Dragan Miliæev

Objektno orijentisano programiranje u realnom vremenu na jeziku C++

Beograd, 1996.

Deo I
Objektno orijentisano programiranje i modelovanje

Uvod

* Jezik C++ je objektno orijentisani programski jezik opšte namene. Veliki deo jezika C++ nasleđen je iz jezika C, pa C++ predstavlja (uz minimalne izuzetke) nadskup jezika C.
* Kurs uvodi u osnovne koncepte objektno orijentisanog programiranja i principe projektovanja objektno orijentisanih softverskih sistema, korišæenjem jezika C++ kao sredstva.
* Kurs je baziran na referencama [ARM] i [Miliæev95]. Knjiga [Miliæev95] predstavlja osnovu ovog kursa, a u ovom dokumentu se nalaze samo glavni izvodi. Kurs sadrži i najvažnije elemente jezika C.

Zašto OOP?

* Objektno orijentisano programiranje (*Object Oriented Programming*, OOP) je odgovor na tzv. krizu softvera. OOP pruža naèin za rešavanje (nekih) problema softverske proizvodnje.
* Softverska kriza je posledica sledeæih problema proizvodnje softvera:

1. Zahtevi korisnika su se *drastièno* poveæali. Za ovo su uglavnom "krivi" sami programeri: oni su korisnicima pokazali šta sve raèunari mogu, i da mogu mnogo više nego što korisnik može da zamisli. Kao odgovor, korisnici su poèeli da traže mnogo više, više nego što su programeri mogli da postignu.
2. Neophodno je poveæati produktivnost programera da bi se odgovorilo na zahteve korisnika. To je moguæe ostvariti najpre poveæanjem broja ljudi u timu. Konvencionalno programiranje je nametalo projektvanje softvera u modulima sa relativno jakom interakcijom, a jaka interakcija između delova softvera koga pravi mnogo ljudi stvara haos u projektovanju.
3. Produktivnost se može poveæati i tako što se neki delovi softvera, koji su ranije veæ negde korišæeni, mogu ponovo iskoristiti, bez mnogo ili imalo dorade. Laku ponovnu upotrebu koda (*software reuse*) tradicionalni naèin programiranja nije omoguæavao.
4. Poveæani su drastièno i troškovi održavanja. Potrebno je bilo naæi naèin da projektovani softver bude èitljiviji i lakši za nadgradnju i modifikovanje. Primer: èesto se dešava da ispravljanje jedne greške u programu generiše mnogo novih problema; potrebno je "lokalizovati" realizaciju nekog dela tako da se promene u realizaciji "ne šire" dalje po ostatku sistema.

* Tradicionalno programiranje nije moglo da odgovori na ove probleme, pa je nastala kriza proizvodnje softvera. Poveæane su režije koje prate proizvodnju programa. Zato je OOP došlo kao odgovor.

Šta daju OOP i C++ kao odgovor?

* C++ je trenutno najpopularniji objektno orijentisani jezik. Osnovna rešenja koja pruža OOP, a C++ podržava su:

1. Apstrakcija tipova podataka (*Abstract Data Types*). Kao što u C-u ili nekom drugom jeziku postoje ugraðeni tipovi podataka (`int`, `float`, `char`, ...), u jeziku C++ korisnik može proizvoljno definisati svoje tipove i potpuno ravnopravno ih koristiti (`complex`, `point`, `disk`, `printer`, `jabuka`, `bankovni_racun`, `klijent` itd.). Korisnik može deklarisati proizvoljan broj promenljivih svog tipa i vršiti operacije nad njima (*multiple instances*, višestruke instance, pojave).
2. Enkapsulacija (*encapsulation*). Realizacija nekog tipa može (i treba) da se sakrije od ostatka sistema (od onih koji ga koriste). Treba korisnicima tipa precizno definisati samo *šta* se sa tipom može raditi, a naèin *kako* se to radi sakriva se od korisnika (definiše se interno).
3. Preklapanje operatora (*operator overloading*). Da bi korisnièki tipovi bili sasvim ravnopravni sa ugraðenim, i za njih se mogu definisati znaèenja operatora koji postoje u jeziku. Na primer, ako je korisnik definisao tip `complex`, može pisati `c1+c2` ili `c1*c2`, ako su `c1` i `c2` promenljive tog tipa; ili, ako je `r` promenljiva tipa `racun`, onda `r++` može da znaèi "dodaj (podrazumevanu) kamatu na raèun, a vrati njegovo staro stanje".
4. Nasleðivanje (*inheritance*). Pretpostavimo da je veæ formiran tip `Printer` koji ima operacije nalik na `print_line`, `line_feed`, `form_feed`, `goto_xy` itd. i da je njegovim korišæenjem veæ realizovana velika kolièina softvera. Novost je da je firma nabavila i štampaèe koji imaju bogat skup stilova pisma i želja je da se oni ubuduæe iskoriste. Nepotrebno je ispoèetka praviti novi tip štampaèa ili prepravljati stari kôd. Dovoljno je kreirati novi tip `PrinterWithFonts` koji je "baš kao i obièan" štampaè, samo"još može da" menja stilove štampe. Novi tip æe *naslediti sve* osobine starog, ali æe još ponešto moæi da uradi.
5. Polimorfizam (*polymorphism*). Pošto je `PrinterWithFonts` veæ ionako `Printer`, nema razloga da ostatak programa ne "vidi" njega kao i obièan štampaè, sve dok mu nisu potrebne nove moguænosti štampaèa. Ranije napisani delovi programa koji koriste tip `Printer` *ne moraju se uopšte prepravljati*, oni æe jednako dobro raditi i sa novim tipom. Pod odreðenim uslovima, stari delovi ne moraju se èak ni ponovo prevoditi! Karakteristika da se novi tip "odaziva" na pravi naèin, iako ga je korisnik "pozvao" kao da je stari tip, naziva se *polimorfizam*.

* Sve navedene osobine mogu se pojedinaèno na ovaj ili onaj naèin realizovati i u tradicionalnom jeziku (kakav je i C), ali je realizacija svih koncepata zajedno ili teška, ili sasvim nemoguæa. U svakom sluèaju, realizacija nekog od ovih principa u tradicionalnom jeziku drastièno poveæava režije i smanjuje èitljivost programa.
* Jezik C++ prirodno podržava sve navedene koncepte, oni su ugraðeni u sâm jezik.

Šta se menja uvoðenjem OOP?

* Jezik C++ nije "èisti" objektno orijentisani programski jezik (*Object-Oriented Programming Language*, OOPL) koji bi korisnika "naterao" da ga koristi na objektno orijentisani (OO) naèin. C++ može da se koristi i kao "malo bolji C", ali se time ništa ne dobija (èak se i gubi). C++ treba koristiti kao sretstvo za OOP i kao smernicu za razmišljanje. C++ ne spreèava da se pišu loši programi, veæ samo omoguæava da se pišu mnogo bolji programi.
* OOP uvodi *drugaèiji naèin razmišljanja* u programiranje!
* U OOP, *mnogo* više vremena troši se na *projektovanje*, a mnogo manje na samu implementaciju (kodovanje).
* U OOP, razmišlja se najpre o *problemu*, ne direktno o programskom rešenju.
* U OOP, razmišlja se o delovima sistema (objektima) koji nešto rade, a ne o tome kako se nešto radi (algoritmima).
* U OOP, pažnja se prebacuje sa realizacije na meðusobne veze izmeðu delova. Težnja je da se te veze što više redukuju i strogo kontrolišu. Cilj OOP je da smanji interakciju izmeðu softverskih delova.

Pregled osnovnih koncepata OOP u jeziku C++

* U ovoj glavi biæe dât kratak i sasvim površan pregled osnovnih koncepata OOP koje podržava C++. Potpuna i precizna objašnjenja koncepata biæe data kasnije, u posebnim glavama.
* Primeri koji se koriste u ovoj glavi nisu usmereni da budu upotrebljivi, veæ samo pokazni. Iz realizacije primera izbaèeno je sve što bi smanjivalo preglednost osnovnih ideja. Zato su primeri èesto i nekompletni.
* Èitalac ne treba da se trudi da posle èitanja ove glave strogo zapamti sintaksu rešenja, niti da otkrije sve pojedinosti koje se kriju iza njih. Cilj je da èitalac samo stekne oseæaj o osnovnim idejama OOP-a i jezika C++, da vidi šta je to novo i šta se sve može uraditi, kao i da proba da sebe "natera" da razmišlja na novi, objektni naèin.

