b[0]=&d1; b[1]=&b1; b[2]=&d2; ...
```

```
f(b,2);
```

Sada se ovo korektno računa, pošto su elementi niza pokazivači (čija veličina ne zavisi od tipa objekta na koji ukazuju)

- ❖ Sada navedena greška nije više moguća, jer se niz tipa *Derived*[]* može konvertovati u *Derived***, ali se to ne može konvertovati implicitno u *Base*** (što bi funkcija sa parametrom tipa *Base*[]* prihvatala):

```
Derived* d[5]; d[0]=&d1; d[1]=&d2; ...
```

```
f(d,2);
```

Greška u prevođenju: Derived[] se ne može implicitno konvertovati u Base***

Reference

❖ Motivacija:

- Zbog orijentacije na to da se i objekti klase i objekti neklasnih tipova koriste što sličnije, objekti klase mogu se koristiti *po vrednosti*: kreirati u svim kategorijama životnog veka, kopirati, prenositi kao argumenti
- Zbog toga postoji opisani problem beskonačne rekurzije za prenos argumenta konstruktora kopije
- U skladu sa istim opredeljenjem, mogu se preklapati operatori za objekte klase; kako postići istu notaciju kao za prenos po vrednosti, za operatore koji treba da menjaju neki od svojih operanada, ili ne žele da kopiraju operative, na primer:

```
complex c1, c2;  
...  
c1 += c2;
```

❖ Zato je na jeziku C++ moguć prenos argumenata *po referenci* (*call by reference*):

```
void f (int i, int &j) {  
    i++;  
    j++;  
}  
  
int main () {  
    int si=0, sj=0;  
    f(si, sj);  
    cout<<"si="<<si<<", sj="<<sj<<"\n";  
}
```

Argument *i* prenosi se po vrednosti, a *j* po referenci

Stvarni argument se neće promeniti

Stvarni argument će se promeniti

Izlaz će biti: *si = 0, sj = 1*

Reference

- ❖ Ceo koncept uopšten je na postojanje složenog tipa *reference* (*reference*): referenca je posrednik do objekta, i za vreme svog životnog veka *upućuje* na (referencira, *refers to*) objekat
- ❖ Referenca je posrednik do objekta, slično kao i pokazivač, ali se i značajno razlikuje od pokazivača po sledećem:
 - referenca nije objekat (pokazivač jeste), već je posebna kategorija entiteta u jeziku; zbog toga, recimo, ne postoje reference na reference, pokazivači na reference, niti nizovi referenci (a postoje reference na pokazivače, pokazivači na pokazivače i nizovi pokazivača); ne postoje ni reference na tip *void*
 - referenca se mora vezati za neki objekat na početku svog životnog veka, odnosno mora biti inicijalizovana vezivanjem za objekat; pokazivač ne mora da se veže za objekat
 - referenca ne može da se preusmeri na drugi objekat, dok pokazivač može
 - referenca ne može da se “raskine”, odnosno da se “razveže” od objekta i da ne upućuje ni na jedan objekat, dok pokazivač može (*null* vrednost pokazivača)
- ❖ Referenca se, u principu, implementira isto kao i pokazivač — njena vrednost sadrži adresu referenciranog objekta, s tim da u određenim situacijama, kada to može da se izvede, prevodilac tu vrednost može da zna i za vreme prevođenja, pa vrednost reference ne mora ni da postoji za vreme izvršavanja (iako najčešće postoji), odnosno ta vrednost ne mora da se smešta nigde za vreme izvršavanja, već postoji samo konceptualno

Reference

- ❖ Ne postoji operacija nad samom referencom koja bi joj promenila vrednost, odnosno "prevezala" je na neki drugi objekat nakon inicijalizacije: *svaka* upotreba reference nakon njenog deklarisanja odnosi se na referencirani objekat; zbog toga se referencia često naziva sinonimom za objekat (ukoliko sama referencia ima svoje ime, što ne mora)

- ❖ Na primer:

```
int i = 1;  
int& j = i;  
i=3;  
j=5;  
int* p = &j;  
j+=1;  
int k=j;  
int m=*p;
```

j je referencia tipa *int&* koja se vezuje za objekat *i* (upućuje na njega)

Svaka upotreba reference *j* u izrazu odnosi se na referencirani objekat - menja se *i*

p je pokazivač na *int*; inicijalizuje se izrazom *&j*; upotreba reference u izrazu odnosi se na referencirani objekat; pošto je inicijalizator izraz, operacija *&j* je operacija uzimanja adrese referenciranog objekta *i*

Isto što i *i+=1*;

Pristup do objekta preko reference ne zahteva operator: sama upotreba reference u izrazu odnosi se na referencirani objekat. Pristup do objekta preko pokazivača zahteva operator.

- ❖ Treba dobro razlikovati znak *&* upotrebljen u deklaraciji reference (sastavni deo tipa), od znaka *&* upotrebljenog u izrazu, kada označava operaciju uzimanja adrese; ako se u ovom drugom slučaju kao operand nađe referencia, uvek se odnosi na referencirani objekat
- ❖ Pristup do objekta preko reference ne zahteva operator: sama upotreba reference u izrazu odnosi se na referencirani objekat; pristup do objekta preko pokazivača zahteva operator

Reference

❖ Slično je za sve druge upotrebe reference, na primer:

- kada je referencia parametar funkcije, ona se inicijalizuje u trenutku poziva funkcije stvarnim argumentom, što znači da se ta referencia vezuje za stvarni argument; pošto se svaka upotreba reference u toj funkciji odnosi na referencirani objekat, svaka operacija u kojoj referencia učestvuje odnosi se na stvarni argument, pa se tako i postiže semantika prenosa po referenci:

```
void inc (int& i) {  
    i++;  
}
```

```
int main () {  
    int s = 0;  
  
    inc(s);  
}
```

Pošto je *i* referencia, operacija se odnosi na referencirani objekat, a to je stvarni argument za koji je referencia vezana u trenutku svoje inicijalizacije, odnosno poziva funkcije

U trenutku poziva funkcije *inc*, formalni argument *i* inicijalizuje se stvarnim argumentom *s*; kako je *i* referencia, vezuje se za stvarni argument u trenutku svoje inicijalizacije, odnosno poziva funkcije

- povratna vrednost funkcije takođe može biti referencia; povratna vrednost funkcije je tako referencia koja postoji kao bezimena vrednost, na mestu poziva funkcije; ona se inicijalizuje u trenutku povratka iz funkcije rezultatom izraza iza naredbe *return*; na taj način, ona se vezuje za objekat na koji se odnosi taj izraz; na primer:

```
int& inc (int& i) {  
    i++;  
    return i;  
}
```

```
int main () {  
    int s = 0;  
  
    inc(inc(s));  
}
```

Funkcija vraća referencu koja se inicijalizuje u trenutku povratka iz funkcije i vezuje za objekat određen izrazom iza narebe *return*; pošto je u tom izrazu referencia, odnosi se na referencirani objekat, a to je stvarni argument

Rezultat prvog poziva funkcije *inc* je referencia koja upućuje na *s*; za njega se vezuje i formalni argument spoljašnjeg poziva, pa *s* na kraju ima vrednost 2

Reference

- ❖ Iako se referenca ne može razvezati od objekta nekom operacijom (takva operacija ne postoji), ona, potpuno isto kao i pokazivač, može postati *viseća (dangling reference)*, ukoliko objekat za koji je referenca vezana prestane da živi dok referenca još uvek živi; tipično, ako funkcija vrati referencu na automatski objekat koji nestaje izlaskom iz funkcije
- ❖ Kao i za pokazivače, obraćanje objektu na koga upućuje viseća referenca ima nedefinisane posledice
- ❖ Na primer, ovo ne valja, iako nije greška u prevodenju (mada prevodilac može izdati upozorenje ako prepozna ovaj problem):

```
int& f () {  
    int r=1;  
    ...  
    return r;  
}
```

Funkcija vraća referencu na automatski objekat koji nestaje pri povratku iz funkcije - viseća referenca

```
int& g (int i) {  
    ...  
    return i;  
}
```

Potpuno isto, i formalni argument je automatski objekat koji nestaje pri povratku iz funkcije - viseća referenca

Reference

- ❖ Reference i pokazivači na objekte imaju mnoge sličnosti:
 - u najvećem broju slučajeva implementiraju se na isti način: imaju vrednost adrese objekta za koji su vezani
 - pristup do objekta i preko pokazivača i preko reference je posredan
 - polimorfizam (dinamičko vezivanje) se aktivira pri posrednom pristupu do objekta i preko pokazivača i preko reference
 - mnoga pravila, a naročito pravila konverzije, npr. pokazivača / reference na izvedenu klasu u pokazivač / referencu na dostupnu osnovnu klasu, važe i za pokazivače i za reference na isti način
- ❖ Međutim, postoje značajne razlike između pokazivača i reference:
 - pokazivač se može preusmeriti tako da ukazuje na drugi objekat; referencia je od trenutka svog nastanka, tj. od inicializacije, trajno vezana za isti objekat
 - pokazivač može da ne ukazuje ni na šta (ima vrednost *null*); referencia uvek, od početka do kraja svog životnog veka, upućuje na jedan (isti) objekat
 - pristup do objekta preko pokazivača vrši se preko operatora *; pristup do objekta preko reference je neposredan, tj. ne zahteva nikakav operator – sama upotreba reference u izrazu odnosi se na referencirani objekat

Reference

- ❖ U principu, reference se upotrebljavaju onda kada se želi notacija prenosa argumenta po vrednosti (umesto da se prenosi pokazivač kao “eksplicitan” posrednik), ali je potrebno da:
 - se izbegne kopiranje stvarnog argumenta u formalni
 - se podrži supstitucija, tj. da stvarni argument može biti specijalizacija tipa formalnog parametra
 - argument bude polimorfan
- ❖ Druga situacija jeste ta kada se želi da funkcija vraća nešto što se može dalje menjati, tipično kod preklapanja operatora, kako bi se ponašali slično operatorima za ugrađene tipove; na primer:

```
vector<complex> v(100);
```

```
v[i]++;
```

Operator [] definisan za klasu `vector<complex>` vraća referencu tipa `complex&`

Klasni tipovi

- ❖ Na jeziku C++, klasni tipovi su *strukture (struct)* i *klase (class)*. Strukture i klase su skoro potpuno izjednačene na jeziku C++, jer se tretiraju na potpuno identičan način (osim dole navedenih izuzetaka):
 - i strukture i klase mogu imati podatke članove i funkcije članice, uključujući i konstruktore, destruktore i operatorske funkcije
 - i strukture i klase mogu imati javne, zaštićene i privatne članove
 - i jedne i druge mogu se izvoditi, mogu imati polimorfne operacije itd.
- ❖ Jedine razlike između strukture i klase su sledeće:
 - ako se u definiciji strukture ne navede specifikator prava pristupa, podrazumeva se *public*; ako se u definiciji klase ne navede specifikator prava pristupa, podrazumeva se *private*
 - ako se u definiciji klase ili strukture, prilikom izvođenja iz druge klase ili strukture (dozvoljeno je sve), za osnovnu strukturu ne navede specifikator prava pristupa, podrazumeva se *public*, dok se za osnovnu klasu podrazumeva *private*
- ❖ Ovo su, ipak, krajnje sporedne razlike i ne treba se na njih oslanjati, jer to može da učini program slabije razumljivim: svakako je bolje uvek navoditi specifikatore prava pristupa eksplisitno, radi razumljivosti
- ❖ Zbog svega ovoga, strukture (*struct*) se koriste samo u izuzetnim situacijama, kada treba predstaviti sasvim jednostavne apstraktne tipove podataka, po pravilu onda kada se oni koriste samo za implementaciju nekih drugih struktura ili apstrakcija; strukture tada po pravilu imaju samo podatke članove i eventualno konstruktore, retko kada i neke jednostavne operacije ili destruktore
- ❖ U svim drugim slučajevima, posebno kada je potrebno da imaju iole složenije operacije ili predstavljaju apstrakciju, treba koristiti klase

Konstantni tipovi i funkcije članice

- ❖ Svaki objektni tip (tj. tip koji nije referenca ili funkcija) može da bude kvalifikovan kao *konstantan* (*constant*). Objekti ovakvog tipa ne mogu se menjati: neposredan pokušaj operacije nad ovakvim objektom koja bi promenila taj objekat prevodilac prijavljuje kao grešku, dok indirektan pokušaj izmene takvog objekta, npr. preko pokazivača ili reference na nekonstantan objekat rezultuje nedefinisanim ponašanjem (može izazvati čak i izuzetak od hardvera ako je konstantan objekat smešten u deo memorije zabranjen za upis)
- ❖ Konstantni objekti moraju biti inicijalizovani u definiciji, jer kasnije svakako ne mogu da se promene. Na primer:

```
const float pi = 3.14;
const char plus = '+';
```

```
pi++;
```

```
plus = '-';
```

Greške u prevodenju

- ❖ Konstantni objekti fundamentalnih tipova mogu da se koriste u konstantnim izrazima koje prevodilac treba da izračuna tokom prevodenja, na primer, u izrazima koje definišu dimenzije nizova:

```
const int MaxNumOfClocks = 100;
```

```
...
```

```
Clock* clocks[MaxNumOfClocks];
```

- ❖ Umesto korišćenja makro zamene `#define`, bolje je koristiti konstantne objekte, jer oni imaju svoj tip koji može biti različit od tipa literala kojim se makro u direktivi `#define` zamenjuje; u određenim situacijama, prevodilac ne mora ni da alocira konstantan objekat za vreme izvršavanja, već da njegovu konstantnu vrednost potpuno iskoristi za vreme prevodenja na svim mestima njenog korišćenja

Konstantni tipovi i funkcije članice

- ❖ Kvalifikator *const* može da se piše i ispred i iza naziva tipa:

```
const char* pc1 = ...;  
char const* pc2 = ...;
```

- ❖ Pokazivač na konstantan objekat definiše se stavljanjem reči *const* ispred (ili iza) tipa objekta (ali ispred znaka *); konstantan pokazivač definiše se stavljanjem reči *const* ispred samog imena pokazivača, odnosno iza znaka *:

```
const char* pc = ...;  
pc[3] = 'a';  
pc = ...;
```

Greška u prevodenju: *pc[3]* je isto što i **(pc+3)*; kako je *pc* tipa *const char** (pokazivač na konstantan *char*), to je **(pc+3)* tipa *const char*, pa se ne može menjati

```
char* const cp = ...;  
cp[3] = 'a';  
cp = ...;
```

Greška u prevodenju: *cp* je tipa *char*const* (konstantan pokazivač na nekonstantan *char*), pa se ne može menjati; *cp[3]* je tipa *char*, pa se može menjati

```
const char* const cpc = ...  
cpc[3] = 'a';  
cpc = ...;
```

Greška u prevodenju: *cpc* je tipa *const char*const* (konstantan pokazivač na konstantan *char*), pa se ne može menjati ni on, ni ono na šta on ukazuje

- ❖ Referenca može biti na konstantan tip, ali sama referenca ne može da se deklariše kao konstantna, jer ona to svakako jeste (ne postoji operacija koja bi promenila referencu nakon inicijalizacije); ako se *const* upotrebi uz referencu indirektno, npt. u *typedef* deklaracijama, ignoriše se

Konstantni tipovi i funkcije članice

❖ Deklarisanjem pokazivača na konstantan objekat programer najavljuje (“obećava”) da ono na šta taj pokazivač ukazuje ne može da se menja *preko tog* pokazivača, što ne znači da je apsolutno konstantno; prevodilac kontroliše ispunjenje te najave dosledno, sprovođenjem sledećih pravila jezika:

- postoji implicitna konverzija iz tipa pokazivača na nekonstantan tip T u tip pokazivača na konstantan tip T : time se samo “zateže” konstantnost, odnosno obećava da se preko nekog drugog pokazivača neće izmeniti objekat, u kontekstu (opsegu važenja) tog pokazivača:

```
char* p = ...;  
const char* q = p;  
...  
q = p;
```

Dozvoljeno: samo znači da se objekat na koga ukazuje q , koji god da je, neće menjati preko tog q , ne znači da on i generalno konstantan

- nije dozvoljena implicitna konverzija iz tipa pokazivača na konstantan tip T u tip pokazivača na nekonstantan tip T , jer bi se time “tiho probila” konstantnost, odnosno omogućilo slučajno narušavanje konstantnosti, bez upozorenja:

```
const char* p = ...;  
char* q = p;  
...  
q = p;
```

Greška u prevodenju, jer bi inače, nakon ovoga, bilo moguće promeniti objekat $*q$

- dozvoljena je eksplisitna konverzija operatorom *const_cast* iz tipa pokazivača na konstantan tip T u tip pokazivača na nekonstantan tip T ; izmena konstantnog objekta preko takvog pokazivača ima nedefinisane efekte:

```
const char* p = ...;  
char* q = const_cast<char*>p;
```

Sada je moguće promeniti objekat $*q$, ali je efekat toga nedefinisan

Konstantni tipovi i funkcije članice

- ❖ String-literali imaju tip *const char[]* (niz konstantnih znakova), pa su u skladu sa tim dozvoljene njihove implicitne konverzije u *const char**, ali ne i u *char**:

```
const char* p = "Hello";  
char* q = "World";
```

Greška u prevodenju

- ❖ Naravno, sve to važi i ukoliko su parametri funkcije pokazivači na konstantan tip: tada funkcija "obećava" da neće izmeniti ono na šta taj parametar ukazuje (jer je opseg važenja parametra lokalna za tu funkciju), pa se funkcija može pozvati i sa pokazivačem na konstantan i sa pokazivačem na nekonstantan objekat (u suprotnom može samo za nekonstantan):

```
void f1 (const char*);  
void f2 (char*);  
char s1 = ...;  
const char s2 = ...;
```

Funkcija *f1* "obećava" da neće menjati ono na šta parametar ukazuje

Funkcija *f2* "ne obećava" da neće menjati ono na šta parametar ukazuje

```
void f () {  
    f1(&s1);  
    f2(&s1);  
    f1(&s2);  
    f2(&s2);  
}
```

U redu: inicijalizacija argumenta je *const char* arg = &s1* (dozvoljena implicitna konverzija)

U redu: inicijalizacija argumenta je *char* arg = &s1* (isti tipovi)

U redu: inicijalizacija argumenta je *const char* arg = &s2* (isti tipovi)

Greška u prevodenju: inicijalizacija argumenta je *char* arg = &s2*

- ❖ Sva navedena pravila konverzija važe potpuno isto i za reference na konstantne / nekonstante tipove

Konstantni tipovi i funkcije članice

- ❖ Nestatičke operacije (funkcije članice) klase se generalno mogu posmatrati razvrstane u dve kategorije:
 - operacije koje *ne menjaju* spolja vidljivo stanje objekta (*externally visible state*), odnosno ništa ne upisuju u atributе objekta, već samo čitaju to stanje / vrednosti atributе i vraćaju informacije o tome; ovakve operacije nazivaju se ponekad *inspektori* (*inspector*) ili *selektori* (*selector*)
 - operacije koje *menjaju* spolja vidljivo stanje objekta, odnosno upisuju nešto u atributе objekta; ovakve operacije ponekad se nazivaju *modifikatori* (*modifier*) ili *mutatori* (*mutator*)
- ❖ Na jeziku C++, operacije koje ne menjaju stanje objekta nazivaju se *konstantne funkcije članice* (*constant member functions*) i označavaju se specifikatorom *const* iza liste argumenata; operacije koje menjaju stanje objekta ne označavaju se posebno:

```
class Clock {  
public:  
    Clock (int hh, int mm, int ss);  
  
    void tick ();  
    string getTime () const; // Konstantna funkcija članica  
    void setTime (int hh, int mm, int ss);  
  
    ...  
};
```

Konstantni tipovi i funkcije članice

- Prevodilac neće dozvoliti poziv nekonstantne funkcije članice za konstantan objekat (uključujući i indirektni pristup preko pokazivača ili reference na konstantan objekat). Na primer:

```
class X {  
public:  
    X (int ii) { write(ii); }  
    int read () const { return i; }  
    void write (int ii) { i = ii; }  
  
private:  
    int i;  
};  
  
...  
  
X x(0);  
const X cx(1);  
X* px = &x;  
const X* pcx = &cx;  
X& rx = x;  
const X& rcx = cx;  
  
x.read();  
x.write();  
cx.read();  
cx.write();  
  
px->read();  
px->write();  
pcx->read();  
pcx->write();  
  
rx.read();  
rx.write();  
rcx.read();  
rcx.write();
```

Sve ovo je u redu, jer se za nekonstantan objekat može pozivati i konstantna i nekonstantna funkcija članica

Greške u prevodenju (označeno crvenim): ne može se pozivati nekonstantna funkcija članica za konstantan objekat

Konstantni tipovi i funkcije članice

❖ Prevodilac zapravo sprovodi ista opšta pravila za pokazivače na (ne)konstantne objekte, jer je:

- u nekonstantnoj, nestatičkoj funkciji članici klase X, pokazivač *this* implicitno deklarisan kao pokazivač tipa *X* const* (konstantan pokazivač na *nekonstantan* objekat)
- u konstantnoj, nestatičkoj funkciji članici klase X, pokazivač *this* je implicitno deklarisan kao pokazivač tipa *const X* const* (konstantan pokazivač na *konstantan* objekat)

```
class X {  
public:  
    X (int ii) { set(ii); }  
  
    int read () const { return i; }  
    void write (int ii) { i = ii; }
```

private:

 int i;

};

...

```
X x(0);  
const X cx(1);  
  
x.read();  
x.write();  
cx.read();  
cx.write();
```

Prilikom poziva, *this* se inicijalizuje ovako:

const X const this* = $\&x$, što je dozvoljeno, jer je *x* tipa X

Prilikom poziva, *this* se inicijalizuje ovako:

X const this* = $\&x$, što je dozvoljeno, jer je *x* tipa X

Prilikom poziva, *this* se inicijalizuje ovako:

const X const this* = $\&cx$, što je dozvoljeno, jer je *cx* tipa *const X*

Greška u prevodenju, jer se *this* inicijalizuje ovako:

X const this* = $\&cx$, što nije dozvoljeno, jer je *cx* tipa *const X*

Konstantni tipovi i funkcije članice

- ❖ Konstantne funkcije članice su one operacije koje (konceptualno) ne menjaju "spolja vidljivo stanje objekta", što ne mora obavezno značiti da ne mogu da upisu vrednost u neki podatak član objekta
- ❖ Tipičan primer jeste situacija kada neka operacija samo izračunava neki podatak, odnosno vraća informaciju (podatak) o stanju objekta, ali je izračunavanje tog podatka veoma složena i zahtevna operacija (memorijski ili vremenski), pa je zgodno uraditi tzv. *memoizaciju (memoization)*: pamćenje izračunatog podatka, kako se sledeći put može samo vratiti ta upamćena vrednost, bez ponovnog izračunavanja. Da bi se ta izračunata vrednost zapamtila u objektu, potrebno je upisati je u neki podatak član predviđen za čuvanje te vrednosti
- ❖ U konstantnoj funkciji članici bi prevodilac sprečio upis u taj podatak član, što se može prevazići eksplicitnom konverzijom operatorom *const_cast*:

```
const_cast<X const*>(this)->member = ...
```

- ❖ U opštem slučaju, ovo može imati nedefinisane efekte, pa na jeziku C++ postoji i direktniji pristup: ovakav podatak član može se označiti kao promenljiv čak i u konstantnim objektima, navođenjem specifikatora *mutable* u deklaraciji tog podatka člana. Na primer:

```
class City {  
public:  
    ...  
    Distance getDistanceFrom (City* other) const;  
protected:  
    Distance computeDistanceFrom (City* other) const;  
private:  
    mutable map<City*,Distance> memoizedDistances;  
};  
...  
Distance City::getDistanceFrom (City* other) const {  
    if (memoizedDistances.count(other)==0)  
        memoizedDistances.insert(computeDistanceFrom(other));  
    return memoizedDistances[other];  
}
```

Funkcija *getDistanceFrom* treba da vrati udaljenost ovog grada od datog drugog grada. Ona je konstantna, jer ne menja spolja vidljivo stanje objekta

Pomoćna funkcija *computeDistanceFrom* sprovodi zahtevan postupak izračunavanja udaljenosti između gradova

Funkcija *insert* klase *map* nije konstantna funkcija članica, pa prevodilac ne bi dozvolio njen poziv za podatak član *memoizedDistances* unutar konsantne funkcije članice ove klase; međutim, ovaj podatak član je deklarisan kao *mutable*, pa je ovo dozvoljeno

Nestalni tipovi

- ❖ Kao *cv-kvalifikator (cv-qualifier)* objektnog tipa, potpuno analogno sa kvalifikatorom *const*, može da se koristi kvalifikator *volatile* (nestalan, nepostojan): označava da se vrednost ovog objekta može promeniti nezavisno od toka kontrole koda u kome se koristi, tipično od strane:
 - hardvera: u memoriju u kojoj je objekat uskladišten neki hardverski uređaj može *asinhrono*, tj. potpuno nezavisno od operacija programa, u proizvoljnim, nepredvidivim trenucima, upisati tj. promeniti njegovu vrednost
 - *signala*, odnosno rutina koje obrađuju asinhronne signale (*signal handler*) koji dolaze od operativnog sistema (ili hardvera, ali koje operativni sistem pretvara u asinhronne signale programu)
- ❖ Ovaj kvalifikator nalaže prevodiocu da ne vrši optimizacije koda i premeštanja operacija sa ovakvim objektom, što preciznije znači da sve operacije koje se po semantici izvršavaju pre neke date operacije čitanja ili promene ovakovog objekta moraju završiti pre te operacije, a one koje su po semantici iza te operacije ne smeju početi pre nego što se ta operacija završi
- ❖ Drugim rečima, prevodilac će svaku operaciju čitanja ili upisa u nestalan objekat izvršiti bez optimizacija i promena redosleda koda, ne sme je pročitati ili upisati pre nego što su sve operacije pre nje završtene i svaki put je čita/upisuje iznova, iako program možda ne menja vrednost tog objekta
- ❖ Kvalifikator *volatile* se koristi potpuno analogno kao i kvalifikator *const*, i sva navedena pravila vezana za pokazivače, reference i konverzije važe na potpuno isti način
- ❖ Objekat može biti istovremeno i *const* i *volatile*

Korisnički definisane konverzije

- ❖ Konstruktor klase X koji se može pozvati samo sa jednim stvarnim argumentom tipa T definiše *korisničku konverziju* (*user-defined conversion*) iz tipa T u tip X i naziva se *konverzionalni konstruktor* (*converting constructor*):

```
class X {  
public:  
    X (int); // Konstruktor konverzije iz tipa int u tip X  
};
```

- ❖ Funkcija se može pozvati sa samo jednim stvarnim argumentom tipa T ako ima samo jedan formalan parametar, ili ako svi drugi formalni parametri imaju podrazumevane vrednosti; taj prvi (ili jedini) parametar mora biti tipa T ili tipa reference na T (obično tipa $const T\&$)
- ❖ Ova konverzija može se vršiti eksplicitno, bilo kojim operatorom konverzije, ali i implicitno, gde god se vrši implicitna konverzija; u nizu tranzitivnih implicitnih konverzija koje se mogu vršiti od datog do odredišnog tipa, samo jedna može biti korisnička konverzija:

```
X f (X x1) {  
    ...  
    return 2;  
}
```

Prilikom povratka iz funkcije f , povratna vrednosti tipa X dobija se konverzijom izraza iza naredbe *return* koji je tipa int : vrši se implicitna konverzija iz int u X pozivom konverzionalnog konstruktora: $X x2(2)$

```
void g () {  
    X x2 = f(1);  
}
```

Prilikom poziva funkcije $f(1)$, formalni argument $x1$ tipa X inicijalizuje se stvarnim argumentom tipa int : vrši se implicitna konverzija iz int u X pozivom konverzionalnog konstruktora: $X x1(2)$

Korisnički definisane konverzije

- ❖ Konverzionim konstruktorom klase X može se definisati korisnička konverzija iz bilo kog drugog tipa, pa i ugrađenog (neklasnog) tipa, u tip X, ali ne i obrnuto, iz tipa X u neki ugrađeni tip
- ❖ Takva konverzija može se definisati *operatorom konverzije*, kao nestatičkom operatorskom funkcijom članicom klase X, koja nema parametre, nema eksplicitan povratni tip, i ima ime u kome se koristi deklarator tipa (*type-id*); ovakva funkcija treba da vrati vrednost datog tipa:

```
class X {  
public:  
    operator int ();           ← Korisnička konverzija iz tipa X u tip int  
    operator Y* ();           ← Korisnička konverzija iz tipa X u tip Y*  
};
```

- ❖ Kao *type-id* može se navesti bilo koji fundamentalni ili složeni tip, ali se ne mogu upotrebljavati simboli za niz [] i funkciju (), osim indirektno, kroz *typedef* sinonime, s tim da odredišni tip ne može biti niz ili funkcija čak ni tako
- ❖ I ova konverzija može se vršiti eksplicitno, bilo kojim operatorom konverzije, ali i implicitno, gde god se vrši implicitna konverzija; na primer:

```
X x;  
int i = x;           ← Korisnički definisana implicitna konverzija iz tipa X u tip int; poziva se x.operator int()  
Y* py = x;           ← Korisnički definisana implicitna konverzija iz tipa X u tip Y*; poziva se x.operator Y*()
```

Korisnički definisane konverzije

- ❖ Ukoliko se želi zabraniti implicitna korisnički definisana konverzija definisana konstruktorom ili operatorom konverzije, odgovarajuća funkcija (konstruktor ili operator) označava se specifikatorom *explicit*; takav konstruktor nije više konverzionalni konstruktor, jer ne definiše implicitnu konverziju; na primer:

```
class X {  
public:  
    explicit X (bool);  
    explicit operator bool ();  
};  
  
X f (X x) {  
    bool b = x;  
    return false;  
}
```

Greška u prevodenju: nije dozvoljena implicitna konverzija iz X u bool

Greška u prevodenju: nije dozvoljena implicitna konverzija iz bool u X

Korisnički definisane konverzije

- Standardne konverzije mogu da se rade implicitno, i to tranzitivno (u nizu), pa jedna korisnički definisana konverzija iz tipa X u ugrađeni tip može da znači i implicitnu konverziju u neki drugi ugrađeni tip; ako se ovo želi sprečiti, ta druga konverzija deklariše se posebno, čime ona nadjačava tranzitivne konverzije, ali se označi kao *obrisana (deleted)*, specifikatorom `=delete` iza zagrada, pa je prevodilac neće dozvoliti. Ovo može da spreči neke nepredviđene upotrebe objekata tipa X, na primer kada se želi upotreba tih objekata kao Bulovih vrednosti (tzv. “problem sigurnog tipa *bool*”, *safe bool problem*):

```
class Assertion {  
public:  
    operator bool ();  
    operator int () = delete;  
};  
  
void f (Assertion x) {  
    x << 1;  
    if (x) ...  
}
```

Greška u prevođenju: ne postoji deklarisan operator `<<` za ovu klasu, dok je operator implicitne konverzije u tip *int* deklarisan kao obrisan. Da ovaj operator nije tako deklarisan, prevodilac bi izvršio implicitnu, korisnički definisani konverziju iz tipa *Assertion* u tip *bool*, pa potom i implicitnu, standardnu konverziju iz tipa *bool* u tip *int* i ovakav kod bez smisla bi bio moguć

Implicitna konverzija iz *Assertion* u *bool* je dozvoljena: objekti ove klase su namenjeni za korišćenje kao Bulovi izrazi, pa je zato definisana ova implicitna konverzija u *bool*

- Slično se može postići i deklarisanjem operatora konverzije u tip *bool* kao *explicit*, jer naredba `if` i naredbe petlji dozvoljavaju korisnički definisani konverziju koja je eksplisitna:

```
class Assertion {  
public:  
    explicit operator bool ();  
};  
  
void f (Assertion x) {  
    x << 1;  
    if (x) ...  
}
```

Greška u prevođenju: nije dozvoljena implicitna konverzija iz tipa *Assertion* u tip *int*, čak ni tranzitivno, jer konverzija iz *Assertion* u *bool* nije implicitna

Ovo je u redu, jer je definisana konverzija iz tipa *Assertion* u tip *bool*

Korisnički definisani literali

- ❖ Kada neki apstraktni tip podataka implementira neku fizičku veličinu, numeričke vrednosti te veličine u programu ne mogu da sadrže informaciju o jedinici u kojoj je izražena ta vrednost. Takav program može biti osetljiv na greške ukoliko se na različitim mestima koriste vrednosti, date kao literali, izražene u različitim jedinicama mere. Na primer:

```
class Pressure {  
public:  
    Pressure (long double); // Pressure in Pa  
    ...  
};  
Pressure atm = 10.e5;  
Pressure p0 = 1.0;
```

Šta ako je ova vrednost greškom izražen u barima?

```
class Pressure {  
private:  
    friend Pressure operator"" _pa (long double);  
    friend Pressure operator"" _bar (long double);  
    Pressure (long double);  
    ...  
};  
Pressure operator"" _pa (long double val) {  
    return Pressure(val);  
}  
Pressure operator"" _bar (long double val) {  
    return Pressure(val*10.e5);  
}  
Pressure atm = 10.e5_pa;  
Pressure p0 = 1.0_bar;
```

Zadatak: rešite ovaj problem bez korišćenja korisnički definisanih literalala!
Uporedite preglednost notacije u oba slučaja.

Poziva se *operator"" _pa (10.e5)*

Poziva se *operator"" _bar (1.0)*

Korisnički definisani literali

- ❖ Korisnički definisani literali imaju isti format kao i ugrađeni literali, s tim što imaju sufiks koji mora da počne donjom crtom `_` i malim slovom iza nje; sami literali mogu biti u formatu celobrojnih, racionalnih, znakovnih ili string-literalata, ali bez sufiksa koji se koriste za takve ugrađene literale
- ❖ Literalski operatori moraju da imaju naziv *operator* "", i za koga sledi identifikator koji počinje donjom crtom i malim slovom iza nje. Kao parametre mogu da imaju samo neke od dozvoljenih tipova, u koje spadaju samo *unsigned long long* za celobrojne literale, *long double* za racionalne literale, *char* i njegove varijacije za znakovne literale i *const char** i varijacije za string literale (za njih se koristi i drugi argument koji daje veličinu niza znakova u string-literalu). Na primer:

```
constexpr long double operator"" _deg (long double deg) {  
    return deg*3.1415926/180;  
}
```

- ❖ Korisnički string-literali mogu da se koriste za identifikaciju instanci apstrakcije koje se ne mogu predstaviti numeričkim literalima; na primer:

```
class Date {  
public:  
    Date (unsigned y, unsigned m, unsigned d);  
    ...  
};  
  
Date operator"" _date (const char*, size_t);  
  
Date today = "04.11.2018."_date;
```

Specifikator *constexpr* navodi da se entitet u deklaraciji može izračunati za vreme prevodenja i koristiti u konstantnim izrazima. Pošto je argument ove funkcije literal (vrednost poznata za vreme prevodenja), i pošto je celo telo funkcije izračunljivo za vreme prevodenja, sve može biti konstantan izraz

Poziva se *operator"" _date("04.11.2018.", 11)*

Glava 8: Deklaracije i opseg važenja

- ❖ Deklaracije
- ❖ Opseg važenja
- ❖ Opseg važenja bloka
- ❖ Opseg važenja klase
- ❖ Prostori imena
- ❖ Enumeracije sa opsegom važenja



Deklaracije

- ❖ Deklaracija uvodi jedno ili više imena u program. Deklaracija može i da se ponovi, odnosno da ponovo uvede isto ime u program, pod uslovom da je identična prethodnoj i da nije definicija
- ❖ Definicija je ona deklaracija koja u potpunosti definiše deklarisani entitet. Ne mogu postojati dve definicije istog imena, čak i ako su identične. Deklaracija nije definicija samo u nekim posebnim, navedenim slučajevima
- ❖ Deklaracija varijable ili funkcije može da počne i specifikatorom *extern* ili *static* koji utiču na način vezivanja deklarisanog imena:
 - Specifikator *extern* deklariše ime sa spoljašnjim (eksternim) vezivanjem; ovakve deklaracije nisu definicije, osim ako imaju inicijalizator
 - Specifikator *static* deklariše ime sa unutrašnjim (internim) vezivanjem

extern const int a;

Deklaracija koja nije definicija, ime sa ekternim vezivanjem

extern const int b = 1;

Definicija imena sa eksternim vezivanjem

static const int c = 1;

Definicija imena sa internim vezivanjem

Deklaracije

- ❖ Deklaracija funkcije koja ne navodi njenog tela nije definicija
- ❖ Deklaracije formalnih parametara u deklaraciji funkcije koja nije definicija takođe nisu definicije. Na takvim mestima nazivi formalnih parametara mogu da se izostave
- ❖ Čak i kada se navodi samo deklaracija funkcije bez tela, poželjno je navesti nazive formalnih parametara radi bolje čitljivosti, odnosno razumljivosti značenja tih parametara, osim ako to značenje nije očigledno; na primer:

```
void strcpy (char* to, const char* from);
```

```
int strcmp (const char*, const char*);
```

Ovde ima smisla navesti imena parametara, da bi bilo jasnije koji niz je izvorišni, a koji odredišni

Ovde nema potrebe navoditi imena parametara, jer su oba ravnopravna i imaju istu ulogu

- ❖ Nakon deklaracije funkcije koja nije definicija, ta funkcija se može pozivati (pod uslovom da su i da su tipovi parametara i povratni tip definisani) i koristiti na svaki drugi način (npr. uzimati njena adresu), pod uslovom da je negde u programu dano njen telo. Ovakva deklaracija bez tela omogućava međusobno rekurzivno pozivanje više funkcija:

```
void f ();
```

```
void g () {  
    ...f(...)  
}
```

```
void f () {  
    ...g(...)  
}
```

Deklaracije

- ❖ Slična potreba međusobnog unakrsnog referenciranja postoji i kod klasa: u definiciji jedne može biti potrebna druga i obratno, recimo, kada su objekti tih klasa obostrano povezani (klase su povezane obostrano navigabilnom asocijacijom); tada se ova cirkularna zavisnost može razrešiti navođenjem *deklaracije - najave klase (forward declaration)* koja nije definicija:

```
class Lobby;
class Clock {
public:
    Clock (Lobby* owner);
...
private:
...
    Lobby* myLobby;
};

class Lobby {
private:
...
    Clock* myClock[MaxNumOfClocks];
};
```

Deklaracija - najava klase

Za deklarisanje pokazivača nije potrebna potpuna definicija klase,
dovoljna je najava

- ❖ Nakon ovakve najave klase unapred, klasa se smatra nekompletnim tipom i ne mogu se definisati objekti te klase, ali se mogu definisati pokazivači i reference na tu klasu; međutim, sa objektima na koje upućuju ti pokazivači i reference se ne može raditi ništa, jer nisu poznati (deklarisani) članovi te klase

Deklaracije

- ❖ Osim navedenog, ovakva tehnika koristi se i za smanjenje međusobnih zavisnosti između fajlova-zaglavlja: ako za definiciju jedne klase *A* nije potrebna potpuna definicija druge klase *B*, onda u zaglavljenu kom je data puna definicija klase *A* ne treba uključivati celo zaglavljenu kom je definicija klase *B*, jer će to značajno produžiti prevođenje ako se prvo zaglavljenu uključuje dalje (što je često slučaj), nego treba postaviti samo deklaraciju - najavu:

```
class B;  
  
class A {  
public:  
    A(B*);  
  
    void doSomething();  
  
...  
  
private:  
    B* myB;  
...  
};
```

Fajl A.h: za ovu definiciju klase *A* nije potrebna puna definicija klase *B*, pa nema potrebe uključiti zaglavljenu *B.h*

Fajl A.cpp: ovde su potrebne pune definicije obe klase, pa se uključuju oba zaglavlja

```
#include "A.h"  
#include "B.h"  
  
A::A(B* aB) : myB(aB) {...}  
  
void A::doSomething() {  
    ...myB->aFunction()...  
}
```

Deklaracije

- ❖ Jezik C++ ima mnogo vrsta deklaracija, od kojih se mnoge mogu navoditi unutar bloka (složene naredbe), odnosno unutar tela funkcija; sve takve deklaracije mogu se navoditi i van tela funkcija
- ❖ Unutar bloka, ovakve deklaracije imaju istu sintaksnu kategoriju kao i naredbe, odnosno mogu se naći bilo gde u bloku, ne samo na njegovom početku; tako se imena (npr. variable) mogu uvoditi na mestu na kom su potrebna, ne obavezno pre toga:

```
int main () {  
    unsigned n; Deklaracija unutar bloka (tela funkcije)  
    cin>>n;  
  
    long sum = 0; Deklaracija unutar bloka  
    for (unsigned i=0; i<n; i++) {  
        int x; Deklaracija unutar bloka  
        cin>>x;  
        sum+=x;  
    }  
  
    cout<<sum;  
}
```

Deklaracije

- ❖ Deklaracije koje uvode varijable ili funkcije imaju sintaksu kao na jeziku C; one mogu biti vrlo komplikovane i nečitke za složene tipove, ali neka osnovna orijentaciona pravila za tumačenje tipa ovakvih imena jesu sledeća:
 - Prilikom tumačenja tipa u deklaraciji, kreće se od mesta imena; ako se radi o složenom tipu za *type-id*, kreće se od mesta gde bi se nalazilo ime, da se radi o deklaraciji imena
 - Iterativno se izgrađuje rečenica koja opisuje dati složeni tip; počinje se od rečenice "... je tipa -"
 - Od tekuće pozicije uvek se ide najpre nadesno, dokle god se ne nađe na zatvorenu zagradu ili na kraj; onda se ide nalevo, sve dok se ne dođe do otvorene zgrade ili do kraja, čime se zaključuje jedan nivo i ponovo kreće nadesno
 - Ako se, pri kretanju nadesno, nađe na znake [], na rečenicu se dodaje iskaz "niz od ... elemenata tipa -"
 - Ako se, pri kretanju nadesno, nađe na listu argumenata unutar zagrade (), na rečenicu se dodaje iskaz "funkcija koja prima argumente tipa (i za svaki argument u listi pročita se tip) i vraća rezultat tipa -"
 - Ako se, pri kretanju nalevo, nađe na znak *, na rečenicu se dodaje iskaz "pokazivač na tip -"
 - Ako se, pri kretanju nalevo, nađe na znak &, na rečenicu se dodaje iskaz "referenca na tip -"
 - Ako se, pri kretanju nalevo, nađe na ime tipa, na rečenicu se dodaje ime tog tipa
 - Ime tipa na početku deklaracije odnosi se na sva deklarisana imena, dok se ostali elementi tzv. deklaratora (npr. znaci za reference i pokazivače) odnose samo na svakog pojedinačno

- ❖ Primeri:

`int *pi, &ri, i, *a[10], (*pa)[], (*pap)[], f(int*), (*pf)(int), (*apf[])(int), &(*pf)(int&)(*int&));`

- ❖ Naravno, preterano složene i nečitke iskaze treba svakako izbegavati i učiniti čitljivijim uvođenjem *typedef* sinonima

Ovo nije dobar stil programiranja!

Deklaracije

- ❖ Veoma često je potrebno deklarisati varijablu koja se inicijalizuje vrednošću nekog izraza, uključujući i poziv funkcije. Relativno često, a posebno u slučaju korišćenja šablonskih klasa iz standardne biblioteke i njihovih operacija, tip takve povratne vrednosti je veoma složen i uključuje parametrizovane šablonske klase, cv-kvalifikatore i slično; slična situacija je i u definisanju samih metoda složenih šablonskih klasa, gde tip neke varijable zavisi od parametara šablona
- ❖ U takvim situacijama programeru može biti teško i nepraktično da obavezno precizira taj tip u deklaraciji varijable, ili bi time program postao teže čitljiv. Sa druge strane, prevodilac tip te povratne vrednosti izraza ili funkcije sasvim pouzdano zna za vreme prevođenja
- ❖ Za ovakve potrebe, moguće je deklarisati varijablu bez eksplisitnog navođenja tipa, uz ključnu reč *auto*. Time varijabla ima i dalje statički definisan tip, ali taj tip određuje sam prevodilac na osnovu tipa inicijalizatora te varijable, i dalje ga koristi kao da je on eksplisitno naveden u deklaraciji
- ❖ Ovo se može koristiti u bilo kojoj deklaraciji koja definiše varijablu sa inicijalizatorom, što može da olakša pisanje koda i učini ga kompaktnijim. Na primer:

```
template<typename T>
T sum (const list<T>& array) {
    T sum = 0;
    for (auto it=array.cbegin(); it!=array.cend(); it++)
        sum += *it;
    return sum;
}
```

Ova funkcija vraća objekat - *iterator* koji je u stanju da iterira kroz kolekciju (ovde listu) sve do kraja i obezbeđuje pristup do tekućeg elementa kolekcije

Tačan tip ovog objekta nije ni bitan, već su bitne samo operacije koje on obezbeđuje (prekolopljeni operatori `++` za pomeranje na sledeći element i `*` za pristup tekućem). Tip ovog objekta prevodilac zna - to je tip povratne vrednosti ove funkcije

Deklaracije

- ❖ Ista stvar može se iskoristiti za lakše pisanje koda bez obaveznog određivanja tipa rezultata izraza, koji može zavisiti od tipova operanada; na primer:

```
auto z = x*exp(y);  
auto u = static_cast<decltype(v)>(z);
```

Tip varijable *z* zavisi od tipa varijable *x* i povratnog tipa one funkcije *exp* koja prima tip varijable *y*

- ❖ U nekim situacijama, posebno kod šablonskih funkcija, povratni tip funkcije je teško ili nemoguće odrediti. Tada se povratni tip funkcije može odrediti na osnovu tipa izraza i specifikatora *decltype* navedenog iza znaka *->*:

```
template<typename T, typename U>  
auto add(T t, U u) -> decltype(t + u) {  
    return t+u;  
}
```

Povratni tip ove funkcije određuje sam prevodilac, kao tip izraza *t+u*, koji zavisi od povratnog tipa operatora + definisanog za operande tipova *T* i *U* koji su parametri šablonu

- ❖ Funkcija može imati povratni tip koji nije eksplicitno naveden, već se takođe zaključuje, ali na osnovu tipa izraza iza naredbe *return*:

```
template<typename T, typename U>  
auto add(T t, U u) {  
    return t+u;  
}
```

Opseg važenja

- ❖ *Oblast ili opseg važenja (scope)* je deo teksta programa, ponekad diskontinualan (razdvojen delovima koda koji ne pripadaju tom opsegu) u kom je neko ime važeće, odnosno u kom se ono može koristiti neposredno, bez posebnih kvalifikacija, odnosno bez upotrebe operatora :: ili nekog drugog operatora ispred tog imena, npr. *Clock::setTime* ili *pc->setTime*
- ❖ Pod *imenom (name)* se podrazumeva identifikator, puno ime operatorske funkcije (npr. *operator+* ili *operator new*), ime korisnički definisanog konverzionog operatora (npr. *operator bool*) ili ime šablonu sa stvarnim argumentima (npr. *stack<int>*)
- ❖ U opsegu važenja nekog imena, prevodilac može da izvrši tzv. *nekvalifikovanu potragu (unqualified lookup)* za deklaracijom tog imena, određujući tako entitet na koji se to ime odnosi. Na primer:

```
class Clock {  
    ...  
    void setTime (int hh, int mm, int ss);  
    ...  
    int h, m, s;  
};
```

Identifikatori *setTime*, *h*, *m*, i *s* su deklarisani kao članovi klase, pa imaju opseg važenja klase

```
void Clock::setTime (int hh, int mm, int ss) {  
    h = (hh>=0 && hh<=23) ? hh : 0;  
    m = (mm>=0 && mm<=59) ? mm : 0;  
    s = (ss>=0 && ss<=59) ? ss : 0;  
}
```

Identifikatori *hh*, *mm*, i *ss* su deklarisani kao parametri funkcije, pa imaju opseg važenja te funkcije

Identifikatori *hh*, *mm*, i *ss*, ali i *h*, *m*, i *s* se ovde mogu koristiti nekvalifikovano, jer je ovo kod unutar njihovog opsega važenja

Opseg važenja

- ❖ Nekvalifikovana pretraga imena podrazumeva da se za dato ime pretražuje određena oblast važenja i, ako se u njoj nađe deklaracija tog imena, prevodilac datu upotrebu imena vezuje za deklarisani entitet; u suprotnom, eventualno pretražuje neku drugu, okružujuću oblast važenja i u njoj traži deklaraciju tog imena itd. Ukoliko takvu deklaraciju ne nađe, upotreba imena nije ispravna
- ❖ Ovakva pretraga najpre po tekućoj oblasti važenja, pa onda po okružujućoj, može da se shvati i kao činjenica da ime deklarisano u ugnezđenoj oblasti važenja *sakriva* isto ime u okružujućoj oblasti važenja.

Na primer:

```
int x = 0;
void f () {
    int x = 1;
    x = 3;
}
struct Dummy {
    int x;
};
```

Globalno ime *x* ima opseg važenja odavde do kraja ove jedinice prevodenja (fajla)

Početak oblasti važenja bloka

Lokalno ime *x* ima oblast važenja odavde do kraja ovog bloka i sakriva globalno *x*

Odnosi se na lokalno ime *x* iz oblasti važenja ovog bloka, a ne na globalno *x*

Završetak oblasti važenja bloka

Ovo ime *x* ima oblast važenja ove strukture/klase

- ❖ Dakle, u oblasti važenja, ime se može koristiti neposredno, nekvalifikovano. Van oblasti se može koristiti ili samo na određene načine, ili nikako
- ❖ Pojam oblasti važenja je isključivo vezana za tekst programa i vreme prevođenja, dakle statički koncept, bez semantike koja se odnosi na vreme izvršanja

Opseg važenja

- ❖ Opseg važenja imena počinje od njegove tzv. *tačke deklarisanja (point of declaration)*, koja se razlikuje za različite entitete, na primer:

- za enumeracije, klase ili šablove, to je odmah iza mesta identifikatora:

```
class X {  
    X (const X&);  
    ...  
};
```

Ovo ime *X* odnosi se baš na ovu klasu *X* koja se definiše

- za varijable, to je odmah iza deklaratora, a pre inicijalizatora:

```
const int x = 1, *p = &x;  
{  
    int* x[x] = {x};  
}
```

Ime *x* već važi, pa je ovo u redu

Ovo ime *x* odnosi se na ono tipa *const int*,

Ovo ime *x* odnosi se na ono tipa *int*[1]*, pa prvi element niza ukazuje na samog sebe

Opseg važenja bloka

- ❖ Potencijalni opseg važenja varijable koja je uvedena deklaracijom unutar bloka (složene naredbe) počinje od tačke deklarisanja i završava se na kraju tog bloka. Stvarni opseg važenja je isti kao i potencijalni opseg važenja, osim ako postoji ugnezđeni blok sa deklaracijom koja uvodi identično ime; u takvom slučaju, iz potencijalnog opsega važenja se isključuje taj ugnezđeni blok:

```
void f () {  
    int x = 1;  
  
    x = 2;  
    {  
        long x = 0;  
        x = 2;  
    }  
    x = 3;  
}  
  
int y = x;
```

Lokalno ime *x*, sakriva isto to globalno ime

Drugo lokalno ime *x*, sakriva isto to ime iz okružujućeg bloka

Ovo ime *x* odnosi se na ono u ugnezđenom bloku, tipa *long*

Ovo ime *x* odnosi se na ono u okružujućem bloku, tipa *int*

Greška u prevodenju, *x* nije u opsegu važenja

- ❖ Potencijalni opseg važenja imena formalnog parametra funkcije počinje od tačke deklarisanja i završava se na kraju cele deklaracije te funkcije
- ❖ Potencijalni opseg važenja imena deklarisanih u hvataču izuzetka (parametar *catch* bloka) počinje od tačke deklarisanja i završava se na kraju tog *catch* bloka, i nije u opsegu važenja u drugom *catch* bloku istog *try* konstrukta

Opseg važenja bloka

- ❖ Inicijalna naredba naredbe `for` može biti deklaracija varijable; njeno ime ima potencijalni opseg važenja od tačke deklarisanja do kraja naredbe koja čini telo te naredbe `for`, i ne važi van te naredbe. Na primer:

```
void read (int a[], int b[], int size) {  
    for (int i=0; i<size; i++)  
        cin>>a[i];  
  
    for (int i=0; i<size; i++)  
        cin>>b[i];  
}
```

- ❖ Međutim:

```
template<typename T>  
int search (T array[], int size, T x) {  
    for (int i=0; i<size && array[i]!=x; i++);  
  
    if (i<size) return i;  
    else return -1;  
}
```

- ❖ Ako je potrebno koristiti ovu varijablu i nakon petlje, potrebno je deklaraciju izvući izvan naredbe `for`:

```
template<typename T>  
int search (T array[], int size, T x) {  
  
    int i=0;  
    for (; i<size && array[i]!=x; i++);  
  
    if (i<size) return i;  
    else return -1;  
}
```

Opseg važenja imena `i` je samo naredba `for`, pa se iza kraja te naredbe može ponovo definisati nova varijabla sa istim imenom, pošto prethodna više ne važi

Opseg važenja imena `i` je samo naredba `for`

Greška u prevodenju, `i` nije u opsegu važenja

Opseg važenja bloka

- ❖ Opštije, varijable se mogu deklarisati na sledećim mestima unutar naredbi, i tada takva imena imaju opseg važenja samo do kraja te naredbe:

- kao inicijalna naredba naredbi *if*, *switch* i *for*:

```
const string myString = "My Hello World Greeting";
```

```
if (const auto it = myString.find("Hello"); it != string::npos)
    cout << it << " Hello\n";
```

Ovo je inicijalna naredba naredbe *if*

Ovo je uslov naredbe *if*

```
if (const auto it = myString.find("World"); it != string::npos)
    cout << it << " World\n";
```

Opseg važenja imena *it* je samo naredba *if*, pa se iza kraja te naredbe može ponovo definisati nova varijabla sa istim imenom

- kao uslov naredbi *if*, *switch*, *while* i *for*:

```
Base* pb = ...;
```

```
if (Derived* pd = dynamic_cast<Derived*>(pb)) {
    pd->f();
    ...
}
```

Opseg važenja imena *pd* je samo naredba *if*, odnosno blokovi njenih *then* i *else* delova, pa se iza kraja te naredbe može ponovo definisati nova varijabla sa istim imenom

Opseg važenja bloka

- ❖ I klasa može biti definisana unutar tela funkcije - tzv. *lokalna klasa (local class)*. Takva klasa nije dostupna van opsega važenja te funkcije i ima neka ograničenja: ne može imati statičke podatke članove, njene funkcije članice moraju biti definisane unutar definicije te klase i drugo
- ❖ Ovakve klase tipično se prave da bi se na licu mesta napravili ad-hoc objekti koji zadovoljavaju neki interfejs i samo redefinišu neke polimorfne operacije, tj. objekti su neke klase izvedene iz neke apstraktne klase, s tim da su takve pojave bitne samo na datom mestu, a ne šire, odnosno takvi objekti se prave samo tu, dok ih svi ostali vide samo kroz te apstraktne, generalizovane interfejse, a nikad kao instance te lokalne klase (pa je zato ta klasa i lokalna, enkapsulirana u funkciju)
- ❖ Na primer, neka funkcija očekuje kao parametar objekat koji zadovoljava neki interfejs, kako bi pozvala neku njegovu operaciju (tzv. *callback* mehanizam):

```
class ISubscriber {
public:
    virtual void notify (Message* );
};

class Publisher {
public:
    static void subscribe (ISubscriber* );
};

void Component::Component () {
    struct Sub : public ISubscriber {
        Sub (Component* c) : comp(c) {}
        virtual void notify (Message* msg) { if (comp) comp->processMessage(msg); }
        Component* comp;
    };
    Publisher::subscribe(new Sub(this));
}
```

Da bi se neko prijavio da prima objave (notifikacije), mora da zadovolji ovaj interfejs, tj. (re)definiše operaciju *notify*

Da bi se neko prijavio da prima objave (notifikacije), mora da se prijavi ovom “oglašivaču” operacijom *subscribe*

Da bi se komponenta prijavila “oglašivaču”, pravi jedan ad-hoc objekat lokalne klase *Sub*, koja služi samo da redefiniše polimorfnu operaciju *notify*, kao i da sačuva pokazivač na komponentu kojoj treba da prosledi poruku kad je primi objavom

Opseg važenja klase

- ❖ Oblast važenja imena deklarisanog unutar definicije klase je od tačke deklaracije do ostatka definicije te klase, ali se proteže i kroz tela svih funkcija članica te klase i izvedenih klasa, čak i ako su ta tela navedena van definicije klase ili pre date deklaracije
- ❖ Unutar definicije klase mogu se navoditi deklaracije ne samo objekata i funkcija, nego i tipova (*typedef* sinonimi, enumeracije, klase itd.); na primer:

```
class DatabaseManager {  
public:  
  
    enum DBStatus { ok, failed, refused };  
  
    DatabaseManager (const char* name);  
  
    DBStatus openConnection();  
    DBStatus closeConnection();  
  
    DBStatus getConnectionStatus();  
  
    DBStatus performQuery (const char* sqlQuery);  
  
private:  
  
    char* name;  
    ...  
};
```

Ova enumeracija bitna je samo za ovu klasu, odnosno za njen interfejs i implementaciju, pa je prirodno da bude logički "upakovana" u tu klasu, kao i da ima opseg važenja te klase; osim toga, nema potrebe da "prlja" globalni prostor imena i generiše potencijalni sukob imena u njemu

Opseg važenja klase

- Ime iz oblasti važenja klase može se koristiti samo na jedan od sledećih načina:
 - Unutar oblasti važenja iste klase ili oblasti važenja izvedenih klasa:

```
DatabaseManager::DatabaseManager(const char* nm) {  
    ...  
    name = new char[sizeof(nm)+1];  
    ...  
}
```

Ime *name* je u opsegu važenja

- Kao desni operand operatora . kom je levi operand objekat te klase ili klase izvedene iz te klase:

```
DatabaseManager* p = new DatabaseManager(...);  
...  
(*p).openConnection();
```

- Kao desni operand operatora -> kom je levi operand pokazivač na objekat te klase ili klase izvedene iz te klase:

```
p->openConnection();
```

Opseg važenja klase

- Kao desni operand operatora :: kom je levi operand ta klasa ili klasa izvedena iz te klase. Na primer, čest je slučaj da redefinisana operacija u izvedenoj klasi treba da uradi isto što i metoda osnovne klase, ali i još nešto (u bilo kom redosledu); drugim rečima, da treba da pozove metodu iz osnovne klase:

```
class Base {  
public:  
    virtual void f();  
};  
  
class Derived : public Base {  
public:  
    virtual void f();  
};  
  
void Derived::f () {  
    ...  
    Base::f();  
    ...  
}
```

Poziv po nekvalifikovanom imenu *f* bi značio rekurzivni poziv iste ove funkcije, što nije korektno

- ❖ Operator ::, koji je inače asocijativan sleva nadesno, navodi da prevodilac vrši *kvalifikovanu potragu* (*qualified lookup*) na sledeći način: *Base::f* znači da se najpre sprovodi nekvalifikovana potraga za imenom *Base*; pronađe se to ime i njegova deklaracija, odnosno deklaracija klase *Base*; zatim se u oblasti važenja te klase traži deklaracija za ime *f*; ako se ne nađe takva deklaracija, traži se u njenoj osnovnoj klasi, ako je ima itd.

Opseg važenja klase

- ❖ Unutar definicije klase, odnosno u oblasti važenja klase mogu biti deklarisani drugi tipovi, uključujući i klase. Svi oni se nazivaju *članovima klase* (*class members*)
- ❖ Takve ugnezđene klase, osim što su u oblasti važenja okružujuće klase, nemaju nikakve druge posebne veze; takvo ugnezđivanje ne znači nikako ugrađivanje objekata, već samo logičko “pakovanje” klase
- ❖ Na primer, za implementaciju liste potrebna je struktura koja predstavlja jedan element liste i skladišti pokazivač(e) na sledeći (i prethodni) takav element liste. Ovakva struktura, dakle, bitna je samo za klasu liste i ni za koga drugog. Štaviše, bitna je samo za njenu implementaciju, pa je logično da bude definisana unutar klase liste, i to kao privatna, da ne bi bila deo njenog interfejsa:

```
template<typename T>
class List {
public:
    List ();
    void put (T);
    ...
private:
    struct ListElement {
        ListElement (const ListElement* next);
        ...
    };
};

void List::put (T t) { ... }

List::ListElement::ListElement (const List::ListElement* next) { ... }
```

Ovoj strukturi *ListElement* ne može da se pristupi van opsega važenja klase *List*

Konstruktor ugnezđene klase *ListElement*

Opseg važenja klase

- ❖ Tip definisan unutar oblasti važenja klase se koristi upravo u ovakvim situacijama, kada je on bitan samo za interfejs ili implementaciju te klase. Umesto ugnezđivanja takvog tipa, mogao bi se on deklarisati i globalno, ali to je lošije nego ugnezđivanje iz sledećih razloga:
 - ugnezđeni tipovi su "logički upakovani" u okružujuću klasu, što povećava razumljivost programa jer naglašava značaj i upotrebu takvog tipa
 - ugnezđeni tipovi pripadaju oblasti važenja klase, pa ne "prljaju" globalni prostor imena, odnosno ne uzrokuju sukob imena
 - mogu biti enkapsulirani, jer mogu biti privatni ili zaštićeni, ukoliko su bitni samo za implementaciju, a ne i za interfejs klase

Prostori imena

- ❖ Pre uvođenja koncepta *prostora imena (namespace)* u jezik C++, imena deklarisana izvan svih definicija klasa i tela funkcija imala su *globalni opseg važenja*, što znači da su bila dostupna od tačke deklarisanja do kraja jedinice prevođenja (fajla)
- ❖ Kada globalna imena imaju eksterno vezivanje, uzrokuju češće pojave sukoba imena (*name clash, name conflict*), pa je vrlo rizično upotrebljavati globalna opšta imena zbog velike šanse da se sukobe sa imenom iz nekog drugog dela programa ili biblioteke, npr. sa nazivima opštih tipova podataka (*date, time* itd.), struktura podataka (*list, stack, queue* itd.) i slično
- ❖ Sa ciljem bolje organizacije programa u velikim projektima, uveden je koncept *prostora imena (namespace)*
- ❖ Prostor imena je posebna oblasti važenja koja, slično klasi, predstavlja “logičko pakovanje” entiteta, odnosno njihovih imena koja se van tog ospega važenja mogu koristiti samo na određeni način, kvalifikovanim imenovanjem preko operatora ::. Za razliku od klase, prostor imena nije tip, pa nema instance kao klasa; prostor imena je samo logički paket entiteta
- ❖ Prostor imena se deklariše pomoću ključne reči *namespace*, iza koje sledi niz bilo deklaracija uokvirenih u velike zagrade {}:

```
namespace users {  
    class User {  
        ...  
    };  
    class Role {  
        ...  
    };  
    class UserGroup {  
        ...  
    };  
}
```

Prostori imena

- ❖ Može biti više različitih definicija istog prostora imena; to je tipično slučaj kada se prostori imena definišu u različitim fajlovima, u kojima se deklarišu različiti entiteti (npr. klase) koji pripadaju istom prostoru imena
- ❖ Sve te definicije istog prostora imena se nadovezuju jedna na drugu; sva imena u uniji svih tih definicija imaju isti opseg važenja - ceo taj prostor imena:

```
// File: user.h

namespace users {

    class User {
        ...
    };

    class Role {
        ...
    };

}

// File: usergroup.h

namespace users {

    class UserGroup {
        ...
    };

}
```

Prostori imena

- ❖ Cela standardna biblioteka jezika C++, odnosno svi entiteti u njoj, definisani su u prostoru imena *std*, iako su deklaracije koje se uključuju u fajlove koji koriste delove te biblioteke grupisane po srodnosti u nekoliko desetina zaglavlja
- ❖ Pre uvođenja prostora imena u jezik C++, zaglavla standardne biblioteke imale su ekstenziju *.h*; aktuelne verzije istih zaglavlja, ali sa prostorom imena *std* nemaju ovu ekstenziju:

- Stara verzija biblioteke:

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

int main () {
    int n;
    cin>>n;

    vector<int> a(n);
    for (int i=0; i<n; i++)
        cin>>a[i];

    ...
}
```

- Aktuelna verzija biblioteke:

```
#include <iostream>
#include <vector>

int main () {
    int n;
    std::cin>>n;

    std::vector<int> a(n);
    for (int i=0; i<n; i++)
        std::cin>>a[i];

    ...
}
```

U staroj verziji biblioteke, imena iz biblioteke su globalna i ne zahtevaju kvalifikaciju (*cin*, *vector*)

Različita zaglavla sadrže različite deklaracije, grupisane po srodnosti, i uključuju se po potrebi

U aktuelnoj verziji biblioteke, imena iz biblioteke su u prostoru imena *std* i zahtevaju kvalifikaciju (*std::cin*, *std::vector*)

Prostori imena

- ❖ Oblast važenja imena deklarisanog unutar definicije prostora imena počinje od tačke deklarisanja i prostire se kroz sve definicije istog prostora imena iza te tačke, kao da su sve te definicije prostora imena spojene
- ❖ Van te oblasti važenja, imenu iz oblasti važenja prostora imena pristupa se kvalifikovano, preko operatora ::; međutim, ime iz prostora imena može se *vesti* u drugu oblast važenja i potom koristiti nekvalifikovano, deklaracijom *using*:

```
#include <iostream>
#include <vector>
using std::cin;
using std::vector;

int main () {
    int n;
    cin>>n;

    vector<int> a(n);
    for (int i=0; i<n; i++)
        cin>>a[i];

    ...
}
```

- ❖ Ako je zmetno navoditi sva potrebna imena, mogu se *vesti* sva imena iz navedenog prostora imena direktivom *using namespace*:

```
#include <iostream>
#include <vector>
using namespace std;
```

Imena *std::cin* i *std::vector* su uključena u tekuću oblast važenja (tj. onu u kojoj je deklaracija *using*) i mogu se koristiti nekvalifikovano

Sva deklarisana imena iz prostora imena *std* (a to su samo ona koja su deklarisana deklaracijama iz uključenih zaglavlja) su uključena u tekuću oblast važenja i mogu se u njoj koristiti nekvalifikovano

Prostori imena

- ❖ Prostori imena se mogu proizvoljno ugnezđivati; kvalifikovani pristup imenu iz ugnezđene oblasti važenja vrši se višestrukim nadovezivanjem operatora ::, npr. *domain::core::users::User*
- ❖ U svakoj jedinici prevođenja podrazumeva se jedan implicitno definisan, bezimeni prostor imena, tzv. *globalni opseg važenja* (*global scope* ili *file scope*); ime deklarisano van svih definicija funkcija, klase ili prostora imena ima *globalnu oblast važenja* ovog implicitnog prostora imena, počev od tačke deklarisanja, do kraja fajla u kom je ta deklaracija
- ❖ U nekoj ugnezđenoj oblasti važenja u kojoj je globalno ime skriveno, tom imenu se može pristupiti kvalifikacijom preko unarnog operatora :: (bez levog operanda):

```
int x = 0;                                Globalno ime x, važi od tačke deklarisanja, do kraja ovog fajla
void f () {                                 Lokalno ime x, sakriva globalno ime x
    int x = 2;                             Odnosi se na lokalno x
    x = 3;                               Odnosi se na globalno x
    ::x = 3;                            Odnosi se na globalno x
}
int* p = &x;                                Odnosi se na globalno x, jer je lokalna oblast važenja bloka završena
```

- ❖ Može se definisati i bezimen prostor imena (*unnamed namespace*): opseg važenja imena deklarisanih u njemu uključuje i okružujuću oblast važenja, ali takva imena imaju interno vezivanje; na ovaj način se mogu deklarisati globalna imena sa internim vezivanjem (umesto stare upotrebe specifikatora *static*), odnosno delovi implementacije nekog modula koji su skriveni od drugih modula:

```
namespace {
    ...
}
```

Enumeracija sa opsegom važenja

- ❖ Enumeracije koje su do sada pominjane uvode enumeratore (simboličke konstante) u okružujući opseg važenja, što znači da se imena tih enumeratora mogu sukobiti sa drugim istim imenima u oblasti važenja u kojoj je ta enumeracija definisana
- ❖ Da bi se ovo sprečilo, enumeracija se može deklarisati tako da bude *sa opsegom važenja (scoped enumeration)*, navođenjem ključne reči *class* ili *struct*, svejedno:

```
enum class Color { red, green, blue };
```

- ❖ Sada su enumeratori u oblasti važenja svoje enumeracije, pa se van te oblasti važenja mogu koristiti samo kvalifikovano:

```
Color r = Color::blue;

switch (r) {
    case Color::red : std::cout << "red\n"; break;
    case Color::green: std::cout << "green\n"; break;
    case Color::blue : std::cout << "blue\n"; break;
}
```

- ❖ Za ovakve enumeracije ne postoji implicitna konverzija u integralne tipove; može se vršiti eksplicitna konverzija operatorom *static_cast*

Glava 9: Izrazi

- ❖ Izrazi
- ❖ Operatori
- ❖ Lvrednosti



Izrazi

- ❖ Izraz (*expression*) je iskaz koji se sastoji od niza operacija (*operation*) nad operandima (*operand*), koji podrazumeva neku obradu i može proizvesti rezultat i bočne efekte (*side effect*); operacije se zadaju operatorima (*operator*)
- ❖ Rezultat operacije (i izraza) naziva se vrednost (*value*)
- ❖ Operacija ima određen broj operanada i može proizvesti rezultat, koji se, ako postoji, potom može koristiti kao operand druge operacije; tako se grade složeni izrazi. Na primer:

a+b*c

i + p->val() - b<<k

```
cout << "Counter = " << p->val() << "\n"
```

- ❖ Kada analizira izraz, predilac kontroliše:
 - sintaksu izraza, uključujući i postojanje zahtevanog broja operanada svakog operatora
 - tipove operanada: ugrađeni operatori zahtevaju operande određenih tipova
 - svojstvo lvalue

Izrazi

- ❖ Postupak određivanja operanada operatora zasniva se na:
 - prioritetu operatora
 - načinu grupisanja (asocijativnosti) operatora: ako se rezultat operatora može koristiti kao operand tog operatora, da li se operacije u nizu izračunavaju sleva nadesno ili zdesna nalevo
 - podrazumevani način se može promeniti zagradama koje uokviruju podizraze
- ❖ Na primer:

$a+b*c$

Izračunava se kao $a+(b*c)$, jer operator * ima viši prioritet nego operator +

$(a+b)*c$

Promena načina izračunavanja zagradama (podizrazima)

$cout << p->inc()$

Izračunava se kao $cout << ((p->inc)())$, jer operatori $->$ i $()$ imaju viši prioritet nego operator $<<$

$**p = 3$

Izračunava se kao $*(*p)$, jer operator * grapiše zdesna nalevo

$a+b+c$

Izračunava se kao $(a+b)+c$, jer operator + grapiše sleva nadesno

Operatori

- ❖ Jezici C i C++ su veoma bogati operatorima. Zapravo je najveći deo obrade u tipičnom C/C++ programu definisan izrazima
- ❖ *Bočni efekat (side effect)* se naziva pojava da funkcija/operacija, za koju se izračunavanje rezultata (povratne vrednosti) smatra primarnim efektom, izvrši neku promenu u svom okruženju, npr. promenu svojih argumenata/operanada. U klasičnoj teoriji programiranja, bočni efekti se smatraju lošom praksom, jer smanjuju razumljivost programa: funkcija/operacija ima efekte koje čitalac ne očekuje i ne vidi direktno, jer je fokusiran na njen rezultat
- ❖ Potpuno suprotno ovom uverenju, mnogi operatori na jezicima C i C++ imaju bočne efekte, tj. menaju neki od svojih operanada. Zapravo je za većinu njih taj bočni efekat upravo njihova primarna uloga!
- ❖ Ovo je posledica izvorne orijentacije jezika C prema konciznim i efikasnim izrazima u kojima programer sugeriše optimizacije prevodiocu: raniji prevodioci nisu bili toliko napredni i prevodenje u optimizovani kod je bilo jednostavnije ukoliko sam izraz definiše način korišćenja neke vrednosti koja je pročitana ili proizvedena kao rezultat u jednoj operaciji kao operand neke naredne

Operatori

- ❖ Na primer, operatori inkrementiranja `++` i dekrementiranja `--` imaju bočne efekte i imaju dva oblika:
 - prefiksni: `++operand` inkrementira operand, a kao rezultat vraća njegovu novu vrednosti
 - postfiksni: `operand++` inkrementira operand, a kao rezultat vraća njegovu staru vrednosti (pre inkrementiranja)

Slično važi i za dekrementiranje. Na primer:

`a = i++;`
`b = --k;`

a dobija vrednost i pre inkrementiranja; b dobija vrednost k nakon dekrementiranja

- ❖ Dodela vrednosti je takođe operator: osim što dodeljuje vrednost (bočni efekat, zapravo primarna uloga), on vraća vrednost; pritom, grupiše zdesna nalevo:

`x = y = f() + z`

Izračunava se: $x = (y = (f() + z))$

- ❖ Postoje operatori složene dodele: `a+=b` znači isto što i `a = a+b`, osim što se operand (ako je podizraz) `a` izračunava samo jednom; na primer:

`*p++ += 3`

Izračunava se kao $((p++)) += 3$. Izraz `*p++ = *p++ + 3` bi inkrementirao p dva puta*

Operatori

- ❖ Redosled izračunavanja operanada neke operacije, kao i redosled izračunavanja vrednosti operacije i njenih bočnih efekata, ne samo u odnosu jednih na druge, nego i u odnosu na druge operacije unutar izraza, nije uvek precizno definisan, što znači da se ostavlja prevodiocu (ili hardveru procesora u vreme izvršavanja) da premešta instrukcije, izvršava ih isprepletano ili uporedo
- ❖ Postoje precizna pravila koja definišu parcijalni redosled izvršavanja, dok ono za šta nije definisano uređenje može da se izvršava proizvoljno i stoga efekti ne moraju biti uvek određeni. Pritom, mnogi operatori uključuju izračunavanje identiteta opranada (onih na koje deluju bočnim efektima), izračunavanje vrednosti i izvršavanje bočnih efekata
- ❖ Pravila jezika samo definišu odnos *sekvenciranja* (*sequencing*), kao parcijalne relacije između dva izračunavanja (vrednosti ili bočnih efekata) koja zahteva da se jedno završi pre nego što drugo počne; ako ne postoji ovakva relacija, izračunavanja se mogu preplitati ili izvršiti sekvencijalno u proizvolnjem redosledu
- ❖ Na primer, redosled izračunavanja operanada nekih operacija nije definisan i nije obavezno sekvenciran:

`i = ++i + i++;`

Nedefinisano ponašanje. Ovo je svakako loš stil programiranja!