Klase

* *Klasa* (*class*) je osnovna organizaciona jedinica programa u OOPL, pa i u jeziku C++. Klasa predstavlja strukturu u koju su grupisani podaci i funkcije:

```
/* Deklaracija klase: */

class Osoba {
public:
  void koSi();               /* funkcija: predstavi se! */
                             /* ... i još nešto */
private:
  char *ime;                 /* podatak: ime i prezime */
  int   god;                 /* podatak: koliko ima godina */
};


/* Svaka funkcija se mora i definisati: */

void Osoba::koSi () {
  cout<<"Ja sam "<<ime<<" i imam "<<god<<" godina.\n";
}
```

* Klasom se definiše novi, korisnièki tip za koji se mogu kreirati instance (primerci, promenljive).
* Instance klase nazivaju se *objekti* (*objects*). Svaki objekat ima one svoje sopstvene elemente koji su navedeni u deklaraciji klase. Ovi elementi klase nazivaju se *èlanovi* klase (*class members*). Èlanovima se pristupa pomoæu operatora "`.`" (taèka):

```
/* Korišæenje klase Osoba: */
/* negde u programu se definišu promenljive tipa osoba, */

Osoba Pera, mojOtac, direktor;


/* a onda se oni koriste: */

Pera.koSi();        /* poziv funkcije koSi objekta Pera */
mojOtac.koSi();     /* poziv funkcije koSi objekta mojOtac */
direktor.koSi();    /* poziv funkcije ko_si objekta direktor */
```

* Ako pretpostavimo da su ranije, na neki naèin, postavljene vrednosti èlanova svakog od navedenih objekata, ovaj segment programa dâje:

```
Ja sam Petar Markovic i imam 25 godina.
Ja sam Slobodan Milicev i imam 58 godina.
Ja sam Aleksandar Simic i imam 40 godina.
```

* Specifikator `public:` govori prevodiocu da su samo èlanovi koji se nalaze iza njega pristupaèni spolja. Ovi èlanovi nazivaju se *javnim*. Èlanovi iza specifikatora `private:` su nedostupni korisnicima klase (ali ne i èlanovima klase) i nazivaju se *privatnim*:

```
/* Izvan èlanova klase nije moguæe: */

Pera.ime="Petar Markovic";     /* nedozvoljeno */
mojOtac.god=55;                /* takoðe nedozvoljeno */


/* Šta bi tek bilo da je ovo dozvoljeno: */
direktor.ime="bu...., kr...., ...";
direktor.god=1000;
/* a onda ga neko pita (što je dozvoljeno): */
direktor.koSi();
/* ?! */
```

Konstruktori i destruktori

* Da bi se omoguæila inicijalizacija objekta, u klasi se definiše posebna funkcija koja se implicitno (automatski) poziva kada se objekat kreira (definiše). Ova funkcija se naziva *konstruktor* (*constructor*) i nosi isto ime kao i klasa:

```
class Osoba {
public:
  Osoba(char *ime, int godine); /* konstruktor */
  void koSi();                   /* funkcija: predstavi se! */
private:
  char *ime;                    /* podatak: ime i prezime */
  int   god;                    /* podatak: koliko ima godina */
};


/* Svaka funkcija se mora i definisati: */

void Osoba::koSi () {
  cout<<"Ja sam "<<ime<<" i imam "<<god<<" godina.\n";
}

Osoba::Osoba (char *i, int g) {
  if (proveri_ime(i))          /* proveri ime da nije ružno */
    ime=i;
  else
    ime="necu da ti kazem ko";
  god=((g>=0 && g<=100)?g:0); /* proveri godine */
}
```

```
/* Korišæenje klase Osoba sada je: */

Osoba Pera("Petar Markovic",25), /* poziv konstruktora osoba */
      mojOtac("Slobodan Milicev",58);

Pera.koSi();
mojOtac.koSi();
```

* Ovakav deo programa može dati rezultate koji su ranije navedeni.
* Moguæe je definisati i funkciju koja se poziva uvek kada objekar prestaje da živi. Ova funkcija naziva se *destruktor*.

Nasleđivanje

* Pretpostavimo da nam je potreban novi tip, `Maloletnik`. Maloletnik je "jedna vrsta" osobe, odnosno "poseduje sve što i osoba, samo ima još nešto", ima staratelja. Ovakva relacija između klasa naziva se *nasleđivanje*.
* Kada nova klasa predstavlja "jednu vrstu" druge klase (*a-kind-of*), kaže se da je ona izvedena iz osnovne klase:

```
class Maloletnik : public Osoba {
public:
  Maloletnik (char*,char*,int);  /* konstruktor */
  void koJeOdgovoran();
private:
  char *staratelj;
};


void Maloletnik::koJeOdgovoran (){
  cout<<"Za mene odgovara "<<staratelj<<".\n";
}


Maloletnik::Maloletnik (char *i, char *s, int g) : Osoba(i,g), staratelj(s) {}
```

* Izvedena klasa `Maloletnik` ima *sve* èlanove kao i osnovna klasa `Osoba`, ali ima još i èlanove `staratelj` i `koJeOdgovoran`. Konstruktor klase `Maloletnik` definiše da se objekat ove klase kreira zadavanjem imena, staratelja i godina, i to tako da se konstruktor osnovne klase `Osoba` (koji inicijalizuje ime i godine) poziva sa odgovarajuæim argumentima. Sâm konstruktor klase `Maloletnik` samo inicijalizuje staratelja.
* Sada se mogu koristiti i nasleđene osobine objekata klase `Maloletnik`, ali su na raspolaganju i njihova posebna svojstva kojih nije bilo u klasi `Osoba`:

```
Osoba otac("Petar Petrovic",40);
Maloletnik dete("Milan Petrovic","Petar Petrovic",12);

otac.koSi();
dete.koSi();
dete.koJeOdgovoran();
otac.koJeOdgovoran();      /* ovo, naravno, ne može! */
```

```
/* Izlaz æe biti:
Ja sam Petar Petrovic i imam 40 godina.
Ja sam Milan Petrovic i imam 12 godina.
Za mene odgovara Petar Petrovic.
*/
```

Polimorfizam

* Pretpostavimo da nam je potrebna nova klasa žena, koja je "jedna vrsta" osobe, samo što još ima i devojaèko prezime. Klasa `Zena` biæe izvedena iz klase `Osoba`.
* I objekti klase `Zena` treba da se "odazivaju" na funkciju `koSi`, ali je teško pretpostaviti da æe jedna dama otvoreno priznati svoje godine. Zato objekat klase `Zena` treba da ima funkciju `koSi`, samo što æe ona izgledati malo drugaèije, svojstveno izvedenoj klasi `Zena`:

```
class Osoba {
public:
  Osoba(char*,int)            /* konstruktor */
  virtual void koSi();        /* virtuelna funkcija */
protected:                    /* dostupno naslednicima */
  char *ime;                  /* podatak: ime i prezime */
  int   god;                  /* podatak: koliko ima godina */
};

void Osoba::koSi () {
  cout<<"Ja sam "<<ime<<" i imam "<<god<<" godina.\n";
}

Osoba::Osoba (char *i, int g) : ime(i), god(g) {}


class Zena : public osoba {
public:
  Zena(char*,char*,int);
  virtual void koSi();              /* nova verzija funkcije koSi */
private:
  char *devojacko;
};

void Zena::koSi () {
  cout<<"Ja sam "<<ime<<", devojacko prezime "<<devojacko<<".\n";
}

Zena::Zena (char *i, char *d, int g) : Osoba(i,g), devojacko(d) {}
```

* Funkcija èlanica koja æe u izvedenim klasama imati nove verzije deklariše se u osnovnoj klasi kao *virtuelna funkcija* (`virtual`). Izvedena klasa može da dâ svoju definiciju virtuelne funkcije, ali i ne mora. U izvedenoj klasi ne mora se navoditi reè `virtual`.
* Da bi èlanovi osnovne klase `Osoba` bili dostupni izvedenoj klasi `Zena`, ali ne i korisnicima spolja, oni se deklarišu iza specifikatora `protected:` i nazivaju *zaštiæenim* èlanovima.
* Drugi delovi programa, korisnici klase `Osoba`, ako su dobro projektovani, ne moraju da vide ikakvu promenu zbog uvoðenja izvedene klase. Oni uopšte ne moraju da se menjaju:

```
/* Funkcija "ispitaj" propituje osobe i ne mora da se menja: */

void ispitaj (Osoba *hejTi) {
  hejTi->koSi();
}
```

```
/* U drugom delu programa koristimo novu klasu Zena: */

Osoba otac("Petar Petrovic",40);
Zena majka("Milka Petrovic","Mitrovic",35);
Maloletnik dete("Milan Petrovic","Petar Petrovic",12);

ispitaj(&otac);                 /* pozvaæe se Osoba::koSi() */
ispitaj(&majka);                /* pozvaæe se Zena::koSi()  */
ispitaj(&dete);                 /* pozvaæe se Osoba::koSi() */


/* Izlaz æe biti:
Ja sam Petar Petrovic i imam 40 godina.
Ja sam Milka Petrovic, devojacko prezime Mitrovic.
Ja sam Milan Petrovic i imam 12 godina.
*/
```

* Funkcija `ispitaj` dobija pokazivaè na tip `Osoba`. Kako je i žena osoba, C++ dozvoljava da se pokazivaè na tip `Zena` (`&majka`) *konvertuje* (pretvori) u pokazivaè na tip `Osoba` (`hejTi`). Mehanizam virtuelnih funkcija obezbeðuje da funkcija `ispitaj`, preko pokazivaèa `hejTi`, pozove pravu verziju funkcije `koSi`. Zato æe se za argument `&majka` pozivati funkcija `Zena::koSi`, za argument `&otac` funkcija `Osoba::koSi`, a za argument `&dete` takoðe funkcija `Osoba::koSi`, jer klasa `Maloletnik` nije redefinisala virtuelnu funkciju `koSi`.
* Navedeno svojstvo da se odaziva prava verzija funkcije klase èiji su naslednici dali nove verzije naziva se *polimorfizam* (*polymorphism*).

Zadaci:

1. Realizovati klasu `Counter` koja æe imati funkciju `inc`. Svaki objekat ove klase treba da odbrojava pozive svoje funkcije `inc`. Na poèetku svog života, vrednost brojaèa objekta postavlja se na nulu, a pri svakom pozivu funkcije `inc` poveæava se za jedan, i vraæa se novodobijena vrednost.

2. Modifikovati klasu iz prethodnog zadatka, tako da funkcija `inc` ima argument kojim se zadaje vrednost poveæanja brojaèa, i vraæa vrednost brojaèa pre poveæanja. Sastaviti glavni program koji kreira objekte ove klase i poziva njihove funkcije `inc`. Pratiti debagerom stanja svih objekata u *step-by-step* režimu.

3. Skicirati klasu koja predstavlja èlana biblioteke. Svaki èlan biblioteke ima svoj èlanski broj, ime i prezime, i trenutno stanje raèuna za naplatu èlanarine. Ova klasa treba da ima funkciju za naplatu èlanarine, koja æe sa raèuna èlana skinuti odgovarajuæu konstantnu sumu. Biblioteka poseduje i posebnu kategoriju poèasnih èlanova, kojima se ne naplaæuje èlanarina. Kreirati niz pokazivaèa na objekte klase èlanova biblioteke, i definisati funkciju za naplatu èlanarine svim èlanovima. Ova funkcija treba da prolazi kroz niz èlanova i vrši naplatu pozivom funkcije klase za naplatu, bez obzira što se u nizu mogu nalaziti i "obièni" i poèasni èlanovi.

Pregled osnovnih koncepata nasleđenih iz jezika C

* Ovo poglavlje predstavlja pregled nekih osnovnih koncepata jezika C++ nasleđenih iz jezika C kao tradicionalnog jezika za strukturirano programiranje.
* Kao u prethodnom poglavlju, detalji su izostavljeni, a prikazani su samo najvažniji delovi jezika C.

Ugrađeni tipovi i deklaracije

* C++ nije èisti OO jezik: ugrađeni tipovi nisu realizovani kao klase, veæ kao jednostavne strukture podataka.
* *Deklaracija* uvodi neko ime u program. Ime se može koristiti samo ako je prethodno deklarisano. Deklaracija govori prevodiocu kojoj jezièkoj kategoriji neko ime pripada i šta se sa tim imenom može raditi.
* *Definicija* je ona deklaracija koja kreira objekat (alocira memorijski prostor za njega) ili daje telo funkcije.
* Neki osnovni ugrađeni tipovi su: ceo broj (`int`), znak (`char`) i racionalni broj (`float` i `double`). Objekat može biti inicijalizovan u deklaraciji; takva deklaracija je i definicija:

```
int i;
int j=0, k=3;
float f1=2.0, f2=0.0;
double PI=3.14;
char a='a', nul='0';
```

Pokazivaèi

* *Pokazivaè* je objekat koji *ukazuje* na neki drugi objekat. Pokazivaè zapravo sadrži adresu objekta na koji ukazuje.
* Ako pokazivaè `p` ukazuje na objekat `x`, onda izraz `*p` oznaèava objekat `x` (operacija dereferenciranja pokazivaèa).
* Rezultat izraza `&x` je pokazivaè koji ukazuje na objekat `x` (operacija uzimanja adrese).
* Tip "pokazivaè na tip `T`" oznaèava se sa `T*`. Na primer:

```
int i=0, j=0; // objekti i i j tipa int;
int *pi;      // objekat pi je tipa "pokazivaè na int" (tip: int*);
pi=&i;        // vrednost pokazivaèa pi je adresa objekta i,
              // pa pi ukazuje na i;
*pi=2;        // *pi oznaèava objekat i; i postaje 2;
j=*pi;        // j postaje jednak objektu na koji ukazuje pi,
              // a to je i;
pi=&j;        // pi sada sadrži adresu j, tj. ukazuje na j;
```

* Mogu se kreirati pokazivaèi na proizvoljan tip na isti naèin. Ako je `p` pokazivaè koji ukazuje na objekat klase sa èlanom `m`, onda je `(*p).m` isto što i `p->m`:

```
Osoba otac("Petar Simiæ",40);  // objekat otac klase Osoba;
Osoba *po;                     // po je pokazivaè na tip Osoba;
po=&otac;                      // po ukazuje na objekat otac;
(*po).koSi();                  // poziv funkcije koSi objekta otac;
po->koSi();                    // isto što i (*po).koSi();
```

* Tip na koji pokazivaè ukazuje može biti proizvoljan, pa i drugi pokazivaè:

```
int i=0, j=0;  // i i j tipa int;
int *pi=&i;    // pi je pokazivaè na int, ukazuje na i;
int **ppi;     // ppi je tipa "pokazivaè na - pokazivaè na - int";
ppi=&pi;       // ppi ukazuje na pi;
*pi=1;         // pi ukazuje na i, pa i postaje 1;
**ppi=2;       // ppi ukazuje na pi,
               // pa je rezultat operacije *ppi objekat pi;
               // rezultat još jedne operacije * je objekat na koji ukazuje
               // pi, a to je i; i postaje 2;
*ppi=&j;       // ppi ukazuje na pi, pa pi sada ukazuje na j,
               // a ppi još uvek na pi;
ppi=&i;        // greška: ppi je pokazivaè na pokazivaè na int,
               // a ne pokazivaè na int!
```

* Pokazivaè tipa `void*` može ukazivati na objekat bilo kog tipa. Ne postoje objekti tipa `void`, ali postoje pokazivaèi tipa `void*`.
* Pokazivaè koji ima posebnu vrednost `0` ne ukazuje ni na jedan objekat. Ovakav pokazivaè se može razlikovati od bilo kog drugog pokazivaèa koji ukazuje na neki objekat.

Nizovi

* *Niz* je objekat koji sadrži nekoliko objekata nekog tipa. Niz je kao i pokazivaè izvedeni tip. Tip "niz objekata tipa `T`" oznaèava se sa `T[]`.
* Niz se deklariše na sledeæi naèin:

```
int a[100]; // a je objekat tipa "niz objekata tipa int" (tip: int[]);
            // sadrži 100 elemenata tipa int;
```

* Ovaj niz ima 100 elemenata koji se indeksiraju od 0 do 99; `i+1`-vi element je `a[i]`:

```
a[2]=5;    // treæi element niza a postaje 5
a[0]=a[0]+a[99];
```

* Elementi mogu biti bilo kog tipa, pa èak i nizovi. Na ovaj naèin se kreiraju višedimenzionalni nizovi:

```
int m[5][7];// m je niz od 5 elemenata;
            // svaki element je niz od 7 elemenata tipa int;
m[3][5]=0;  // pristupa se èetvrtom elementu niza m;
            // on je niz elemenata tipa int;
            // pristupa se zatim njegovom šestom elementu i on postaje 0;
```

* Nizovi i pokazivaèi su blisko povezani u jezicima C i C++. Sledeæa tri pravila povezuju nizove i pokazivaèe:

1. Svaki put kada se ime niza koristi u nekom izrazu, osim u operaciji uzimanja adrese (operator `&`), implicitno se konvertuje u pokazivaè na svoj prvi element. Na primer, ako je `a` tipa `int[]`, onda se on konvertuje u tip `int*`, sa vrednošæu adrese prvog elementa niza (to je poèetak niza).
2. Definisana je operacija sabiranja pokazivaèa i celog broja, pod uslovom da su zadovoljeni sledeæi uslovi: pokazivaè ukazuje na element nekog niza i rezultat sabiranja je opet pokazivaè koji ukazuje na element istog niza ili za jedno mesto iza poslednjeg elementa niza. Rezultat sabiranja `p+i`, gde je `p` pokazivaè a `i` ceo broj, je pokazivaè koji ukazuje `i` elemenata iza elementa na koji ukazuje pokazivaè `p`. Ako navedeni uslovi nisu zadovoljeni, rezultat operacije je nedefinisan. Analogna pravila postoje za operacije oduzimanja celog broja od pokazivaèa, kao i inkrementiranja i dekrementiranja pokazivaèa.
3. Operacija `a[i]` je po definiciji ekvivalentna sa `*(a+i)`.