Operatori

- ❖ Neka od (brojnih i složenih) pravila redosleda izračunavanja (*order of evaluation*) su:
 - sva izračunavanja vrednosti i bočnih efekata celog izraza završavaju se pre nego što počne izračunavanje vrednosti i bočnih efekata narednog izraza
 - izračunavanje svih argumenata funkcije i svih operacija pre poziva funkcije završava se pre bilo kog izvršavanja unutar pozvane funkcije; analogno važi za povratak iz funkcije
 - izračunavanje vrednosti (ali ne obavezno i bočnih efekata) svih operanada bilo kog operatora završava se pre izračunavanja vrednosti (ali ne obavezno i bočnih efekata) tog operatora
 - za neke operatore definisan je i sekvenciran redosled izračunavanja operanada (za neke samo izračunavanja vrednosti, za neke i bočnih efekata): `++`, `--`, `&&`, `||`, `?::`, `,`, `=`, `@=` (operatori složene dodele), `[]`, `>>`, `<<`, dok za ostale nije
- ❖ Logički operatori `&&` (logičko I) i `||` (logičko ILI) izračunavaju se po pravilu "kratkog spoja" (*short-circuit evaluation*): ako prvi operand definitivno određuje vrednost rezultata, drugi se i ne izračunava:

`if (p && p->f()) ...`

Ovo je bezbedno, jer ako pokazivač `p` ima vrednost `null`, drugi operand operatora `&&` neće ni biti izračunavan, pa ovaj pokazivač neće biti korićen neispravno

Lvrednosti

- ❖ Zbog čega su sledeći izrazi ispravni:

```
&a  
*(p+3)  
x += 1
```

a sledeći nisu?

```
&(i+3)  
(i+j) += k
```

- ❖ Intuitivno je jasno da u nekim od ovih situacija operacija nije dozvoljena jer njen operand ne predstavlja nekakav "čvrst" objekat koji se može identifikovati, koji zauzima neku memoriju, ali je to suviše neformalno tumačenje koje prevodilac ne može da sprovodi: potrebna su precizna, formalna pravila kojima prevodilac može da proverava ispravnost ovakvih izraza
- ❖ Još je u jeziku C postojao koncept (on vodi poreklo iz nekih starijih jezika) tzv. *lvrednosti* (*lvalue*), kao "nečega što može da stoji sa leve strane znaka =" i koji se koristio za ovaku namenu; ovaj koncept u jeziku C++ ima složenije značenje, ali i veći značaj, pošto se operatori mogu preklapati, pa operandi mogu biti i objekti klase
- ❖ Pored sintaksne provere i provere tipova operanada, prevodilac proverava i svojstvo *lvrednosti* za izraze

Lvrednosti

- ❖ *Vrednost (value)* je rezultat izraza (uključujući i pojedinačnu operaciju kao prost izraz)
- ❖ Svaka vrednost ima svoju *vrednosnu kategoriju (value category)* koja može, ali ne mora biti *lvrednost* (pored još nekih). Dakle, *lvrednost* je binarno svojstvo vrednosti: neka vrednost ili jeste, ili nije *lvrednost*
- ❖ Lvrednosti su sledeće stvari:
 - ime varijable, funkcije ili podatka člana, bez obzira na tip
 - string-literal (ali ne i ostali literali)
 - rezultat poziva funkcije, uključujući i operatorske funkcije, ako je taj rezultat referenca, koja se zato naziva referenca na lvrednost (*lvalue reference*)
 - rezultat sledećih ugrađenih operatora:
 - dodele $a=b$ i složene dodele $a+=b$ itd.
 - prefiksнog inkrementiranja $++a$ i dekrementiranja $--a$ (ali ne i postfiksног)
 - indirekcije $*p$
 - indeksiranja $a[i]$, gde je jedan operand pokazivač ili niz koji je lvrednost
 - izraza za pristup članu objekta ($a.m$) ili indirektan pristup članu objekta ($p->m$), osim ako je m enumerator ili nestatička funkcija članica
 - izraza a, b , ako je b lvrednost
 - izraza $a?b:c$, ako je b ili c lvrednost, u zavisnosti od toga šta je rezultat izraza a

Lvrednosti

- ❖ Neki ugrađeni operatori zahtevaju kao neki svoj operand lvrednost, pa se kao taj operand može pojaviti samo izraz koji jeste lvrednost; ako operator ima bočni efekat nad operandom, onda taj operand mora biti lvrednost
- ❖ Činjenica da rezultat nekog operatora jeste lvrednost prestavlja činjenicu da ta vrednost upućuje na neki "čvrst" entitet, koji ima svoj identitet, a koja se onda može upotrebiti kao operand operatora koji zahteva lvrednost
- ❖ Tako prevodilac proverava ispravnost izraza. Na primer:
- Operator dodele = zahteva kao svoj levi operand lvrednost, da bi nad njim proizveo bočni efekat, ali ne zahteva da njegov desni operand bude lvrednost; operator + ne proizvodi lvrednost:
 - a = (b+c) Ispravno pod uslovom da je a ime variable, što jeste lvrednost; desni operand operatora = ne mora biti lvrednost
 - (b+c) = a Greška u prevodenju, jer levi operand operatora = nije lvrednost
- I rezultat operatora dodele jeste lvrednost, koja se odnosi na istu stvar na koju i njegov levi operand (koji svakako jeste lvrednost), pa se rezultat ovog operatora može upotrebiti tamo gde se zahteva lvrednost; na primer:

$(a = b) = c$

Rezultat izraza $a=b$ jeste lvrednost koja se odnosi na a , pa se njemu dodeljuje vrednost c

Lvrednosti

- Operator `&` zahteva operand koji jeste lvrednost, ali mu rezultat nije lvrednost. Nasuprot tome, operator indirekcije `*` ne zahteva da operand bude lvrednost, ali mu rezultat jeste lvrednost:

`&a`

Ispravno pod uslovom da je `a` ime varijable ili funkcije, što jeste lvrednost

`&a = ...`

Greška u prevodenju, jer levi operand operatora `=` nije lvrednost

`&(a+3)`

Greška u prevodenju, jer operand operatora `&` nije lvrednost

`*(p+3)`

Ispravno, jer operator `*` ne zahteva da operand jeste lvrednost, dok mu rezultat jeste lvrednost

- Operator `a << b` daje rezultat koji predstavlja binarnu vrednost celobrojnog operanda `a` pomerenu za `b` bita uлево. Kako znati da li taj operator proizvodi bočni efekat ili ne, tj. da li menja (pomera) svoj levi operand? Ako ga menja, taj operand bi morao biti lvrednost, ali ovaj operator ne zahteva da levi operand bude lvrednost, pa onda sigurno ne proizvodi bočni efekat

- Rezultat poziva funkcije je lvrednost akko funkcija vraća referencu na lvrednost, na primer:

`int& f();`

Funkcija `f` vraća referencu na lvrednost, pa rezultat poziva te funkcije jeste lvrednost

`f() += 1;`

Rezultat poziva funkcije `f` je referenca koja upućuje na neki objekat tipa `int`

`X& X::operator+=(int);`

Ova operatorska funkcija klase `X` ima parametar tipa `int` i vraća referencu na lvrednost

`X x;`

`X* p = &(x += 3);`

Za operaciju `x+=3` poziva se `x.operator+=(3)`. Ova operatorska funkcija vraća referencu na neki objekat tipa `X` i taj rezultat jeste lvrednost, pa može biti operand operatora `&`

Lvrednosti

- ❖ Termin “lvrednost” ima poreklo u tome što “može da stoji sa leve strane znaka dodele”, ali to ne važi za svaku lvrednost: samo *promenljive lvrednosti (modifiable lvalues)* mogu da stoje sa leve strane znaka dodele (i složene dodele)
- ❖ Ime niza, funkcije ili konstantnog objekta nije promenljiva lvrednost, pa ne može stajati sa leve strane operatora dodele ili složene dodele:

```
int a[5], f();
```

```
const int c = 3;
```

```
a = ...  
f = ...  
c = ...
```

```
&a  
&f  
&c
```

Greška u prevodenju, jer *a, f i c* nisu promenljive lvrednosti

Ispravno, jer *a, f i c* jesu lvrednosti

- ❖ Pojam lvrednosti je na jeziku C++ upravo značajan zbog toga što postoji mogućnost korišćenja objekata klase po vrednosti, preklapanja operatora za korisničke tipove (klase), sa notacijom kao za ugrađene tipove i njihove operande; blisko povezan sa svim tim jeste i pojam reference na lvrednost, koja je i uvedena sa ciljem podrške takvom korišćenju, ali je sve to i značajno zakomplikovalo jezik

Glava 10: Funkcije

- ❖ Deklaracija i definicija funkcije
- ❖ *Inline* funkcije
- ❖ Podrazumevani argumenti
- ❖ Preklapanje funkcija
- ❖ Poziv funkcije



Deklaracija i definicija funkcije

- ❖ Funkcije su jedini oblik potprograma na jezicima C i C++: procedure su specijalne vrste funkcija koje imaju povratni tip *void* (ne vraćaju rezultat)
- ❖ Deklaracija funkcije koja nije definicija može da se pojavi u bilo kom opsegu važenja (na bilo kom mestu)
- ❖ Deklaracija funkcije u opsegu važenja klase deklariše funkciju članicu, osim ako je deklarisana specifikatorom *friend*
- ❖ Ako deklaracija funkcije uključuje i njen telo, onda je definicija
- ❖ Definicija funkcije može da se pojavi samo u opsegu važenja prostora imena i unutar definicije klase. Definicija funkcije ne može da se pojavi unutar druge funkcije - ne postoji staticko ugnezđivanje funkcija kao npr. na jeziku Pascal. Naravno, dozvoljeno je dinamičko ugnezđivanje poziva funkcija, pa i rekurzija
- ❖ Definicija funkcije može, umesto tela, da sadrži = *delete*. Ovakva funkcija naziva se *obrisanom (deleted)*. Svaka upotreba ovakve funkcije, npr. njen poziv, je neispravna i uzrokuje grešku u prevođenju; ovako se mogu sprečiti npr. neke implicitne konverzije ili inicijalizacije implicitno generisanim konstruktorima; na primer:

```
class X {  
public:  
    X (const X&) = delete;  
    ...  
};
```

Konstruktor kopije je obrisan, pa se objekti ove klase ne mogu inicijalizovati kopiranjem

Deklaracija i definicija funkcije

- ❖ Funkcija može, a ne mora imati parametre. Funkcija koja nema parametre deklariše se kao $f()$ ili $f(void)$, svejedno (radi se o istoj stvari)
- ❖ U tip funkcije ulazi povratni tip, kao i tipovi svih parametara, pri čemu se ne pravi razlika između sledećih tipova parametara:
 - niza elemenata tipa T sa dimenzijom ili bez nje i pokazivača na T
 - funkcije nekog tipa i pokazivača na funkciju istog tipa
 - parametra sa cv-kvalifikacijom i bez nje

Na primer, sledeće deklaracije istih imena su deklaracije istih funkcija (a deklaracije različitih imena su deklaracije različitih funkcija):

```
void f(int);
void f(const int);

void g(const int* );
void g(const int* const);

void h(int[]);
void h(int[5]);
void h(int* );
```

Deklaracija i definicija funkcije

- ❖ Povratni tip funkcije ne može biti funkcija ili niz (ali može biti pokazivač ili referenca na funkciju ili niz)
- ❖ Povratni tip funkcije može da se navede i iza parametara i znaka `->`, što olakšava pisanje i čitanje deklaracija ako je povratni tip složen, ili ako se ne može odrediti, npr. zato što zavisi od tipova argumenata unutar šablonu:

```
auto redirect (int*)(int)) -> int(*)(int*)(int));
```

```
template <typename U, typename V> Ovo bi bilo prilično teško napisati klasičnom notacijom  
auto combine (U u, V v) -> decltype(u+v);
```

- ❖ Povratni tip ne mora da se navodi eksplisitno, nego da se ostavi prevodiocu da ga sam izvede, na osnovu tipa izraza iza naredbe `return`; svi tipovi iza višestrukih naredbi `return` moraju biti konzistentni; ova mogućnost nije dozvoljena za virtuelne funkcije:

Povratni tip ove funkcije je `decltype(u*scalar1 + v*scalar2)`

```
template <typename U, typename V>  
auto combine (U u, V v, double scalar1, double scalar2) {  
    return u*scalar1 + v*scalar2;  
}
```

Inline funkcije

- ❖ Kada se u složenom programu temeljno sprovedu svi elementi objektne dekompozicije, tipična situacija jeste ta da većina metoda (tela funkcija) ima vrlo malo linija koda, vrlo retko više od 5-10 linija; sve preko toga može biti signal da je propuštena prilika za apstrakcijom ili dekompozicijom (makar algoritamskom). Duža tela funkcija svakako nisu preporučljiva, jer smanjuju razumljivost
- ❖ Tipični ekstremni primeri su metode koje rade vrlo proste operacije, na primer:
 - samo vraćaju ili postavljaju vrednost atributa (*getter* i *setter* operacije):

```
string Person::getName () const { return this->name; }  
Person& Person::setName (const string& newName) { this->name = newName; return *this; }
```

- predstavljaju “omotače” (*wrapper*) oko neke druge operacije, sa ciljem enkapsulacije:

```
Clock* Clock::create (...) { return new Clock(...); }
```

 - delegiraju poziv jednoj ili nekim drugim operacijama, uz eventualne konverzije argumenata, kao posledica dekompozicije i lokalizacije zajedničkih delova, odnosno svođenja na već postojeće:

```
Clock::Clock (int hh, int mm, int ss) { setTime(hh,mm,ss); }  
bool operator!= (const complex& c1, const complex& c2) { return !(c1==c2); }
```

- ❖ Nije neobičan utisak koji se može steći o dobro dekomponovanom programu, da na neki način “većina metoda ne radi ništa posebno, već sve prepušta drugima, samo delegira pozive drugima”, a ceo program ipak radi složen posao!
- ❖ Ovo je posledica dobre dekompozicije, ali i puno implicitne semantike koja je sadržana u konstruktima jezika, kao što su npr. polimorfni pozivi, pozivi konstruktora osnovnih klasa, konverzije i slično

Inline funkcije

- ❖ Međutim, takve funkcije koje samo delegiraju poziv drugim funkcijama prave nepotreban režijski trošak u vreme izvršavanja: smeštanje povratne adrese na stek, izvršavanje instrukcije skoka u potprogram, indirektni povratak iz potprograma preko povratne adrese skinute sa steka, pa u mnogim slučajevima i kopiranje argumenata na stek i njihovo skidanje sa steka su potpuno nepotreban trošak za funkciju koja će samo ponovo izvršiti poziv druge funkcije ili prosto pročitati ili upisati podatak
- ❖ Zbog ovoga je još odavno osmišljena optimizaciona tehnika u prevodiocima *neposrednog ugrađivanja koda* pozvanog potprograma na mesto poziva (*inlining*): umesto koda za skok u potprogram, sa prenosom argumenata i čuvanjem povratne adrese na steku, kod pozvanog potprograma se neposredno ugrađuje u kod pozivaoca; ovo uzrokuje kraće vreme izvršavanja, ali i nešto veći “memorijski otisak” (*memory footprint*), odnosno veličinu programa
- ❖ U svakom slučaju, navedeni primeri trivijalnih metoda koje delegiraju pozive drugim ili samo čitaju/upisuju podatak sprovođenjem ove optimizacije ne prave nikakav trošak, a imaju značaj u dizajnu programa (bolja dekompozicija, enkapsulacija)
- ❖ Na neki način, ovo se može smatrati korakom unazad u evolutivnom razvoju programiranja: koncept potprograma je u najstarije programske jezike uveden da bi se smanjila redundantnost i unapredila dekompozicija, tj. da se ne bi ponavljaо isti kod na svakom mestu korišćenja, već se on izdvaja u potprogram koji se poziva, potencijalno parametrizovano, sa različitim mesta. Neposredno ugrađivanje u kod radi obrnutu stvar i pravi redundansu, ali je razlika velika:
 - ovaj postupak radi prevodilac potpuno automatski i skriveno od programera
 - redundansa postoji samo u mašinskom kodu, dok je izvorni kod dekomponovan i bez redundanse

Inline funkcije

- ❖ U drugim jezicima, neposredno ugrađivanje u kod je isključivo optimizaciona tehnika čije je sprovođenje diskreciono pravo prevodioca (može da ga sprovodi, ali ne mora, po svom nahođenju) i potpuno je nevidljiva za programera
- ❖ U jezik C++ je ova tehnika uvedena kako koncept jezika, tako što programer može deklarisati funkciju kao *inline*, kao preporuku prevodiocu da izvrši ovu optimizaciju, odnosno telo te funkcije ugradi u kod na mestu poziva
- ❖ Međutim, važno je naglasiti da se semantika programa ni na koji način ne menja, bez obzira na to da li je funkcija *inline* ili ne: sva semantička pravila važe na potpuno isti način; na primer, formalni parametri se inicijalizuju stvarnim argumentima i na kraju funkcije uništavaju na potpuno isti način
- ❖ Štaviše, prevodilac može, ali uopšte ne mora da ispoštuje zahtev za neposredno ugrađivanje u kod. Neki prevodioci će, na primer, odbiti da to urade ako funkcija ima petlju ili lokalni statički objekat, a svakako ne mogu to da urade ako je funkcija rekurzivna
- ❖ Bez obzira na to da li prevodilac uradi ovu optimizaciju ili ne, semantika programa se svakako ne menja
- ❖ U svakom slučaju, kao *inline* treba deklarisati samo funkcije koje su jednostavne i kratke, poput onih navedenih u datim primerima

Inline funkcije

- ❖ Funkcija se može deklarisati kao *inline* navođenjem ovog specifikatora u deklaraciji. Funkcije članice klase X, kao i prijateljske funkcije klasi X koje su definisane u definiciji date klase X (u definiciji klase im je navedeno i telo) su implicitno *inline*, čak i ako se to ne naglasi:

```
inline void strcpy(char* to, const char* from) { while (*to++ = *from++); }
```

```
class Person {
public:
    string getName () const { return name; }
    Person& setName (const string& newName) { name = newName; return *this; }
    ...
};
```

ili:

```
class Person {
public:
    inline string getName () const;
    inline Person& setName (const string& newName);
    ...
};

string Person::getName () const { return name; }
Person& Person::setName (const string& newName) { name = newName; return *this; }
```

Ove funkcije su implicitno *inline*

Inline funkcije

- ❖ Da bi ugradio kod *inline* funkcije na mestu poziva, prevodilac mora da ima kompletну definiciju (tela) te funkcije. Zato su za ove funkcije dozvoljene višestruke definicije u istom programu, ali samo po jedna u svakoj jedinici prevođenja; po pravilu, definicije *inline* funkcija navode se u zaglavljima
- ❖ Čak i ako se nalaze u različitim jedinicama prevođenja, višestruke definicije *inline* funkcija odnose se na istu funkciju i semantika te funkcije se ne menja: ona će imati istu adresu, lokalni statički objekti su jedinstveni za sve definicije (sve iste definicije takvog objekata odnose se na jedan jedinstveni entitet) i slično
- ❖ Sve definicije iste *inline* funkcije u različitim jedinicama prevođenja moraju biti identične; ako nisu, ponašanje programa je nedefinisano
- ❖ Sa jedne strane, prevodilac ne mora ispoštovati zahtev za ugrađivanje *inline* funkcije u kod pozivaoca, već generisati kod za uobičajen poziv. Sa druge strane, ne brani se prevodiocu da kod i neke druge funkcije koja nije *inline* ugradi na mesto poziva kao optimizaciju. Zbog toga se osnovni smisao *inline* funkcija zapravo gubi, pa *inline* formalno ne znači ništa više nego to da su dozvoljene višestruke definicije iste funkcije u različitim jedinicama prevođenja u istom programu
- ❖ Zbog toga je u jezik uveden i koncept *inline* statičkih varijabli (objekata ili referenci): dozvoljene su višestruke definicije *inline* statičke varijable u različitim jedinicama prevođenja istog programa (ali samo po jedna definicija u jednoj jedinici), pod uslovom da su te definicije identične; sve one se odnose na isti entitet
- ❖ Ovo omogućava kompletne definicije biblioteka u fajlovima zaglavljima, bez potrebe za definisanje objekata u *.cpp* fajlovima

inline istream cout;

Podrazumevani argumenti

- ❖ Vrlo često je potrebno da funkcija ima nekoliko varijanti, odnosno da se može pozvati sa nekim argumentom ili bez njega, pri čemu izostavljeni argument treba da uzme neku *podrazumevanu vrednost (default argument)*; na primer:

```
complex::complex(double re=0.0, double im=0.0);
```

```
template <typename T>
list<T> list::insert (T t, int at=0);
```

Podrazumevani argument

Podrazumevani argument, sa značenjem da se podrazumeva početak ako se izostavi

- ❖ Ako se neki stvarni argument u pozivu ovakve funkcije izostavi, taj argument dobiće podrazumevanu vrednost navedenu u deklaraciji funkcije:

```
complex c1(1.,1.), c2(1.), c3;
```

```
list<complex> lst;
lst.insertAt(c1).insertAt(c2,1).insertAt(c3);
```

Inicijalizacija: *c1(1.,1.), c2(1.,0.0), c3(0.0,0.0)*

insertAt(c1,0).insertAt(c2,1).insertAt(c3,0)

Podrazumevani argumenti

- ❖ Podrazumevani argumenti nisu deo tipa funkcije. Ako se deklaracije funkcije ponavljaju, ne smeju ponovo navoditi podrazumevani argument za isti parametar, čak i ako je identičan. Na mestu poziva funkcije, podrazumevani argumenti predstavljaju uniju svih do tada deklarisanih podrazumevanih argumenata, s tim da ne sme postojati parametar koji nema podrazumevani argument iza parametra koji ima podrazumevani argument:

```
void f(int p1, int p2 = 2,int p3);
void f(int p1, int p2 = 2,int p3);
void f(int p1 = 1, int p2, int p3);
f(3);
void f(int p1, int p2,int p3=3);
f(0);
```

Greška u prevodenju: ponovljena definicija podrazumevanog argumenta

Greška u prevodenju: ne može da bude
void f(int=0,int=0,int)

Poziv: *f(0,2,3)*

- ❖ U principu, ovaj koncept predstavlja notacionu pogodnost: da ga nema, bilo bi potrebno pisati više varijanti funkcija sa različitim parametrima; na primer, umesto:

```
double log (double x, double base=10.0);
```

moralo bi da se piše:

```
double log (double x, double base);
double log (double x) { return log(x,10.0); }
```

Preklapanje funkcija

- ❖ Ponekad postoji potreba da se naprave potprogrami koje rade logički istu stvar, samo sa drugačijim brojem ili tipovima parametara. U tradicionalnim jezicima, za ovakve potprograme morala bi da se osmisle različita imena, jer prevodilac poziv potprograma vezuje sa pozvanim potprogramom samo na osnovu imena
- ❖ Na jeziku C++, različite funkcije mogu imati isto ime, ukoliko se dovoljno razlikuju po broju ili tipovima parametara; ovo se naziva *preklapanje* (ili *preopterećenje*) funkcija (*function overloading*)
- ❖ Prevodilac vezuje poziv funkcije za pozvanu funkciju ne samo na osnovu imena, nego i u zavisnosti od broja i tipova stvarnih argumenata koje uparuje sa tipovima formalnih parametara, pri čemu se pretražuju oblasti važenja u zavisnosti od toga kojim oblastima važenja pripadaju argumenti (tzv. *argument-dependent lookup*, ADL i *overload resolution*). Na primer:

```
double max (double, double);
const char* max (const char*, const char*);
...
const char* s = max("March", "January");
double d = max(3.6, 5);
```

Ove funkcije vraćaju "veći" od dva parametra, šta go da su

Poziva se *max(const char*, const char*)*

Poziva se *max(double, double)*

Preklapanje funkcija

- ❖ U poređenje tipova ulaze i cv-kvalifikacije, što znači da se razlikuju tipovi koji jesu ili nisu kvalifikovani kao konstantni, uključujući i objekte i njihove funkcije članice koje jesu ili nisu konstantne. Na primer:

```
Task& TaskQueue::at (int position = 0);
```

```
inline const Task& TaskQueue::at (int position = 0) const;  
    return const_cast<TaskQueue*>(this)->at(position);  
}
```

Sada se na sledećim mestima pozivaju odgovarajuće funkcije, u zavisnosti od konstantnosti objekta za koji se one pozivaju:

```
const TaskQueue* pcq = ...;
```

```
TaskQueue* pq = ...;
```

```
const Task& ct1 = pcq->at();
```

Poziva se *const Task& TaskQueue::at (int = 0) const*

```
const Task& ct2 = pq->at();
```

Poziva se *Task& TaskQueue::at (int = 0)*

```
Task& ct3 = pq->at();
```

Poziva se *Task& TaskQueue::at (int = 0)*

```
Task& ct4 = pcq->at();
```

Greška u prevodenju: poziva se *const Task& TaskQueue::at (int = 0) const*

Preklapanje funkcija

- ❖ Jasno je da će se u slučaju da se formalni parametri neke od preklopljenih funkcija po broju i tipovima u potpunosti slažu sa stvarnim argumentima pozvati baš ta funkcija, ali šta ako takva funkcija ne postoji? Šta ako postoji jedna ili više funkcija koje se ipak mogu pozvati, uz dozvoljene implicitne konverzije tipova stvarnih argumenata u tipove formalnih parametara?
- ❖ Prevodilac sprovodi vrlo složen postupak potrage za funkcijom koja odgovara pozivu na osnovu tipova argumenata i parametara deklarisanih funkcija (ADL i *overload resolution*). Ishod ovog postupka može da bude trojak:
 - Nijedna funkcija ne odgovara pozivu, odnosno nijedna se ne može pozvati čak ni implicitnim konverzijama tipova argumenata u tipove parametara; ovakav poziv onda nije ispravan i prevodilac prijavljuje grešku
 - Postoji više kandidata - funkcija koje se mogu pozvati uz implicitne konverzije tipova argumenata u tipove parametara, s tim da postupak ne preferencira nijednu od njih (nijedna ne odgovara više nego neka od ostalih); i ovaj poziv nije ispravan i prevodilac prijavljuje grešku tipa "višeoznačan poziv funkcije" (*ambiguous function call*)
 - Jedna od funkcija - kandidata bolje odgovara od ostalih, na osnovu kriterijuma preferenciranja funkcija (ukoliko se neka potpuno poklapa po broju i tipovima parametara sa tipovima argumenata, onda je ona sigurno ta, ali to nije jedini slučaj); jedino ovakav poziv je ispravan i za njega će prevodilac generisati kod za poziv odabrane funkcije
- ❖ Ponekad prevodilac može jednoznačno odrediti funkciju koja se poziva, iako to čoveku koji tumači program možda nije najjasnije, jer su implicitne konverzije komplikovane i skrivene. Takve situacije treba izbegavati i takve pozive, odnosno preklopljene funkcije drugačije rešiti:
 - eksplicitnom konverzijom stvarnih argumenata u tačne tipove formalnih parametara, kako bi bilo nedvosmisleno jasno koja funkcija treba da se pozove
 - drugačijim imenovanjem funkcija, kako bi se poziv razrešio prostim imenovanjem funkcije

Preklapanje funkcija

- ❖ U skup funkcija kandidata u postupku ADL ulaze samo funkcije članice klase koja predstavlja tip objekta čija se funkcija poziva, ne i one iz osnovne klase. To znači da funkcija sa istim imenom, ukoliko ne redefiniše virtualnu funkciju osnovne klase, sakriva sve funkcije iz osnovne klase sa istim imenom (isto važi i za druge vrste članova). Na primer:

```
struct B {  
    void f (int);  
};
```

```
struct D : B {  
    void f ();
```

Ova funkcija ne redefiniše, već sakriva funkciju *B::f(int)*

```
int main () {  
    D d;  
    d.f(1);
```

Greška u prevodenju: u skupu kandidata ne postoji funkcija *f* koja može da primi *int*.

Moglo bi ovako: *d.B::f(int)*

- ❖ Ovakva pojava nije dobra, jer iako izvedena klasa nasleđuje funkcije, one više ne čine njen interfejs (ukoliko se objektu pristupa kao instanci izvedene, a ne osnovne klase), što narušava semantiku nasleđivanja. Svi entiteti iz osnovne klase sa datim imenom mogu se uvesti u opseg pretrage unutar izvedene klase direktivom *using*:

```
struct D : B {  
    using B::f;  
    void f() {}  
};
```

```
int main () {  
    D d;  
    d.f(1);
```

U opseg važenja klase *D* uključene su sve funkcije sa imenom *f* iz klase *B*

Sada je ovo ispravno, poziva se *d.B::f(1)*

Preklapanje funkcija

- Ukoliko je funkcija jednoznačno odabrana ovim postupkom potrage kao ona koja treba da bude pozvana, a ta funkcija je označena kao *obrisana (deleted)*, specifikatorom `=delete` u deklaraciji, poziv neće biti ispravan, prevodilac će prijaviti grešku i neće tražiti neku drugu preklopljenu funkciju koju bi mogao da pozove uz implicitne konverzije argumenata. Ovako se mogu zabraniti neke implicitne konverzije, odnosno pozivi neke funkcije za neke tipove argumenata koji bi mogli biti konvertovani u tipove parametara postojećih funkcija. Na primer:

```
int f (double) { return 0; }

int f (int) = delete;

int main () {
    f(1);
}
```

Greška u prevodenju: `f(int)` je obrisana, ne poziva se `f(double)`

- Analogno važi i za proveru prava pristupa: ukoliko odabrana funkcija nije dostupna na mestu poziva, poziv neće biti ispravan, prevodilac će prijaviti grešku i neće tražiti drugu preklopljenu funkciju koju može da pozove, a koja je dostupna; provera prava pristupa vrši se nakon i nezavisno od potrage i odabira pozvane preklopljene funkcije. Na primer:

```
class X {
public:
    int f (double) { return 0; }

private:
    int f (int) { return 1; }
};

int main () {
    X x;
    x.f(1);
}
```

Greška u prevodenju: `X::f(int)` nije dostupna, ne poziva se `X::f(double)`

Poziv funkcije

- ❖ Poziv funkcije obavlja se ugrađenim operatorom poziva funkcije (), čiji je levi operand funkcija, pokazivač na funkciju ili referenca na funkciju, a unutar zagrada su izrazi, razdvojeni zarezima, čije vrednosti predstavljaju stvarne argumente
- ❖ Prilikom poziva funkcije, formalni parametri se, kao objekti, inicijalizuju stvarnim argumentima, uz eventualne implicitne konverzije po potrebi; ukoliko se radi o objektima klasa, pozivaju se njihovi konstruktori:

```
struct X { X (int); }
```

```
X f (X x1) {  
    return 2;  
}
```

```
int main () {  
    f(1);  
}
```

Formalni parametar $x1$ inicijalizuje se stvarnim argumentom 1 kao $X x1(1)$

Povratna vrednost funkcije je objekat tipa X koji se inicijalizuje izrazom iza naredbe *return* pozivom konstruktora $X(2)$

- ❖ Rezultat funkcije jeste vrednost izraza iza izvršene naredbe *return*; ukoliko je povratni tip funkcije *void*, rezultat funkcije nije vrednost (funkcija nema rezultat)

Poziv funkcije

- ❖ Ukoliko je funkcija virtualna funkcija članica, poziv funkcije je polimorfan (tj. polimorfizam se aktivira) ako i samo ako se poziv vrši preko pokazivača ili reference na objekat:

```
class Base {  
    public: virtual void f();  
};
```

```
class Derived : public Base {  
    public: virtual void f() override;  
};
```

```
void g (Base* pb) {  
    pb->f();  
}
```

```
void main () {  
    Derived d;  
    g(&d);  
  
    Base b;  
    g(&b);  
}
```

Ovaj poziv je uvek polimorfan; isto je i za poziv preko reference

Poziva se *Derived::f()*

Poziva se *Base::f()*

Poziv funkcije

- ❖ U suprotnom, ukoliko se poziv funkcije članice vrši preko objekta, poziv nije polimorfan:

```
void h (Base b) {  
    b.f();  
}
```

```
void main () {  
    Derived d;  
    h(d);  
  
    Base b;  
    h(b);  
}
```

Objekat *b* je direktna instanca klase *Base* i sigurno ništa osim toga, pa poziv nije polimorfan. Ovakve pozive prevodilac može da reši statičkim vezivanjem (za vreme prevođenja)

Formalni parametar *b* je objekat klase *Base* i nezavisan je od stvarnog argumenta kojim se inicijalizuje prilikom poziva funkcije

Poziva se *Base::f()* u oba slučaja

- ❖ Dinamičko vezivanje je implementaciona tehnika kojom se realizuju polimorfni pozivi. Treba primetiti to da bi u ovakvima pozivima za objekat (ne preko pokazivača ili reference), čak i ako bi taj poziv bio izvršen dinamičkim vezivanjem, rezultat bio isti (jer VTP objekta tipa *Base* ukazuje na VT baš te klase). Za pozive preko pokazivača ili reference, prevodilac, u opštem slučaju, ne može znati šta se krije iza objekta, jer to može biti i promenljivo (npr. za parametre funkcija), pa će te pozive uvek implementirati dinamičkim vezivanjem

Glava 11: Životni vek varijabli

- ❖ Trajanje skladišta i životni vek
- ❖ Automatski životni vek
- ❖ Statički životni vek
- ❖ Dinamički životni vek
- ❖ Životni vek vezan za nit
- ❖ Privremeni objekti
- ❖ Ugrađeni objekti



Trajanje skladišta i životni vek

- ❖ Svaki *objekat* u programu ima određenu kategoriju *trajanja skladišta (storage duration)*: vreme u toku izvršavanja programa za koje je prostor u memoriji za taj objekat alociran (smatra se zauzetim za potrebe smeštanja tog objekta); te kategorije su:
 - *statičko trajanje (static storage duration)*: prostor za objekat se alocira na početku izvršavanja programa i dealocira na kraju izvršavanja programa; po pravilima jezika, za svaku definiciju ovakvog objekta postoji jedna instanca za vreme izvršavanja; zato prevodilac može (i po pravilu to i radi) alocirati taj prostor u prevedenom fajlu koji predstavlja memorijsku mapu programa prilikom pokretanja
 - *automatsko trajanje (automatic storage duration)*: prostor za objekat alocira se na ulasku u blok i dealocira pri izlasku iz tog bloka; za svaku aktivaciju bloka i svaku definiciju automatskog objekta postoji posebna instanca objekta, pa se ovaj prostor alocira na kontrolnom steku programa, dok se objekti adresiraju relativno u odnosu na vrh steka, kako bi se pristupilo aktuelnoj instanci objekta na koji se definicija odnosi (npr. pri rekurzijama)
 - *dinamičko trajanje (dynamic storage duration)*: prostor za objekat se alocira i dealocira na eksplicitan zahtev, pozivom odgovarajućih funkcija koje alociraju i dealociraju prostor dinamički, sledom izvršavanja programa
 - *trajanje vezano za nit (thread)*: prostor za objekat se alocira kada počinje izvršavanje niti, a dealocira kada se izvršavanje završi; svaka nit ima svoju instancu ovakvog objekta

Trajanje skladišta i životni vek

- ❖ Svaka *varijabla* (objekat ili referenca) u programu ima svoj *životni vek* (*lifetime*): vreme tokom izvršavanja programa za koje ta varijabla živi i za koje joj se može pristupati:
 - životni vek objekata klasnog tipa i agregata takvih objekata, kao i njihovih podobjekata, počinje nakon što se završi njihova inicijalizacija (poziv konstruktora), osim ako se inicijalizuju tzv. trivijalnim konstruktorom (implicitni konstruktor koji nema baš nikakve efekte za vreme izvršavanja)
 - životni vek objekata klasnog tipa završava se kada započne izvršavanje destruktora (osim ako je destruktur trivijalan - nema baš nikakve efekte)
 - za sve druge objekte (objekte neklasnih tipova, objekte sa trivijalnim konstruktorima i destruktorma, nizovi takvih objekata), životni vek počinje kada se za njih alocira prostor, a završava kada se prostor dealocira
 - životni vek reference počinje kada se završi njena inicijalizacija, a traje kao da je ona skalarni objekat
- ❖ Prema tome, iz ovoga sledi:
 - životni vek objekta je isti ili je ugnezđen u vreme trajanja njegovog skladišta (memorijskog prostora)
 - pre početka životnog veka objekta klasnog tipa uvek se (bez ikakvog izuzetka) poziva njegov konstruktor, a nakon završetka životnog veka uvek (bez ikakvog izuzetka) njegov destruktur
 - životni vek referenciranog objekta (na koga ukazuje pokazivač ili upućuje referenca) može biti kraći od životnog veka tog pokazivača ili reference - problem visećih pokazivača / referenci (*dangling reference*)

Trajanje skladišta i životni vek

- ❖ Nakon što je alociran prostor za objekat, a pre nego što je započeo njegov životni vek, kao i nakon što je završen životni vek, a pre nego što je dealociran njegov prostor, neke operacije imaju nedefinisane efekte, na primer pristup do nestatičkog podatka člana ili poziv nestatičke funkcije članice tog objekta
- ❖ Pojam životnog veka je ortogonalan pojmu opsega važenja:
 - životni vek je koncept vezan za izvršavanje, dok je opseg važenja vezan za prevođenje programa
 - životni vek je vremenski koncept, opseg važenja je prostorni (vezan za deo izvornog koda programa)
- ❖ Svaka varijabla ima jednu od sledećih kategorija životnog veka:
 - statički
 - automatski
 - dinamički
 - lokalni za nit (*thread local*)
 - privremeni (*temporary*)
- ❖ Jedan od osnovnih principa dizajna jezika C++ je bio taj da objekti svih tipova (i klasnih i neklasnih) mogu biti svih kategorija životnog veka. Ovo je jedan od najvažnijih uzroka koji je doveo do ogromne složenosti ovog jezika; mnoga složena pravila, kao i koncepti, posledica su ove odluke: konstruktori kopije, preklapanje operatora, operator dodele, konstruktor premeštanja, reference, lvrednosti i reference na lvrednosti, dvrednosti i reference na dvrednosti itd.
- ❖ Zato je većina drugih, novijih OO jezika krenula drugačijim putem: u njima postoji stroga podela, tako da objekti klase mogu biti samo dinamički i anonimni, dok instance ugrađenih tipova mogu biti svih drugih kategorija životnog veka i samo oni mogu biti imenovane variable; i obratno: dinamički mogu biti samo objekti klasa, ostali ne mogu. Ovo je značajno pojednostavilo semantiku tih jezika

Trajanje skladišta i životni vek

- ❖ Statički životni vek, odnosno statički podaci su najstarija kategorija životnog veka podataka u računarstvu. Oni vode poreklo iz najstarijih programa i programske jezike, iz vremena kada su programi imali jednostavnu strukturu i zadatok:
 - programi nisu bili interaktivni
 - sve svoje ulazne podatke imali su definisane zajedno sa instrukcijama (naredbama za njihovu obradu) u samom programu
 - program je imao zadatok da te ulazne podatke, definisane statički u okviru programa obradi i ispiše rezultate te obrade (tipično neka matematička, numerička izračunavanja) na izlazni uređaj (tipično linijski štampač)
 - za ovu obradu program je mogao da koristi neke varijable koje su ponovo statički definisane - prostor za njih se unapred alocira, ponovo statički
- ❖ Sa pojavom proceduralnog programiranja pojavljuje se potreba za lokalnim podacima, čiji je životni vek vezan za aktivaciju potprograma - žive i koriste se samo u tom potprogramu
- ❖ Ako ne postoji rekurzija, i ovakvi podaci se mogu alocirati statički (iako su dostupni samo lokalno u potprogramu i iako se reinicijalizuju pri svakoj aktivaciji potprograma), jer u svakom trenutku izvršavanja postoji najviše jedna aktivacija datog potprograma; zato se ti podaci tu mogu adresirati apsolutno (memorijski direktnim adresiranjem)
- ❖ Ako postoji mogućnost rekurzije, ovo više nije moguće, jer u datom trenutku može postojati više (i to unapred nepoznat broj) aktivacija istog potprograma, pa se ovakvi podaci moraju alocirati i dealocirati za vreme izvršavanja, i adresirati relativno, tako da se adresiranje u nekoj operaciji unutar potprograma odnosi na trenutno aktuelnu instancu
- ❖ Zato se koristi stek: prilikom ulaska u potprogram, na vrhu steka se formira novi blok lokalnih podataka (tzv. aktivacioni blok); operacije u potprogramu adresiraju podatke iz bloka na vrhu steka (relativno adresiranje u odnosu na vrh steka); prilikom povratka iz potprograma, aktivacioni blok se skida sa vrha steka i ponovo postaje aktuelan onaj koji se odnosi na proceduru iz koje je ova pozvana

Trajanje skladišta i životni vek

- ❖ Razvojem programiranja, a posebno interaktivnih programa i složenih struktura podataka, statički i automatski (lokalni) podaci nisu više dovoljni:
 - nije moguće i nije dovoljno unapred alocirati podatke, statički, jer se ne može znati unapred njihov broj i/ili struktura
 - životni vek vezan za aktivaciju potprograma je previše kratak: potrebno je da podaci nadžive aktivaciju potprograma, ali da ne žive za sve vreme izvršavanja programa
- ❖ Zato su potrebni *dinamički objekti*, čije se kreiranje i uništavanje odvija po potrebama programa, odnosno njegove dinamike i logike: takav dinamički objekat može se kreirati u jednom scenariju, u jednom pozivu potprograma, a potom uništiti u nekom sasvim drugom scenartiju, u pozivu nekog drugog potprograma, sve po potrebi logike programa
- ❖ Uvođenjem konkurentnog programiranja i pojma niti (*thread*), pojavljuje se i prirodna potreba da se životni vek objekta veže za nit, ali i ne samo to, nego da se jedno isto deklarisano ime vezuje za po jednu instancu u svakoj niti, i da se operacije sa takvim imenom odnose na onu instancu koja je u opsegu te niti (slično kao što se operacije nad lokalnom varijablom vezuju za onu instancu koja se odnosi na tekuću aktivaciju potprograma)

Automatski životni vek

- ❖ Varijable sa automatskim životnim vekom su lokalne varijable koje nisu označene kao *static*, *extern* ili *thread_local*
- ❖ Automatska varijabla inicijalizuje se svaki put kada izvršavanje dođe do mesta njene definicije; svaki nailazak izvršavanja na definiciju pravi novu instancu varijable
- ❖ Automatska varijabla se uništava (pozivom destruktora, ako je objekat klase) kada izvršavanje napusti blok u kom je definisana, i to na bilo koji način: prolaskom do kraja bloka, naredbom *return*, ili zbog bačenog izuzetka

```
int f (...) {  
    int i = ...;  
    ... f(...) ...  
  
    for (i=0; i<10; i++) {  
        int j = i+5;  
        ...  
    }  
}
```

Automatski objekat: alocira se i inicijalizuje pri svakom pozivu funkcije *f*

Rekurzija: ugnezđeni poziv kreiraće novi skup svojih automatskih varijabli

Automatski objekat: *j* se kreira i inicijalizuje u svakoj iteraciji ove petlje

j prestaje da živi

i prestaje da živi

- ❖ Za svako *izvršavanje* definicije ovakvog objekta postoji posebna instanca za vreme izvršavanja; zato prevodilac ne može alocirati prostor za takve objekte u prevedenom fajlu statički, pošto može biti više nezavršenih aktivacija datog bloka (rekurzija ili više niti); zato se ovakvi objekti alociraju na steku