Na primer:

```
int a[10]; // a je niz objekata tipa int;
int *p=&a; // p ukazuje na a[0];
a[2]=1;    // a[2] je isto što i *(a+2); a se konvertuje u pokazivaè
           // koji ukazuje na a[0]; rezultat sabiranja je pokazivaè
           // koji ukazuje na a[2]; dereferenciranje tog pokazivaèa (*)
           // predstavlja zapravo a[2]; a[2] postaje 1;
p[3]=3;    // p[3] je isto što i *(p+3), a to je a[3];
p=p+1;     // p sada ukazuje na a[1];
*(p+2)=1;  // a[3] postaje sada 1;
p[-1]=0;   // p[-1] je isto što i *(p-1), a to je a[0];
```

Izrazi

* *Izraz* je iskaz u programu koji sadrži operande (objekte, funkcije ili literale nekog tipa), operacije nad tim operandima i proizvodi rezultat taèno definisanog tipa. Operacije se zadaju pomoæu operatora ugraðenih u jezik.
* Operator može da prihvata jedan, dva ili tri operanda strogo definisanih tipova, i proizvodi rezultat koji se može koristiti kao operand nekog drugog operatora. Na ovaj naèin se formiraju složeni izrazi.
* Prioritet operatora definiše redosled izraèunavanja operacija unutar izraza. Podrazumevani redosled izraèunavanja može se promeniti pomoæu zagrada `()`.
* C i C++ su prebogati operatorima. Zapravo najveæi deo obrade u jednom programu predstavljaju izrazi.
* Mnogi ugraðeni operatori imaju sporedni efekat: pored toga što proizvode rezultat, oni menjaju vrednost nekog od svojih operanada.
* Postoje operatori za inkrementiranje (`++`) i dekrementiranje (`--`), u prefiksnoj i postfiksnoj formi. Ako je `i` nekog od numerièkih tipova ili pokazivaè, `i++` znaèi "inkrementiraj `i`, a kao rezultat vrati njegovu staru vrednost"; `++i` znaèi "inkrementiraj `i` a kao rezultat vrati njegovu novu vrednost". Analogno važi za dekrementiranje.
* Dodela vrednosti se vrši pomoæu operatora dodele `=`: `a=b` znaèi "dodeli vrednost izraza `b` objektu `a`, a kao rezultat vrati tu dodeljenu vrednost". Ovaj operator grupiše sdesna ulevo. Tako:

```
a=b=c; // dodeli c objektu b i vrati tu vrednost; zatim dodeli tu vrednost u a;
       // prema tome, c je dodeljen i objektu b i objektu a;
```

* Postoji i operator složene dodele: `a+=b` znaèi isto što i `a=a+b`, samo što se izraz `a` samo jednom izraèunava:

```
a+=b;    // isto što i a=a+b;
a-=b;    // isto što i a=a-b;
a*=b;    // isto što i a=a*b;
a/=b;    // isto što i a=a/b;
```

Naredbe

* *Naredba* podrazumeva neku obradu ali ne proizvodi rezultat kao izraz. Postoji samo nekoliko naredbi u jezicima C i C++.
* Deklaracija se sintaksno smatra naredbom. Izraz je takoðe jedna vrsta naredbe. *Složena naredba* (ili *blok*) je sekvenca naredbi uokvirena u velike zagrade `{}`. Na primer:

```
{                     // poèetak složene naredbe (bloka);
  int a, c=0, d=3;    // deklaracija kao naredba;
  a=(c++)+d;          // izraz kao naredba;
  int i=a;            // deklaracija kao naredba;
  i++;                // izraz kao naredba;
}                     // kraj složene naredbe (bloka);
```

* *Uslovna naredba* (*if* naredba): `if (`*izraz*`)` *naredba* `else` *naredba*. Prvo se izraèunava *izraz*; njegov rezultat mora biti numerièkog tipa ili pokazivaè; ako je rezultat razlièit od nule (što se tumaèi kao "taèno"), izvršava se prva

naredba; inaèe, ako je rezultat jednak nuli (što se tumaèi kao "netaèno"), izvršava se druga *naredba* (*else* deo). Deo *else* je opcioni:

```
if (a++) b=a;  // inkrementiraj a; ako je a bilo razlièito od 0,
               // dodeli novu vrednost a objektu b;

if (c) a=c;    // ako je c razlièito od 0, dodeli ga objektu a,
else a=c+1;    // inaèe dodeli c+1 objektu a;
```

* Petlja (*for* naredba): `for (`*inicijalna_naredba izraz1*`;` *izraz2*`)` *naredba*. Ovo je petlja sa izlaskom na vrhu (petlja tipa *while*). Prvo se izvršava *inicijalna_naredba* samo jednom pre ulaska u petlju. Zatim se izvršava petlja. Pre svake iteracije izraèunava se *izraz1*; ako je njegov rezultat jednak nuli, izlazi se iz petlje; inaèe, izvršava se iteracija petlje. Iteracija se sastoji od izvršavanja *naredbe* i zatim izraèunavanja *izraza2*. Oba izraza i *inicijalna_naredba* su opcioni; ako se izostavi, uzima se da je vrednost *izraza1* jednaka 1. Na primer:

```
for (int i=0; i<100; i++) {
  //... Ova petlja se izvršava taèno 100 puta
}

for (;;) {
  //... Beskonaèna petlja
}
```

Funkcije

* Funkcije su jedina vrsta potprograma u jezicima C i C++. Funkcije mogu biti èlanice klase ili globalne funkcije (nisu èlanice nijedne klase).
* Ne postoji statièko (sintaktièko) ugnežðivanje tela funkcija. Dinamièko ugnežðivanje poziva funkcija je dozvoljeno, pa i rekurzija.
* Funkcija može, ali ne mora da ima argumente. Funkcija bez argumenata se deklariše sa praznim zagradama. Argumenti se prenose samo po vrednostima u jeziku C, a mogu se prenositi i po referenci u jeziku C++.
* Funkcija može, ali ne mora da vraæa rezultat. Funkcija koja nema povratnu vrednost deklariše se sa tipom `void` kao tipom rezultata.
* Deklaracija funkcije koja nije i definicija uključuje samo zaglavlje sa tipom argumenata i rezultata; imena argumenata su opciona i nemaju znaèaja za program:

```
int stringCompare (char*,char*);  // deklaracija globalne funkcije;
                                  // prima dva argumenta tipa char*,
                                  // a vraæa tip int;
void f();                         // globalna funkcija bez argumenata
                                  // koja nema povratnu vrednost;
```

* Definicija funkcije daje i telo funkcije. Telo funkcije je složena naredba (blok):

```
int Counter::inc () {  // definicija funkcije èlanice; vraæa int;
  return count++;      // vraæa se rezultat izraza;
}
```

* Funkcija može vratiti vrednost koja je rezultat izraza u naredbi `return`.
* Mogu se definisati lokalna imena unutar tela funkcije (taènije unutar svakog ugnežðenog bloka):

```
int Counter::inc () {
  int temp;       // temp je lokalni objekat
  temp=count+1;   // count je èlan klase Counter
  count=temp;
  return temp;
}
```

* Funkcija èlanica neke klase može pristupati èlanovima sopstvenog objekta bez posebne specifikacije. Globalna funkcija mora specifikovati objekat èijem èlanu pristupa.
* Poziv funkcije obavlja se pomoæu operatora `()`. Rezultat ove operacije je rezultat poziva funkcije:

```
int f(int);     // deklaracija globalne funkcije
Counter c;      // objekat c klase Counter
int a=0, b=1;
a=b+c.inc();    // poziv funkcije c.inc koji vraæa int
a=f(b);         // poziv globalne funkcije f
```

* Može se deklarisati i pokazivaè na funkciju:

```
int f(int);      // f je tipa "funkcija koja prima jedan argument tipa int
                 // i vraæa int";
int (*p)(int);   // p je tipa
                 // "pokazivaè na funkciju
                 // koja prima jedan argument tipa int
                 // i vraæa int";
p=&f;            // p ukazuje na f;
int a;
a=(*p)(1);       // poziva se funkcija na koju ukazuje p, a to je funkcija f;
```

Struktura programa

* Program se sastoji samo od deklaracija (klasa, objekata, ostalih tipova i funkcija). Sva obrada koncentrisana je unutar tela funkcija.
* Program se fizièki deli na odvojene jedinice prevoðenja - datoteke. Datoteke se prevode odvojeno i nezavisno, a zatim se povezuju u izvršni program. U svakoj datoteci se moraju deklarisati sva imena pre nego što se koriste.
* Zavisnosti između modula - datoteka definišu se pomoæu *datoteka-zaglavlja*. Zaglavlja sadrže deklaracije svih entiteta koji se koriste u datom modulu, a definisani su u nekom drugom modulu. Zaglavlja (`.h`) se uključuju u tekst datoteke koja se prevodi (`.cpp`) pomoæu direktive `#include`.
* Glavni program (izvor toka kontrole) definiše se kao obavezna funkcija `main`. Primer jednog jednostavnog, ali kompletnog programa:

```
class Counter {
public:
  Counter();
  int inc(int by);
private:
  int count;
};


Counter::Counter () : count(0) {}

int Counter::inc (int by) {
  return count+=by;
}


void main () {
  Counter a,b;
  int i=0, j=3;
  i=a.inc(2)+b.inc(++j);
}
```

Elementi jezika C++ koji nisu objektno orijentisani

Oblast važenja imena

* Oblast važenja imena je onaj deo teksta programa u kome se deklarisano ime može koristiti.
* Globalna imena su imena koja se deklarišu van svih funkcija i klasa. Njihova oblast važenja je deo teksta od mesta deklaracije do kraja datoteke.
* Lokalna imena su imena deklarisana unutar bloka, uključujući i blok tela funkcije. Njihova oblast važenja je od mesta deklarisanja, do završetka bloka u kome su deklarisane.

```
int x;              // globalni x

void f () {
  int x;            // lokalni x, sakriva globalni x;
  x=1;              // pristup lokalnom x
  {
    int x;          // drugi lokalni x, sakriva prethodnog
    x=2;            // pristup drugom lokalnom x
  }
  x=3;              // pristup prvom lokalnom x
}

int *p=&x;          // uzimanje adrese globalnog x
```

* Globalnom imenu se može pristupiti, iako je sakriveno, navođenjem operatora "::" ispred imena:

```
int x;              // globalni x

void f () {
  int x=0;          // lokalni x
  ::x=1;            // pristup globalnom x;
}
```

* Za formalne argumente funkcije smatra se da su lokalni, deklarisani u krajnje spoljašnjem bloku tela funkcije:

```
void f (int x) {
  int x;      // pogrešno
}
```

* Prvi izraz u naredbi `for` može da bude definicija promenljive. Tako se dobija lokalna promenljiva za blok u kome se nalazi `for`:

```
{
  for (int i=0; i<10; i++) {
    //...
    if (a[i]==x) break;
    //...
  }
  if (i==10) // može se pristupati imenu i
}
```

* Oblast važenja klase imaju svi članovi klase. To su imena deklarisana unutar deklaracije klase. Imenu koje ima oblast važenja klase, van te oblasti, može se pristupiti preko operatora "`.`" i "`->`", gde je levi operand objekat, odnosno pokazivač na objekat date klase ili klase izvedene iz date klase, ili preko operatora "`::`", gde je levi operand ime klase:

```
class X {
public:
  int x;
  void f();
};

void X::f () {/*...*/}
X xx;
xx.x=0;
xx.X::f();  // može i ovako
```

* Oblast važenja funkcije imaju samo labele (za `goto` naredbe). One se mogu navesti bilo gde (i samo) unutar tela funkcije, a vide se u celoj funkciji.

Objekti i lvrednosti

* Objekat je neko područje u memoriji podataka, u toku izvršavanja programa. To može biti promenljiva (globalna ili lokalna), privremeni objekat koji se kreira pri izračunavanja izraza, ili jednostavno memorijska lokacija na koju pokazuje neki pokazivač. Uopšte, objekat je primerak nekog tipa (ugrađenog ili klase), ali ne i funkcija.
* Samo nekonstantni objekat se u jeziku C++ naziva promenljivom.
* *lvrednost* (*lvalue*) je izraz koji upućuje na objekat. *lvalue* je kovanica od "nešto što može da stoji sa leve strane znaka dodele vrednosti", iako ne mogu sve lvrednosti da stoje sa leve strane znaka =, npr. konstanta.
* Za svaki operator se definiše da li zahteva kao operand lvrednost, i da li vraća lvrednost kao rezultat. "Početna" lvrednost je ime objekta ili funkcije. Na taj način se rekurzivno definišu lvrednosti.
* Promenljiva lvrednost (*modifiable lvalue*) je ona lvrednost, koja nije ime funkcije, ime niza, ili konstantni objekat. Samo ovakva lvrednost može biti levi operand operatora dodele.
* Primeri lvrednosti:

```
int i=0;     // i je lvrednost, jer je ime koje upućuje
             // na objekat - celobrojnu promenljivu u memoriji

int *p=&i;  // i p je ime, odnosno lvrednost

*p=7;        // *p je lvrednost, jer upućuje na objekat koga
             // predstavlja ime i; rezultat operacije * je
             // lvrednost

int *q[100];
*q[a+13]=7; // *q[a+13] je lvrednost
```

Životni vek objekata

* Životni vek objekta je vreme u toku izvršavanja programa za koje taj objekat postoji (u memoriji), i za koje mu se može pristupati.
* Na početku životnog veka, objekat se kreira (poziva se njegov konstruktor ako ga ima), a na kraju se objekat ukida (poziva se njegov destruktor ako ga ima). Sinonim za kreiranje objekta je inicijalizacija objekta.

```
int glob=1;        // globalni objekat; životni vek mu
                   // je do kraja programa;

void f () {
  int lok=2;       // lokalni objekat; životni vek mu je do
                   // izlaska iz spoljnjeg bloka funkcije;
  static int sl=3;// lokalni statièki objekat; oblast
                   // važenja je funkcija, a životni vek je ceo
                   // program; inicijalizuje se samo jednom;
  for (int i=0; i<sl; i++) {
    int j=i;       // j je lokalni za for blok
    //...
  }
}
```

* U odnosu na životni vek, postoje automatski, statièki, dinamièki i privremeni objekti.
* Životni vek *automatskog* objekta (lokalni objekat koji nije deklarisan kao `static`) traje od nailaska na njegovu definiciju, do napuštanja oblasti važenja tog objekta. Automatski objekat se kreira iznova pri svakom pozivu bloka u kome je deklarisan. Definicija objekta je izvršna naredba.
* Životni vek *statièkih* objekata (globalni i lokalni `static` objekti) traje od izvršavanja njihove definicije do kraja izvršavanja programa. Globalni statièki objekti se kreiraju samo jednom, na poèetku izvršavanja programa, pre korišæenja bilo koje funkcije ili objekta iz istog fajla, ne obavezno pre poziva funkcije `main`, a prestaju da žive po završetku funkcije `main`. Lokalni statièki objekti poènju da žive pri prvom nailasku toka programa na njihovu definiciju.
* Životni vek *dinamièkih* objekata neposredno kontroliše programer. Oni se kreiraju operatorom `new`, a ukidaju operatorom `delete`.
* Životni vek *privremenih* objekata je kratak i nedefinisan. Ovi objekti se kreiraju pri izraèunavanju izraza, za odlaganje meðurezultata ili privremeno smeštanje vraæene vrednosti funkcije. Najèešæe se uništavaju èim više nisu potrebni.
* Životni vek èlanova klase je isti kao i životni vek objekta kome pripadaju.
* Formalni argumenti funkcije se, pri pozivu funkcije, kreiraju kao automatski lokalni objekti i inicijalizuju se stvanim argumentima. Semantika inicijalizacije formalnog argumenta je ista kao i inicijalizacija objekta u definiciji.
* Primer:

```
int a=1;

void f () {
  int b=1;          // inicijalizuje se pri svakom pozivu
  static int c=1;   // inicijalizuje se samo jednom
  printf(" a = %d ",a++);
  printf(" b = %d ",b++);
  printf(" c = %d\n",c++);
}

void main () {
  while (a<4) f();
}

// izlaz æe biti:
// a = 1 b = 1 c = 1
// a = 2 b = 1 c = 2
// a = 3 b = 1 c = 3
```

O konverziji tipova

* C++ je strogo tipizirani jezik, što je u duhu njegove objektne orjentacije.
* Tipizacija znaèi da svaki objekat ima svoj taèno odreðeni tip. Svaki put kada se na nekom mestu oèekuje objekat jednog tipa, a koristi se objekat drugog tipa, potrebno je izvršiti *konverziju* tipova.
* Konverzija tipa znaèi pretvaranje objekta datog tipa u objekat potrebnog tipa.