Automatski životni vek

- ❖ Parametri funkcije jesu lokalne, automatske varijable koje se inicijalizuju stvarnim argumentima u trenutku poziva te funkcije:

```
void f (X x1) {...}
```

```
void g () {  
    X x2;  
    ...f(x2)...  
}
```

Prilikom poziva funkcije *f* parametar *x1* inicijalizuje se kao automatska varijabla: *X x1=x2*, kakav god da je tip *X*

- ❖ Pored tradicionalnog oblika *for* naredbe kao na jeziku C, jezik C++ ima još jedan oblik ove naredbe, za iteriranje u opsegu (*range*) neke sekvene:

```
int a[] = {0, 1, 2, 3, 4, 5};  
std::vector<int> v = {0, 1, 2, 3, 4, 5};
```

```
void f () {  
    for (int i : a) { cout<<i; }  
    for (int& i : a) { i++; }  
  
    for (auto i : v) { cout<<i; }  
    for (auto& i : v) { i++; }  
}
```

i je objekat tipa *int* koji se inicijalizuje (kopijom vrednosti) svakog elementa
i je referenca na objekat tipa *int* koji se inicijalizuje tekućim elementom

- ❖ Katakteristike ovog oblika naredbe *for* su:

- tip deklarisane varijable treba da bude tip elementa sekvene (uređene kolekcije) kojom se inicijalizuje ili referenca na taj tip; najčešće se koristi *auto* ili *auto&*
- sekvenca *range* kroz koju se iterira može da bude niz ili objekat klase koja ima članove sa imenom *begin* i *end*
- petlja iterira kroz dati niz (mora biti poznate dimenzije), ili počev od *range.begin()* do *range.end()*
- u svakoj iteraciji petlje deklarisana varijabla inicijalizuje se tekućim elementom sekvence

Automatski životni vek

- ❖ Kada izvršavanje napušta blok na bilo koji način (prolaskom kroz kraj bloka, naredbom *return* ili zbog bačenog izuzetka), propisno se uništavaju svi automatski objekti tog bloka koji su kreirani, ali i samo oni: ako neki objekat nije kreiran (recimo zato što je pre izvršavanja njegove definicije bačen izuzetak), objekat neće biti ni uništen
- ❖ Ovo važi i za napuštanje bloka zbog podignutog izuzetka: svi objekti koji su kreirani, a čiji se blokovi napuštaju do ulaska u odgovarajući *catch* blok, propisno se uništavaju (pozivom destruktora); ovo obuhvata i okružujuće blokove, odnosno blokove funkcija koje su pozvane, a nisu završene
- ❖ Analogno važi i za objekte koji su samo delimično kreirani, jer je izuzetak podignut tokom njihove inicijalizacije (poziva konstruktora): svi njihovi podobjekti osnovnih klasa i članovi koji su kreirani biće propisno uništeni, a oni koji nisu, neće
- ❖ Ovaj postupak naziva se *razmotavanje steka* (*stack unwinding*):

```
void f () {
    try {
        X x1;
        g();
    }
    catch (...) {}
}

void g () {
    X x2;
    h();
}

void h () {
    X x3;
    throw 0;
}
```

Ulaskom u ovaj *catch* zbog izuzetka bačenog u funkciji *h*
biće uništeni objekti *x3*, *x2* i *x1*

- ❖ Automatski objekti se uništavaju po redusledu uvek tačno obrnutom od onog kojim su kreirani

Statički životni vek

❖ Varijable sa statičkim životnim vekom su sledeće:

- varijable deklarisane u oblasti prostora imena (*namespace*), uključujući i globalni prostor
- ostale varijable (deklarisane u oblasti bloka ili klase) koje su deklarisane kao *static* ili *extern*

osim ako imaju specifikator *thread_local*

```
namespace N {
```

```
    int i;
```

Statički objekat u oblasti prostora imena

```
    struct S {  
        static int i;  
    };
```

Statički objekat u oblasti klase (statički podatak član)

```
    void f () {  
        static int i;  
    }
```

Statički lokalni objekat

```
}
```

❖ Za svaku definiciju ovakvog objekta postoji jedna instanca za vreme izvršavanja; zato prevodilac može (i po pravilu to i radi) alocirati prostor za takve objekte u prevedenom fajlu koji predstavlja memorijsku mapu programa prilikom pokretanja. U svakom slučaju, memorijski prostor za ovakve objekte alocira se na početku izvršavanja programa i dealocira na kraju izvršavanja programa

Statički životni vek

- ❖ Inicijalizacija statičkih varijabli, odnosno početak njihovog životnog veka razlikuje se za lokalne i ostale statičke varijable
- ❖ Statičke varijable koje nisu lokalne (tj. one iz prostora imena ili statički podaci članovi) inicijalizuju se na sledeći način:
 - najpre se vrši *statička inicijalizacija* (*static initialization*), što znači sledeće:
 - *konstantna inicijalizacija* (*constant initialization*) onih varijabli koje su deklarisane kao *constexpr* ili koje su inicijalizovane konstantnim izrazom; po pravilu, ovo prevodilac vrši još za vreme prevođenja (u prevedeni kod upiše vrednosti tih varijabli u statički alociran prostor za njih); čak i ako to ne uradi za vreme prevođenja, mora da obezbedi da se to radi za vreme izvršavanja pre svega ostalog
 - *inicijalizacija nulom* (*zero initialization*): za sve ostale situacije, objekti se inicijalizuju tako da dobiju binarnu vrednost 0, sa značenjem očekivano konvertovane vrednosti (npr. pokazivač će imati *null* vrednost čak i ako se ona ne implementira binarnim nulama)
 - *dinamička inicijalizacija* (*dynamic initialization*) podrazumeva izračunavanje inicijalizatora (kao izraza) za vreme izvršavanja programa, kao i poziv konstruktora objekta klase koji se inicijalizuje, odnosno upis izračunate vrednosti u varijablu koja nije klasnog tipa; ukoliko može, prevodilac dinamičku inicijalizaciju takođe može uraditi za vreme prevođenja

Statički životni vek

- ❖ Na primer:

```
int i, *pi;
```

Statička inicijalizacija nulama: *i* ima vrednost 0, *p* ima vrednost *null*

```
constexpr double pi = 3.1415926;
```

Statička konstantna inicijalizacija

```
double radius = 20.;
```

```
inline double area (double r) { return r*r*pi; }
```

```
double a = area(radius);
```

Dinamička inicijalizacija (koju bi prevodilac mogao da izvrši i za
vreme prevođenja)

```
struct S {
```

```
    S () { cout<<"S::S()\n"; }
```

```
};
```

```
S s;
```

Dinamička inicijalizacija - poziv konstruktora *S::S()*

- ❖ Dinamička inicijalizacija statičkih varijabli unutar iste jedinice prevođenja obavlja se redosledom njihovih definicija. Redosled inicijalizacije varijabli iz različitih jedinica nije uređen
- ❖ Ukoliko se tokom inicijalizacije ovakvih statičkih varijabli dogodi izuzetak, poziva se *std::terminate()*

Statički životni vek

- ❖ Za ovakve nelokalne varijable koje se inicijalizuju i dinamičkom inicijalizacijom, tačan trenutak izvršavanja te dinamičke inicijalizacije nije precizno određen: ona može, ali ne mora biti izvršena pre početka izvršavanja funkcije *main*, jer prevodilac ne može to uvek da obezbedi (zbog nezavisnog prevođenja fajlova)
- ❖ Jedino što se garantuje jeste to da su statičke varijable propisno inicijalizovane pre nego što se na bilo koji način pristupi bilo kojoj statičkoj varijabli definisanoj u istoj jedinici prevođenja
- ❖ Zbog ovoga, korišćenje nelokalnih statičkih objekata klase (statičkih podataka članova ili statičkih objekata unutar prostora imena) nije bezbedno: ukoliko se njima pristupa na mestu izvan fajla u kom su definisani, može se dogoditi da se taj pristup radi pre nego što su oni propisno inicijalizovani
- ❖ Zato se ne preporučuje definisanje statičkih objekata klasa koji nisu lokalni, posebno ako njihovi konstruktori imaju neke vidljive efekte. Na primer:

```
// File: a.cpp
#include "a.h"

A sa; —————— sa je globalni statički objekat; vreme njegove
A::A () : m(true) {}

// File: b.cpp
#include "a.h"

bool f (A* pa) { return pa->m; }

int main () {
    if (f(&sa)) ...
}
```

Fajl *a.h* sadrži samo sledeće:

```
struct A {
    bool m;
    A();
};

extern A sa;
```

Kada se ovo izvršava, *sa* možda nije uopšte inicijalizovan, pa će *f* možda vratiti *true*, a možda i *false*

Statički životni vek

- ❖ Sa druge strane, inicijalizacija lokalnih statičkih varijabli je precizno definisana: takva varijabla inicijalizuje se kada kontrola toka prvi put nađe na njenu definiciju; ako više niti nailazi na tu definiciju, samo prva će izvršiti inicijalizaciju. Svako naredno izvršavanje preskače (ignoriše) tu definiciju
- ❖ Ukoliko izvršavanje nikada ne dođe do ovakve definicije, statički objekat neće biti ni inicijalizovan (pa stoga ni uništen pozivom destruktora), na primer ako se funkcija ne pozove ili ne izvrši grana u kojoj je definicija
- ❖ S obzirom na svoj statički životni vek, statičke lokalne varijable nadživljavaju izvršavanje bloka (pa i poziv funkcije) u kom su deklarisane, iako imaju lokalni opseg važenja, pa nisu dostupne van te oblasti. Na primer:

```
int a = 1;           a je globalni statički objekat: dostupan u celom fajlu, statički inicijalizovan
void f () {
    int b = 1;         b je lokalni automatski objekat: inicijalizuje se pri svakom pozivu funkcije
    static int c=1;    c je lokalni statički objekat: dostupan samo u bloku, inicijalizuje se pri prvom
                      izvršavanju definicije
    cout<<" a = "<<a++;
    cout<<" b = "<<b++;
    cout<<" c = "<<c++<<' \n' ;
}                         a i c nadživljavaju izvršavanje funkcije, b nestaje izlaskom iz funkcije
int main () {
    while (a<4) f();
}
```

Ispisaće se:
a = 1 b = 1 c = 1
a = 2 b = 1 c = 2
a = 3 b = 1 c = 3

Statički životni vek

- ❖ Zbog svega ovoga, umesto statičkih objekata koji nisu lokalni (npr. globalni ili podaci članovi), bolje je koristiti lokalne statičke objekte, odnosno statičke objekte “umotati” u funkciju (po pravilu nečlanicu ili statičku članicu). Tako se garantuje propisna inicijalizacija, ali i bolja enkapsulacija. Na primer:

```
class Clock {  
public:  
    Clock (...) { getClockRegister()->add(this); }  
  
    static const ClockRegister* getClocks () { return getClockRegister(); }  
  
private:  
    static ClockRegister* getClockRegister ();  
};  
  
ClockRegister* Clock::getClockRegister () {  
    static ClockRegister clockRegister(...);  
    return &clockRegister;  
}
```

Lokalni statički objekat. Enkapsuliran je u ovu funkciju, pa mu se može pristupiti samo preko nje. Pri prvom pozivu te funkcije (a drugačije mu se i ne može pristupiti), on će biti inicijalizovan pozivom konstruktora, koji može da uradi bilo kakvu dinamičku inicijalizaciju.

- ❖ Svi statički objekti žive do kraja programa i uništavaju se nakon završetka funkcije *main*. Ako neki statički objekat (npr. lokalni) nije inicijalizovan, neće biti ni uništen (neće biti pozvan njegov destruktor)

Dinamički životni vek

- ❖ Dinamički životni vek objekata neposredno se kontroliše logikom i dinamikom programa: dinamički objekti se prave i uništavaju eksplisitno:
 - svako izvršavanje izraza (operatora) *new* pravi nov dinamički objekat
 - tako napravljen dinamički objekat živi dok se ne uništi operatorom *delete*

```
Clock* pc = new Clock(9,0,0);
```

Pravljenje dinamičkog objekta

...

```
delete pc;
```

Uništavanje dinamičkog objekta

- ❖ Životni vek dinamičkih objekata nije implicitan i vezan za neki opseg, kao što je to slučaj sa automatskim i statičkim objektima; on se kontroliše eksplisitno, pa dinamički objekti mogu da nadžive izvršavanje funkcije u kojoj su napravljeni, što tipično i jeste slučaj: oni se prave u jednom scenariju, izvršavanjem jedne funkcije, a uništavaju možda u nekom poptuno drugom scenariju i u drugoj funkciji:

```
template <typename T>
List<T>& List<T>::addBack (T t) {
    ListElem<T>* e = new ListElem<T>(t, tail, nullptr);
    if (!head) head = e;
    tail = e;
    return *this;
}
```

Pravljenje dinamičkog objekta

```
template <typename T>
T List<T>::removeFront () {
    ListElem<T>* e = head;
    ...
    delete e;
    ...
}
```

Uništavanje dinamičkog objekta

Dinamički životni vek

- ❖ Izraz (operator) *new* uvek radi sledeće stvari, ovim redom:
 1. alocira prostor za smeštanje jednog objekta datog tipa ili niza objekata datog tipa, ako se pravi niz
 2. inicijalizuje jedan objekat ili svaki objekat u nizu (ako se pravi niz objekata) zadatim inicijalizatorom
 3. vraća vrednost tipa pokazivača na dati tip koji ukazuje na napravljeni objekat, ili na prvi element napravljenog niza (ako se pravi niz objekata datog tipa)
- ❖ Alokacija prostora (prvi korak) vrši se pozivom neke od (preklopljenih) operatorskih funkcija koje su standardno definisane jezikom (postoje i ovakve operatorske funkcije sa još nekim parametrima):

void* operator new (std::size_t count); — Poziva se ako se pravi jedan objekat
void* operator new[] (std::size_t count); — Poziva se ako se pravi niz objekata
- ❖ Tip *size_t* je neoznačeni celobrojni tip deklarisan u nekoliko zaglavlja standardne biblioteke koji se koristi za izražavanje veličina tipova; ovo je tip rezultata operatorka *sizeof*
- ❖ Parametar ovih funkcija tipa *size_t* prenosi veličinu prostora koji treba alocirati (u jedinicama *sizeof(char)*)
- ❖ Podrazumevana implementacija ovih funkcija upravlja slobodnom memorijom (*heap, free store*), odnosno radi dinamičku alokaciju i dealokaciju u za to predviđenom delu memorije programa
- ❖ Programer može promeniti način alokacije prostora na neki od sledećih načina:
 - zameniti implementaciju neke od ovih funkcija svojom implementacijom, čime će promeniti način alokacije dinamičke memorije (preusmeriti na svoj alokator); dovoljno je samo negde u programu definisati takvu funkciju, bez posebne deklaracije na nekom drugom mestu
 - definisati ove funkcije za svoje klasne tipove, čime će promeniti način alokacije prostora samo za objekte te klase

Dinamički životni vek

- ❖ Izraz (operator) *new* ima redom sledeće elemente iza ključne reči *new*:

- opcioni argumenti za tzv. *parametre smeštanja* (*placement parameters*)
- specifikator objektnog tipa, kao *type-id*, za objekat koji se kreira
- opcioni inicijalizator, kao inače pri inicijalizaciji varijabli na drugim mestima

```
new (std::nothrow) Clock (9, 0, 0)
```

Argumenti za parametre smeštanja

Inicijalizator

Specifikator tipa (*type-id*)

- ❖ Specifikator tipa je *type-id* koji mora biti objektni tip (ne tip reference), i može biti tip koji predstavlja jedan objekat ili niz objekata nekog tipa:

```
new (Clock*) (nullptr)
```

Pravi se jedan objekat tipa *Clock**

```
new (Clock*[n]) {}
```

Pravi se niz od *n* objekata tipa *Clock**

- ❖ Ukoliko je specifikovan tip niza, sve dimenzije tog niza osim prve moraju biti date kao konstatni izrazi sa pozitivnom celobrojnom vrednošću; prva dimenzija može biti izraz koji se izračunava u trenutku izvršavanja izraza *new* - tako se jedino i mogu praviti dinamički dimenzionisani nizovi (sa dimenzijom poznatom tek za vreme izvršavanja):

```
const int Max = 100;  
extern int n;  
new (Clock[2*n][Max]) (nullptr)  
new (Clock[2*n][n]) {}
```

Greška u prevođenju: druga dimenzija nije konstantan izraz

- ❖ U specifikatoru tipa (*type-id*) može da se koristi i *auto*, s tim da je inicijalizator tada obavezan, jer se na osnovu njegovog tipa određuje tip objekta koji se kreira

Dinamički životni vek

- ❖ Opcioni argumenti za parametre smeštanja mogu se koristiti za prosleđivanje dodatnih informacija (parametara) operatorskim funkcijama koje izraz *new* poziva za potrebe alokacije prostora za dinamički objekat. Dve standardno podržane mogućnosti za ove parametre su sledeće:
 - parametar za tzv. *placement new* (po kom su ovi parametri i dobili ime, jer su uvedeni u jezik kao uopštenje ove mogućnosti)
 - argument *nothrow*
- ❖ Tzv. *placement new* omogućava da se dinamički objekat smesti u prostor već alociran za neki drugi objekat, odnosno u skladište koje još uvek traje. Na primer:

```
char* ptr = new char[sizeof(T)];
```

Alokacija prostora za niz znakova date dimenzije

```
T* tptr = new(ptr) T;
```

Placement new: pravi se objekat tipa *T* u prostoru na koji ukazuje argument *ptr*

```
tptr->~T();
```

Objekat tipa *T* se uništava eksplicitnim pozivom destruktora

```
delete [] ptr;
```

Dealokacija prostora

- ❖ Podrazumevano, izraz *new*, tačnije, operatorska funkcija koju on poziva za alokaciju prostora baca izuzetak ukoliko ne može da alocira taj prostor; ako se kao argument dostavi predefinisani objekat *std::nothrow*, neće biti bačen izuzetak, nego će izraz *new* vratiti *null* vrednost u ovom slučaju:

```
auto p = new (std::nothrow) (Clock[2*n][Max]) {}  
if (p) ...p[i][j]...
```

Dinamički životni vek

- ❖ *Placement new* zapravo omogućava inicijalizaciju objekta pozivom konstruktora za deo memorije koji je već određen za smeštanje objekta; drugačije nije moguće pozvati konstruktor koji inicijalizuje objekat na definisanom mestu u memoriji
- ❖ *Placement new* se može koristiti u sistemskim rutinama niskog nivoa, recimo u delovima sistema (npr. operativnog sistema) koji upravljaju alokacijom memorije, keširanjem blokova sa diska, implementaciji fajl sistema i slično
- ❖ Na primer, želimo da dealocirane segmente memorije uvezujemo u listu, s tim da strukturu *FreeSeg* koja predstavlja jedan slobodan segment smestimo baš na početak tog slobodnog segmenta (a ne da alociramo poseban deo memorije koji bi se trošio za evidenciju slobodne memorije):

```
class FreeStore {  
public:  
    inline void free (void* addr, size_t sz);  
    ...  
private:  
    class FreeSeg {  
public:  
        FreeSeg (FreeSeg* nxt, size_t sz) : next(nxt), size(sz) {}  
    private:  
        FreeSeg* next;  
        size_t size;  
    };  
    FreeSeg* head;  
};  
  
void FreeStore::free (void* addr, size_t sz) {  
    head = new (addr) FreeSeg(head,sz);  
}
```

Zadatak: implementirati funkciju
void* FreeStore::alloc(size_t size);

Pitanje: kako postići isto bez
korišćenja *placement new*?

Pitanje: šta radi i šta vraća ugrađena sistemska
operatorska funkcija koja se poziva za *placement new*?
void* operator new (std::size_t count, void* ptr);

Dinamički životni vek

- ❖ Izraz *new* vraća vrednost koja je po tipu pokazivač na tip specifikovan pomoću *type-id* i koji ukazuje na objekat napravljen ovim izrazom, odnosno na prvi objekat u nizu, ako je napravljen niz
- ❖ Izrazom *new* mogu se praviti objekti (ne i reference), pošto ne postoje pokazivači na reference (pa operator *new* ne može da vrati pokazivač na dinamički kreiranu referencu)
- ❖ Ova vraćena vrednost je jedina veza do napravljenog dinamičkog objekta koji je inače anoniman (nema ime); ukoliko se ova veza izgubi, objekat ostaje nedostupan, pa se ne može ni uništiti:

```
int i = new int(2);
```

Greška u prevodenju: ne postoji implicitna konverzija iz *int** u *int*

```
int j = *new int(2);
```

Problem: pristup do dinamičkog objekta je trajno izgubljen

```
int& r = *new int(2);
```

Referenca upućuje na dinamički objekat, pa je ona sada njegovo ime

```
delete &r;
```

Dinamički životni vek

-
- ❖ Izraz (operator) *delete* ima jedan od sledeća dva oblika:

```
delete expression
```

```
delete [] expression
```

- ❖ Prvi oblik se koristi ako se uništava jedan objekat; drugi oblik je obavezan ako se uništava niz objekata. Ako se u ovome pogreši, efekat je nedefinisan (greška u izvršavanju, jer se pozivaju pogrešne operatorske funkcije za dealokaciju)
- ❖ Izraz koji je operand ovog operatora mora biti tipa pokazivača na objektni tip. On mora ukazivati na objekat, odnosno niz objekata napravljen pomoću *new*, ili na podobjekat osnovne klase objekta napravljen pomoću *new*, ili imati vrednost *null* (u kom slučaju ovaj operator nema efekta); u suprotnom, efekat je nedefinisan
- ❖ Izraz (operator) *delete* uvek radi sledeće stvari (osim ako pokazivač ima *null* vrednost, kada nema efekta), ovim redom:
 1. poziva destruktor objekta na koji pokazivač ukazuje, ili destruktor svakog elementa niza, ukoliko je pokazivač na klasni tip; ako pokazivač ukazuje na podobjekat osnovne klase, a destruktor je virtuelan, poziv je polimorfan; ako pokazivač ukazuje na podobjekat osnovne klase, a destruktor nije virtuelan, ponašanje je nedefinisano
 2. oslobađa (deallocira) prostor koji je zauzimao objekat, odnosno niz, pozivom odgovarajuće operatorske funkcije za dealokaciju
- ❖ Operator *delete* uvek ima povratni tip *void*
- ❖ Potpuno analogno alokaciji, delokacija prostora (drugi korak) vrši se pozivom neke od (preklopljenih) operatorskih funkcija koje su standardno definisane jezikom (postoje i ovakve operatorske funkcije sa još nekim parametrima) i koje se mogu zameniti ili redefinisati za klase:

```
void operator delete (void* ptr);  
void operator delete[] (void* ptr);
```

Dinamički životni vek

- ❖ Tzv. *curenje memorije* (*memory leak*) je problem koji može nastupiti nekorektnim rukovanjem dinamičkim objektima, tako što se dinamički objekti repetitivno prave, ali se ne uništavaju, jer je programer zaboravio da propisno uništava dinamičke objekte koji više nisu potrebni (a program iznova pravi nove kada su potrebni)
- ❖ Nakon dužeg izvršavanja programa, slobodna memorija će biti iscrpljena, pa memorije za nove dinamičke objekte više neće biti, i naredna operacija *new* baciće izuzetak (ili vratiti *null* vrednost); nakon toga, program više neće radi kako se od njega očekuje
- ❖ Na primer, sledeći deo koda je banalan, ali i očigledan primer curenja memorije:
`int i = *new int(0);` Problem: ovom objektu ne može se pristupiti, pa se on ne može ni obrisati
- ❖ Tipičan uzrok jeste nepažljivo dodeljena odgovornost za brisanje dinamički napravljenih objekata. Na primer, korisniku neke funkcije koja pravi nov dinamički objekat i vraća pokazivač (ili referencu) na njega nije jasno da je odgovornost za brisanje tog objekta na njemu (a ne na onom ko ga je koristio), pa će zaboraviti da ga obriše:

```
extern X* getAnX (...);
```

```
X* px = getAnX(...);
```

Nije sasvim jasno da li pozivalac ima odgovornost za brisanje objekta na kog ukazuje vraćeni pokazivač

- ❖ Uzrok može biti i situacija u kojoj se objekat ne obriše zbog izuzetka:

```
void f () {  
    int* p = new int(1);  
    g();  
    delete p;  
}
```

Ako funkcija *g* baci izuzetak, dinamički objekat neće biti obrisan

Dinamički životni vek

- ❖ Pažljivim projektovanjem programa mora se unapred odrediti odgovornost za uništavanje dinamičkih objekata koji se kreiraju. Dobro imenovanje funkcija i njihovo dokumentovanje pomaže u sprečavanju ovog problema. Na primer:
`X* px = createAnX(...);` Ovaj naziv funkcije sugerira pozivaocu da obrati pažnju na odgovornost za uništavanje objekta koji je napravljen u ovoj funkciji
- ❖ Postoje i razvojni alati koji nadziru izvršavanje programa i mogu da daju izveštaj o dinamičkim objektima koji su kreirani, a nisu uništeni tokom izvršavanja programa
- ❖ U nekim drugim programskim jezicima (npr. Java), dinamički objekat uništava se implicitno, kada poslednja referenca koja na objekat ukazuje prestane da živi (pošto se tada objektu svakako više ne može pristupiti, jer se objektima u tim jezicima ne može pristupiti drugačije nego preko referenci)
- ❖ Ovo implicitno brisanje naziva se “skupljanje đubre” (*garbage collection*) i obavlja ga poseban deo izvršnog okruženja jezika (*garbage collector*) povremeno, po nahodjenju izvršnog okruženja
- ❖ Ovakav pristup značajno smanjuje pojavu curenja memorije, ali ne može da je potpuno spreči, jer mogu postojati trajne reference koje ukazuju na objekat i tako ne dozvoljavaju njegovo brisanje, iako on zapravo nije neophodan; na primer, kada grupa objekata međusobno ciklično ukazuju referencama jedan na drugog
- ❖ Postoje i sofisticirane metode otkrivanja ovakvih pojava, ali one ne mogu nikada biti potpuno delotvorne, pa problem curenja memorije i dalje ostaje kao pretnja. Osim toga, na sličan način mogu “curiti” bilo koji drugi resursi koji se dinamički alociraju (recimo niti), a ne uništavaju kada je potrebno

Dinamički životni vek

- ❖ Sličan pristup za smanjenje problema curenja memorije postoji i na jeziku C++, kao pouzdanije rešenje ovog problema kroz korišćenje tzv. *pametnih pokazivača* (*smart pointer*): pokazivača za koje se vodi evidencija o broju onih koji ukazuju na isti objekat i za koje je obezbeđeno implicitno uništavanje dinamičkog objekta kada poslednji od njih prestane da ukazuje na taj objekat
- ❖ U standarnoj biblioteci jezika C++ postoji više šablonskih klasa za pametne pokazivače:
 - *std::unique_ptr*: pametni pokazivač koji je jedini “vlasnik” svog objekta; kada ovakav pokazivač koji ukazuje na dati objekat prestane da ukazuje na taj objekat (zbog kraja životnog veka ili zato što je promenio vrednost dodelom druge vrednosti), i dinamički objekat na koga on ukazuje se implicitno briše
 - *std::shared_ptr*: pametni pokazivač koji je deljeni “vlasnik” svog objekta; kada poslednji pokazivač koji ukazuje na dati objekat prestane da ukazuje na taj objekat (zbog kraja životnog veka ili zato što je promenio vrednost dodelom druge vrednosti), i dinamički objekat na koga on ukazuje se implicitno briše
 - *std::weak_ptr*: pametni pokazivač koji je tzv. “slaba referenca” (*weak reference*) na objekat; objekat može biti uništen i ako na njega ukazuju samo slabi pokazivači, ali se preko ovog pokazivača može bezbedno pristupiti objektu, pri čemu se on tada mora konvertovati u *shared_ptr*; ovakvi pokazivači mogu se koristiti i za raskidanje cikličnih referenciranja objekata pomoću pametnih pokazivača
- ❖ Na primer:

```
{  
    std::shared_ptr<X> p = new X;  
{  
    std::shared_ptr<X> q = p;  
    ...q->...  ...*p...  
}  
}
```

Za pametne pokazivače preklopljeni su operatori `*`, `->` i `=`

Ovde, po prestanku života pametnog pokazivača *p*, i dinamički objekat klase *X* se uništava implicitno

Životni vek vezan za nit

- ❖ Varijable koje bi inače bile statičke po životnom veku (lokalne, u prostoru imena ili u oblasti klase) mogu biti deklarisane kao *thread_local*; tada imaju životni vek vezan za nit
- ❖ Ovakve varijable nastaju pri kreiranju svake nove niti i nestaju kada se nit završi
- ❖ Svaka nit ima svoju instancu ove varijable, nezavisnu od ostalih niti; svako obraćanje toj varijabli odnosi se na onu instancu koja pripada niti u čijem kontekstu se izvršava taj pristup. Na primer:

```
thread_local int i=0;  
void f (int ii) {  
    i = ii;  
}  
  
void tf (int id) {  
    f(id);  
    std::cout<<++i<<std::endl;  
}  
  
int main () {  
    i = 10;  
    std::thread t1(tf,1);  
    std::thread t2(tf,2);  
    std::thread t3(tf,3);  
  
    std::cout<<i<<std::endl;  
}
```

Životni vek vezan za nit

Odnosi se na instancu *i* koja pripada niti u čijem kontekstu se ovo izvršava

Odnosi se na instancu *i* koja pripada niti u čijem kontekstu se ovo izvršava

Funkcija *main* se izvršava u kontekstu početne, “glavne” niti programa

Odnosi se na instancu *i* koja pripada niti u čijem kontekstu se ovo izvršava

Privremeni objekti

- ❖ Privremeni objekti su anonimni (bezimeni) objekti klase čiji je životni vek najčešće kratak; oni nastaju implicitno, u određenim situacijama, i nestaju implicitno, pod kontrolom prevodioca
- ❖ Osnovna potreba za privremenim objektom jeste u tome da se rezultat nekog izraza, uključujući i poziv funkcije (i operatorske funkcije) može dalje iskoristiti kao operand neke druge operacije (ili argument funkcije)
- ❖ Bez obzira na to kada se tačno privremeni objekat inicijalizuje i uništava, uvek se pri inicijalizaciji poziva njegov konstruktor, a pri uništavanju njegov destruktur
- ❖ Pre verzije jezika C++17, rezultat poziva funkcije (uključujući i operatorske) koja ne vraća referencu (na vrijednost), odnosno čiji je povratni tip klasa (a ne referenca na nju), bio je uvek privremeni objekat: u trenutku povratka iz funkcije, na mestu poziva funkcije, pravi se privremeni objekat koji se inicijalizuje izrazom iza naredbe *return*:

```
class X {...};  
X f (X x) {  
    return x;  
}  
  
int main () {  
    X x1;  
    ...f(x1)...  
    ...f( f(x1) )...  
}
```

Do C++17, rezultat ove funkcije u svakom pozivu je bezimeni, privremeni objekat koji se inicijalizuje izrazom iza naredbe *return* u trenutku povratka iz funkcije

Do C++17, rezultat ugnezđenog poziva *f* je privremeni objekat tipa *X* koji prihvata rezultat tog poziva; njime se inicijalizuje stvarni argument poziva okružujućeg poziva funkcije *f*, čiji rezultat ponovo poredstavlja privremeni objekat

Privremeni objekti

Do C++17, rezultat ovog eksplisitnog poziva konstruktora, kao i bilo koje konverzije u klasni tip, jeste privremeni objekat tipa *complex*, baš kao i rezultat svake od ovih operatorskih funkcija `+ i * čiji je povratni tip complex`

- ❖ Pre verzije C++17, privremeni objekti obavezno su se pravili i u sledećim situacijama, pored još nekih:

- konverzija koja ne vraća vrednost uključujući i eksplisitni poziv konstruktora; na primer:

```
complex c1(3.0,0.0), c2(0.0,4.0), c3;  
c3 = (c1 + c2) * (complex(1.,0.) + c2);
```

Do C++17, rezultat inicijalizatorskog izraza sa desne strane znaka = je privremeni objekat tipa *complex* kojim se inicijalizuje objekat *c4* pozivom konstruktora kopije

- prilikom inicijalizacije kopiranjem (*copy initialization*, inicijalizacija sa znakom =):

```
complex c4 = (c1 + c2) * (complex(1.,0.) + c2);
```

- kada se referenca inicijalizuje izrazom drugog, ali konvertibilnog tipa

- ❖ Pritom, za navedeni drugi slučaj inicijalizacije kopiranjem, kao i na nekim drugim mestima, prevodiocu se dopušta (ali se ne obavezuje) optimizacija *izostavljanja kopiranja* (*copy elision*): umesto da se najpre napravi privremeni objekat kao rezultat inicijalizacionog izraza, a potom njime inicijalizuje deklarisani objekat konstruktorom kopije, prevodilac može odmah da inicijalizuje deklarisani objekat rezultatom inicijalizacionog izraza, izostavljajući kopiranje, čak i ako konstruktor kopije ima vidljive bočne efekte, ali samo pod uslovom da su sve odgovarajuće funkcije, uključujući i konstruktor kopije, dostupne na mestu deklaracije (proveravaju se prava pristupa kao da se kopiranje ne izostavlja, tzv. *as-if* pravilo)

Privremeni objekti

- ❖ Počev od verzije jezika C++17, ova semantika je značajno izmenjena: rezultat poziva funkcije (uključujući i operatorske funkcije), baš kao i rezultat bilo koje operacije i izraza jeste *vrednost (value)*, a ne privremeni objekat
- ❖ Vrednost je poseban entitet u programu i odnosi se na rezultat izraza koji se dalje može koristiti kao argument poziva funkcije. Na taj način se između poziva funkcija u izrazima prosleđuju vrednosti
- ❖ Pravljenje privremenog objekta se sada maksimalno odlaže do trenutka kada postane zaista neophodno, i vrši se samo u nekim situacijama; ovo se naziva *materijalizacija privremenog objekta (temporary materialization)*; na primer, navedeno izostavljanje kopiranja kod inicijalizacije, kao i na mnogim drugim mestima, gde god je moguće kopiranje izostaviti, nije više dozvoljena, ali neobavezna optimizacija prevodioca, već je definisana semantika: na navedenom mestu se privremeni objekat *nikada* ne pravi
- ❖ Referenca (na lvrednost) se može inicijalizovati izrazom koji nije lvrednost samo ako je ta referenca na konstantu; tada se (i pre, i od verzije C++17), pravi privremeni objekat za koji se vezuje ta referenca:

```
complex& r1 = complex(1.,0.);  
const complex& r2 = complex(1.,0.);
```

Greška u prevodenju: referenca na nekonstantnu lvrednost ne može se inicijalizovati izrazom koji nije lvrednost

- ❖ Zbog ovoga, ako neka funkcija ima parametar koji je referenca (na lvrednost), ona treba da bude referenca na konstantu, inače se ta funkcija ne bi mogla pozvati sa argumentima koji to nisu:

```
complex operator+ (const complex&, const complex&);  
...c1+complex(1.,0.)
```

Ovaj poziv ne bi bio ispravan kada bi parametar bio referenca na nekonstantu

Privremeni objekti

- ❖ Uništavanje svih privremenih objekata napravljenih pri izračunavanju nekog potpunog izraza (izraza čiji rezultat više nije operand nekog okružujućeg izraza) vrši se kao poslednji korak u izračunavanju tog izraza u koji je pravljenje tih privremenih objekata leksički ugrađeno; drugim rečima, svi privremeni objekti napravljeni tokom izračunavanja izraza ne nadživljavaju pun izraz u kom su leksički napravljeni (ne uključuje dinamički ugnezđene izraze izvršene unutar pozvanih funkcija):

```
c3 = (c1 + c2) * (complex(1., 0.) + c2);
```

Svi privremeni objekti napravljeni u ovom izrazu uništavaju se na kraju izračunavanja tog izraza

- ❖ Ako je tokom izračunavanja izraza napravljeno više privremenih objekata, oni se uništavaju po obrnutom redosledu od redosleda njihove inicijalizacije
- ❖ Sve ovo važi čak i ako je izračunavanje izraza bacilo izuzetak
- ❖ Ako je za privremeni objekat vezana referenca, životni vek tog privremenog objekta produžava se do kraja životnog veka te reference, ali ne tako da on nadživi poziv funkcije u kojoj je napravljen: svi privremeni objekti obavezno se uništavaju pre povratka iz funkcije u kojoj su napravljeni, pa referenca vraćena iz funkcije i dalje može biti viseća

Ugrađeni objekti

- ❖ Objekti ugrađeni u druge objekte složenih tipova nazivaju se *podobjektima*; to su sledeći:
 - podobjekat osnovne klase unutar objekta izvedene klase
 - objekat koji je manifestacija nestatičkog podatka člana, tzv. objekat član (*member object*), ugrađen u objekat klase čiji je taj podatak član
 - element niza
- ❖ Objekti koji nisu podobjekti drugih objekata nazivaju se *kompletnim objektima* (*complete object*)
- ❖ Kompletni objekti, objekti članovi i objekti koji su elementi niza nazivaju se i *najizvedenijim objektima* (*most-derived object*), da bi se razlikovali od podobjekata osnovnih klasa ugrađene u objekte izvedenih klasa
- ❖ Životni vek ugrađenih objekata određen je životnim vekom objekta u koji su ugrađeni i vezan je za njega: ugrađeni objekti nastaju zajedno sa okružujućim objektom (inicijalizuju se tokom inicijalizacije okružujućeg objekta) i nestaju zajedno sa njim (uništavaju se tokom uništavanja okružujućeg objekta)
- ❖ Redosled inicijalizacije je uvek određen za objekte ugrađene u isti objekat. Redosled njihovog uništavanja je takođe određen i uvek je suprotan redosledu njihove inicijalizacije
- ❖ Elementi niza inicijalizuju se po redosledu njihovih indeksa (počev od elementa na poziciji 0 ka rastućim pozicijama); uništavaju se obrnutim redosledom

Ugradjeni objekti

- ❖ Inicijalizatori za podobjekte osnovnih klasa i za objekte članove navode se u zaglavlju konstruktora (izvedene) klase; svako pominjanje podatka člana u telu konstruktora klase više nije inicijalizacija, nego neka druga operacija nad već inicijalizovanim objektom članom:

```
class Whole : public Entity {  
public:  
    Whole ();  
    ~Whole ();  
private:  
    Part part;  
};  
  
Whole::Whole () : Entity(), part(this) {  
    ...part...  
}  
  
Whole::~Whole () {...}
```

Destruktor

Ovo je inicijalizacija podobjekta osnovne klase

Ovo je inicijalizacija podobjekta člana

Ovde više ne može da se izvede inicijalizacija podobjekata

Ovde se implicitno pozivaju destruktori člana *part* i podobjekta osnovne klase

- ❖ Dakle, u telu konstruktora klase ne može se inicijalizovati objekat član, već mu se eventualno može dodeliti vrednost operatorom dodele, što je potpuno druga operacija od inicijalizacije (iako za neklasne tipove ima isti efekat - prosto kopiranje vrednosti); naravno, ako je nestatički podatak član tipa reference ili konstantnog tipa, on se mora inicijalizovati, svakako mu se ne može dodeliti vrednost:

```
struct X {  
    X (int&);  
    int& r;  
    const int c;  
};  
  
X::X (int& i) : r(i), c(i) {}
```

Ugradjeni objekti

- ❖ Kada se inicijalizuje objekat klase, redosled inicijalizacije je sledeći:
 - ako se radi o najizvedenijem objektu (tj. ovo je “prva” inicijalizacija objekta u hijerarhiji nasleđivanja), inicijalizuju se podobjekti virtualnih osnovnih klasa, po redosledu obilaska grafa nasleđivanja po dubini, sleva nadesno (po redosledu kako su te osnovne klase navedene u deklaracijama izvedenih klasa)
 - inicijalizuju se podobjekti direktnih osnovnih klasa (iz koje je ova klasa neposredno izvedena), po redosledu njihovog navođenja u deklaraciji izvedene klase
 - inicijalizuju se objekti članovi
 - izvršava se telo konstruktora klase čiji je objekat direktna instanca
- ❖ Treba primetiti da svako pominjanje “inicijalizacije (pod)objekta” u navedenim pravilima znači rekurzivnu inicijalizaciju podobjekata osnovnih klasa, objekata članova i izvršavanje tela konstruktora za taj (pod)objekat
- ❖ Redosled uništavanja (destrukcije) objekta uvek je tačno obrnut:
 - izvršava se telo destruktora klase čiji je objekat direktna instanca
 - uništavaju se objekti članovi, po redosledu obrnutom od redosleda inicijalizacije
 - uništavaju se podobjekti direktnih osnovnih klasa, po redosledu obrnutom od redosleda inicijalizacije
 - ako se radi o najizvedenijem objektu, uništavaju se podobjekti virtualnih osnovnih klasa, po redosledu obrnutom od redosleda inicijalizacije

Ugradjeni objekti

❖ Na primer:

```
struct D { D(char c) { cout<<"D"<<c; } };  
  
struct A {  
    D d;  
    A() : d('a') { cout<<"A\n"; }  
};  
  
struct B : A {  
    D d;  
    B() : d('b') { cout<<"B\n"; }  
};  
  
struct C {  
    D d;  
    C() : d('c') { cout<<"C\n"; }  
};  
  
struct X : B, C {  
    D d;  
    D a[3];  
    X() : C(), B(), d('x'), a{'0','1','2'} { cout<<"X"; }  
};  
  
int main () {  
    X x;  
}
```

Pre izvršavanja tela konstruktora izvedene klase, najpre se inicijalizuju podobjekti osnovnih klasa pozivom njihovih konstruktora, pa onda i objekti članovi. Bez obzira na redosled ovde navedenih inicijalizatora, redosled inicijalizacije je uvek određen redosledom navođenja osnovnih klasa i podataka članova u definiciji izvedene klase

Ispis će biti:
*DaA
DbB
DcC
DxD0D1D2X*

Zadatak: odrediti redosled uništavanja, odnosno poziva destruktora; napisati destruktore koji ispisuju slične poruke

Glava 12: Konstruktor, destruktor i operator dodele

- ❖ Konstruktor
- ❖ Podrazumevani konstruktor
- ❖ Konstruktor kopije
- ❖ Destruktor
- ❖ Operator dodele kopiranjem



Konstruktor

- ❖ Konstruktor (*constructor*) je specijalna nestatička funkcija članica klase koja se koristi za inicijalizaciju objekata te klase
- ❖ Konstruktor može imati proizvoljne parametre, pa klasa može imati više konstruktora. Postupak izbora konstruktora koji će biti pozvan na mestu inicijalizacije određuje se po istim pravilima kao i za ostale preklopljene funkcije, na osnovu tipova argumenata u inicijalizaciji
- ❖ Konstruktor nema povratni tip (čak ni *void*)
- ❖ Konstruktor ima pokazivač *this*, kao i svaka druga nestatička funkcija članica
- ❖ Kao i svaka druga funkcija članica, konstruktor može biti javan, zaštićen ili privatan. Na mestu inicijalizacije, prevodilac proverava dostupnost konstruktora koji je odabran. Prema tome, pošto se pri inicijalizaciji objekta izvedene klase inicijalizuje i podobjekat osnovne klase pozivom konstruktora, ako su svi konstruktori neke klase privatni, iz te klase se ne može izvesti klasa za koju se mogu praviti objekti ili podobjekti
- ❖ Konstruktor ne može biti virtualan, niti može biti cv-kvalifikovan: cv-kvalifikacija nekog objekta stupa na snagu kada je u potpunosti završena njegova inicijalizacija
- ❖ Konstruktor zapravo nema ime i ne može se pozvati eksplisitno - on se uvek poziva implicitno za potrebe inicijalizacije kompletног objekta ili podobjekta
- ❖ Ako se tokom izvršavanja tela konstruktora ili u inicijalizaciji podobjekata pozove virtualna funkcija tog objekta, pozvaće se implementacija te funkcije koja pripada toj klasi čiji se konstruktor izvršava. Ovo je stoga što generisani kod za svaki konstruktor postavlja vrednost VTP tog objekta tako da ukazuje na VT baš te klase, pa konstruktor izvedene klase postavlja na svoju VT prepisujući prethodnu vrednost itd.