* Sluèajevi kada se može desiti da se oèekuje jedan tip, a dostavlja se drugi, odnosno kada je potrebno vršiti konverziju su:
1. operatori za ugrađene tipove zahtevaju operande odgovarajuæeg tipa;
2. neke naredbe (`if`, `for`, `do`, `while`, `switch`) zahtevaju izraze odgovarajuæeg tipa;
3. pri pozivu funkcije, kada su stvarni argumenti drugaèijeg tipa od deklarisanih formalnih argumenata; i operatori za korisnièke tipove (klase) su specijalne vrste funkcija;
4. pri povratku iz funkcije, ako se u izrazu iza `return` koristi izraz drugaèijeg tipa od deklarisanog tipa povratne vrednosti funkcije;
5. pri inicijalizaciji objekta jednog tipa pomoæu objekta drugog tipa; sluèaj pod 3 se može svesti u ovu grupu, jer se formalni argumenti inicijalizuju stvarnim argumentima pri pozivu funkcije; takoðe, sluèaj pod 4 se može svesti u ovu grupu, jer se privremeni objekat, koji prihvata vraæenu vrednost funkcije na mestu poziva, inicijalizuje izrazom iza naredbe `return`.

* Konverzija tipa može biti ugrađena u jezik (standardna konverzija) ili je definiše korisnik (programer) za svoje tipove (korisnièka konverzija).
* Standardne konverzije su, na primer, konverzije iz tipa `int` u tip `float`, ili iz tipa `char` u tip `int` itd.
* Prevodilac može sam izvršiti konverziju koja mu je dozvoljena, na mestu gde je to potrebno; ovakva konverzija naziva se *implicitnom*. Programer može eksplicitno navesti koja konverzija treba da se izvrši; ova konverzija naziva se *eksplicitnom*.
* Jedan naèin zahtevanja eksplicitne konverzije je pomoæu operatora *cast*: `(`*tip*`)` *izraz*.
* Primer:

```
char f(float i, float j) {
  //...
}

int k=f(5.5,5);  // najpre se vrši konverzija float(5),
                 // a posle i konverzija vraæene vrednosti
                 // iz char u int
```

Konstante

* Konstantni tip je izvedeni tip koji se iz nekog osnovnog tipa dobija stavljanjem specifikatora `const` u deklaraciju:

```
const float pi=3.14;
const char plus='+';
```

* Konstantni tip ima sve osobine osnovnog tipa, samo se objekti konstantnog tipa ne mogu menjati. Pristup konstantama kontroliše se u fazi prevoðenja, a ne izvršavanja.
* Konstanta mora da se inicijalizuje pri definisanju.
* Prevodilac èesto ne odvaja memorijski prostor za konstantu, veæ njeno korišæenje razrešava u doba prevoðenja.
* Konstante mogu da se koriste u konstantnim izrazima koje prevodilac treba da izraèuna u toku prevoðenja, na primer kao dimenzije nizova.
* Pokazivaè na konstantu definiše se stavljanjem reèi `const` ispred cele definicije. Konstantni pokazivaè definiše se stavljanjem reèi `const` ispred samog imena:

```
const char *pk="asdfgh";          // pokazivaè na konstantu
pk[3]='a';                        // pogrešno
pk="qwerty";                      // ispravno

char *const kp="asdfgh";          // konstantni pokazivaè
kp[3]='a';                        // ispravno
kp="qwerty";                      // pogrešno

const char *const kpk="asdfgh";  // konst. pokazivaè na konst.
kpk[3]='a';                       // pogrešno
kpk="qwerty";                     // pogrešno
```

* Navoðenjem reèi `const` ispred deklaracije formalnog argumenta funkcije koji je pokazivaè, obezbeðuje se da funkcija ne može menjati objekat na koji taj argument ukazuje:

```
char *strcpy(char *p, const char *q); // ne može da promeni *q
```

* Navodjenjem reèi `const` ispred tipa koji vraæa funkcija, definiše se da æe privremeni objekat koji se kreira od vraæene vrednosti funkcije biti konstantan, i njegovu upotrebu kontroliše prevodilac. Za vraæenu vrednost koja je pokazivaè na konstantu, ne može se preko vraæenog pokazivaèa menjati objekat:

```
const char* f();
*f()='a';         // greška!
```

* Preporuka je da se umesto tekstualnih konstanti koje se ostvaruju pretprocesorom (kao u jeziku C) koriste konstante na opisani naèin.
* Dosledno korišæenje konstanti u programu obezbeðuje podršku prevodioca u spreèavanju grešaka - korektnost konstantnosti.

Dinamièki objekti

* Operator `new` kreira jedan dinamièki objekat, a operator `delete` ukida dinamièki objekat nekog tipa `T`.
* Operator `new` za svoj argument ima identifikator tipa i eventualne argumente konstruktora. Operator `new` alocira potreban prostor u slobodnoj memoriji za objekat datog tipa, a zatim poziva konstruktor tipa sa zadatim vrednostima. Operator `new` vraæa pokazivaè na dati tip:

```
complex *pc1 = new complex(1.3,5.6),
        *pc2 = new complex(-1.0,0);

*pc1=*pc1+*pc2;
```

* Objekat kreiran pomoæu operatora `new` naziva se dinamièki objekat, jer mu je životni vek poznat tek u vreme izvršavanja. Ovakav objekat nastaje kada se izvrši operator `new`, a traje sve dok se ne oslobodi operatorom `delete` (može da traje i po završetku bloka u kome je kreiran):

```
complex *pc;

void f() {
  pc=new complex(0.1,0.2);
}

void main () {
  f();
  delete pc;    // ukidanje objekta *pc
}
```

* Operator `delete` ima jedan argument koji je pokazivaè na neki tip. Ovaj pokazivaè mora da ukazuje na objekat kreiran pomoæu operatora `new`. Operator `delete` poziva destruktor za objekat na koji ukazuje pokazivaè, a zatim oslobaða zauzeti prostor. Ovaj operator vraæa `void`.
* Operatorom `new` može se kreirati i niz objekata nekog tipa. Ovakav niz ukida se operatorom `delete` sa parom uglastih zagrada:

```
comlex *pc = new complex[10];
//...
delete [] pc;
```

* Kada se alocira niz, nije moguæe zadati inicijalizatore. Ako klasa nema definisan konstruktor, prevodilac obezbeðuje podrazumevanu inicijalizaciju. Ako klasa ima konstruktore, da bi se alocirao niz potrebno je da postoji konstruktor koji se može pozvati bez argumenata.
* Kada se alocira niz, operator `new` vraæa pokazivaè na prvi element alociranog niza. Sve dimenzije niza osim prve treba da budu konstantni izrazi, a prva dimenzija može da bude i promenljivi izraz, ali takav da može da se izraèuna u trenutku izvršavanja naredbe sa operatorom `new`.

Reference

* U jeziku C prenos argumenata u funkciju bio je iskljuèivo po vrednosti (*call by value*). Da bi neka funkcija mogla da promeni vrednost neke spoljne promenljive, trebalo je preneti pokazivaè na tu promenljivu.
* U jeziku C++ moguæ je i prenos po referenci (*call by reference*):

```
void f(int i, int &j) {
                    // i se prenosi po vrednosti, j po referenci
  i++;              // stvarni argument se neæe promeniti
  j++;              // stvarni argument æe se promeniti
}


void main () {
  int si=0,sj=0;
  f(si,sj);
  cout<<"si="<<si<<", sj="<<sj<<"\n";
}


// Izlaz æe biti:
// si=0, sj=1
```

* C++ ide još dalje, postoji izvedeni tip *reference* na objekat (*reference type*). Reference se deklarišu upotrebom znaka `&` ispred imena.
* Referenca je alternativno ime za neki objekat. Kada se definiše, referenca mora da se inicijalizuje nekim objektom na koga æe upuæivati. Od tada referenca postaje sinonim za objekat na koga upuæuje i svaka operacija nad referencom (uključujuæi i operaciju dodele) je ustvari operacija nad referenciranim objektom:

```
int i=1;               // celobrojni objekat i
int &j=i;              // j upuæuje na i
i=3;                   // menja se i
j=5;                   // opet se menja i
int *p=&j;             // isto što i &i
j+=1;                  // isto što i i+=1
int k=j;               // posredan pristup do i preko reference
int m=*p;              // posredan pristup do i preko pokazivaèa
```

* Referenca se realizuje kao (konstantni) pokazivaè na objekat. Ovaj pokazivaè pri inicijalizaciji dobija vrednost adrese objekta kojim se inicijalizuje. Svako dalje obraæanje referenci podrazumeva posredni pristup objektu preko ovog pokazivaèa. Nema naèina da se, posle inicijalizacije, vrednost ovog pokazivaèa promeni.
* Referenca lièi na pokazivaè, ali se posredan pristup preko pokazivaèa na objekat vrši operatorom `*`, a preko reference bez oznaka. Uzimanje adrese (operator `&`) reference znaèi uzimanje adrese objekta na koji ona upuæuje.
* Primeri:

```
int &j = *new int(2);  // j upuæuje na dinamièki objekat 2
int *p=&j;             // p je pokazivaè na isti objekat
(*p)++;                // objekat postaje 3
j++;                   // objekat postaje 4
delete &j;             // isto kao i delete p
```

* Ako je referenca tipa reference na konstantu, onda to znaèi da se referencirani objekat ne sme promeniti posredstvom te reference.
* Referenca može i da se vrati kao rezultat funkcije. U tom sluèaju funkcija treba da vrati referencu na objekat koji traje (živi) i posle izlaska iz funkcije, da bi se mogla koristiti ta referenca:

```
// Može ovako:
int& f(int &i) {
  int &r=*new int(1);
  //...
  return r;  // pod uslovom da nije bilo delete &r
}

// ili ovako:
int& f(int &i) {
  //...
  return i;
}

// ali ne može ovako:
int& f(int &i) {
  int r=1;
  //...
  return r;
}

// niti ovako:
int& f(int i) {
  //...
  return i;
}

// niti ovako:
int& f(int &i) {
  int r=*new int(1);
  //...
  return r;
}
```

* Prilikom poziva funkcije, kreiraju se objekti koji predstavljaju formalne argumente i inicijalizuju se stvarnim argumentima (semantika je ista kao i pri definisanju objekta sa inicijalizacijom). Prilikom povratka iz funkcije, kreira se privremeni objekat koji se inicijalizuje objektom koji se vraæa, a zatim se koristi u izrazu iz koga je funkcija pozvana.
* Rezultat poziva funkcije je lvrednost samo ako funkcija vraæa referencu.
* Ne postoje nizovi referenci, pokazivaèi na reference, ni reference na reference.

Funkcije

Deklaracije funkcija i prenos argumenata

* Funkcije se deklarišu i definišu kao i u jeziku C, samo što je moguæe kao tipove argumenata i rezultata navesti korisnièke tipove (klase).
* U deklaraciji funkcije ne moraju da se navode imena formalnih argumenata.
* Pri pozivu funkcije, upoređuju se tipovi stvarnih argumenata sa tipovima formalnih argumenata navedenim u deklaraciji, i po potrebi vrši konverzija. Semantika prenosa argumenata jednaka je semantici inicijalizacije.
* Pri pozivu funkcije, inicijalizuju se formalni argumenti, kao automatski lokalni objekti pozvane funkcije. Ovi objekti se konstruišu pozivom odgovarajuæih konstruktora, ako ih ima. Pri vraæanju vrednosti iz funkcije, semantika je ista: konstruiše se privremeni objekat koji prihvata vraæenu vrednost na mestu poziva:

```
class Tip {
//...
public:
  Tip(int i);              // konstruktor
};

Tip f (Tip k) {
  //...
  return 2;                // poziva se konstruktor Tip(2)
}

void main () {
  Tip k(0);
  k=f(1);                  // poziva se konstruktor Tip(1)
  //...
}
```

Neposredno ugraðivanje u kôd

* Èesto se definišu vrlo jednostavne, kratke funkcije (na primer samo presleðuju argumente drugim funkcijama). Tada je vreme koje se troši na prenos argumenata i poziv veæe nego vreme izvršavanja tela same funkcije.
* Ovakve funkcije se mogu deklarisati tako da se neposredno ugraðuju u kôd (*inline* funkcije). Tada se telo funkcije direktno ugraðuje u pozivajuæi kôd. Semantika poziva ostaje potpuno ista kao i za obiènu funkciju.
* Ovakva funkcija deklariše se kao `inline`:

```
inline int inc(int i) {return i+1;}
```

* Funkcija èlanica klase može biti *inline* ako se definiše unutar deklaracije klase, ili izvan deklaracije klase, kada se ispred njene deklaracije nalazi reè `inline`:

```
class C {
  int i;
public:
  int val () {return i;}  // ovo je inline funkcija
};

// ili:

class D {
  int i;
public:
  int val ();
};

inline int D::val() {return i;}
```

* Prevodilac ne mora da ispoštuje zahtev za neposredno ugraðivanje u kôd. Za korisnika ovo ne treba da predstavlja nikakvu prepreku, jer je semantika ista. *Inline* funkcije samo mogu da ubrzaju program, a nikako da izmene njegovo izvršavanje.
* Ako se inline funkcija koristi u više datoteka, u svakoj datoteci mora da se naðe njena potpuna definicija (najbolje pomoæu datoteke-zaglavlja).

Podrazumevane vrednosti argumenata

* C++ obezbeðuje i moguænost postavljanja podrazumevanih vrednosti za argumente. Ako se pri pozivu funkcije ne navede argument za koji je definisana podrazumevana vrednost (u deklaraciji funkcije), kao vrednost stvarnog argumenta uzima se ta podrazumevana vrednost:

```
complex::complex (float r=0, float i=0) // podrazumevana
  {real=r; imag=i;}                     // vrednost za r i i je 0

void main () {
  complex c; // kao da je napisano "complex c(0,0);"
  //...
}
```

* Podrazumevani argumenti mogu da budu samo nekoliko poslednjih iz liste:

```
complex::complex(float r=0, float i)  // greška
  { real=r; imag=i; }
```

Preklapanje imena funkcija

* Èesto se javlja potreba da se u programu naprave funkcije koje realizuju logièki istu operaciju, samo sa razlièitim tipovima argumenata. Za svaki od tih tipova mora, naravno, da se realizuje posebna funkcija. U jeziku C to bi moralo da se realizuje tako da te funkcije imaju razlièita imena. To, međutim, smanjuje èitljivost programa.
* U jeziku C++ moguæe je definisati više razlièitih funkcija sa istim identifikatorom. Ovakav koncept naziva se *preklapanje imena funkcija* (engl. *function overloading*). Uslov je da im se razlikuje broj i/ili tipovi argumenata. Tipovi rezultata ne moraju da se razlikuju:

```
char* max (const char *p, const char *q)
    { return (strcmp(p,q)>=0)?p:q; }
double max (double i, double j) { return (i>j) ? i : j; }


double r=max(1.5,2.5);          // poziva se max(double,double)
char *q=max("Pera","Mika");     // poziva se max(const char*,const char*)
```

* Koja æe se funkcija stvarno pozvati, određuje se u fazi prevođenja prema slaganju tipova stvarnih i formalnih argumenata. Zato je potrebno da prevodilac može jednoznaèno da odredi koja funkcija se poziva.
* Pravila za razrešavanje poziva su veoma složena [ARM, Miliæev95], pa se u praksi svode samo na dovoljno razlikovanje tipova formalnih argumenata preklopljenih funkcija. Kada razrešava poziv, prevodilac otprilike ovako prioritira slaganje tipova stvarnih i formalnih argumenata:

1. najbolje odgovara potpuno slaganje tipova; tipovi `T*` (pokazivaè na `T`) i `T[]` (niz elemenata tipa `T`) se ne razlikuju;
2. sledeæe po odgovaranju je slaganje tipova korišæenjem standardnih konverzija;
3. sledeæe po odgovaranju je slaganje tipova korišæenjem korisnièkih konverzija;
4. najlošije odgovara slaganje sa tri taèke (`...`).

Operatori i izrazi

* Pregled operatora dat je u sledeæoj tabeli. Operatori su grupisani po prioritetima, tako da su operatori u istoj grupi istog prioriteta, višeg od operatora koji su u narednoj grupi. U tablici su prikazane i ostale važne osobine: naèin grupisanja (asocijativnost, L - sleva udesno, D - sdesna ulevo), da li je rezultat lvrednost (D - da, N - nije, D/N - zavisi od nekog operanda, pogledati specifikaciju operatora u [ARM, Miliæev95]), kao i naèin upotrebe. Prazna polja ukazuju da svojstvo grupisanja nije primereno datom operatoru.