Konstruktor

- ❖ Konstruktor se ne nasleđuje, u sledećem smislu: ako osnovna klasa ima konstruktor sa određenim parametrima, ne znači da i izvedena klasa implicitno ima (nasleđuje) takav konstruktor, odnosno da se i objekti te izvedene klase mogu inicijalizovati istim argumentima, već takav konstruktor mora eksplicitno da se definiše u izvedenoj klasi, ako je potreban
- ❖ Međutim, konstruktori se mogu i naslediti (*inheriting constructors*), upotrebom direktive *using* u izvedenoj klasi: ako se u definiciji izvedene klase *Derived* navede direktiva *using Base::Base*, gde je *Base* direktna osnovna klasa, onda se u opseg potrage za konstruktorima pri inicijalizaciji objekta izvedene klase uvode svi konstruktori te osnovne klase; ako se pri inicijalizaciji objekta izvedene klase odabere neki od tih konstruktora, on će inicijalizovati podobjekat te osnovne klase, dok će objekti članovi te izvedene klase i ostale njene osnovne klase biti inicijalizovani podrazumevanom inicijalizacijom; pritom, konstruktor izvedene klase sakriva ovakav nasleđeni konstruktor sa istim potpisom. Na primer:

```
struct Base {  
    Base (int, int);  
    Base (const char*);  
};  
  
struct Derived : Base {  
    using Base::Base;  
    Derived (const char*);  
};  
  
int main () {  
    Derived d1(1,2); // U redu: podobjekat Base unutar d1 se inicijalizuje pozivom Base::Base(int,int)  
    Derived d2("Hello"); // Objekat d2 se inicijalizuje pozivom Derived::Derived(const char*)  
}
```

Konstruktor

- ❖ U definiciji konstruktora klase, pre njegovog tela, a iza liste parametara i znaka :, može se navesti *lista inicijalizatora članova (member initializer list)*, u kojoj se navode inicijalizacije objekata članova i podobjekata osnovnih klasa razdvojene zarezima. Ove inicijalizacije završavaju se pre ulaska u telo konstruktora klase
- ❖ Ako neki podobjekat nema podrazumevanu inicijalizaciju, ovde se mora navesti njegova inicijalizacija, u suprotnom, prevodilac će prijaviti grešku. Na primer:

```
struct Base {  
    Base (int, int);  
};
```

Klasa *Base* nema podrazumevanu inicijalizaciju

```
struct Derived : Base {  
    Base b;  
    int& r;  
    Derived ();  
};
```

Greška u prevodenju: podobjekat osnovne klase *Base*, objekat član *b*, kao i referenca *r* nemaju podrazumevanu inicijalizaciju

```
Derived::Derived () {}
```

- ❖ Ako nestatički podatak član ima podrazumevani inicijalizator u definiciji klase, a naveden je u ovoj listi, biće inicijalizovan kao što piše u toj listi u konstruktoru, a podrazumevana inicijalizacija biće ignorisana:

```
struct X {  
    int i = 1;  
    X ();  
};
```

Podrazumevana inicijalizacija podatka člana

```
X::X () : i(2) {}
```

Objekat član *i* imaće vrednost 2

- ❖ Izuzeci tokom ove inicijalizacije mogu se hvatati *try-catch* konstruktom na nivou cele funkcije

Konstruktor

- ❖ Lista inicijalizatora članova u konstruktoru klase može imati i samo jednu inicijalizaciju (i ništa više osim nje), koja navodi ime te iste klase i inicijalizator (listu argumenata u zagradi). Tada se radi o *delegiranju* inicijalizacije: posmatrani konstruktor je *delegirajući* (*delegating constructor*), a onaj određen inicijalizacijom u listi je *ciljni konstruktor* (*target constructor*)

- ❖ Na primer:

```
class Matrix {  
public:  
    Matrix (int m, int n);  
    Matrix (int m, int n, long copyFrom[]);  
    ...  
  
private:  
    long (*mat)[N][N];  
    int m, n;  
};  
  
Matrix::Matrix (int mm, int nn) : m(mm), n(nn) {  
    if (m>N || n>N) throw MatrixTooLarge();  
    mat = new long[m][N];  
}  
  
Matrix::Matrix (int m, int n, long a[]) : Matrix(m,n) {  
    for (int i=0; i<m; i++)  
        ...  
}
```

Ciljni konstruktor

Delegirajući konstruktor

Zadatak: implementirati ovo isto bez delegirajućeg konstruktora

- ❖ Prilikom inicijalizacije, najpre se izvršava ciljni konstruktor, određen na osnovu uobičajenih pravila odabira konstruktora prema stvarnim argumentima, a onda se izvršava telo delegirajućeg konstruktora
- ❖ Na ovaj način se pravilna algoritamska dekompozicija može implementirati nešto kompaktnije, jer se zajednički deo dva konstruktora ne mora izdvajati u posebnu pomoćnu operaciju koja se poziva iz oba konstruktora, već se poziv onog prostijeg može "ugraditi" u onaj složeniji

Podrazumevani konstruktor

- ❖ Konstruktor koji se može pozvati bez stvarnih argumenata, što znači da ili nema nijedan parametar, ili svi parametri imaju podrazumevane vrednosti argumenata, naziva se *podrazumevani konstruktor (default constructor)*
- ❖ Ovaj konstruktor poziva se pri podrazumevanoj inicijalizaciji, tj. kada se za inicijalizaciju nekog objekta ne navede eksplicitan inicijalizator, ili se navede prazan inicijalizator. Na primer:

```
struct X {  
    X (int=0);  
    ...  
};
```

```
struct Y {  
    X x1, x2;  
    Y () : x2() {}  
    ...  
};
```

```
int main () {  
    X x;  
}
```

Podrazumevani konstruktor

Ovde se poziva podrazumevani konstruktor X::X(0) za objekte članove x1 i x2

Ovde se poziva podrazumevani konstruktor X::X(0) za objekat x

Podrazumevani konstruktor

- ❖ Ako klasa nema nijedan eksplisitno deklarisan konstruktor (tj. korisnički definisan konstruktor), prevodilac će implicitno deklarisati jedan podrazumevani konstruktor koji je javan, *inline* i koji vrši podrazumevanu inicijalizaciju podobjekata osnovnih klasa i objekata članova. Na primer:

```
struct X {  
    ...  
};  
  
struct Y {  
    X x1, x2;  
    Y () {}  
    ...  
};  
  
X x;
```

Ovde nema deklarisanih konstruktora, prevodilac automatski generiše podrazumevani konstruktor

Sve ovo je u redu, jer klasa X ima podrazumevani konstruktor

- ❖ Ako klasa ima neki konstruktor (pa prevodilac ne generiše implicitni podrazumevani konstruktor), programer ipak može forsirati automatsko generisanje podrazumevanog konstruktora koji bi prevodilac implicitno generisao specifikatorom *=default*:

```
struct X {  
    X (int);  
    X () = default;  
};
```

Forsiranje generisanja podrazumevanog konstruktora sa podrazumevanom inicijalizacijom podobjekata

- ❖ Može se i sprečiti automatsko generisanje ovog podrazumevanog konstruktora, ako se on označi kao obrisan specifikatorom *=delete*. Tada će prevodilac sprečiti pokušaj inicijalizacije svakog objekta te klase koja bi zahtevala poziv podrazumevanog konstruktora
- ❖ Ukoliko prevodilac implicitno deklariše podrazumevani konstruktor ili je on eksplisitno deklarisan kao *=default*, a neki od podobjekata nema podrazumevanu inicijalizaciju (npr. nema podrazumevani konstruktor, ili je referenca ili konstantni objekat i slično), prevodilac će smatrati ovaj podrazumevani konstruktor obrisanim: to znači da će prijaviti grešku samo ako se pravi objekat ove klase za koji se traži podrazumevana inicijalizacija

Konstruktor kopije

- ❖ Konstruktor klase X čiji je prvi parametar tipa $X\&$, $const X\&$, $volatile X\&$ ili $const volatile X\&$, a koji ili nema druge parametre, ili svi ostali parametri imaju podrazumevane vrednosti, naziva se *konstruktor kopije (copy constructor)*
- ❖ Ovaj konstruktor poziva se kada se objekat klase X inicijalizuje objektom istog tipa (osim ako postupak odabira konstruktora ne odabere neki drugi konstruktor koji više odgovara pozivu), kao što su sledeće situacije:
 - inicijalizacija objekta: $X x1 = x2$ ili $X x1(x2)$, gde je $x2$ objekat klase X ili iz nje izvedene klase
 - prenos argumenta pri pozivu funkcije $f(x)$ koja ima taj parametar tipa X, gde je x objekat klase X ili iz nje izvedene klase
 - povratak vrednosti iz funkcije f koja ima povratni tip X naredbom $return x$, gde je x objekat klase X ili iz nje izvedene klase
- ❖ Ako se objekat izvedene klase inicijalizuje objektom osnovne klase, i ako se odabere ovaj konstruktor, kao i na svim drugim mestima, referenca na osnovnu kalsu inicijalizuje se referencom na podobjekat osnovne klase unutar objekta izvedene klase:

```
struct Base {  
    Base (const Base&);  
    ...  
};  
  
struct Derived : Base {  
    Derived ();  
    ...  
};  
  
Derived d;  
Base b(d);
```

Konstruktor kopije

Ovde se poziva konstruktor kopije osnovne klase $Base::Base(const Base&)$, čiji se parametar - referenca vezuje za podobjekat osnovne klase unutar objekta d

Konstruktor kopije

- ❖ Ako klasa *X* nema nijedan eksplisitno deklarisan konstruktor kopije (tj. korisnički deklarisan konstruktor kopije), prevodilac će implicitno deklarisati konstruktor kopije koji je javan, *inline*, nije *explicit*, i za koji važi:
 - ako svaka direktna i virtuelna osnovna klasa *B* ima konstruktor kopije koji ima parametar tipa *const B&* ili *const volatile B&*, i ako svaka klasa *M* podatka člana ima konstruktor kopije koji prima parametar tipa *const M&* ili *const volatile M&*, onda i ovaj implicitno generisani konstruktor kopije ima parametar tipa *const X&*
 - u suprotnom, ovaj konstruktor kopije ima parametar tipa *X&*
- ❖ Klasa može imati i više konstruktora kopije, recimo onaj koji prima *X&* i onaj koji prima *const X&*
- ❖ Ako klasa ima neki konstruktor kopije (pa prevodilac ne generiše implicitni konstruktor kopije), programer ipak može forsirati automatsko deklarisanje konstruktora koji bi prevodilac implicitno deklarisao specifikatorom *=default*:

```
struct X {  
    X (X&);  
    X (const X&) = default;  
};
```

- ❖ Ako podobjekti osnovnih klasa i objekti članovi ne mogu da se kopiraju, recimo zato što su objekti klasa koje nemaju dostupne konstruktore kopije, ili ako klasa ima korisnički, eksplisitno deklarisan konstruktor premeštanja (*move constructor*) ili operator dodele premeštanjem (*move assignment operator*), onda će ovaj implicitno deklarisani konstruktor kopije biti obrisan (smatraće se da njegov poziv nije dozvoljen, iako je on deklarisan)
- ❖ U suprotnom, ako ovaj implicitno deklarisani ili podrazumevani konstruktor kopije nije obrisan, on će biti definisan i vršiće podrazumevano kopiranje podobjekata osnovnih klasa i objekata članova, po istom redosledu kao i u inicijalizaciji; ako su ti podobjekti objekti nekih klasa, pozivaju se njihovi konstruktori kopije, u suprotnom, vrši se prosto kopiranje vrednosti

Destruktor

- ❖ Destruktor (*destructor*) je posebna nestatička funkcija članica klase koja se poziva uvek na kraju životnog veka objekta. Svrha destruktora je da osloboди resurse koje je objekat eventualno zauzimao tokom svog životnog veka
- ❖ Destruktor se deklariše kao funkcija članica klase sa imenom te klase i znakom ~ ispred imena klase:

```
struct X {  
    ~X ();  
    ...  
};
```

Destruktor

- ❖ Destruktor nikada nema parametre, pa klasa ima najviše jedan destruktorn (ne može se preklopiti). Destruktor nema povratni tip. Destruktor ima pokazivač *this*, kao i svaka nestatička funkcija članica
- ❖ Destruktor se poziva implicitno uvek na kraju životnog veka objekta, bez obzira na taj životni vek:
 - na kraju izvršavanja programa, za statičke objekte
 - po završetku izvršavanja niti, za objekte sa životnim vekom vezanim za nit (*thread local*)
 - po izlasku iz opsega važenja bloka, za automatske objekte napravljene u tom bloku i privremene objekte čiji je životni vek produžen jer je za njih vezana referenca
 - izrazom *delete*, za objekte sa dinamičkim životnim vekom
 - završetkom celog izraza, za privremene objekte napravljene u tom izrazu
 - prilikom razmotavanja steka kod bačenog izuzetka, za sve automatske objekte koji su napravljeni, a čiji se blokovi napuštaju do hvatanja izuzetka
- ❖ Destruktor se može pozvati i eksplisitno, za objekte koji su napravljeni pomoću operacije *placement new*. Ako se destruktur pozove eksplisitno za objekat čiji se destruktur kasnije poziva i implicitno, ponašanje je nedefinisano

Destruktor

- ❖ Ako klasa X nema eksplisitno deklarisan destruktor (tj. korisnički definisan destruktor), prevodilac će implicitno deklarisati destruktor koji je javan i *inline*
- ❖ Programer može zahtevati automatsko generisanje destruktora koji bi prevodilac implicitno deklarisao specifikatorom =*default*
- ❖ Ako iz bilo kog razloga nije moguće uništavanje podobjekata osnovnih klasa i objekata članova (npr. jer su im destruktori nedostupni), destruktor se smatra obrisanim (nije moguć njegov implicitan ili eksplisitan poziv)
- ❖ Ako implicitno deklarisan ili podrazumevani destruktor nije obrisan, prevodilac generiše njegovu definiciju sa praznim telom
- ❖ Kada se uništava neki objekat, najpre se izvršava telo destruktora, a nakon toga se implicitno pozivaju destruktori objekata članova (ako su objekti klasa) i destruktori osnovnih klasa, uvek po redosledu obrnutom od redosleda inicijalizacije

Destruktor

- ❖ Destruktor može biti i virtuelan; destruktur deklarisan kao virtuelan u osnovnoj klasi ostaje virtuelan i u izvedenim klasama:

```
struct Base {  
    virtual ~Base ();
```

Destruktor je virtuelan i u izvedenim klasama

```
};
```

- ❖ Tada je poziv destruktora polimorfan u situacijama kada se objektu pristupa posredno, preko pokazivača ili reference na osnovnu klasu:

```
Base* pb = new Derived;  
delete pb;
```

Pozvaće se destruktur izvedene klase

- ❖ U ovakvoj situaciji, biće pozvan najpre destruktur izvedene klase, koji uvek implicitno poziva i destruktore osnovnih klasa, pa u telu destruktora nikada ne treba pisati eksplisitni poziv destruktora osnovne klase
- ❖ Ukoliko destruktur ne bi bio virtuelan, u ovakvim situacijama ponašanje bi bilo nedefinisano. Zbog toga se preporučuje da destruktur klase uvek bude virtuelan, ako je klasa polimorfna (ako ima bar jednu virtuelnu funkciju članicu), jer se njenim objektima pristupa polimorfno: računa se na to da se objektu pristupa kao generalizovanom entitetu, bez znanja o konkretnom tipu tog objekta, što važi i za uništavanje
- ❖ Destruktor može da se deklariše i kao čisto virtuelan (=0), na primer u osnovnoj klasi koja treba da bude apstraktna, a nema drugu pogodnu funkciju članicu koja bi bila čisto virtuelna. Takvi destruktori ipak moraju da imaju definiciju, jer se destruktur osnovne klase uvek poziva kada se uništava objekat izvedene klase

Operator dodele kopiranjem

- ❖ Operator dodele kopiranjem (*copy assignment operator*) klase X je nestatička operatorska funkcija članica te klase sa imenom **operator=** čiji je jedini parametar tipa X, X&, const X&, volatile X& ili const volatile X&
- ❖ Ova funkcija poziva se kada se objekat klase X nalazi kao levi operand operatora dodele =, i kada se baš ona odabere u postupku odabira preklopljene operatorske funkcije za desni operand tog operatora:

```
struct X {  
    X& operator= (const X&);  
    ...  
};  
X x1, x2 = x1, x3;  
x3 = x2;
```

Operator dodele kopiranjem

Ovde se poziva konstruktor kopije

Ovde se poziva podrazumevani konstruktor

Ovde se poziva operator dodele kopiranjem: x3.operator=(x2)

- ❖ Operacija dodele kopiranjem je potpuno drugačija operacija od inicijalizacije kopiranjem:
 - inicijalizacija kopiranjem se vrši na početku životnog veka objekta, i tada se poziva konstruktor kopije; pre toga objekat koji se inicijalizuje ne postoji
 - dodela kopiranjem se vrši u izrazu, kao operacija dodele, i tada se poziva operator dodele; pre toga objekat kome se dodeljuje već postoji i inicijalizovan je ranije

Operator dodele kopiranjem

- ❖ Ako klasa *X* nema nijedan eksplisitno deklarisan operator dodele kopiranjem (tj. korisnički definisan takav operator), prevodilac će implicitno deklarisati operator dodele kopiranjem koji je javan, *inline*, i za koji važi:
 - ako svaka direktna osnovna klasa *B* ima operator dodele kopiranjem koji ima parametar tipa *B&*, *const B&* ili *const volatile B&*, i ako svaka klasa *M* podatka člana ima operator dodele kopiranjem koji prima parametar tipa *M&*, *const M&* ili *const volatile M&*, onda i ovaj implicitno deklarisani operator dodele ima oblik *X& operator=(const X&)*
 - u suprotnom, ovaj operator dodele ima oblik *X& operator=(X&)*
- ❖ Klasa može imati i više operatora dodele kopiranjem, recimo onaj koji prima *X&* i onaj koji prima *const X&*
- ❖ Ako klasa ima neki korisnički definisan operator dodele kopiranjem (pa prevodilac ne generiše onaj implicitni), programer ipak može forsirati generisanje podrazumevanog operatora dodele kopiranjem specifikatorom *=default*:

```
struct X {  
    X& operator= (X&);  
    X& operator= (const X&) = default;  
};
```

- ❖ Pošto klasa tako uvek ima deklarisan operator dodele kopiranjem, on uvek sakriva takav operator iz osnovne klase
- ❖ Implicitno deklarisan operator dodele kopiranjem biće smatrano obrisanim ako ta klasa ima korisnički definisan konstruktor premeštanja (*move constructor*) ili operator dodele premeštanjem (*move assignment operator*)
- ❖ Ako podobjekti osnovnih klasa i objekti članovi ne mogu da se dodeljuju kopiranjem, recimo zato što su objekti klasa koje imaju nedostupne operatore dodele kopiranjem, ili ako je član konstanta ili referenca, onda će podrazumevani operator dodele kopiranjem biti obrisan (smatraće se da njegov poziv nije dozvoljen, iako je on deklarisan)
- ❖ U suprotnom, ako ovaj implicitno deklarisani ili podrazumevani operator dodele kopiranjem nije obrisan, on će biti definisan i vršiće podrazumevanu dodelu kopiranjem podobjekata osnovnih klasa i objekata članova, po istom redosledu kao i u inicijalizaciji; ako su ti podobjekti objekti nekih klasa, pozivaju se njihovi operatori dodele kopiranjem, u suprotnom, vrši se prosto kopiranje vrednosti

Glava 13: Objekti sa zauzetim resursima

- ❖ Objekti sa zauzetim resursima
- ❖ Kopiranje objekata
- ❖ Premeštanje resursa
- ❖ Kategorije vrednosti
- ❖ Reference na dvrednosti
- ❖ Konstruktor premeštanja
- ❖ Operator dodele premeštanjem
- ❖ Semantika vrednosti u C++17



Objekti sa zauzetim resursima

- ❖ U mnogim situacijama objekti neke klase imaju zauzete (alocirane), pridružene resurse, kao što su, na primer:
 - dinamički alocirani objekti
 - otvoreni fajlovi
 - drugi resursi alocirani uslugom operativnog sistema ili izvršnog okruženja programa, kao što su npr. memorija, niti, semafori, priključnice (*socket*), neki drugi kanali komunikacije, "ključevi" (*lock, mutex*) i sl.
- ❖ Ovi resursi po pravilu se alociraju pri inicijalizaciji objekta, što obezbeđuju odgovarajući konstruktori
- ❖ Na primer, posmatrajmo klasu *string* koja treba da realizuje apstraktni tip podataka za nizove znakova, za koje želimo da se mogu dimenzionisati dinamički, u trenutku pravljenja, na osnovu potrebe onoga čime se inicijalizuju; osim toga, želimo da implementiramo operacije konkatenacije dva stringa (npr. preklapanjem operatora +) i mnoge druge. Zbog toga nam je potrebno da objekti ove klase imaju pridruženi dinamički niz znakova, jer se samo tako može napraviti niz čija se dimenzija zna tek prilikom pravljenja. Početna skica ove klase:

```
class string {  
public:  
    string () : str(nullptr) {}  
    string (const char*);  
    ...  
  
private:  
    char* str;  
};  
  
string::string (const char* s) : string() {  
    if (!s) return;  
    str = new char[std::strlen(s)+1];  
    std::strcpy(str,s);  
}
```

Omogućava podrazumevanu inicijalizaciju:
string s;

Omogućava inicijalizaciju ugrađenim nizovima znakova:
string s("Hello");

Objekti sa zauzetim resursima

- ❖ Naravno, ove zauzete resurse treba propisno i oslobođiti (deallocate) kada objekat prestaje da živi. Ako se predviđa korišćenje ovih objekata po vrednosti (svih kategorija životnog veka), odnosno ugradnjom (kao podobjekti, npr. objekti članovi), njihovo uništavanje je implicitno. Na primer:

```
int main () {
    string str("Hello");
    ...
}
```

Ovde se završava životni vek objekta *str*

- ❖ Prema već navedenim pravilima, ako se u klasi ne navede eksplicitno destruktor, prevodilac će generisati implicitni destruktor koji vrši destrukciju objekata članova pozivom njihovih destruktora; međutim, za članove koji su ugrađenih tipova, destrukcija nema nikakvog efekta, što je ovde slučaj, pošto je član *str* pokazivač. Prema tome, u ovom slučaju, dinamički niz znakova neće biti dealociran, pa će postojati problem curenja memorije (*memory leak*)
- ❖ Zato nam je ovde neophodan korisnički definisan destruktor koji vrši potrebnu dealokaciju:

```
class string {
public:
    string () : str(nullptr) {}
    string (const char*);

    ~string () { delete [] str; str = nullptr; }
    ...
};

};
```

Objekti sa zauzetim resursima

- ❖ Dalje, ako se predviđa korišćenje ovih objekata po vrednosti (svih kategorija životnog veka), korisnici ove klase očekivaće i mogućnost inicijalizacije kopiranjem. Na primer:

```
int main () {  
    string s1("Hello"), s2(s1);  
    ...  
}
```

Objekat *s2* inicijalizuje se kopiranjem objekta *s1*

- ❖ Prema već navedenim pravilima, ako se u klasi ne navede eksplisitno konstruktor kopije, prevodilac će generisati implicitni konstruktor kopije koji vrši inicijalizaciju kopiranjem objekata članova pozivom njihovih konstruktora kopije; međutim, za članove koji su ugrađenih tipova, vrši se prosto kopiranje vrednosti, što je ovde slučaj, pošto je član *str* pokazivač. Prema tome, u ovom slučaju, objekti klase *string* biće *plitko kopirani* (*shallow copy*), što znači da će se kopirati samo pokazivač, a ne i dinamički niz na koji on ukazuje
- ❖ Ovo nije željeno ponašanje, jer će dovesti do toga da kopirani objekti (npr. *s1* i *s2* u primeru gore) dele isti dinamički niz, pa će promene učinjene u jednom biti vidljive i u drugom. Ovde je namera da ti objekti postoje kao nezavisni entiteti i da se njihove izmene rade nezavisno
- ❖ Zato nam je ovde neophodan korisnički definisan konstruktor kopije koji vrši *duboko kopiranje* (*deep copy*):

```
class string {  
public:  
    string () : str(nullptr) {}  
    string (const char*);  
    string (const string& s) : string(s.str) {}  
  
    ~string () { delete [] str; str = nullptr; }  
    ...  
};
```

Objekti sa zauzetim resursima

- ❖ Potpuno analogno, ako se predviđa korišćenje ovih objekata po vrednosti (svih kategorija životnog veka), korisnici ove klase očekivaće i mogućnost dodele kopiranjem. Na primer:

```
int main () {
    string s1("Hello"), s2;
    s2 = s1;
    ...
}
```

- ❖ Prema već navedenim pravilima, ako se u klasi ne navede eksplisitno operator dodele kopiranjem, prevodilac će generisati implicitni operator dodele kopiranjem koji vrši dodelu kopiranjem objekata članova pozivom njihovih operatora dodele kopiranjem; međutim, za članove koji su ugrađenih tipova, vrši se prosto kopiranje vrednosti, što je ovde slučaj, pošto je član *str* pokazivač. Prema tome, i u ovom slučaju objekti klase *string* biće dodelom plitko kopirani
- ❖ Zato nam je ovde neophodan i korisnički definisan operator dodele kopiranjem koji vrši duboko kopiranje, ali za razliku od konstruktora kopije, on mora najpre da oslobodi postojeći dinamički niz (ono što radi destruktur), pa onda alocira i kopira novi niz (ono što radi konstruktor kopije):

```
string& string::operator= (const string& s) {
    if (this == &s) return *this;
    delete [] str; str = nullptr;
    if (!s.str) return;
    str = new char[std::strlen(s.str)+1];
    std::strcpy(str,s.str);
    return *this;
}
```

U slučaju poziva *s=s*, ne radi dalje, jer bi to obrisalo dinamički niz objekta *s*

Zadatak:

Ova implementacija obezbeđuje osnovnu garanciju sigurnosti od izuzetaka. Objasniti zašto i prepraviti je tako da obezbeđuje jaku garanciju.

Objekti sa zauzetim resursima

- ❖ Pritom, ove operacije imaju neke zajedničke delove, pa je dobro *refaktorisati* (*refactor*) ovu klasu tako da se ti zajednički delovi izdvoje u pomoćne potprograme i tako eliminišu dupliranja koda:
 - konstruktor kopije radi alokaciju i kopiranje
 - destruktur radi dealokoaciju
 - operator dodele radi najpre dealokaciju (kao destruktur), pa onda alokaciju i kopiranje (kao konstruktor kopije)

```
class string {
public:
    string () : str(nullptr) {}
    string (const char* s) : string() { allocate(s); copy(s); }
    string (const string& s) : string(s.str) {}
    string& operator= (const string& s);

    ~string () { release(); }
    ...

protected:
    void allocate (const char* s) { if (s) str = new char[std::strlen(s)+1]; }
    void copy (const char* s) { if (s) std::strcpy(str,s); }
    void release () { delete [] str; str = nullptr; }

    ...
};

inline string& string::operator= (const string& s) {
    if (this!=&s) {
        release(); allocate(s.str); copy(s.str);
    }
    return *this;
}
```

Zadatak:

Ova implementacija ima nedostatak zbog toga što operacije *allocate* i *copy* dva puta prolaze kroz ceo izvorni niz znakova (to rade funkcije *strlen* i *strcpy*). Prepraviti je tako da se to radi samo jednom.

Objekti sa zauzetim resursima

- ❖ Prema tome, u ovakvim situacijama potreban je posebno definisan konstruktor kopije, operator dodele kopiranjem i destruktur; zato postoji preporuka da se, ako za klasu postoji potreba za jednom od ovih operacija, obrati pažnja i verovatno naprave sve tri, jer je u pitanju možda slučaj objekta sa zauzetim resursima koji se koristi i ugrađuje "po vrednosti"
- ❖ Treba primetiti da je sve ovo posledica ključne odluke da se objekti klase mogu koristiti i ugrađivati "po vrednosti", odnosno da se mogu koristiti na isti način kao i objekti ugrađenih (neklassnih) tipova i biti svih kategorija po životnom veku
- ❖ Ukoliko to ne bi bio slučaj, kao što i nije u mnogim drugim novijim objektno orijentisanim jezicima, sve ove komplikacije ne bi bilo; u tim jezicima važi:
 - objekti su samo dinamički i uvek anonimni
 - objektima se pristupa samo preko posrednika (pokazivača, odnosno referenci)
 - postoje samo operacije tih klasa koje se pozivaju eksplicitno; nema operatorskih funkcija
 - nema ugrađenih objekata članova klase (samo pokazivača / referenci), nema automatskih i statičkih objekata (samo pokazivača / referenci) itd.
 - nema implicitnih kopiranja prilikom inicijalizacije, dodele, prenosa argumenata i povratne vrednosti (kopiraju se i prenose samo pokazivači / reference na objekte)

Kopiranje objekata

- Prepostavimo da smo implementirali operatorsku funkciju *operator+* koja vraća rezultat spajanja dva niza znakova iz operanada tipa *string*; taj rezultat mora biti objekat tipa *string* (po vrednosti), jer je on novonapravljeni objekat koji sadrži poseban dinamički niz znakova:

```
string operator+ (const string&, const string&);
```

- Osim toga, implementirali smo i nestatičku funkciju članicu *substr* koja vraća (ponovo po vrednosti) nov objekat tipa *string* koji sadrži samo podniz datog niza znakova objekta domaćina, počev od zadate pozicije i zadate dužine; za potrebe implementacije ove funkcije, napravili smo još nekoliko funkcija članica klase *string*:

```
inline void allocate (size_t sz) {
    if (sz+1==0) throw std::length_error;
    if (sz) str = new char[sz+1];
}

inline void copy (const char* s, std::size_t count) {
    if (!s || !count) return;
    size_t i = 0;
    while (i<count && *s) str[i++] = *s++;
    str[i] = '\0';
}

inline size_t size () const { return str?std::strlen(str):0; }

inline string substr (size_t pos, size_t count) const {
    size_t sz = size();
    if (pos>=sz) throw std::out_of_range;
    if (pos+count>sz) count = sz - pos;
    string s;
    s.allocate(count);
    s.copy(str+pos, count);
    return s;
}
```

Pomoćna, zaštićena funkcija članica klase *string*

Pomoćna, zaštićena funkcija članica klase *string*

Javna funkcija članica klase *string*

Javna funkcija članica klase *string*

Kopiranje objekata

- ❖ Pre verzije jezika C++17, svaki rezultat izraza klasnog tipa, uključujući poziv funkcije koja ima povratni tip klase i eksplicitan poziv konstruktora klase, kao rezultat proizvodi privremeni, bezimeni objekat koji se pravi na mestu poziva funkcije, tokom izvršavanja izraza koji sadrži taj poziv i inicijalizuje u trenutku povratka iz te funkcije izrazom iza naredbe *return*
- ❖ Na primer, osnovna semantika (pre verzije C++17) sledeće inicijalizacije

```
string s = s1 + s2;
```

je sledeća: rezultat izraza *s1+s2* je privremeni objekat klase *string* koji je inicijalizovan prilikom povratka iz funkcije odgovarajućim konstruktorom; nakon povratka, objekat *s* klase *string* inicijalizuje se pozivom konstruktora kopije sa tim privremenim objektom kao argumentom poziva konstruktora; naravno, ti konstruktori moraju da budu dostupni na odgovarajućim mestima

- ❖ Potpuno ista situacija je i u sledećem slučaju:

```
string s = string("Hello");
```

- ❖ U određenim situacijama, prevodiocu je na raspolaganju optimizacija *izostavljanja kopiranja* (*copy elision*) koju može, ali ne mora da izvrši (pre verzije C++17), čak i kada konstruktori čiji se pozivi izostavljaju imaju bočne efekte

Kopiranje objekata

- ❖ Jedan slučaj u kom se može (pre verzije C++17), odnosno mora (od verzije C++17) vršiti izostavljanje kopiranja jeste inicijalizacija objekta privremenim objektom koji je rezultat izraza, uključujući i poziv konstruktora u izrazu: kada se objekat klase *string* inicijalizuje izrazom koji vraća rezultat tipa *string*, za rezultat tog izraza ne mora da se pravi privremeni objekat, već taj rezultat može neposredno da se izgradi u objektu koji se inicijalizuje, tako što se taj objekat inicijalizuje isto kao što bi se inicijalizovao taj privremeni objekat:

```
string sa = s1 + s2;  
string sb = string("Hello");
```

- ❖ Tada će naredba *return* u pozivu operatorske funkcije *operator+(s1,s2)* konstruisati rezultat u samom memorijskom prostoru objekta *sa* koji se rezultatom ovog poziva operatorske funkcije inicijalizuje, pa konstruktor kopije neće biti pozivan; slično važi i za konstruktor *string("Hello")*
- ❖ Ovo važi za sve inicijalizacije objekata, uključujući i parametre funkcija pri pozivu funkcije, kada se oni inicijalizuju stvarnim argumentima koji su rezultati izraza
- ❖ Ovo se implementira tako što se funkciji u pozivu zapravo dostavlja, kao jedan od skrivenih parametara, adresa memorijskog prostora za objekat u koji treba da konstruiše i smesti rezultat; ako je to privremeni objekat, na mestu poziva odvaja se taj prostor i pozvanoj funkciji prosleđuje njegova adresa; ako je to objekat koji se inicijalizuje uz izostavljanje kopiranja, onda se dostavlja adresa tog objekta
- ❖ Pritom, odgovarajući konstruktor kojim bi se inicijalizovao privremeni objekat, kao i konstruktor kopije moraju biti dostupni kao da se pozivaju (iako se zapravo ne izvršavaju), pa prevodilac sprovodi formalna pravila isto kao da se oni pozivaju, odnosno kao da se ova optimizacija ne vrši (tzv. *as if* pravilo)

Kopiranje objekata

- ❖ Kada je privremeni objekat kojim se vrši inicijalizacija operand naredbe *return*, ova ista optimizacija izostavljanja kopiranja naziva se *optimizacija povratne vrednosti (return value optimization, RVO)*; na primer:

```
string concat (const string& s1, const string& s2) {  
    return s1+s2;  
}
```

RVO: rezultat izraza iza *return* biće konstruisan u prostoru privremenog objekta koji ova funkcija vraća, pa neće biti pozivan konstruktor kopije

- ❖ Moguće su i ovakve višestruke, vezane optimizacije koje izbegavaju višestruka kopiranja
- ❖ U opštem slučaju, ukratko, ako je X klasni tip, navedene optimizacije izgledaju ovako:

```
X f() {  
    return X();  
}
```

RVO: samo jedan jedini poziv podrazumevanog konstruktora X() koji ininijalizuje privremeni objekat koji je rezultat konačnog izraza *f()*

```
f();
```

```
X x = X(X(f()));
```

Samo jedan jedini poziv podrazumevanog konstruktora X() koji ininijalizuje objekat *x*

- ❖ Do verzije jezika C++17, ovo su bile opcione (neobavezne) optimizacije, iako ih većina prevodilaca sprovodi; od verzije C++17 one su obavezne (uvek se sprovode)
- ❖ Programi čija semantika zavisi od toga da li se ove optimizacije sprovode ili ne, odnosno u kojima postoje bočni efekti konstruktoru kopije i operatora dodele kopiranjem nisu dobri

Kopiranje objekata

- ❖ Još jedna varijanta izbegavanja kopiranja jeste i dalje opcionala (i u verziji C++17) i naziva se *optimizacija imenovane povratne vrednosti* (*named return value optimization, NRVO*): ako je operand naredbe *return* ime (ali ne nepostojanog) automatskog objekta koji nije parametar te funkcije ili parametar *catch* bloka, i koji je istog tipa kao povratni tip funkcije (uz ignorisanje cv-kvalifikacije), zapravo se taj automatski objekat izgrađuje (inicijalizuje) u prostoru objekta koji se vraća iz funkcije; sve operacije nad tim automatskim objektom vrše se u tom memorijskom prostoru. Na primer:

```
string join (const string& s1, const string& s2) {  
    string s = s1 + " " + s2;  
    return s;  
}
```

NRVO: automatski objekat *s* zauzimaće memorijski prostor povratne vrednosti i njegova inicijalizacija i sve druge operacije nad njim vršiće se u tom prostoru

- ❖ U opštijem slučaju:

```
T f () {  
    T t;  
    ...  
    return t;  
}
```

NRVO: automatski objekat *t* zauzimaće memorijski prostor povratne vrednosti i njegova inicijalizacija i sve druge operacije nad njim vršiće se u tom prostoru

- ❖ Ova optimizacija implementira se na isti opisani način: funkcija prima kao skriveni parametar adresu prostora u koji treba da vrati rezultat; pošto na osnovu koda tela funkcije prevodilac može lako da zaključi da se iza *return* imenuje automatski objekat, sve operacije nad tim objektom (uključujući i poziv konstruktora) može da usmeri na taj memorijski prostor, odnosno tu adresu smatra adresom tog automatskog objekta

Kopiranje objekata

- ❖ Treba primetiti da ovakva optimizacija ne može da “pređe granice” poziva funkcije (osim eventualno za *inline* funkcije), jer prevodilac u opštem slučaju ne može da zna kako izgleda upotreba parametara u telu funkcije na mestu poziva te funkcije, a parametar mora svakako da se inicijalizuje stvarnim argumentom prilikom poziva. Upravo zato NRVO isključuje parametre, iako su i oni automatski po trajanju skladišta. Na primer:

```
T f (T t) {  
    return t;  
}  
  
T x;  
  
f(x);
```

Ovde se ne vrši NRVO: konstruktor kopije pozvaće se i za inicijalizaciju parametra stvarnim argumentom ($T t(x)$), a potom i za inicijalizaciju privremenog objekta koji predstavlja rezultat poziva funkcije objektom ia *return* ($T temp(t)$)

- ❖ Takođe treba primetiti da se, ukoliko funkcija ima parametre tipa reference, ta referenca samo vezuje za stvarni argument; ukoliko je stvarni argument privremeni objekat, pa zato nije vrijednost, referenca mora biti na konstantu, inače ova inicijalizacija nije dozvoljena. U takvom slučaju, kopiranja svakako nema:

```
T f (const T& t);  
  
f(T());
```

Pošto je parametar funkcije referenca, nema kopiranja stvarnog argumenta

- ❖ Ako je parametar klasnog tipa (a ne referenca), kopiranja uvek ima i samo ponekad se može izostaviti:

```
T f (T t);  
  
T x;  
  
f(x);  
  
f(T());
```

Pošto je parametar funkcije klasnog tipa, stvarni argument se kopira u parametar, tj. poziva se konstruktor kopije

Ovde se može (ili mora, za C++17) izostaviti kopiranje

Kopiranje objekata

- ❖ Posmatrajmo sada izračunavanje sledećeg izraza, pri čemu su sve imenovane varijable objekti tipa *string*:

```
s = s1 + " " + s2.substr(2,7) + " " + s3.substr(0,2);
```

- ❖ Izračunavanje ovog izraza teče ovako (pretpostavka je da je verzija jezika pre C++17):

- drugi argument poziva *operator+(s1, " ")* je tipa *const char[]*, pa se implicitno konvertuje u *const char** ugrađenom konverzijom (“rastakanje niza u pokazivač”), a potom i korisnički definisanom konverzijom u *string*, pozivom konstruktora konverzije *string(const char*)*; rezultat konverzije je privremeni objekat kojim se inicijalizuje formalni parametar; pošto je to referenca, ona se vezuje za taj privremeni objekat
- rezultat ovog poziva biće jedan privremeni objekat tipa *string*, označimo ga ovde sa *x*, koji je inicijalizovan povratnim izrazom; taj privremeni objekat ima svoj pridruženi dinamički niz znakova koji sadrži konkatenaciju niza znakova iz *s1* i niza znakova “ ”; ukoliko prevodilac ne sprovodi *RVO*, ovaj privremeni objekat biće inicijalizovan pozivom konstruktora kopije, ako je operand izvršene naredbe *return* tipa *string*
- unutar pozvane funkcije *s2.substr(2,7)* pravi se automatski objekat *s* koji alocira prostor za traženi podstring; ovim automatskim objektom se inicijalizuje privremeni objekat koji predstavlja rezultat ovog poziva, označimo ga ovde za *y*; ukoliko prevodilac ne sprovodi *NRVO*, ponovo će se pozivati konstruktor kopije
- poziva se sada operatorska funkcija za drugi operator + u izrazu; parametri ove funkcije su reference koje se vezuju za navedene privremene objekte, tj. poziva se *operator+(x,y)*

i tako dalje. Na kraju, poziva se operatorska funkcija *operator=* čiji je desni operand privremeni objekat napravljen kao rezultat podizraza sa desne strane znaka =

Kopiranje objekata

- ❖ Prema tome, u ovakvim i sličnim izrazima, vrši se potencijalno mnogo poziva konstruktora kopije ili operatora dodele kopiranjem koji kopiraju sadržaj, tj. alociraju nove pridružene resurse samo da bi kopirali taj isti sadržaj iz privremenih objekata, koji ubrzo potom nestaju
- ❖ Kopiranje resursa iz privremenog objekta je prilično beskoristan posao i čist režijski trošak za vreme izvršavanja, pošto privremeni objekat iz kog se pridruženi sadržaj kopira (uz potencijalnu alokaciju novog resursa) ubrzo nestaje, i *sigurno se ne može* koristiti za bilo šta drugo, jer je on nedostupan - on je svakako implicitan rezultat izraza koji se koristi eventualno samo kao operand (argument) naredne operacije i ni za šta više
- ❖ Količina kopiranja je utoliko veća ukoliko:
 - prevodilac ne sprovodi optimizaciju izostavljanja kopiranja na svim mestima na kojima je to moguće i dozvoljeno pravilima jezika
 - se parametri u funkcije prenose po vrednosti, a ne po referenci
 - funkcije vraćaju rezultate kopiranjem automatskih objekata
 - se objekti ugrađuju u druge objekte po vrednosti (a ne preko pokazivača), posebno u apstraktne strukture podataka (kolekcije, kontejnere), što je često slučaj, pa se umetanjem u strukturu i vraćanjem vrednosti iz strukture objekti kopiraju (operatorima dodele ili konstruktorima kopije)

Premeštanje resursa

- ❖ Opisani problem predstavlja izvornu motivaciju za uvođenje tzv. *semantike premeštanja* (*move semantics*) u jezik C++ (počev od verzije C++11): umesto da se alocirani resursi pridruženi privremenom objektu kopiraju, oni mu se mogu *preoteti*, tj. *premestiti* u objekat koji se tim privremenim objektom inicijalizuje ili kom se taj privremeni objekat dodeljuje, pošto privremenom objektu svakako više nisu potrebni
- ❖ Tako je (od verzije C++11) moguće definisati konstruktore i operatore dodele, pa i one koji kao argument mogu da prime objekat istog tipa, koji će se pozivati u slučaju da se kao argument pojavi *dvrednost* (*rvalue*), odnosno rezultat izraza koji nije *lvrednost* (*lvalue*)
- ❖ Konstruktor koji prima referencu na dvrednost iste klase naziva se *konstruktor premeštanja* (*move constructor*); nestatička operatorska funkcija članica *operator=* koja prima ovakvu referencu na dvrednost iste klase naziva se *operator dodele premeštanjem* (*move assignment*)
- ❖ Parametri koji se vezuju za dvrednosti su tzv. *reference na dvrednosti* (*rvalue reference*) i označavaju se u deklaraciji sa dva znaka &:

```
class string {  
public:  
    ...  
    string (const string& s);  
    string& operator= (const string& s);  
  
    string (string&& s);  
    string& operator= (string&& s);  
    ...  
};
```

Konstruktor kopije

Operator dodele kopiranjem

Konstruktor premeštanja

Operator dodele premeštanjem

Premeštanje resursa

- ❖ Konstruktor premeštanja poziva se kada se objekat inicijalizuje izrazom koji predstavlja dvrednost, a takav je privremeni objekat, pod uslovom da prevodilac ne izostavlja kopiranje:

```
string s = s1 + s2;
```

Ukoliko se ne izostavlja kopiranje, pozvaće se konstruktor premeštanja sa argumentom koji je rezultat izraza $s1+s2$

```
void f (string);
```

```
f(s1+s2);
```

Ukoliko se ne izostavlja kopiranje, pozvaće se konstruktor premeštanja kojim se inicijalizuje parametar funkcije rezultatom izraza $s1+s2$

- ❖ Osim toga, kao specijalan slučaj, konstruktor premeštanja se poziva i kada se povratna vrednost funkcije inicijalizuje imenovanim objektom sa automatskim trajanjem skladišta, ovaj put uključujući i formalni parametar, opet osim ako prevodilac ne vrši NRVO:

```
inline string substr (size_t pos, size_t count) const {  
    ...  
    string s;  
    ...  
    return s;  
}
```

Ukoliko se ne vrši NRVO, poziva se konstruktor premeštanja koji inicijalizuje privremeni objekat koji prihvata rezultat poziva funkcije sa argumentom s

- ❖ Ukoliko klasa nema konstruktor premeštanja, u ovakvim slučajevima pozvaće se konstruktor kopije, pa postojanje konstruktora premeštanja ne treba da menja semantiku programa: konstruktor premeštanja može da popravi performanse i smanji količinu kopiranja resursa, ali pošto njegova aktivacija nije uvek određena (u slučajevima opcionih izbegavanja kopiranja), logika programa ne sme da zavisi od njegovog ponašanja
- ❖ Slično važi i za operator dodele premeštanjem: poziva se kada je desni operand dvrednost

Premeštanje resursa

- ❖ Pošto većina prevodilaca izostavlja kopiranje čak i kada je to neobavezno, a od verzije C++17 mnoge od navedenih optimizacija postale su obavezne, premeštanje zbog izbegavanja kopiranja resursa iz privremenih objekata gubi na značaju
- ❖ Međutim, semantika premeštanja ipak ima svoj značaj (koji postoji oduvek) i koji je možda i važniji od navedenog: premeštanje se može vršiti i na mestima za koje nisu predviđene optimizacije, ukoliko je ono efikasnije i nema potrebe za kopiranjem
- ❖ Na primer, bibliotečna šablonska klasa *vector* predstavlja niz promenljivih dimenzija koji se implicitno proširuje po potrebi (operacija *resize*). U tom slučaju za niz elemenata alocira se nov prostor, a elementi vektora se po vrednosti kopiraju na novo alocirano mesto. Ukoliko tip elementa vektora ima semantiku premeštanja, biće upotrebljeno premeštanje umesto kopiranja; ako odgovarajuće operacije premeštanja nisu definisane, vršiće se kopiranje
- ❖ Osim toga, neke apstrakcije ne dozvoljavaju kopiranje (jer to nema smisla), ali se njihovi alocirani resursi mogu premeštati; jedan takav primer je apstrakcija ulaznog ili izlaznog znakovnog toka (*istream* i *ostream*)
- ❖ U ovakvim situacijama moguće je i eksplisitno zahtevati semantiku premeštanja, iako bi se podrazumevano pozivao konstruktor kopije ili operator dodele kopiranjem: izraz koji je lvrednost se može eksplisitno konvertovati u dvrednost pozivom bibliotečne funkcije *std::move* koja vraća referencu na dvrednost za argument koji može biti i lvrednost; za ovakav rezultat onda se vezuju funkcije čiji parametri primaju reference na dvrednosti, pa i konstruktor premeštanja ili operator dodele premeštanjem:

```
string s1("Hello");
string s2 = std::move(s1);
```

Objekat *s2* biće inicijalizovan konstruktorom premeštanja i preoteće resurse iz objekta *s1*, recimo zato što objekat *s1* više nema potrebe za tim resursom

Premeštanje resursa

- ❖ Konstruktor premeštanja i operator dodele premeštanjem treba da preotmu, tj. premeste resurse iz izvorišnog u odredišni objekat, ali tako da izvorišni objekat ostave u konzistentnom stanju, tako da on i dalje bude validan objekat koji se svakako može uništiti (jer se on svakako uništava pozivom destruktora); zato ove operacije po pravilu imaju parametar koji je referenca na nekonstantu
- ❖ Za primer klase *string*, to znači da se objekat član *str* u izvorišnom objektu postavi na *null* vrednost, a ne ostavi na staroj vrednosti, jer preuzeti dinamički niz znakova više nije njegov; u svakom slučaju, njegov destruktur ne sme uništiti taj niz jer on pripada drugom objektu (premešten je):

```
class string {
public:
    string () : str(nullptr) {}
    string (const char* s) : string() { allocate(s); copy(s); }

    string (const string& s) : string(s.str) {}
    string& operator= (const string& s);

    string (string&& s) : string() { move(s); }
    string& operator= (string&& s);

    ~string () { release(); }
    ...

protected:
    void allocate (const char* s) { if (s) str = new char[std::strlen(s)+1]; }
    void copy (const char* s) { if (s) std::strcpy(str,s); }
    void release () { delete [] str; str = nullptr; }

    void move (string&& s) { str = s.str; s.str = nullptr; }
    ...

};

inline string& string::operator= (string&& s) {
    if (this!=&s) {
        release(); move(s);
    }
    return *this;
}
```

Premeštanje resursa

- ❖ Vratimo se na implementaciju operatora dodele kopiranjem:

```
inline string& string::operator= (const string& s) {  
    if (this != &s) {  
        release();  
        allocate(s.str);  
        copy(s.str);  
    }  
    return *this;  
}
```

- ❖ Ova implementacija ima sledeća dva nedostatka:

- provera na “samodeljivanje” (*self assignment*) u naredbi *if* izvršava se uvek i troši režijsko vreme u svakom pozivu, iako se verovatno nikada u programu neće dogoditi problematična situacija samodeljivanja (jer nema smisla)
- ova implementacija pruža samo osnovni (slabi) nivo garancije od izuzetaka: ako alokacija prostora u funkciji *allocate* ne uspe i baci izuzetak, objekat domaćin (na koga ukazuje *this*) već je izmenjen u operaciji *release*; bilo bi bolje da se obezbedi jaka garancija

Premeštanje resursa

❖ Oba problema se mogu prevazići *idiomom kopiranja i zamene (copy-and-swap idiom)*:

- ako je desni operand operatora dodele ivrednost, najpre se konstruktorom kopije pravi lokalni automatski objekat - parametar operatorske funkcije, koji kopira resurs argumenta u formalni parametar
- potom se radi prosta i efikasna razmena (*swap*) resursa između parametra i objekta domaćina (onog na koga ukazuje *this*); ta zamena radi se pomoćnom operacijom *swap* koja ne baca izuzetak (*non-throwing*)
- kada se izlazi iz ove operatorske funkcije, parametar, kao lokalni automatski objekat biće uništen pozivom destruktora, koji će obrisati prethodni resurs koji je objekat domaćin imao u sebi:

```
inline string& string::operator= (string other) {  
    swap(*this,other);  
    return *this;  
}
```

Funkcija *swap* će samo razmeniti vrednosti pokazivača *this->str* i *other.str*, bez bacanja izuzetka

❖ Sada će se dešavati sledeće:

Objekat *other* se ovde uništava i njegov destruktur briše niz koji je ranije bio pridružen objektu **this*

- ako je desni operand operatora dodele dvrednost, npr. privremeni objekat, formalni parametar operatorske funkcije biće inicijalizovan njime i to ili direktnom konstrukcijom, zbog izostavljanja kopiranja (svakako za C++17), ili konstruktorom premeštanja ako ove optimizacije nema:

```
s = string("Hello");
```

Formalni parametar *other* biće inicijalizovan ili pozivom konstruktora konverzije *string(const char*)*, ili pozivom konstruktora premeštanja *string(string&&)* (pre C++17)

- ako je desni operand operatora dodele ivrednost, formalni parametar operatorske funkcije biće inicijalizovan pozivom konstruktora kopije; ako se pri alokaciji njegovog resursa baci izuzetak, to će biti urađeno pre ulaska u operatorsku funkciju, pa objekat domaćin (levi operand operatora dodele) neće biti promenjen:

```
s1 = s2;
```

Formalni parametar *other* biće inicijalizovan pozivom konstruktora kopije *string(const string&)*

Premeštanje resursa

- ❖ Pomoćna operacija *swap* izgleda ovako:

```
friend void swap (string& first, string& second) {  
    using std::swap;  
    swap(first.str,second.str);  
}
```

- ❖ Ona je deklarisana kao prijateljska funkcija koja nije članica klase *string*, kako bi bila deklarisana u prostoru imena u kom je i klasa *string*, da bi je postupak potrage po argumentu (*argument dependent lookup, ADL*) pronašao za nekvalifikovanu pretragu u tom opsegu (inače ne bi, da je statička funkcija članica); ovo je samo zato da bi ta funkcija bila dostupna i na drugim mestima za slične potrebe, recimo u apstraktnim strukturama podataka koje treba da rade iste stvari (zamenu vrednosti npr. u svojim *swap* funkcijama koje koriste za premeštanje)

- ❖ Treba primetiti da

```
using std::swap;  
swap(first.str,second.str);
```

u opštem slučaju ne znači isto što i:

```
std::swap(first.str,second.str);
```

- ❖ Prva varijanta dozvoljava da postupak ADL za argumente poziva funkcije *swap* nađe i funkciju koja bolje odgovara tipovima argumenata a nije u prostoru imena *std*, gde je najpre i traži, ali da ako takvu ne nađe, drugu traži i u prostoru imena *std*. Druga varijanta zahteva da se takva funkcija strogo traži samo u prostoru imena *std*
- ❖ Za konkretan slučaj, pošto se radi o prostim pokazivačima tipa *char**, rezultat će biti isti, ali u slučaju drugih, npr. korisničkih tipova, rezultat može biti različit

Premeštanje resursa

- ❖ Ostaje još i da se definiše konstruktor premeštanja, koji može da bude pozivan za argumente koji su dvrednosti. On vrši jednostavnu zamenu operacijom *swap*, pri čemu je najpre svoj pokazivač *str* postavio na vrednost *null* podrazumevanom konstrukcijom:

```
inline string::string (string&& other) : string() {  
    swap(*this,other);  
}
```

- ❖ Privremeni objekat kojim se inicijalizuje objekat za koji se konstruktor izvršava neće imati ništa da uništava, jer će u njegov pokazivač *str* ovom zamenom biti upisana *null* vrednost
- ❖ Kako sve operacije *swap* takve da ne bacaju izuzetke (*non-throwing, noexcept*), ovakva implementacija obezbediće jaku garanciju od izuzetaka; osim toga, one su veoma efikasne, jer vrše prostu i brzu razmenu vrednosti

Premeštanje resursa

- ❖ Vratimo se na implementaciju funkcije *substr*:

```
inline string string::substr (size_t pos, size_t count) const {  
    size_t sz = size();  
    if (pos>=sz) throw std::out_of_range;  
    if (pos+count>sz) count = sz - pos;  
    string s;  
    s.allocate(count);  
    s.copy(str+pos, count);  
    return s;  
}
```

- ❖ Ukoliko se ova funkcija pozove za privremeni objekat kao objekat domaćin (**this*), bespotrebno će alocirati prostor za vraćeni objekat tipa *string*, ako već alocirani niz znakova u tom objektu ubrzano nestaje; umesto toga, za traženi podniz može se iskoristiti isti taj prostor koji se može preoteti od privremenog objekta domaćina
- ❖ Kako se radi o nestatičkoj funkciji članici, potrebno je razlikovati funkcije koje će biti pozvane u slučajevima da je objekat domaćin privremeni objekat ili nije, odnosno da je lvrednost ili dvrednost; to se može uraditi navođenjem znakova & (za lvrednost) ili && (za dvrednost) u deklaraciji nestatičke funkcije članice:

```
class string {  
    ...  
    string substr (size_t pos, size_t count) const &;  
    string substr (size_t pos, size_t count) &&;  
    ...  
};  
  
string s1 = "Hello world!";  
string s2 = s1.substr(0,5);  
string s3 = string("Hello world!").substr(0,5) + (s1+s2).substr(0,5);
```

Zadatak: implementirati funkciju *string::substr(...)&*

Zadatak: implementirati istu funkciju ali tako da vraća referencu na dvrednost

Funkcija koja se poziva za lvrednost

Funkcija koja se poziva za dvrednost

Poziva se *string::substr(...)* const&

Poziva se *string::substr(...)* &&

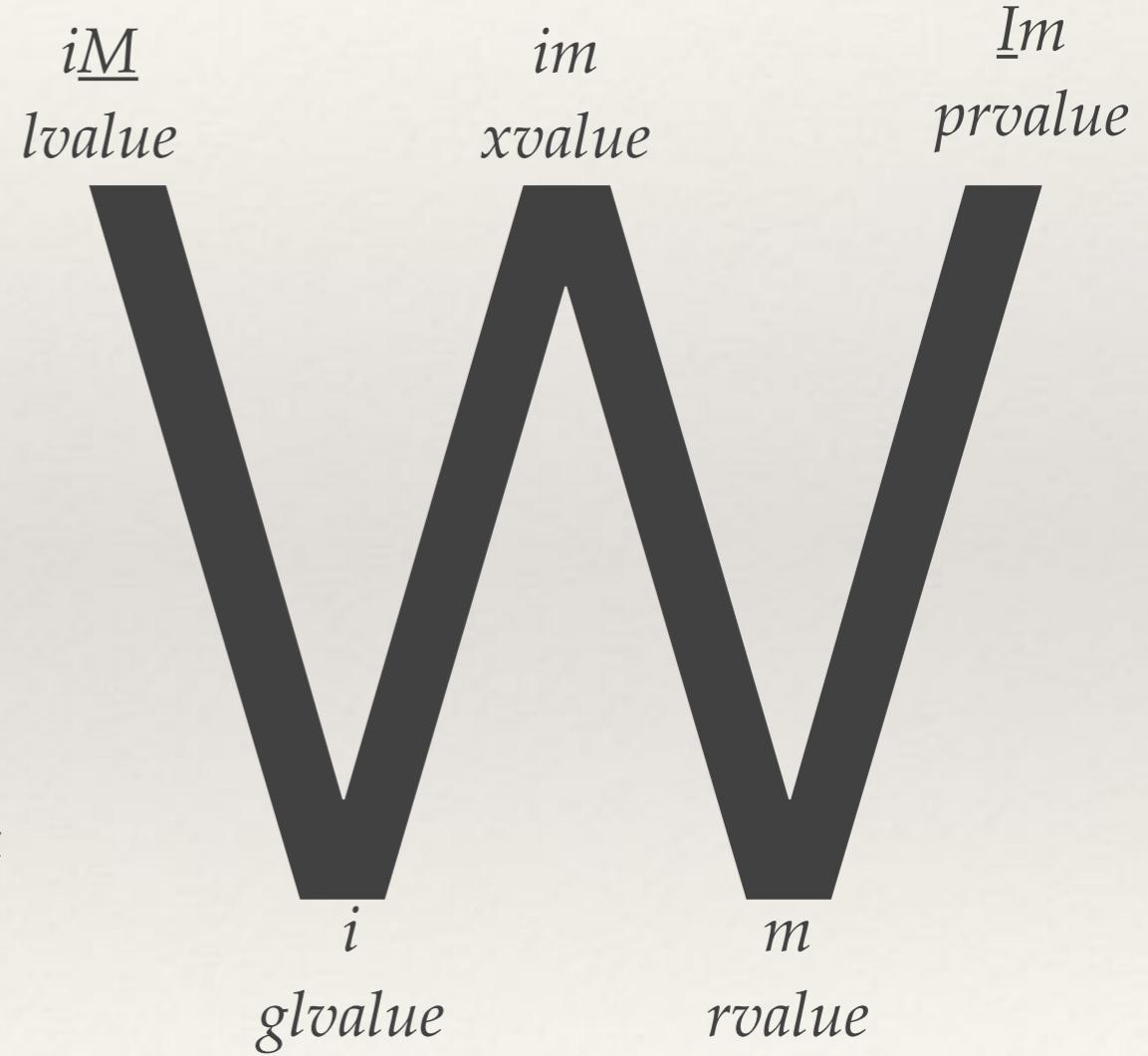
Poziva se *string::substr(...)* &&

Kategorije vrednosti

- ❖ Svaki izraz (rezultat operatora, literal ili ime varijable) ima dva svojstva: *tip (type)* i *kategoriju vrednosti (value category)*
- ❖ Dakle, *vrednost (value)* je izraz i svaki izraz jeste vrednost koja pripada jednoj od kategorija vrednosti
- ❖ Ideja za razvrstavanje na kategorije potiče od sledećih svojstava entiteta kog vrednost predstavlja:
 - entitet se može (ili ne može) *identifikovati (identifiable)*: može se odrediti da li se jedan izraz odnosi na isti entitet kao drugi izraz, recimo poređenjem adresa objekata ili funkcija koje ti izrazi identifikuju
 - entitet se može (ili ne može) *promešta (movable)*: konstruktor premeštanja, operator dodele premeštanjem i druge preklopljene funkcije koje implementiraju premeštanje mogu da se vežu za takav izraz
- ❖ Izrazi mogu biti jedne od sledećih disjunktnih kategorija:
 - ako ima identitet i ne može se premeštati, ima kategoriju *lvrednosti (lvalue, od left value)*
 - ako ima identitet i može se premeštati, ima kategoriju *xvrednosti (xvalue, nema posebne motivacije za naziv, mada se često asocira sa expiring value)*
 - ako nema identitet i može se premeštati, ima kategoriju *čdvrednosti (prvalue, od pure rvalue, "čista dvrednost")*
 - izrazi koji nemaju identitet i ne mogu se premeštati se ne koriste (ne postoje jer nemaju smisla)
- ❖ Izraz koji ima identitet, bez obzira na to da li se može ili ne može premeštati, dakle koji je lvrednost ili xvrednost, naziva se *glvrednost (glvalue, od generalized lvalue)*
- ❖ Izraz koji se može premeštati, bez obzira na to da li ima ili nema identitet, dakle koji je čdvrednost ili xvrednost, naziva se *dvrednost (rvalue, od right lvalue)*

Kategorije vrednosti

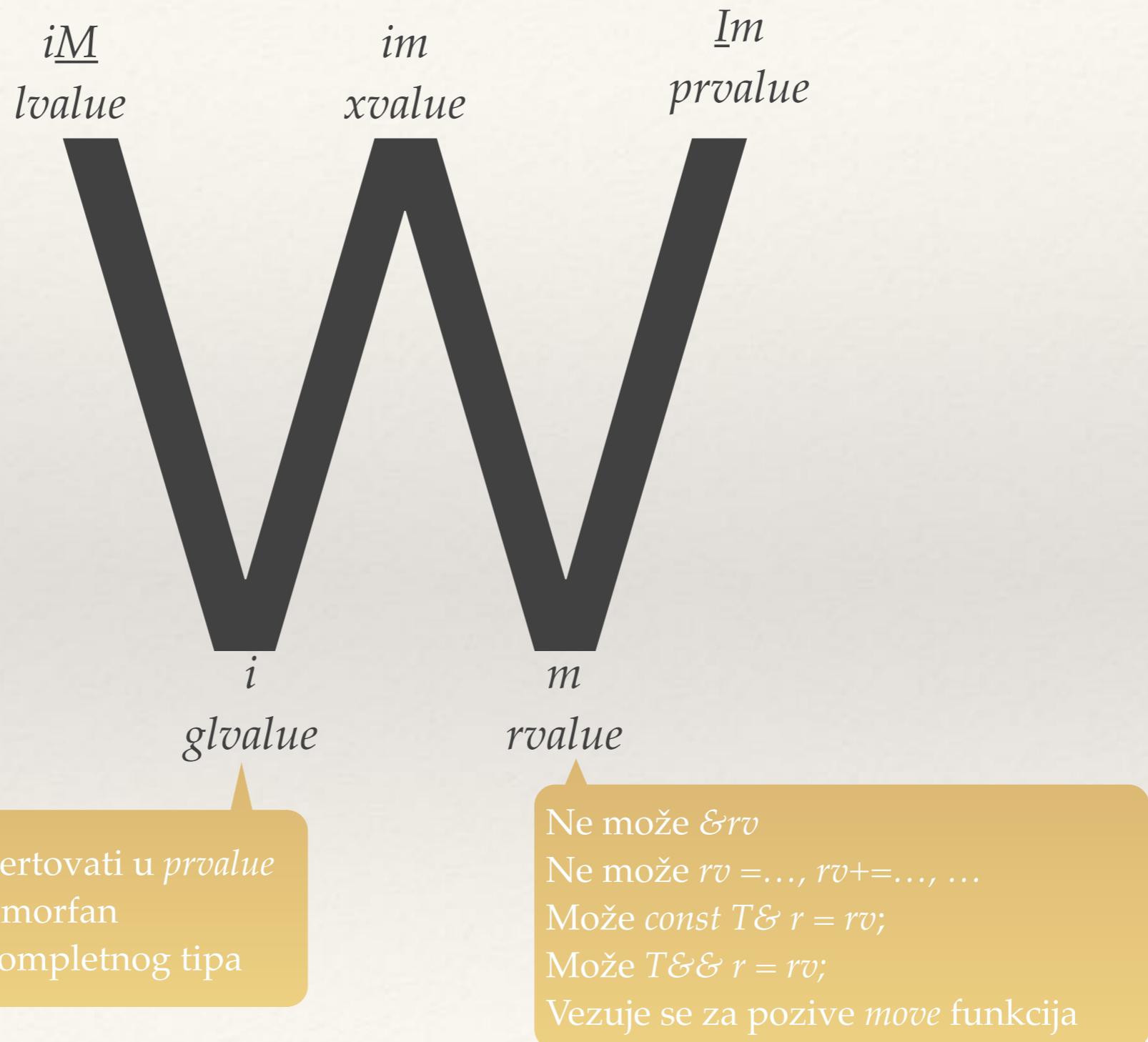
- ❖ Sledeći dijagram (oblika slova W) prikazuje ovu kategorizaciju, pri čemu oznaka i znači *identifiable*, m znači *movable*, a podvučeno veliko slovo znači da dato svojstvo i ili m ne važi
- ❖ Kategorije gore su disjunktne: svaka vrednost pipada jednoj i samo jednoj od te tri kategorije, *lvalue*, *xvalue* ili *prvalue*
- ❖ Kategorije dole su generalizacije (unije) po dve susedne kategorije:
 - $glvalue = lvalue \cup xvalue$
 - $rvalue = xvalue \cup prvalue$
- ❖ Neka vrednost je ili lvrednost (*lvalue*) ili dvrednost (*rvalue*), nikad oba i ništa osim toga
- ❖ Neka vrednost je ili glvrednost (*glvalue*) ili čdvrednost (*prvalue*), nikad oba i ništa osim toga



Kategorije vrednosti

- ❖ Glvrednost (*glvalue*) je lvrednost (*lvalue*) ili xvrednost (*xvalue*) i za svaku glvrednost (a to znači i za lvrednost i za xvrednost) važi:
 - može se implicitno konvertovati u čdvrednost (*prvalue*) konverzijom lvrednosti u dvrednost (*rvalue*), niza u pokazivač i funkcije u pokazivač
 - može biti polimorfna: dinamički tip objekta koji identificuje ne mora biti isti kao i statički tip određen za vreme prevodenja (za objekte na koje ukazuju reference i pokazivači)
 - može imati nekompletan tip tamo gde je to dozvoljeno izrazom
- ❖ Dvrednost (*rvalue*) je čdvrednost (*prvalue*) ili xvrednost (*xvalue*) i za svaku dvrednost (a to znači i za čdvrednost i za xvrednost) važi:
 - ne može se uzimati njena adresa
 - ne može biti levi operand operatora dodele i složene dodele
 - može se koristiti za inicijalizaciju reference na konstantnu lvrednost (*const T&*) ili reference na dvrednost (*T&&*); u slučaju da identificuje privremeni objekat, životni vek tog objekta se produžava do kraja trajanja reference, ali najviše do izlaska iz funkcije u kom je taj privremeni objekat napravljen
 - kada se koristi kao argument poziva funkcije, i ako su na raspolaganju dve preklopljene funkcije, jedna koja prima parametar koji je referenca na dvrednost (*T&&*) a druga koja prima referencu na konstantnu lvrednost (*const T&*), biće pozvana ona koja prima referencu na dvrednost; prema tome, ako postoji i konstruktor kopije i konstruktor pomeranja, odnosno operator dodele kopiranjem i premeštanjem, biće pozvan onaj za premeštanje

Kategorije vrednosti



Kategorije vrednosti

- ❖ Lvrednost (*lvalue*) su, pored još nekih, sledeći izrazi:
 - ime varijable, funkcije ili podatka člana, bez obzira na tip; čak i kad je tip varijable referenca na dvrednost, izraz koji se sastoji od njenog imena je lvrednost
 - poziv funkcije ili operator za preklopljenu operatorsku funkciju, ako je povratni tip referenca na lvrednost
 - $a=b$, $a+=b$, $a\%=b$ i svi drugi ugrađeni operatori složene dodele
 - $++a$ i $--a$, ugrađeni prefiksni operatori inkrementiranja i dekrementiranja
 - $*p$, ugrađeni operator indirekcije
 - $a[n]$ i $p[n]$, ugrađeni operator indeksiranja, pod uslovom da je jedan operand izraza $a[n]$ niz koji je lvrednost
 - $x.m$, član objekta, osim kad je m enumerator član ili nestatička funkcija članica i osim kada je x dvrednost i m je nestatički podatak član tipa koji nije referenca
 - $p->m$, ugrađeni operator člana pokazivanog objekta, osim kad je m enumerator član ili nestatička funkcija članica
 - a,b , ugrađeni operator zarez, ako je b lvrednost
 - $a?b:c$, ugrađeni ternarni uslovni operator pod određenim uslovima (ako je izabrani izraz lvrednost)
 - string literal, npr. "Hello"
 - rezultat konverzije u referencu na lvrednost, npr. `static_cast<int&>(x)`
- ❖ Svojstva lvrednosti:
 - sve kao za glvrednosti
 - može se uzimati adresa $\&l$
 - promenljiva lvrednost može biti levi operand ugrađenih operatora dodele i složene dodele, $lv = \dots$, $lv+=\dots$ itd.
 - njome se može inicijalizovati referenca na lvrednost koja onda upućuje na objekat koji taj izraz identifikuje: $T\& r = lv$;

Kategorije vrednosti

❖ Čdvrednost (*prvalue*) su, pored još nekih, sledeći izrazi:

- enumerator i literal (osim string literal), npr. `23`, `12.34`, `nullptr` ili `true`
- poziv funkcije ili operator za preklopljenu operatorsku funkciju, ako povratni tip nije referenca, npr. `s.substr(1,2)` ili `s1+s2`
- `a++` i `a--`, ugrađeni postfiksni operatori inkrementiranja i dekrementiranja
- `a+b`, `a-b`, `a%b`, `a>>b` i svi drugi ugrađeni aritmetički operatori
- `a&&b`, `a||b`, `!a`, ugrađeni logički operatori
- `a<b`, `a>b`, `a==b`, `a!=b` i svi drugi ugrađeni operatori poređenja
- `&p`, ugrađeni operator uzimanja adrese
- `x.m`, član objekta, ako je `m` enumerator član ili nestatička funkcija članica
- `p->m`, ugrađeni operator člana pokazivanog objekta, ako je `m` enumerator član ili nestatička funkcija članica
- `a,b`, ugrađeni operator zarez, ako je `b` dvrednost
- `a?:b:c`, ugrađeni ternarni uslovni operator pod određenim uslovima (ako je izabrani izraz dvrednost)
- rezultat konverzije u tip koji nije referenca, npr. `static_cast<int>(x)`
- pokazivač `this`

❖ Svojstva čdvrednosti:

- sve kao za dvrednosti
- ne može biti polimorfni: dinamički tip uvek je jednak statičkom tipu određenom za vreme prevođenja
- čdvrednost koja nije klasnog tipa i nije tipa niza ne može biti cv-kvalifikovana
- ne može biti nekompletног tipa, tipa apstraktne klase ili niza takvog tipa

Kategorije vrednosti

- ❖ Xvrednost (*xvalue*) su, pored još nekih, sledeći izrazi:
 - poziv funkcije ili operator za preklopljenu operatorsku funkciju, ako je povratni tip referenca na dvrednost, npr. *std::move(x)*
 - *a[n]*, pod uslovom da je jedan operand niz koji je dvrednost
 - *x.m*, član objekta, ako je *x* dvrednost a *m* nestatički podatak član koji nije tipa reference
 - *a?b:c*, ugrađeni ternarni uslovni operator pod određenim uslovima
 - rezultat konverzije u referencu na dvrednost objektnog tipa, npr. *static_cast<int&&>(x)*
 - bilo koji izraz koji se odnosi na privremeni objekat nakon materijalizacije privremenog objekta (od verzije C++17)
- ❖ Svojstva xvrednosti:
 - sve kao za dvrednosti i za glvrednosti, konkretno:
 - kao i dvrednosti, xvrednosti se vezuju za reference na dvrednosti, pa se za njih pozivaju funkcije premeštanja
 - kao i glvrednosti, mogu biti polimorfne

Kategorije V

Može se konvertovati u *prvalue*
Može biti polimorfna
Može biti nekompletog tipa
Može *const T& r = xv;*
Može *T&& r = xv;*
Vezuje se za pozive *move* funkcija

Može *&lv*
Može *lv =..., lv+=..., ...*
Može *T& r = lv;*
Može se konvertovati u *prvalue*
Može biti polimorfna
Može biti nekompletog tipa



Može se konvertovati u *prvalue*
Može biti polimorfna
Može biti nekompletog tipa

Im
prvalue

Ne može biti polimorfna
Ne može biti nekompletog tipa
Ne može biti tipa apstraktne klase
Ne može biti tipa niza navedenih tipova
Može *const T& r = prv;*
Može *T&& r = prv;*
Vezuje se za pozive *move* funkcija

Ne može *&rv*
Ne može *rv =..., rv+=..., ...*
Može *const T& r = rv;*
Može *T&& r = rv;*
Vezuje se za pozive *move* funkcija

Reference na dvrednosti

- ❖ Za podršku semantici premeštanja u jezik (počev od verzije C++11) uveden je tip *reference na dvrednosti (rvalue reference)*
- ❖ Reference na dvrednosti deklarišu se sa dva znaka & umesto jednim
- ❖ Reference na dvrednosti se u mnogim aspektima ponašaju slično referencama na lvrednosti (npr. inicijalizacija, cv-kvalifikacija, konverzije itd.)
- ❖ Reference na dvrednosti mogu se inicijalizovati slično kao i reference na lvrednosti:

- u deklaraciji sa inicijalizatorom:

```
T&& r = object;
```

- prilikom poziva funkcije koja ima referencu kao parametar:

```
void f(T&& r);
```

```
f(object);
```

- prilikom povratka iz funkcije koja ima referencu kao povratni tip:

```
T&& f() {
```

```
...
```

```
    return object;
```

```
}
```

- kada se inicijalizuje nestatički podatak član koji je tipa reference:

```
C::C (...) : r(object) {...}
```

- ❖ Referenca na dvrednost može se inicijalizovati izrazom koji je dvrednost (*rvalue*) ili koji se u taj tip može implicitno konvertovati

Reference na dvrednosti

- ❖ Ključna razlika i smisao uvođenja referenci na dvrednosti jeste u sledećem: kada argument poziva funkcije dvrednost, i ako su na raspolaganju dve preklopljene funkcije, jedna koja prima parametar koji je referenca na dvrednost ($T\&\&$) a druga koja prima referencu na konstantnu lvrednost ($const\ T\&$), biće pozvana ona koja prima referencu na dvrednost; prema tome, ako postoji i konstruktor kopije i konstruktor pomeranja, odnosno operator dodele kopiranjem i premeštanjem, biće pozvan onaj za premeštanje
- ❖ Na primer:

```
T f (const T&);
```

```
T f (T&&);
```

```
T t;
```

Pošto je t lvrednost, poziva se $T f(const\ T\&)$

```
f(t);
```

Pošto je $f(t)$ dvrednost, poziva se $T f(T\&\&)$

```
f(f(t));
```

Pošto je $std::move(t)$ vraća $T\&\&$, poziva se $T f(T\&\&)$

```
f(std::move(t));
```

```
f (T());
```

Pošto je $T()$ dvrednost (čdvrednost), poziva se $T f(T\&\&)$

Konstruktor premeštanja

- ❖ Konstruktor klase X čiji je prvi parametar tipa $X\&\&$, $const\ X\&\&$, $volatile\ X\&\&$ ili $const\ volatile\ X\&\&$, a koji ili nema druge parametre, ili svi ostali parametri imaju podrazumevane vrednosti, naziva se *konstruktor premeštanja (move constructor)*
- ❖ Ovaj konstruktor poziva se kada se objekat klase X inicijalizuje dvrednošću (do verzije C++17), odnosno xvrednošću (od verzije C++17) istog tipa, uključujući:
 - inicijalizaciju objekta: $X\ x1 = std::move(x2)$ ili $X\ x1(std::move(x2))$, gde je $x2$ objekat klase X ili iz nje izvedene klase
 - prenos argumenta pri pozivu funkcije $f(std::move(x))$, koja ima taj parametar tipa X: $void\ f(X\ x)$
 - povratak vrednosti iz funkcije f koja ima povratni tip X naredbom $return\ x$, gde je x objekat klase X ili iz nje izvedene klase
- ❖ Do verzije C++17, ako je inicijalizator čdvrednost (*prvalue*), poziv ovog konstruktora je bio najčešće izostavljan zbog optimizacije, dok je od verzije C++17 ta optimizacija obavezna, pa se konstruktor premeštanja nikada ne poziva za inicijalizaciju pomoću čdvrednosti (poziva se za inicijalizaciju samo pomoću xvrednosti)
- ❖ Svrha konstruktora premeštanja jeste da preotme (premesti) resurse iz objekta kojim se dati objekat domaćin inicijalizuje, ostavljajući taj objekat u validnom, konzistentnom stanju (barem tako da se on može ispravno uništiti)

Konstruktor premeštanja

- ❖ Klasa može imati i više konstruktora premeštanja, recimo onaj koji prima `X&&` i onaj koji prima `const X&&`
- ❖ Ako klasa `X` nema nijedan eksplisitno deklarisan konstruktor premeštanja (tj. korisnički deklarisan konstruktor premeštanja) i pod uslovom da ta klasa nema korisnički definisan konstruktor kopije, operator dodele kopiranjem, operator dodele premeštanjem i destruktor, prevodilac će implicitno deklarisati konstruktor premeštanja koji je javan, *inline* i nije *explicit*
- ❖ Ako klasa ima neki konstruktor premeštanja (pa prevodilac ne generiše implicitni konstruktor premeštanja), programer ipak može forsirati automatsko deklarisanje konstruktora koji bi prevodilac implicitno deklarisao specifikatorom `=default`
- ❖ Ako podobjekti osnovnih klasa ili objekti članovi ne mogu da se premeštaju ili uništavaju, recimo zato što su objekti klase koje nemaju dostupne konstruktore premeštanja ili destruktore, onda će ovaj implicitno deklarisani konstruktor premeštanja biti obrisan (smatraće se da njegov poziv nije dozvoljen, iako je on deklarisan)
- ❖ U suprotnom, ako ovaj implicitno deklarisani ili podrazumevani konstruktor premeštanja nije obrisan, on će biti definisan i vršiće podrazumevanu inicijalizaciju podobjekata osnovnih klasa i objekata članova xvrednostima, po istom redosledu kao i u inicijalizaciji; ako su ti podobjekti objekti nekih klasa, pozivaju se njihovi konstruktori premeštanja, ako ih imaju, odnosno konstruktori kopije u suprotnom

Operator dodele premeštanjem

- ❖ Operator dodele premeštanjem (*move assignment operator*) klase X je nestatička operatorska funkcija članica te klase sa imenom *operator=* čiji je jedini parametar tipa *X&&, const X&&, volatile X&& ili const volatile X&&*
- ❖ Ova funkcija poziva se kada se objekat klase X nalazi kao levi operand operatora dodele = i kada se baš ona odabere u postupku odabira preklopljene operatorske funkcije za desni operand tog operatora, odnosno kada je desni operand dvrednost ili se može konvertovati u dvrednost
- ❖ I za operator dodele premeštanjem važe slična pravila kao i za konstruktor premeštanja, odnosno za operator dodele kopiranjem: ako klasa nema nijedan eksplicitno deklarisan operator dodele premeštanjem (tj. korisnički deklarisan operator dodele premeštanjem) i pod uslovom da ta klasa nema korisnički definisan konstruktor kopije, konstruktor premeštanja, operator dodele kopiranjem i destruktorn, prevodilac će implicitno deklarisati konstruktor premeštanja koji je javan, *inline* i nije *explicit*
- ❖ Ovaj implicitno deklarisani operator dodele premeštanja može se smatrati i obrisanim pod odgovarajućim uslovima, a ako nije obrisan ili je podrazumevan, on vrši dodelu premeštanjem podobjekata osnovnih klasa i objekata članova, odnosno dodelu kopiranjem ako premeštanje nije definisano

Semantika vrednosti od C++17

- ❖ Do verzije C++17, optimizacija izostavljanja kopiranja bila je neobavezna i važilo je *as-if* pravilo: čak i kada se kopiranje izbegava, odgovarajući konstruktor kopije ili premeštanja morao je da bude definisan za navedene situacije inicijalizacije privremenim objektima, čak i kada je sigurno da se ti konstruktori nikada ne pozivaju (jer prevodilac uvek vrši ovu optimizaciju i programer to može da zna)
- ❖ Zato su morali da se prave prestupi i definišu ovakvi konstruktori i za klase koje prirodno ne treba da budu takve da se njihovi objekti mogu kopirati ili premeštati, ili su pravljena drugačija zaobilazna rešenja, ako se želi prenos po vrednosti
- ❖ Počev od verzije C++17, optimizacija izbegavanja kopiranja (*copy elision*) obavezna je (a ne samo opcionala) na sledećim mestima:
 - u naredbi *return*, kada je operand ove naredbe čdvrednost (*prvalue*) istog klasnog tipa (ignorišući cv-kvalifikacije) kao i povratni tip funkcije (RVO):

```
X f () {  
    return X();  
}  
f();
```

Samo jedan poziv podrazumevanog konstruktora X()

- pri inicijalizaciji varijable, kada je inicijalizator čdvrednost (*prvalue*) istog klasnog tipa (ignorišući cv-kvalifikacije) kao i varijabla:

```
X x = X(X(f()));
```

Samo jedan poziv podrazumevanog konstruktora X() koji inicijalizuje x

- ❖ Odgovarajući konstruktori kopije i premeštanja čak više ne moraju ni da postoje niti da budu dostupni, jer semantika jezika garantuje da se oni ne pozivaju na ovim mestima
- ❖ Optimizacija NRVO je i dalje opcionala: ako se ona ne vrši, poziva se odgovarajući konstruktor premeštanja ili kopije prilikom inicijalizacije povratne vrednosti ako je operand naredbe *return* imenovana automatska varijabla (ali ne parametar)

Semantika vrednosti od C++17

- ❖ Zapravo je formalna semantika povratnih vrednosti poziva funkcija (uključujući i rezultate operatorskih funkcija koje se pozivaju za preklopljene operatore) značajno promenjena od verzije C++17: ako funkcija vraća rezultat koji nije referenca, taj rezultat *nije privremeni objekat*, kao što je to ranije bilo, nego samo *vrednost (value)*, i to *čdvrednost (prvalue)*
- ❖ Čdvrednost predstavlja “potrebu za inicijalizacijom”, ali ne obavezno privremenog objekta (koja se opcionalno može i preskočiti): ta inicijalizacija obaviće se onako kako je definisano operandom naredbe *return* koja vraća tu vrednost, a obaviće se u zavisnosti od konteksta u kom se koristi
- ❖ Iz funkcija se tako, formalno, vraćaju samo vrednosti i one se dalje prosleđuju kao argumenti/operandi ili inicijalizatori dalje u izrazima ili inicijalizacijama
- ❖ Privremeni objekat se pravi samo u određenim situacijama, odnosno u zavisnosti od konteksta, kada se ta vrednost *materijalizuje* u privremeni objekat (*temporary materialization*)
- ❖ Materijalizacija privremenog objekta se maksimalno odlaže do trenutka kada se taj objekat mora napraviti, a sve do tada se samo koristi vrednost

Semantika vrednosti od C++17

- ❖ To se u mnogim slučajevima svodi na obavezu izbegavanja kopiranja. Na primer, sledeća inicijalizacija formalno se tumači ovako:

```
X f () { return X(1); }
X x = X(X(f()));
```

- Funkcija f u naredbi *return* vraća vrednost koja je čdvrednost i koja “predstavlja” inicijalizaciju nekog objekta (ali tek kada se on odredi kontekstom i ne obavezno privremenog) pozivom konstruktora $X(1)$
 - Ova čdvrednost prosleđuje se kao “zahtev” za inicijalizaciju ponovo čdvrednosti $X(f())$, dakle čdvrednost koju vraća $X(f())$, predstavlja istu prethodno navedenu inicijalizaciju; slično važi za još jednu čdvrednost $X(X(f()))$
 - najzad, tom čdvrednošću $X(X(f()))$ inicijalizuje se objekat x , tako da se navedena inicijalizacija pozivom konstruktora $X(1)$ konačno “vezuje” za objekat x
- ❖ Implementaciono, sve se opet svodi na isti mehanizam sprovođenja izostavljanja kopiranja RVO tako što pozvana funkcija, ukoliko vraća čdvrednost, dobija adresu mesta (objekta) koji treba da inicijalizuje u naredbi *return*, i ta adresa prosleđuje se tranzitivno
 - ❖ Međutim, formalna semantika jezika je promenjena i sada konstruktori kopije i premeštanja ne moraju uopšte da budu definisani da bi ovo bilo moguće:

```
struct X {
    X (int);
    X (const X&) = delete;
    X (X&&) = delete;
};

X f () { return X(1); }
X x = f();
```

Do C++17 ovo ne bi bilo moguće. U C++17 je sasvim ispravno i radi $X x(1)$

Semantika vrednosti od C++17

- ❖ Situacije kada se vrši materijalizacija privremenog objekta zapravo predstavljaju slučajeve u kojima se mora obezbediti mesto za objekat koji će se na kraju ovakvog lanca inicijalizovati onim što predstavlja čdvrednost i čija će adresa biti prosleđivana svim funkcijama koje se pozivaju, a vraćaju čdvrednost. Pored još nekih, to su sledeće situacije:
 - kada se referenca vezuje za čdvrednost:

```
const X rl& = X(1);  
X rr&& = X(1);
```

- kada se pristupa članu čdvrednosti tipa klase:

```
int i = X(1).i;
```

Pravi se privredni objekat koji se inicijalizuje pomoću X(1) i pristupa se njegovom članu *i*

- kada se niz koji je čdvrednost (npr. kao objekat član izraza koji je čdvrednost) konvertuje u pokazivač ili kada se pristupa njegovom elementu
- kada je operand operatora *sizeof* čdvrednost
- kada je čdvrednost izraz čiji se rezultat odbacuje (levi operand operatora zarez i izraz kao naredba):

```
X f() { return X(1); }  
f();
```

Funkcija *f* vraća čdvrednost, a poziv je sam u naredbi, pa mu se rezultat odbacuje, ali mora da se materijalizuje kao privredni objekat koji se inicijalizuje sa X(1)

Glava 14: Inicijalizacija

- ❖ Inicijalizacija
- ❖ Konstantna inicijalizacija
- ❖ Inicijalizacija nulom
- ❖ Podrazumevana inicijalizacija
- ❖ Inicijalizacija vrednošću
- ❖ Inicijalizacija kopiranjem
- ❖ Direktna inicijalizacija
- ❖ Inicijalizacija listom
- ❖ Agregatna inicijalizacija
- ❖ Inicijalizacija referenci



Inicijalizacija

- ❖ Sve nelokalne varijable (objekti i reference) sa statičkim trajanjem skladišta, a to obuhvata statičke podatke članove i varijable definisane u oblasti važenja prostora imena, inicijalizuju se pri pokretanju programa, pre poziva funkcije *main* (mada to nije garantovano i nije uvek slučaj, kao što je ranije objašnjeno)
- ❖ Sve varijable sa životnim vekom vezanim za nit (*thread_local*) inicijalizuju se pri pokretanju niti, pre izvršavanja funkcije niti
- ❖ Obe ove kategorije varijabli inicijalizuju se u dve faze, sledećim redom:
 - Statička inicijalizacija (*static initialization*)
 - Dinamička inicijalizacija (*dynamic initialization*)
- ❖ Statička inicijalizacija vrši se za vreme prevođenja
- ❖ Dinamička inicijalizacija se konceptualno vrši nakon statičke inicijalizacije, a praktično za vreme izvršavanja (pokretanja) programa, mada je u nekim slučajevima moguće, a to je prevodiocu i dozvoljeno, da i nju izvrši za vreme prevođenja (tzv. rana dinamička inicijalizacija, *early dynamic initialization*)

Inicijalizacija

- ❖ Statička inicijalizacija radi sledeće:
 - Ukoliko je dozvoljeno, vrši se *konstantna inicijalizacija* (*constant initialization*); u praksi, ova inicijalizacija obavlja se za vreme prevodenja tako što prevodilac izračunava konstantne izraze koji su inicijalizatori, a izračunate vrednosti upisuju se u prostor alociran za varijable u prevedenim fajlovima; na primer:

```
const int n = 5;
int x = 2*n+1, *p = &x;
size_t s = sizeof(X);
```

 - U onim slučajevima u kojima se ne vrši konstantna inicijalizacija, vrši se *inicijalizacija nulom* (*zero initialization*); u praksi, za ove varijable ne odvaja se prostor u prevedenim fajlovima, već program u izvršavanju koristi uslugu operativnog sistema koja alocira segment memorije inicijalizovan nulama; na primer:

```
int n, *p;
```

 - ❖ Dinamička inicijalizacija vrši se po redosledu koji određuju specifična pravila jezika; za nelokalne varijable koje ne spadaju u neke posebne kategorije (npr. statički podaci članovi šablonskih klasa ili *inline* varijable pod određenim uslovima), redosled je određen redosledom definisanja u jednoj jedinici prevodenja
 - ❖ Ako dinamička inicijalizacija ne menja vrednost nijednog objekta iz prostora imena pre njegove inicijalizacije, i ako bi statička inicijalizacija proizvela isti rezultat kao i dinamička, prevodiocu je dozvoljeno da dinamičku inicijalizaciju obavi kao statičku, zapravo za vreme prevodenja (*rana dinamička inicijalizacija*, *early dynamic initialization*)
 - ❖ Kao što je već objašnjeno, dinamička inicijalizacija ne mora biti završena pre početka izvršavanja funkcije *main*
 - ❖ Ako dinamička inicijalizacija baci izuzetak, poziva se funkcija *std::terminate*

Inicijalizacija

- ❖ Inicijalizacija lokalne statičke varijable vrši se kada bilo koje izvršavanje (u bilo kojoj niti) prvi put nađe na njenu definiciju; svaki sledeći nailazak preskače inicijalizaciju
- ❖ Ako ova inicijalizacija baci izuzetak, statička varijabla se ne smatra inicijalizovanom i inicijalizacija će biti pokušana pri sledećem nailasku izvršavanja na istu definiciju
- ❖ Nestatički podatak član može biti inicijalizovan na dva načina:
 - u listi inicijalizatora članova (*member initializer list*) u konstruktoru klase:

```
X::X (int j) : i(j+1) {}
```

Lista inicijalizatora članova

- podrazumevanim inicijalizatorom člana (*default member initializer*) u definiciji klase, koji se koristi ako se član izostavi iz liste inicijalizatora članova u konstruktoru; ako se član sa podrazumevanim inicijalizatorom pojavi i u listi inicijalizatora članova, ta podrazumevana inicijalizacija se ignoriše:

```
struct X {  
    int i = 1;  
    string s{'H', 'e', 'l', 'l', 'o'};  
    X () {}  
    X (int j) : i(j+1) {}  
};
```

Podrazumevani inicijalizatori

Članovi *i* i *s* se inicijalizuju podrazumevanim inicijalizatorima

Član *s* se inicijalizuje podrazumevanim inicijalizatorom, a *i* izrazom *j+1*

Konstantna inicijalizacija

- ❖ Konstantna inicijalizacija se vrši za varijable sa statičkim trajanjem skladištenja i onima vezanim za niti (*thread local*), samo pod uslovom da su inicijalizovani konstantnim izrazom
- ❖ *Konstantan izraz (constant expression)* je izraz koji se može izračunati za vreme prevođenja i može se koristiti gde god je potreban konstantan izraz (npr. za dimenzije nizova ili inicijalizaciju konstanti)
- ❖ Ovakvi su izrazi koji uključuju npr. samo literale, konstante primitivnih tipova i operatore koji se mogu izračunati za vreme prevođenja
- ❖ Međutim, od verzije C++11, u konstantnim izrazima mogu učestvovati i varijable, uključujući i objekte klase, pa čak i pozivi korisničkih funkcija, uključujući i pozive funkcija članica klase, pod uslovom da su te funkcije deklarisane kao *constexpr*, i da se te funkcije, uključujući i konstruktore, mogu izvršiti za vreme prevođenja, što znači da su im argumenti konstantni izrazi, da su inicijalizatori svih podobjekata konstantni izrazi, da su svi podaci članovi inicijalizovani i slično
- ❖ Ovakve klase i funkcije moraju zadovoljiti niz definisanih uslova, pri čemu su se ti uslovi menjali kroz verzije jezika, i dalje se menjaju u novijim verzijama jezika u pravcu relaksacije ograničenja; na primer, od verzije C++20, ove funkcije mogu da budu čak i virtuelne
- ❖ Jedno od osnovnih ograničenja jeste to da se te funkcije ne mogu oslanjati na pozive funkcija koje nisu označene kao *constexpr* ili vrednosti varijabli koje nisu označene kao *constexpr* ili podataka članova koji nemaju inicijalizatore
- ❖ Funkcije koje su označene kao *constexpr*, uključujući i konstruktore, mogu se pozivati i van konstantnih izraza, odnosno za vreme izvršavanja
- ❖ Objekti deklarisani kao *constexpr* su implicitno konstantni i moraju biti inicijalizovani konstantnim izrazima; funkcije koje su označene kao *constexpr* su implicitno *inline*

Konstantna inicijalizacija

- ❖ Ideja ove tehnike je zapravo to da se čitav deo programa, uključujući definicije klasa, kreiranje njihovih objekata i pozive funkcija, a ne samo izračunavanje prostih izraza, izvršava za vreme prevodenja, kako bi se proizveo odgovarajući željeni rezultat
- ❖ Ova tehnika može se koristiti na primer za:
 - prekonfigurisanje, odnosno inicijalizaciju složenijih struktura podataka ili objekata i njihovih veza koja se može izvršiti za vreme prevodenja, a koje se onda koriste u izvršavanju programa
 - složenija izračunavanja parametara programa

koja se mogu izvršiti za vreme prevodenja, kako se ne bi trošilo vreme i zauzimao memorijski prostor prilikom svakog izvršavanja programa, tipično prilikom pokretanja programa

- ❖ Ovo može da bude značajno na primer za sistemski, ugrađeni (*embedded*) softver, posebno za onaj za koji se zahteva mali memorijski otisak (*memory footprint*), brzo pokretanje i izvršavanje, ili što manja potrošnja energije pri izvršavanju

Konstantna inicijalizacija

- ❖ Na primer, *enumeratorske klase*, tj. klase sa konstantnim skupom instanci predstavljaju uopštenje pojma enumeracije, jer su tipovi čiji skupovi instanci ne mogu da se menjaju tokom izvršavanja programa, ali su, za razliku od enumeracija čije su instance proste, skalarne simboličke vrednosti, instance ovakvih klasa objekti sa svim svojim svojstvima
- ❖ Na primer, želimo da napravimo predefinisan, prekonfigurisan skup korisničkih uloga u programu koje se inicijalizuju statički, u toku prevođenja, pri čemu su te uloge predstavljene objektima koji imaju odgovarajuće usluge (npr. proveru da li korisnik sa tom ulogom može da izvrši datu komandu):

```
class UserRole {  
public:  
    constexpr UserRole (const char* name, ...);  
    const char* name = nullptr;  
  
    bool isAuthorizedFor (Command* cmd) const;  
    ...  
};  
  
constexpr UserRole ordinary("Ordinary user", ...);  
constexpr UserRole privileged("Privileged user", ...);  
constexpr UserRole admin("Administrator", ...);
```

Konstruktor je *constexpr*, pa se može koristiti za konstantnu inicijalizaciju statičkih objekata u toku prevođenja, pod uslovom da se poziva sa argumentima koji su konstantni izrazi

Ovi objekti se inicijalizuju statički, konstantnom inicijalizacijom i konstantni su

Konstantna inicijalizacija

- ❖ Primer: struktura za alokaciju prostora za objekte tipa T , bez fragmentacije (ostataka slobodnog prostora):

```
template <class T, int size>
class Storage {
public:
    constexpr Storage () : head(slots) { slots[size-1].next = nullptr; }

    void* alloc () { Slot* p=head; if (p) head=p->next; return p->slot; }
    void free (void* addr) { head = new (addr) Slot(head); }
```

private:

```
struct Slot {
    constexpr Slot () : next(this+1) {}
    Slot (Slot* nxt) : next(nxt) {}

    union {
        Slot* next;
        char slot[sizeof(T)];
    };
};

Slot* head;
Slot slots[size];
};
```

```
Storage<Clock,50> clockStorage;
```

Konstruktor je *constexpr*, pa se može koristiti za konstantnu inicijalizaciju statičkih objekata u toku prevodenja

Članovi moraju imati inicijalizatore koji su konstantni izrazi

Konstruktor strukture *Slot* je *constexpr*, pa se može pozvati iz konstruktora *Storage* koji je *constexpr*

Ovaj statički objekat bi mogao da se inicijalizuje konstantnom inicijalizacijom u toku prevodenja, ali nažalost, izgleda da još uvek nema načina da se ovo garantuje na prenosiv način u opštem slučaju, pa će inicijalizacija ipak po pravilu biti dinamička. Ako bi se ovaj objekat deklarisao kao *constexpr*, njegova inicijalizacija bi bila konstantna, ali bi i on bio konstantan i ne bi mogao da se menja

Zadatak:

Funkcija *Store::free* nije pouzdana ako joj se dostavi pokazivač koji ne ukazuje na neki od elemenata niza *slots*. Napraviti je tako da bude otporna na ovakve greške.

Inicijalizacija nulom

- ❖ *Inicijalizacija nulom (zero initialization)* se obavlja u sledećim slučajevima:
 - Za svaku imenovanu varijablu sa statičkim trajanjem skladišta ili trajanjem skladišta vezanim za nit (*thread local*), pod uslovom da nije inicijalizovana konstantnom inicijalizacijom, a pre svake druge inicijalizacije:
static T t;
 - Kao deo postupka inicijalizacije vrednošću, za neklasne tipove i za članove klasnih tipova koji su inicijalizovani vrednošću i koji nemaju konstruktore, uključujući inicijalizaciju vrednošću aggregata za koje nisu zadati inicijalizatori:
T();
T t = {};
T{};
 - Kada se niz znakova inicijalizuje string literalom koji je kraći od tog niza, ostatak niza se inicijalizuje nulama:
char a[n] = " ";

Inicijalizacija nulom

❖ Inicijalizacija nulom radi sledeće:

- Ako je T skalarni tip, inicijalna vrednost objekta je celobrojna konstanta nula eksplisitno konvertovana u tip T ; za pokazivače, to je uvek *null* vrednost, čak i ako se ona ne predstavlja binarnom vrednošću nula
- Ako je T klasni tip, svi podobjekti osnovne klase i članovi se inicijalizuju nulama, a konstruktori se ignorišu
- Ako je T niz, elementi se inicijalizuju nulama
- Za reference se ništa ne radi

```
static T t;
```

❖ Na primer:

```
struct X {  
    int m;  
};  
  
int i;  
int* p;  
  
int main () {  
    X x{};  
  
    int a[2]{1};  
    int j{};  
  
    delete p;  
    cout<<x.m<<i<<a[1]<<j;  
}
```

Kao statički objekti, inicijalizuju se nulama

Inicijalizuje se nulama kao deo inicijalizacije vrednošću

$a[1]$ se inicijalizuje nulom kao deo inicijalizacije vrednošću, jer nije zadat inicijalizator za taj element

Inicijalizuje se nulama kao deo inicijalizacije vrednošću, jer je *int* neklasni tip

Bezbedno je raditi *delete* za pokazivač koji ima *null* vrednost

Podrazumevana inicijalizacija

❖ Podrazumevana inicijalizacija (*default initialization*) se obavlja u sledećim slučajevima:

- Kada se varijabla sa automatskim, statičkim ili trajanjem skladišta vezanim za nit deklariše bez inicijalizatora:

`T t;`

- Kada se pravi dinamički objekat izrazom *new* bez navedenog inicijalizatora:

`new T`

- Kada se osnovna klasa ili nestatički podatak član ne navede u listi inicijalizatora članova u konstruktoru klase, a taj konstruktor se poziva:

```
struct B {...};  
  
struct D : B {  
    B b;  
    D () {}  
}  
  
D d;
```

❖ Podrazumevana inicijalizacija radi sledeće:

- Ako je *T* klasni tip, poziva se podrazumevani konstruktor bez argumenata
- Ako je *T* niz, svi elementi se inicijalizuju podrazumevanom inicijalizacijom
- Inače, ne radi se ništa; zbog toga automatski i dinamički objekti (i njihovi podobjekti) neklasnih tipova imaju neodređene vrednosti (šta god se zateklo u memoriji)

Podrazumevana inicijalizacija

- ❖ Prema tome, automatski i dinamički objekti neklasnih tipova imaju nedefinisane podrazumevane vrednosti; statički i oni vezani za nit se inicijalizuju nulom. Reference se ne mogu podrazumevano inicijalizovati
- ❖ Na primer:

```
struct X {  
    int m;  
    X () {}  
};  
  
int i;  
  
int main () {  
    int j;  
    X x;
```

m nije pomenut u listi inicijalizatora članova, pa se podrazumevano inicijalizuje (nema akcije jer je tipa *int*)

Inicijalizuje se najpre nulama, a onda podrazumevano (ništa), pa ostaje nula

Podrazumevana inicijalizacija bez efekta, ima neodređenu vrednost

Podrazumevana inicijalizacija, poziva se podrazumevani konstruktor,
x.m ima neodređenu vrednost

- ❖ Motivacija za ovakvo ponašanje je efikasnost: u nekim situacijama nije neophodna nikakva određena inicijalna vrednost, pa nema potrebe gubiti vreme na inicijalizaciju (cilj je da prevedeni kod bude jednako efikasan kao da je pisani):

```
int m;  
cin>>m;
```

Inicijalizacija vrednošću

- ❖ Kao što je pokazano, podrazumevana inicijalizacija u nekim situacijama poziva podrazumevane konstruktore, dok za automatske i dinamičke objekte neklasnih tipova ne radi nikakvu inicijalizaciju, ostavljajući ih sa nedefinisanim vrednostima. U mnogim situacijama potrebno je da takvi objekti budu definisani određenom vrednošću, odnosno inicijalizovani nulom
- ❖ U verziji jezika C++03, u kontekstima gde je to bilo moguće, ovo se postizalo inicijalizatorom sa praznim zagradama:

```
C::C () : t() {}  
T* p = new T();
```

- ❖ Međutim, u mnogim kontekstima to nije moguće, jer takva notacija znači deklaraciju funkcije bez parametara koja vraća tip T , a ne objekta tipa T :

```
T t();
```

- ❖ U takvim situacijama morala se koristiti drugačija inicijalizacija, ali ona ima i drugu semantiku (inicijalizacije kopiranjem ili agregatne inicijalizacije):

```
T t = T();
```

Inicijalizacija kopiranjem, uz moguće izostavljanje kopiranja

```
T t = {};
```

- ❖ Ili nešto drugo, u zavisnosti od tipa, ali problem ostaje kada je tip nepoznat, kao što je slučaj sa šablonima:

```
int i = 0;
```

Inicijalizacija nulom

```
Clock clk;
```

Podrazumevana inicijalizacija

```
int a[10]{}
```

Agregatna inicijalizacija

Inicijalizacija vrednošću

- ❖ Zato je počev od verzije C++11 u jezik uvedena *inicijalizacija vrednošću* (*value initialization*), čija je ideja da vrši podrazumevanu inicijalizaciju za objekte klase, a inicijalizaciju nulom za objekte neklasnih tipova, i to u svim kontekstima na uniforman način:

- kada se inicijalizuje bezimeni privremeni objekat sa praznim zagradama (običnim ili velikim):

`T()`

`T{}`

- kada se inicijalizuje dinamički objekat u izrazu *new* sa praznim zagradama (običnim ili velikim):

`new T()`

`new T{}`

- kada se inicijalizuje nestatički podatak član ili osnovna klasa u inicijalizatoru članova u konstruktoru klase sa praznim zagradama (običnim ili velikim):

`X::X (...) : T(), t() {...}`

`X::X (...) : T{}, t{} {...}`

- kada se inicijalizuje imenovana varijabla sa praznim velikim zagradama:

`T t{};`

Inicijalizacija vrednošću

❖ Pritom:

- u svim navedenim slučajevima kada se koriste velike zagrade {}, ako je T agregatni tip (niz ili klasa pod nekim uslovima, tipično struktura), vrši se agregatna inicijalizacija umesto inicijalizacije vrednošću
- ako je T klasa koja nema podrazumevani konstruktor, ali ima konstruktor koji prihvata argument tipa $std::initializer_list$, vrši se inicijalizacija listom umesto inicijalizacije vrednošću

❖ Inicijalizacija vrednošću vrši sledeće:

- Ako je T klasa bez podrazumevanog konstruktora, ili sa eksplicitno korisnički definisanim ili obrisanim podrazumevanim konstruktorom, objekat se inicijalizuje podrazumevanom inicijalizacijom (koja nije dozvoljena ako podrazumevani konstruktor ne postoji ili je obrisan)
- Ako je T klasa sa podrazumevanim konstruktorom koji nije ni eksplicitno korisnički definisan niti obrisan, objekat se najpre inicijalizuje nulom, a potom podrazumevanom inicijalizacijom ako ima netrivialan podrazumevani konstruktor
- Ako je T niz, svaki element se inicijalizuje vrednošću
- Inače, objekat se inicijalizuje nulom

❖ Uprošćeno, ako je korisnik sam napravio podrazumevani konstruktor, onda ova inicijalizacija poziva taj konstruktor i radi podrazumevanu inicijalizaciju definisanu u njemu; u suprotnom, inicijalizuje objekat nulom, a onda poziva podrazumevane konstruktore podobjekata. Na primer:

```
struct X {  
    int i;  
    X () {}  
};
```

Eksplicitno definisan korisnički konstruktor, ne inicijalizuje i

```
struct Y {  
    int j;  
    X x;  
};
```

Implicitno definisan podrazumevani konstruktor

```
cout<<X().i<<endl<<Y().j<<endl<<Y().x.i;
```

U privremenom objektu $X()$, i će imati neodređenu vrednost.
U privremenom objektu $Y()$, j i $x.i$ će imati vrednost 0.

Inicijalizacija kopiranjem

- ❖ *Inizijalizacija kopiranjem (copy initialization)* se obavlja kada se objekat (ne referenca) tipa T (koji je objektni tip) inicijalizuje u sledećim slučajevima:

- Kada se imenovani objekat (automatski, statički ili vezan za nit) inicijalizuje izrazom iza znaka `=`:

```
T t = expression;
```

- Kada se argument prenosi u pozvanu funkciju po vrednosti (ne referenci):

```
void f (T t);
```

```
f(expression);
```

- Kada se vraća iz funkcije koja vraća vrednost (ne referencu):

```
T f () {
```

```
...
```

```
return expression;
```

```
}
```

- Kada se baca ili hvata izuzetak po vrednosti (ne referenci):

```
throw expression;
```

```
catch (T t) {...}
```

- Kao deo agregatne inicijalizacije, za inicijalizaciju svakog elementa za koji je zadat inicijalizator:

```
T a[N] = {expression, expression, ...};
```

Inicijalizacija kopiranjem

❖ Inizijalizacija kopiranjem radi sledeće:

- Ako je T klasa, a izraz kojim se inicijalizuje jeste čvrednost (*prvalue*) koja je istog tipa T (bez cv-kvalifikacije), onda se izostavlja kopiranje i objekat inicijalizuje onim što ta čvrednost predstavlja, kao što je ranije objašnjeno (pre verzije C++17, pravio se privremeni objekat kao rezultat izraza koji se premeštao ili kopirao u objekat koji se inicijalizuje, uz moguće izostavljanje kopiranja):

```
X x = X();
```

```
extern X f(...);  
X x = f(...);
```

- Ako je T klasa, a tip izraza kojim se objekat inicijalizuje (bez cv-kvalifikacije) klasa izvedena iz te klase, pretražuju se konstruktori klase T koji nisu *explicit* i koji mogu da prihvate tip izraza kao argument i bira se onaj konstruktor koji najbolje odgovara, po pravilima rezolucije za preklapanje funkcija; objekat se onda inicijalizuje tim konstruktorom:

```
Derived d;
```

```
Base b = d;
```

- Ako je T klasa, a tip izraza kojim se objekat inicijalizuje (bez cv-kvalifikacije) nije klasa izvedena iz te klase, ili T nije klasa, a tip izraza jeste klasa, traže se korisnički definisane konverzije koje mogu da inicijalizuju objekat tipa T iz tipa izraza; ako je odabrana korisnička konverzija zadata konverzionim konstruktorom, objekat se inicijalizuje tim konstruktorom direktno (bez pravljenja privremenog objekta, kao ranije):

```
X x = y;
```

- U preostalim slučajevima, kada ni T ni tip izraza nisu klase, vrednost izraza se prosto kopira u objekat, ako je izraz istog tipa T , ili se po potrebi koriste standardne konverzije da konvertuju vrednost izraza u tip objekta:

```
Base* pb = new Derived;
```

Direktna inicijalizacija

- ❖ *Direktna inicijalizacija (direct initialization)* inicijalizuje objekat eksplisitnim skupom argumenata konstruktora u sledećim situacijama:

- Inicijalizacija imenovanog objekta nepraznom listom izraza unutar zagrada:

`T t (expression, ...)`

- Inicijalizacija objekta neklasnog tipa jednim izrazom unutar velikih zagrada:

`T t {expression}`

- Inicijalizacija privremenog objekta kao čvrednosti nepraznom listom izraza unutar zagrada:

`T (expression, ...)`

- Inicijalizacija privremenog objekta kao čvrednosti izraza *static_cast*:

`static_cast<T>(expression)`

- Inicijalizacija dinamičkog objekta u izrazu *new* pomoću inicijalizatora sa nepraznim zagradama:

`new T (expression, ...)`

- Inicijalizacija podobjekta osnovne klase ili podobjekta člana unutar liste inicijalizatora u konstruktoru za nepraznom listom unutar zagrada:

`C::C (...), B (expression, ...), m (expression,...) {...}`

Direktna inicijalizacija

- ❖ Direktna inicijalizacija radi sledeće:

- Ako je T klasa, a jedini izraz kojim se inicijalizuje jeste čvrednost (*prvalue*) koja je istog tipa T (bez cv-kvalifikacije), onda se izostavlja kopiranje i objekat inicijalizuje onim što ta čvrednost predstavlja, kao što je ranije objašnjeno (pre verzije C++17, pravio se privremeni objekat kao rezultat izraza koji se premeštao ili kopirao u parametar konstruktora kojim se objekat inicijalizuje, uz moguće izostavljanje kopiranja):

`X x1 (x2);`

- Ako je T klasa, a tip izraza kojim se objekat inicijalizuje (bez cv-kvalifikacije) nije jedini ili nije istog tipa, pretražuju se konstruktori klase T koji mogu da prihvate te argumente i bira se onaj konstruktor koji najbolje odgovara, po pravilima rezolucije za preklapanje funkcija; objekat se onda inicijalizuje tim konstruktorom
- Ako T nije klasa, a tip izraza kojim se objekat inicijalizuje (bez cv-kvalifikacije) jeste klasa, traže se korisnički definisane konverzionate operatorske funkcije klase izraza ili njene osnovne klase, ako ih ima, i bira se ona koja najbolje odgovara, i objekat tipa T se onda inicijalizuje tom konverzijom
- Inače, ako je T tip *bool*, a izvorni tip `std::nullptr_t` (tip konstante *nullptr*), vrednost se inicijalizuje na *false*
- U preostalim slučajevima, kada ni T ni tip izraza nisu klase, vrednost izraza se prosto kopira u objekat, ako je izraz istog tipa T , ili se po potrebi koriste standardne konverzije da konvertuju vrednost izraza u tip objekta:

`Base* pb(new Derived);`

- ❖ Direktna inicijalizacija je manje restriktivna i više dozvoljava nego inicijalizacija kopiranjem: inicijalizacija kopiranjem uzima u obzir samo korisnički definisane konverzije (konstruktore konverzije i konverzionate funkcije) koje nisu *explicit*, dok direktna inicijalizacija uzima u obzir i njih

Inicijalizacija listom

- ❖ Inicijalizacija listom omogućava da se variable inicijalizuju na kompaktan način, čitavim listama izraza unutar velikih zagrada (*braced-init-list*) proizvoljne dužine. Na primer:

```
Task t1, t2, ...;
```

```
TaskList myTasks {&t1, &t2, &t3};
```

```
TaskList yourTasks {&t4, &t5};
```

- ❖ Slično važi i za pozive funkcija kojima se mogu prosleđivati liste kao argumenti; tada se njihovi parametri inicijalizuju listom; na primer:

```
myTasks.add({&t4, &t5, t6});
```

```
myTasks.add({&t7});
```

- ❖ Liste izraza unutar velikih zagrada nisu izrazi pa nemaju svoj tip; osim toga, ovakva inicijalizacija ne dozvoljava neke implicitne konverzije (uključujući i neke standardne)

Inicijalizacija listom

- ❖ Tip `std::initializer_list` iz standardne biblioteke predstavlja tip objekata posrednika koji prenose vrednosti iz inicijalizatorske liste u konstruktore klase i druge funkcije čiji se parametri inicijalizuju listom izraza unutar velikih zagrada. Na primer:

```
class TaskList {  
public:  
    TaskList (std::initializer_list<Task*> lst) { this->add(lst); }  
    TaskList& add (std::initializer_list<Task*>);  
    ...  
};  
  
TaskList& TaskList::add (std::initializer_list<Task*> lst) {  
    for (auto t : lst) {  
        //... add t to the list  
    }  
    return *this;  
}  
  
int main () {  
    Task t1, t2, t3, t4, t5, t6;  
  
    TaskList tlst{&t1, &t2, &t3};  
  
    tlst.add({&t4, &t5}).add({&t6});  
}
```

Inicijalizacija listom

- ❖ *Inicijalizacija kopiranjem-listom (copy-list-initialization)* se obavlja kada se varijabla tipa T inicijalizuje listom izraza unutar velikih zagrada u sledećim slučajevima:

- Kada se imenovana varijabla inicijalizuje listom iza znaka =:

```
T t = {expression, expression, ...};
```

- Kada se argument prenosi u pozvanu funkciju, a parametar inicijalizuje listom:

```
void f (T t);
```

```
f({expression, expression, ...});
```

- Kada se povratna vrednost funkcije inicijalizuje listom:

```
T f () {  
    ...  
    return {expression, expression, ...};  
}
```

- U izrazu za indeks kod korisnički definisanog preklopljenog operatora $[]$, kada se listom inicijalizuje parametar tog operatora:

```
object[{expression, expression, ...}]
```

- U desnom operandu korisnički definisanog preklopljenog operatora dodele, kada se listom inicijalizuje parametar tog operatora:

```
object = {expression, expression, ...}
```

- U izrazu eksplicitne konverzije u obliku funkcionalnog poziva i u eksplicitnim pozivima konstruktora, kada se parametar odgovarajuće konverzione funkcije ili konstruktora inicijalizuje listom (njome se ne inicijalizuje rezultat, nego parametar):

```
X({expression, expression, ...})
```

- Kada se nestatički podatak član inicijalizuje listom iza znaka =:

```
class X { T t = {expression, expression, ...}; };
```

Inicijalizacija listom

- ❖ *Direktna inicijalizacija listom (direct-list-initialization)* inicijalizuje varijablu listom izraza unutar velikih zagrada u sledećim situacijama:

- Kada se imenovana varijabla inicijalizuje listom bez znaka =:

`T t {expression, expression, ...}`

- Kada se privremeni objekat inicijalizuje listom:

`T {expression, expression, ...}`

- Kada se dinamički objekat inicijalizuje u izrazu *new* listom:

`new T {expression, expression, ...}`

- Kada se nestatički podatak član inicijalizuje listom bez znaka =:

`class X { T t {expression, expression, ...}; };`

- Kada se podobjekat osnovne klase ili podobjekat član inicijalizuje listom, unutar liste inicijalizatora u konstruktoru klase:

`C::C (...), B {expression, ...}, m {expression,...} {...}`

Inicijalizacija listom

- ❖ Inizijalizacija varijable tipa T listom radi sledeće (nabrojana su samo neka pravila, i to okvirno):
 - Ako je T agregatni tip, a lista sadrži samo jedan element istog ili tipa izvedenog iz T (bez cv-kvalifikacija), objekat se inicijalizuje tim elementom (inicijalizacijom kopiranjem ili direktnom inicijalizacijom, kako je objašnjeno)
 - Inače, ako je T niz znakova, a lista sadrži samo jedan element koji je string literal odgovarajućeg tipa, niz znakova se inicijalizuje string literalom kao što je uobičajeno
 - Ako je T agregatni tip, primenjuje se agregatna inicijalizacija
 - Inače, ako je lista prazna, a T klasa sa podrazumevanim konstruktorom, primenjuje se inicijalizacija vrednošću
 - Inače, ako je T specijalizacija tipa `std::initializer_list`, objekat se inicijalizuje (direktno ili kopiranjem) iz čvrednosti istog tipa koja se odnosi na datu listu
 - Inače, traži se konstruktor klase T koji može da se pozove sa jednim argumentom tipa `std::initializer_list` i on se poziva sa datom listom
 - Inače, ako T nije klasa, a lista ima samo jedan element, i T je ili referenca kompatibilna sa tipom elementa ili nije refenca, inicijalizuje se direktno ili kopiranjem tog elementa
 - Inače, ako je T referenca, vezuje se za privremeni objekat tipa na koji upućuje referenca, koji se inicijalizuje listom
 - Inače, ako lista nema elemente, vrši se inicijalizacija vrednošću

Aggregatna inicijalizacija

- ❖ Aggregatna inicijalizacija (*aggregate initialization*) je specijalan slučaj inicijalizacije listom, kada je tip objekta koji se inicijalizuje tzv. *aggregatni tip* (*aggregate type*), u sledećim situacijama:

```
T t = {expression, expression, ...};
```

```
T t {expression, expression, ...};
```

- ❖ *Aggregatni tip* je tip niza ili klasni tip koji nema:

- privatne ili zaštićene nestatičke podatke članove
- korisnički deklarisane ili nasleđene konstruktore
- privatne, zaštićene ili virtuelne osnovne klase
- virtuelne funkcije članice

- ❖ U praksi je to najčešće niz, struktura (bez konstruktora) ili unija, a aggregatna inicijalizacija služi da neposredno inicijalizuje elemente niza, odnosno podobjekte strukture (osnovne klase i članove) elementima liste. Na primer:

```
struct Coord {  
    double x, y, z;  
};
```

```
Coord points[] = {{0.,0.}, {3.,0.}, {3.,4.}};
```

Niz *points* imaće tri elementa, svaki predstavlja instancu strukture *Coord* inicijalizovanu jednom od ugnezđenih listi

Agregatna inicijalizacija

❖ Agregatna inizijalizacija radi sledeće:

- Inicijalizuje kopiranjem svaki element niza, odnosno podobjekat osnovne klase i podobjekat član, po redosledu indeksa, odnosno deklarisanja, odgovarajućim elementom liste
- Ako je element liste ugnezđena lista, odgovarajući element/podobjekat se inicijalizuje tom listom (što opet može biti agregatna inicijalizacija, koja je stoga rekurzivna)
- Ako je tip objekta koji se inicijalizuje niz sa nepoznatim brojem elemenata, taj broj elemenata određen je brojem elemenata u listi:

```
Coord points[] = {{0.,0.}, {3.,0.}, {3.,4.}};
```

Niz *points* ima tri elementa

- Ako elemenata u listi ima više nego elemenata niza/podobjekata koje treba inicijalizovati, prijavljuje se greška
- Ako elemenata u listi ima manje nego elemenata niza/podobjekata koje treba inicijalizovati, preostali se inicijalizuju praznim listama, po pravilima inicijalizacije praznom listom (inicijalizacija vrednošću):

```
Coord points[3] = {{3.,0.}, {3.,4.}};
```

Treći element niza biće inicijalizovan nulom

Inicijalizacija reference

- ❖ Reference obe vrste (i na lvrednosti i na dvrednosti) mogu se inicijalizovati na sve navedene načine (u svim slučajevima umesto jednog znaka & za referencu na lvrednost, može stajati i dva znaka && za referencu na dvrednost):

- Kada se imenovana referencia deklariše sa inicijalizatorom:

```
T& ref = expression;  
T& ref(expression);  
T& ref = {expression, expression, ...};  
T& ref{expression, expression, ...};
```

- Kada se argument prenosi u pozvanu funkciju, a parametar je referencia:

```
void f (T& t);  
  
f(expression);  
f({expression, expression, ...});
```

- Kada je povratna vrednost funkcije referencia:

```
T& f () {  
    ...  
    return expression;  
}
```

- Kada se nestatički podatak član inicijalizuje u listi inicijalizatora članova u konstruktoru klase:

```
X::X (...) : ref(expression) {...}
```

Inicijalizacija reference

- ❖ Ako se za inicijalizaciju upotrebljava lista, onda se primjenjuje inicijalizacija listom: pravi se privremeni objekat tipa T i taj objekat se inicijalizuje datom listom:

```
int (&ref)[] = {0, 1, 2};
```

- ❖ Ako je referenca na lurednost, a izraza kojim se inicijalizuje je lurednost tipa T ili tipa izvedenog iz T , referenca se vezuje za objekat koji identificira taj izraz, onosno na njegov podobjekat osnovne klase tipa T :

```
Base& ref = *new Derived;
```

- ❖ Ako je referenca na lurednost, može se inicijalizovati izrazom koji je lurednost samo ako je referenca na konstantu; tada se opet pravi (materijalizuje) privremeni objekat za koji se vezuje referenca:

```
const int& ref = a+b;
```

- ❖ Takav privremeni objekat nastavlja da živi dok je na njega vezana referenca, ali ne nakon izlaska iz funkcije u kojoj se to dešava
- ❖ Izraz kojim se inicijalizuje referenca može biti i tipa za koji postoji konverzija koja vraća odgovarajuću referencu

Glava 15: Preklapanje operatora

- ❖ Preklopljeni operatori
- ❖ Preporučeni načini preklapanja operatora
- ❖ Operatori *new* i *delete*



Prekopljeni operatori

- ❖ *Prekopljeni operatori (overloaded operator)* su funkcije sa posebnim imenom (simbol @ označava simbol nekog operatora ugrađenog u jezik, a T neki tip):

```
operator @  
operator T  
  
operator new  
operator new []  
  
operator delete  
operator delete []  
  
operator "" suffix
```

- ❖ Kada se neki od operatora pojave u izrazu sa nekim operandom koji je tipa klase ili enumeracije, prevodilac traži korisnički definisanu funkciju koja odgovara tipovima i broju operanada tog operatora po opštim pravilima postupka *ADL*:

```
complex c1=..., c2=..., c3; c3 = c1+c2;
```

- ❖ Ove funkcije mogu da se pozivaju i neposredno:

```
c3.operator=(operator+(c1,c2));
```

- ❖ Prekopljene operatorske funkcije mogu biti nestatičke funkcije članice klase ili funkcije nečlanice koje imaju barem jedan parametar koji prima argumente tipa klase ili enumeracije

Prekopljeni operatori

- ❖ Postoje sledeća ograničenja:
 - Ne mogu da se preklope sledeći operatori: `::`, `.`, `.`^{*} i `?:`, kao ni operatori koji nisu simboli, već ključne reči (*sizeof*, *const_cast*, *static_cast*, *dynamic_cast*, *reinterpret_cast* itd.)
 - Ne mogu da se definišu novi operatori, npr. `**` ili `<>`
 - Prekopljeni operatori `&&` i `||` gube svojstvo izračunavanja prečicom (*short-circuit evaluation*)
 - Prekopljeni operatori `=`, `()`, `[]`, i `->` moraju biti nestatičke funkcije članice
 - Prekopljeni operator `->` mora da vrati ili običan pokazivač, ili objekat (po vrednosti ili referenci) za koji je ponovo definisan prekopljeni operator `->`
 - Ne može da se promeni broj operanada, način grupisanja i prioritet operatora; zapravo se ne može promeniti način tumačenja (parsiranja) izraza i pridruživanja operanada operatorima, jer to određuju pravila jezika; ali kada pridruži operative operatorima, prevodilac traži odgovarajuću prekopljenu funkciju za operative koji su tipa klase ili enumeracije
- ❖ Povratni tip prekopljenog operatora može biti bilo koji, pa i `void`

Prekopljeni operatori

- ❖ Prekopljeni operatori mogu biti deklarisani ili kao nestatičke funkcije članice ili kao funkcije nečlanice (često prijateljske); neki operatori mogu biti prekopljeni na bilo koji od ta dva načina, a neki samo kao članice
- ❖ Prekopljeni operatori mogu biti unarni (imaju jedan operand) ili binarni (imaju dva operanda)
- ❖ Unarni operator @ ima jedan operand, pa može biti prekopljen kao:
 - nestatička funkcija članica klase X bez parametara:

T X::**operator@** ();

tada se operacija @x tumači kao

x.**operator@** ()

- jedini operand je objekat čija se funkcija članica poziva

- funkcija nečlanica klase X sa jednim parametrom koji prihvata argument tipa X:

T **operator@** (X);

tada se operacija @x tumači kao

operator@ (x)

- jedini operand se prenosi kao argument funkcije

Preklopljeni operatori

- ❖ Binarni operator @ ima dva operanda, pa može biti preklopljen kao:

- nestatička funkcija članica klase X sa jednim parametrom:

T X::**operator@** (Y);

tada se operacija $x@y$ tumači kao

x.**operator@** (y)

- prvi operand je objekat čija se funkcija članica poziva, a drugi operand se prenosi kao argument

- funkcija nečlanica klase X sa dva parametra:

T **operator@** (X, Y);

tada se operacija $x@y$ tumači kao

operator@ (x, y)

- oba operanda se prenose kao argumenti

Prekopljeni operatori

- ❖ Kada traži prekopljeni operator za date tipove operanada, isto kao i za ostale pozive funkcija, prevodilac sprovodi postupak *potrage zavisne od argumenata* (*argument dependent lookup, ADL*); pojednostavljeno rečeno, ovaj postupak traži funkciju, uključujući i prekopljen operator, ne samo prema uobičajenim pravilima potrage za nekvalifikovanim imenom u tekućoj oblasti važenja, nego i po klasama kojima pripadaju argumenti (i njihovim osnovnim klasama), ali i po prostorima imena kojima pripadaju te klase
- ❖ Ovo omogućava da prekopljeni operatori budu deklarisani u različitim prostorima imena, a da ipak budu odabrani za argumente određenog tipa; na primer:

```
namespace complex {  
  
    class complex {  
        ...  
        friend std::ostream& operator<< (std::ostream& os, const complex& c) {  
            return os<<'('<<c.real<<', '<<c.imag<<')';  
        }  
    };  
}  
  
int main () {  
    complex::complex c1, c2;  
    ...  
    std::cout<<c1<<','<<c2;  
}
```

Poziva se prekopljeni operator `<<` za tipove `char` i `double`, definisan u prostoru imena `std`, jer je operand `os` tipa `std::ostream`.
Ovaj izraz vraća referencu na isti objekat klase `ostream` koji je dostavljen kao argument

Poziva se prekopljeni operator `<<` za tip `complex`, definisan u prostoru imena `complex`, jer je operand `c1` tipa `complex::complex`.
Pošto je ovaj operator vratio referencu na objekat tipa `ostream`, može se upotrebiti kao levi operand istog operatora

Prekopljeni operatori

- ❖ Jedan aspekt može biti veoma bitan pri odlučivanju o tome da li neki prekopljeni operator treba napraviti kao nestatičku funkciju članicu ili kao funkciju nečlanicu (ako je to moguće):
 - poziv nestatičke funkcije članice za neki izraz ne dozvoljava “promociju” tog izraza, odnosno bilo kakvu implicitnu konverziju levog operanda operatora `:` zahteva se da klasa kojoj pripada taj objekat, ili neka njena osnovna klasa, ima navedenu funkciju, inače poziv nije dozvoljen:

```
complex complex::operator+ (complex c) {...}  
complex::complex (double d) {...}
```

```
complex c1(3., 4.); double d = 5.0;  
complex c2 = d+c1;
```

Konstruktor konverzije, definiše implicitnu konverziju iz `double` u `complex`

Greška u prevodenju: tumači se kao `d.operator+(c1)`, a levi operand operatora `+` ne konvertuje se u objekat

- pri pozivu funkcije nečlanice vrše se dozvoljene i definisane implicitne konverzije argumenata; za navedeni primer klase `complex`, ukoliko se operator `+` definiše kao funkcija nečlanica i definiše navedeni konstruktor konverzije iz tipa `double`, moguće su sve varijante poziva za različite tipove argumenata:

```
complex operator+ (complex c1, complex c2) {...}  
complex::complex (double d) {...}
```

```
complex c1(3., 4.); double d = 5.0; int i = 3;  
complex c2 = d+c1, c3 = c2+i, c4 = c2+c3;
```

Preklopljeni operatori

- ❖ Zbog toga je uobičajena praksa da se operatori preklapaju na sledeći način:
 - operatori kod kojih operandi nisu na neki način ravnopravni, simetrični, prvenstveno zbog toga što dati operator ima bočni efekat, odnosno menja svoj operand, definišu se kao nestatičke funkcije članice; na primer, iako operatori složene dodele (+= i ostali) ili operatori ++ i -- mogu, što se tiče jezika, da se definišu i kao funkcije nečlanice, obično se definišu kao nestatičke funkcije članice jer menjaju svoj (levi) operand, pa zato zahtevaju to da taj operand bude baš objekat nad kojim deluju, a ne neka konvertovana vrednost (privremeni objekat):

```
complex complex::operator+=(complex c) {...}
```

```
complex c1(3., 4.), c2;  
c2 += c1;
```

Tumači se kao *c2.operator+=(c1)*

- operatori kod kojih su operandi na neki način ravnopravni (simetrični), odnosno koji ne menjaju svoje operande (npr. binarni aritmetički i relacioni operatori, operatori koji rade sa bitima itd.), preklapaju se kao funkcije nečlanice:

```
complex operator+(complex c1, complex c2) {...}  
complex::complex(double d) {...}
```

```
complex c1(3., 4.); double d = 5.0; int i = 3;  
complex c2 = d+c1, c3 = c2+i, c4 = c2+c3;
```

Prekopljeni operatori

- ❖ Operatori `++` i `--` imaju prefiksni i postfiksni oblik koji se mogu razlikovati tako što se prekopljeni postfiksni operatori definišu sa parametrom koji prihvata drugi operand tipa `int`; pri pozivu prekopljene operatorske funkcije, ovaj argument ima vrednost 0:

```
complex& complex::operator++ () {...}
```

Prefiksni operator, vraća lurednost

```
complex complex::operator++ (int) {...}
```

Postfiksni operator, vraća čdvrednost

```
complex c(3.,4.);
```

Tumači se kao `c.operator++(0)`

```
c++;
```

Tumači se kao `c.operator++()`

- ❖ Prekopljeni operator `->` mora biti nestatička funkcija članica koja nema parametre i koja mora da vrati ili običan pokazivač, ili (referencu na) objekat za koji je takođe prekopljen operator `->`:

```
template<typename T>
T* smart_ptr<T>::operator-> () {...}
```

```
smart_ptr<Clock> p = &clk;
```

```
p->tick();
```

Tumači se kao `(p.operator->())->tick();`

Preporučeni načini preklapanja operatora

- ❖ Osim navedenih ograničenja, pravila jezika ne nameću bilo koja druga, pa stoga:
 - preklopljeni operator može imati bilo kakve efekte i bilo kakvo značenje; na primer, operator dodele = uopšte ne mora raditi dodelu kopiranjem, ili operator + uopšte ne mora raditi sabiranje (šta god to značilo za dati tip)
 - preklapanje jednog operatora ne znači implicitno i preklapanje nekog drugog, blisko povezanog operatora; na primer, ako je za klasu preklopljen operator ==, nije implicitno preklopljen i operator !=, ili, ako je za klasu preklopljen operator <, nije implicitno preklopljen i operator > i slično
 - veze koje postoje za ugrađene operatore ne moraju biti takve i za preklopljene operatore; na primer, za ugrađene operatore +, = i += važi odgovarajuća sprega, koja uopšte ne mora da važi za preklopljene operatore (da $a+=b$ ima efekat kao $a=a+b$)
 - preklopljeni operatori mogu da vraćaju bilo koji tip, pa i tip *void*, iako ugrađeni operatori vraćaju odgovarajuće tipove; na primer, ugrađeni operator dodele vraća vrednost koja upućuje na levi operand, dok preklopljeni operator dodele može vratiti bilo koji tip i bilo kakvu vrednost

Preporučeni načini preklapanja operatora

- ❖ Međutim, preklopljene operatore treba praviti tako da imaju očekivano značenje i efekte, kao i da njihova upotreba što više liči na upotrebu ugrađenih operatora, jer to i jeste osnovna motivacija i namera celog koncepta preklapanja operatora - da se korisnički definitisani tipovi mogu upotrebljavati na isti način kao i ugrađeni tipovi, pa i u izrazima
- ❖ Prvo, preklopljeni operatori treba da imaju očekivane, logične efekte; na primer, operator dodele = treba da radi dodelu kopiranjem ili premeštanjem, a operator + (ako je definisan) treba da radi sabiranje (šta god to značilo za dati tip)

Preporučeni načini preklapanja operatora

- ❖ Skup preklopljenih operatora treba da bude “zatvoren”, u smislu da su preklopljeni svi operatori koji se očekuju zajedno; to nikako ne znači da za klasu treba preklapati sve moguće operatore, već samo da skup operatora treba kompletirati na one očekivane; na primer:
 - ako je preklopljen operator `==`, treba preklopiti i operator `!=`, i to baš tako da vraća negaciju operatora `==` za iste operande, ali ne obavezno i operatore `< i >` (jer za klasu možda nema smisla takva relacija uređenja)
 - ako je za klasu preklopljen operator `== i <`, treba preklopiti i operatore `!=, >, <= i >=`; od verzije jezika C++20, u jeziku postoji operator opštег poređenja `<=>` koji vraća vrednost manju od 0 ako je prvi operand manji od drugog, veću od 0 ako je prvi veći od drugog, a nulu ako su operandi jednaki; dovoljno je preklopiti samo ovaj operator i svi drugi vezani operatori poređenja na jednakost i nejednakost biće implicitno definisani na odgovarajući način
 - ako je za klasu preklopljen operator `+` i postoji operator `=`, treba preklopiti i operator `+=` i to sa očekivanim spregnutim značenjem, jer će se i on očekivati,
i tako dalje

Preporučeni načini preklapanja operatora

- ❖ Veze koje postoje za ugrađene operatore treba da budu takve i za preklopljene operatore; na primer, za ugrađene operatore `+`, `=` i `+=` važi odgovarajuća sprega, koja treba da važi i za preklopljene operatore (da `a+=b` ima efekat kao `a=a+b`)
- ❖ Preklopljeni operatori treba da vraćaju očekivani tip, po uzoru na ugrađene operatore, i istu kategoriju vrednosti: na primer, ugrađeni operatori dodele i složene dodele vraćaju vrednost koja upućuje na levi operand, pa to isto treba raditi i za preklopljene operatore, kako bi moglo da se radi i npr. sledeće:

`(a = b) = c;`

`(a = b) ++;`

- ❖ Za preklopljene operatore dodele = kopiranjem i premeštanjem preporučuje se sledeće:
 - da vraćaju referencu na svoj levi operand (kao nekonstantu)
 - da ne rade ništa u slučaju samodeljivanja (`x=x`)
 - da operator dodele premeštanjem ostavi desni operand (argument) u konzistentnom stanju, barem tako da se može propisno uništiti i da ne baca izuzetke (*noexcept*)

Preporučeni načini preklapanja operatora

- ❖ Binarni, infiksni aritmetički operatori, kao i binarni, infiksni operatori nad bitima se obično preklapaju kao funkcije nečlanice, jer su im operandi ravnopravni (simetrični)
- ❖ Pošto su uobičajeno praćeni operatorima složene dodele, obično se aritmetički operatori svode na operatore složene dodele na sledeći način:

```
X& X::operator+= (const X& other) {  
    ...  
    return *this;  
}  
  
X operator+ (const X& x1, const X& x2) {  
    X temp(x1);  
    temp += x2;  
    return temp;  
}
```

Preporučeni načini preklapanja operatora

- ❖ Binarni, infiksni relacioni operatori se obično preklapaju kao funkcije nečlanice, jer su im operandi ravnopravni
- ❖ Preklapaju se tako što se implementira operator == i, ako je potrebno, operator <, a onda ostali svode na njih:

```
bool operator== (const X& x1, const X& x2) {  
    ...  
}  
  
inline bool operator!= (const X& x1, const X& x2)  
{ return !(x1==x2); }
```

```
bool operator< (const X& x1, const X& x2) {  
    ...  
}  
  
inline bool operator> (const X& x1, const X& x2)  
{ return x2<x1; }
```

```
inline bool operator<= (const X& x1, const X& x2)  
{ return (x1<x2)|| (x1==x2); }
```

```
inline bool operator>= (const X& x1, const X& x2)  
{ return x2<=x1; }
```

Preporučeni načini preklapanja operatora

- ❖ Od verzije C++20 postoji operator `<=>` koji se može preklopiti, a kad je on preklopljen, implicitno su definisani i svi drugi relacioni operatori u prirodnom obliku:

```
int operator<=> (const string& s1, const string& s2) {
    return strcmp(s1.str,s2.str);
}
```

- ❖ Kad je ovaj operator preklopljen, implicitno je definisano i svih šest ostalih relacionih operatara u prirodnom obliku:

```
inline bool operator== (const string& s1, const string& s2) { return (s1<=>s2)==0; }
inline bool operator!= (const string& s1, const string& s2) { return !(s1==s2); }
inline bool operator< (const string& s1, const string& s2) { return (s1<=>s2) < 0; }
inline bool operator> (const string& s1, const string& s2) { return s2<s1; }
inline bool operator<= (const string& s1, const string& s2)
{ return (s1<s2)||(s1==s2); }
inline bool operator>= (const string& s1, const string& s2) { return (s2<=s1); }
```

- ❖ Može se tražiti i generisanje podrazumevane implementacije ovog operatora, koja se svodi na isti operator primjenjen redom na podobjekte, u poretku njihovog deklarisanja:

```
bool operator<=> (const X&, const X&) = default;
```

Preporučeni načini preklapanja operatora

- ❖ Kada se preklapaju operatori `<< i >>` za klase sa ciljem implementacije izlaznih odnosno ulaznih operacija preko znakovnih tokova (*I/O stream*), moraju se definisati kao funkcije nečlanice, jer im je drugi (desni) operand onaj za koji se definišu
- ❖ Oni treba da vraćaju referencu na isti objekat koji je dostavljen kao prvi operand:

```
std::ostream& operator<< (std::ostream& os, const T& object) {  
    // Write object to output stream os  
  
    return os;  
}  
  
std::istream& operator>> (std::istream& is, T& object) {  
    // Read object from input stream is  
  
    if /* T could not be created */ )  
        is.setstate(std::ios::failbit);  
  
    return is;  
}
```

Preporučeni načini preklapanja operatora

- ❖ Operatori prefiksnog i postfiksnog inkrementiranja i dekrementiranja se obično preklapaju na sledeći način:

```
X& X::operator++ () {
```

```
...
```

```
    return *this;
```

```
}
```

```
X X::operator++ (int) {
```

```
    X temp(*this);
```

```
    operator++();
```

```
    return temp;
```

```
}
```

Preporučeni načini preklapanja operatora

- ❖ Unarni logički operator negacije `!` se preklapa za klase čiji se objekti koriste u kontekstu gde se očekuju Bulovi izrazi
- ❖ Zbog toga ove klase najčešće imaju i operator konverzije u tip `bool` koji je *explicit*, a operator `!` svodi se na negaciju te konvertovane vrednosti:

```
X::operator bool () {  
    ...  
}  
  
inline bool operator! (const X& x) {  
    return !bool(x);  
}
```

Preporučeni načini preklapanja operatora

- ❖ Operator [] se mora preklopiti kao nestatička funkcija članica; ona može imati parametar bilo kog tipa; taj parametar prima vrednost izraza između uglastih zagrada (drugog operanda):

`x[expression]`

tumači se kao:

`x.operator[](expression)`

- ❖ Ovaj operator se tipično preklapa tako da postoje varijante koje omogućavaju i čitanje iz konstantnih i nekonstantnih objekata, kao i upis u nekonstantne objekte, odnosno varijante koje su konstantna i nekonstantna funkcija članica i koje vraćaju odgovarajuće vrednosti (reference na konstantu i nekonstantu, respektivno):

```
class X {  
    ...  
    T& operator[] (size_t);  
    const T& operator[] (size_t) const;  
    ...  
};
```

Preporučeni načini preklapanja operatora

- ❖ Operator () se preklapa za klase čiji se objekti mogu posmatrati kao tzv. *funkcijski objekti (function object)*, tj. objekti koji se mogu koristiti kao funkcije, jer se na njih može primeniti operator poziva funkcije (kao za obične funkcije):

`x(expression1, expression2)`

tumači se kao:

`x.operator()(expression1, expression2)`

- ❖ Za razliku od običnih funkcija, funkcijski objekti, kao i svaki drugi objekti, mogu imati (i tipično imaju) svoje stanje, koje "nose" sa sobom (svaki objekat svoje nezavisno stanje) i mogu da ga menjaju i akumuliraju
- ❖ Ovakvi objekti se tipično prenose, slično kao i pokazivači na funkcije, kao argumenti nekih operacija koje sprovode određene postupke, odnosno algoritme, za koji su im potrebne implementacije elementarnih operacija-koraka tog algoritma; zato takav algoritam poziva nazad dostavljenu funkciju, odnosno funkcijski objekat (tzv. *callback mehanizam*), ili ga primenjuje na neke elemente koje obilazi tokom algoritma (npr. obilasci raznih struktura)
- ❖ Na primer, funkcija *for_each* iz standardne biblioteke iterira kroz kolekciju, dok kao treći parametar očekuje funkcijski objekat koga primenjuje na svaki posećeni element, dostavljajući mu taj posećeni element kao argument operatora ():

```
struct Sum {  
    int sum;  
    Sum () : sum(0) {}  
    void operator() (int n) { sum += n; }  
};  
  
std::vector<int> v{...};  
  
Sum s = std::for_each(v.begin(), v.end(), Sum());
```

Operatori *new* i *delete*

- ❖ Kao što je već objašnjeno, izraz oblika

```
new (placement_params) T (init)
```

uvek radi sledeće tri stvari:

1. Poziva neku od ugrađenih globalnih alokatorskih funkcija oblika

```
void* ::operator new (size_t size, placement_params)
```

```
void* ::operator new [] (size_t size, placement_params)
```

2. Inicijalizuje objekat odgovarajućom inicijalizacijom

3. Vraća pokazivač na napravljeni dinamički objekat

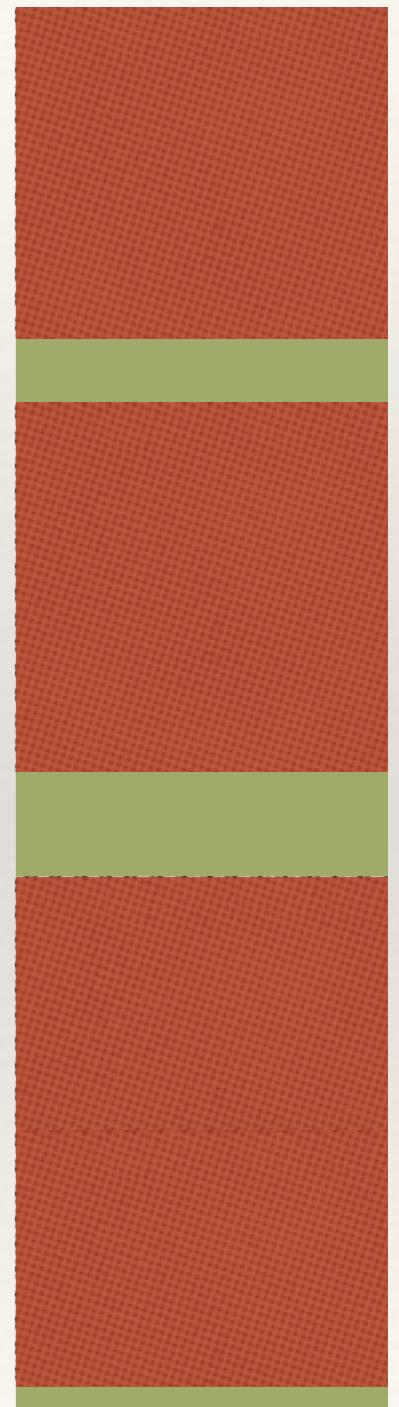
- ❖ Ovaj postupak je uvek isti i ne može se promeniti, u smislu skupa i redosleda koraka, ali se poziv alokatorske funkcije u prvom koraku može preusmeriti na drugu funkciju, pa i onu definisanu posebno za neku klasu preklapanjem alokatorske funkcije *new* za tu klasu

- ❖ U prvom koraku, ugrađene alokatorske funkcije ::*new* podrazumevano rade sa slobodnim delom memorije u kom pronalaze slobodan prostor tražene veličine *size* i alociraju ga

- ❖ Ove funkcije mogu da se zamene drugim, korisnički definisanim, sa istim potpisom; dovoljno je definisati neku od njih u nekom fajlu

Operatori *new* i *delete*

- ❖ Posmatrajmo izgled slobodnog prostora tokom sukcesivnih operacija alokacije i dealokacije delova memorije različite veličine: kada se dealocira neki dinamički objekat, u oslobođeni prostor njegove veličine može se alocirati nov dinamički objekat koji je najčešće manji (može biti i jednak, što je manje verovljano, ali nikako veći) od tog dela; tako ostaje *fragment* slobodne memorije koji možda nije dovoljan da se u njega smesti bilo koji nov objekat
- ❖ Posle dužeg rada programa može se dogoditi to da je slobodna memorija rascepka (fragmentirana) na male slobodne delove, pa iako je ukupno ima sasvim dovoljno, nijedan slobodan deo nije dovoljno velik da bi se u njega smestio traženi objekat; tada program više ne može alocirati prostor za traženi novi dinamički objekat i više ne radi regularno
- ❖ Ovakav problem naziva se *fragmentacija memorije* (*memory fragmentation*)
- ❖ Ovaj problem nije primetan ni značajan kod programa koji rade neko vreme i onda se gase, npr. kod interaktivnih uslužnih programa, ali kod programa koji rade neprekidno, npr. onih ugrađenih u hardver koji radi neprekidno (*embedded software*) može da bude veoma ozbiljan



Operatori *new* i *delete*

- ❖ Alokatorska funkcija *new* može se preklopiti za neku klasu *T* kao statička funkcija članica te klase (čak i ako se ne deklariše kao *static*, uvek je implicitno statička):

```
void* T::operator new (size_t size, placement_params)
```

```
void* T::operator new [] (size_t size, placement_params)
```

- ❖ Ove funkcije mogu da se preklope u svim oblicima u kojima postoje i takve ugrađene funkcije
- ❖ Ove funkcije ne treba da pozivaju konstruktore eksplisitno; konstruktor se uvek poziva kao drugi korak u izvršavanju izraza *new* čiji samo jedan korak poziv ove alokatorske funkcije
- ❖ Ove funkcije imaju zadatak da pronađu “sirov” memorijski prostor zadate veličine i ništa više, nikako da od njega prave objekat (što radi konstruktor)
- ❖ Upravo zato i vraćaju pokazivač tipa *void** koji sadrži adresu alociranog prostora i upravo zato su one statičke funkcije članice klase, jer se pozivaju pre pravljenja objekta te klase
- ❖ Ako klasa ima ovaku funkciju, ona će biti pozivana u izvršavanju prvog koraka izraza *new* kada se alociraju objekti, odnosno nizovi objekata te klase
- ❖ Ovakva funkcija može alocirati dinamičke objekte nekog drugog tipa ili na drugi način rukovati prostorom

Operatori *new* i *delete*

- ❖ Analogno, dealokatorska funkcija *delete* može se preklopiti za neku klasu *T* kao statička funkcija članica te klase (čak i ako se ne deklariše kao *static*, uvek je implicitno statička):

```
void T::operator delete (void*)
```

```
void T::operator delete [] (void*)
```

- ❖ Ove funkcije mogu da se preklope u još nekim dostupnim oblicima (npr. onim kojim primaju veličinu prostora koji se dealocira)
- ❖ Ove funkcije ne treba da pozivaju destruktore eksplisitno; destruktur se uvek poziva kao korak u izvršavanju izraza *delete*
- ❖ Ove funkcije imaju zadatak da memorijski prostor na zadatoj adresi proglase slobodnim i ništa više
- ❖ Upravo zato i primaju pokazivač tipa *void** koji sadrži adresu prostora koji treba dealocirati (a ne pokazivač na objekat klase, jer je objekat već uništen)
- ❖ Ako klasa ima ovaku funkciju, ona će biti pozivana u izvršavanju drugog koraka izraza *delete* kada se dealociraju objekti, odnosno nizovi objekata te klase

Operatori *new* i *delete*

- ❖ Jedna ideja za alokaciju prostora za objekte klase X koja nema problem fragmentacije, jer objekte smešta u niz slotova veličine tipa X, pri čemu slobodne slotove ulančava u listu (pa su operacije alokacije i dealokacije kompleksnosti $O(1)$):

```
template <class T, int size>
class Storage {
public:

    Storage () : head(slots) { slots[size-1].next = nullptr; }

    void* alloc () { Slot* p=head; if (p) head=p->next; return p?p->slot:nullptr; }

    void free (void* addr) { head = new (addr) Slot(head); }

private:

    struct Slot {
        Slot () : next(this+1) {}
        Slot (Slot* nxt) : next(nxt) {}

        union {
            Slot* next;
            char slot[sizeof(T)];
        };
    };

    Slot* head;
    Slot slots[size];
};
```

Poziva se ugrađena alokatorska funkcija za *placement new* i klasu *Slot*

Operatori *new* i *delete*

- ❖ Sada se ova pomoćna klasa za alokaciju može koristiti na sledeći način:

```
class X {  
public:  
    ...  
  
    void* operator new (size_t) {  
        void* addr = storage.alloc();  
        if (!addr) throw std::bad_alloc;  
        else return addr;  
    }  
  
    void operator delete (void* addr) { storage.free(addr); }  
  
private:  
    static Storage<X, 2000> storage;  
};  
  
Storage<X, 2000> X::storage;
```

- ❖ Nedostatak ovog rešenja jeste to što se unapred, statički mora zadati veličina skladišta; potrebnu veličinu je često teško odrediti, jer zavisi od dinamičke prirode programa, a čak i ako se odredi, može biti premala (da ne može da zadovolji trenutne potrebe, iako slobodne memorije i dalje ima) ili prevelika (pa se memorija nepotrebno rezerviše i drži zauzetom za skladište, iako nije potrebna za objekte). Ova veličina može se zadati i dinamički, ali pri njegovoj inicijalizaciji (unaprediti ovo rešenje na taj način)
- ❖ Postoje i naprednija rešenja koja nemaju ovo ograničenje; zadatak: osmisliti i implementirati neko takvo rešenje

Operatori *new* i *delete*

- ❖ Evo još jedne ideje elegantnog rešenja čiji je način korišćenja skoro isti, a koje nema problem fragmentacije niti potrebe za dimenzionisanjem skladišta; proučiti njegovo delovanje:

```
template <class T>
class RecycleBin {
public:
    void* alloc () {
        Slot* p=head;
        if (p) head=p->next; else p = new Slot;
        return p;
    }

    void free (void* addr) { head = new (addr) Slot(head); }

private:
    struct Slot {
        Slot (Slot* nxt=nullptr) : next(nxt) {}

        union {
            Slot* next;
            char slot[sizeof(T)];
        };
    };

    Slot* head = nullptr;
};
```

Poziva se ugrađena alokatorska funkcija za klasu *Slot* koja radi sa ugrađenim alokatorom za slobodnu memoriju