Operator	Znaèenje	Grup.	lvred.	Upotreba
`::`	razrešavanje oblasti važenja	L	D/N	*ime_klase* `::` *èlan*
`::`	pristup globalnom imenu		D/N	`::` *ime*
`[]`	indeksiranje	L	D	*izraz*`[`*izraz*`]`
`()`	poziv funkcije	L	D/N	*izraz* (*lista_izraza*)
`()`	konstrukcija vrednosti		N	*ime_tipa* (*lista_izraza*)
`.`	pristup èlanu	L	D/N	*izraz* `.` *ime*

`->`	posredni pristup èlanu	L	D/N	*izraz* `->` *ime*
`++`	postfiksni inkrement	L	N	*lvrednost*`++`
`--`	postfiksni dekrement	L	N	*lvrednost*`--`
`++`	prefiksni inkrement	D	D	`++`*lvrednost*
`--`	prefiksni dekrement	D	D	`--`*lvrednost*
`sizeof`	velièina objekta	D	N	`sizeof` *izraz*
`sizeof`	velièina tipa	D	N	`sizeof(`*tip*`)`
`new`	kreiranje dinamièkog objekta		N	`new` *tip*
`delete`	ukidanje dinamièkog objekta		N	`delete` *izraz*
`~`	komplement po bitima	D	N	`~`*izraz*
`!`	logièka negacija	D	N	`!`*izraz*
`-`	unarni minus	D	N	`-`*izraz*
`+`	unarni plus	D	N	`+`*izraz*
`&`	adresa	D	N	`&`*lvrednost*
`*`	dereferenciranje pokazivaèa	D	D	`*`*izraz*
`()`	konverzija tipa (*cast*)	D	D/N	`(`*tip*`)` *izraz*
`.*`	posredni pristup èlanu	L	D/N	*izraz* `.*` *izraz*
`->*`	posredni pristup èlanu	L	D/N	*izraz* `->*` *izraz*
`*`	množenje	L	N	*izraz* `*` *izraz*
`/`	deljenje	L	N	*izraz* `/` *izraz*
`%`	ostatak	L	N	*izraz* `%` *izraz*
`+`	sabiranje	L	N	*izraz* `+` *izraz*
`-`	oduzimanje	L	N	*izraz* `-` *izraz*
`<<`	pomeranje ulevo	L	N	*izraz* `<<` *izraz*
`>>`	pomeranje udesno	L	N	*izraz* `>>` *izraz*
`<`	manje od	L	N	*izraz* `<` *izraz*
`<=`	manje ili jednako od	L	N	*izraz* `<=` *izraz*
`>`	veæe od	L	N	*izraz* `>` *izraz*
`>=`	veæe ili jednako od	L	N	*izraz* `>=` *izraz*
`==`	jednako	L	N	*izraz* `==` *izraz*
`!=`	nije jednako	L	N	*izraz* `!=` *izraz*
`&`	I po bitima	L	N	*izraz* `&` *izraz*
`^`	isključèivo ILI po bitima	L	N	*izraz* `^` *izraz*
`|`	ILI po bitovima	L	N	*izraz* `|` *izraz*
`&&`	logièko I	L	N	*izraz* `&&` *izraz*
`||`	logièko ILI	L	N	*izraz* `||` *izraz*
`? :`	uslovni operator	L	D/N	*izraz* `?` *izraz* `:` *izraz*
`=`	prosto dodeljivanje	D	D	*lvrednost* `=` *izraz*
`*=`	množenje i dodela	D	D	*lvrednost* `*=` *izraz*
`/=`	deljenje i dodela	D	D	*lvrednost* `/=` *izraz*
`%=`	ostatak i dodela	D	D	*lvrednost* `%=` *izraz*
`+=`	sabiranje i dodela	D	D	*lvrednost* `+=` *izraz*
`-=`	oduzimanje i dodela	D	D	*lvrednost* `-=` *izraz*
`>>=`	pomeranje udesno i dodela	D	D	*lvrednost* `>>=` *izraz*
`<<=`	pomeranje ulevo i dodela	D	D	*lvrednost* `<<=` *izraz*
`&=`	I i dodela	D	D	*lvrednost* `&=` *izraz*
`|=`	ILI i dodela	D	D	*lvrednost* `|=` *izraz*
`^=`	isključèivo ILI i dodela	D	D	*lvrednost* `^=` *izraz*
`,`	sekvenca	L	D/N	*izraz* `,` *izraz*

Zadaci:

4. Realizovati funkciju `strclone` koja prihvata pokazivaè na znakove kao argument, i vrši kopiranje niza znakova na koji ukazuje taj argument u dinamièki niz znakova, kreiran u dinamièkoj memoriji, na koga æe ukazivati pokazivaè vraæen kao rezultat funkcije.

5. Modifikovati funkciju iz prethodnog zadatka, tako da funkcija vraæa pokazivaè na konstantni (novoformirani) niz znakova. Analizirati moguænosti upotrebe ove modifikovane, kao i polazne funkcije u glavnom programu, u pogledu

izmene kreiranog niza znakova. Izmenu niza znakova pokušati i posredstvom vraæene vrednosti funkcije, i preko nekog drugog pokazivaèa, u koji se prebacuje vraæena vrednost funkcije.

6. Realizovati klasu èiji æe objekti služiti za izdvajanje reèi u tekstu koji je dat u nizu znakova. Jednom reèju se smatra niz znakova bez blanko znaka. Klasa treba da sadrži èlana koji je pokazivaè na niz znakova koji predstavlja ulazni tekst, i koji æe biti inicijalizovan u konstruktoru. Klasa treba da sadrži i funkciju koja, pri svakom pozivu, vraæa pokazivaè na dinamièki niz znakova u koji je izdvojena naredna reè teksta. Kada naiðe na kraj teksta, ova funkcija treba da vrati nula-pokazivaè. U glavnom programu isprobati upotrebu ove klase, na nekoliko objekata koji deluju nad istim globalnim nizom znakova.

Klase

Klase, objekti i èlanovi klase

Pojam i deklaracija klase

* *Klasa* je je realizacija apstrakcije koja ima svoju internu predstavu (svoje atribute) i operacije koje se mogu vršiti nad njom (javne funkcije èlanice). Klasa definiše tip. Jedan primerak takvog tipa (instanca klase) naziva se *objektom* te klase (engl. *class object*).
* Podaci koji su deo klase nazivaju se *podaci èlanovi* klase (engl. *data members*). Funkcije koje su deo klase nazivaju se *funkcije èlanice* klase (engl. *member functions*).
* Èlanovi (podaci ili funkcije) klase iza kljuène reèi `private:` zaštiæeni su od pristupa spolja (enkapsulirani su). Ovim èlanovima mogu pristupati samo funkcije èlanice klase. Ovi èlanovi nazivaju se *privatnim èlanovima klase* (engl. *private class members*).
* Èlanovi iza kljuène reèi `public:` dostupni su spolja i nazivaju se *javnim èlanovima klase* (engl. *public class members*).
* Èlanovi iza kljuène reèi `protected:` dostupni su funkcijama èlanicama date klase, kao i klasa izvedenih iz te klase, ali ne i korisnicima spolja, i nazivaju se *zaštiæenim èlanovima klase* (engl. *protected class members*).
* Redosled sekcija `public`, `protected` i `private` je proizvoljan, ali se preporuèuje baš navedeni redosled. Podrazumevano (ako se ne navede specifikator ispred) su èlanovi privatni.
* Kaže se još da klasa ima svoje unutrašnje stanje, predstavljeno atributima, koje menja pomoæu operacija. Javne funkcije èlanice nazivaju se još i *metodima* klase, a poziv ovih funkcija - *upuæivanje poruke* objektu klase. Objekat klase menja svoje stanje kada se pozove njegov metod, odnosno kada mu se uputi poruka.
* Objekat unutar svoje funkcije èlanice može pozivati funkciju èlanicu neke druge ili iste klase, odnosno uputiti poruku drugom objektu. Objekat koji šalje poruku (poziva funkciju) naziva se objekat-*klijent*, a onaj koji je prima (èija je funkcija èlanica pozvana) je objekat-*server*.
* Preporuka je da se klase projektuju tako da nemaju javne podatke èlanove.
* Unutar funkcije èlanice klase, èlanovima objekta èija je funkcija pozvana pristupa se direktno, samo navoðenjem njihovog imena.
* Kontrola pristupa èlanovima nije stvar objekta, nego klase: jedan objekat neke klase iz svoje funkcije èlanice može da pristupi privatnim èlanovima drugog objekta iste klase. Takoðe, kontrola pristupa èlanovima je potpuno odvojena od koncepta oblasti važenja: najpre se, na osnovu oblasti važenja, određuje entitet na koga se odnosi dato ime na mestu obraæanja u programu, a zatim se određuje da li se tom entitetu može pristupiti.
* Moguæe je preklopiti (engl. *overload*) funkcije èlanice, uključujuæi i konstruktore.
* Deklaracijom klase smatra se deo kojim se specifikuje ono što korisnici klase treba da vide. To su uvek javni èlanovi. Meðutim, da bi prevodilac korektno zauzimao prostor za objekte klase, mora da zna njegovu velièinu, pa u deklaraciju klase ulaze i deklaracije privatnih podataka èlanova:

```
// deklaracija klase complex:
class complex {
  public:
    void  cAdd(complex);
    void  cSub(complex);
    float cRe();
    float cIm();
    //...
  private:
    float real,imag;
};
```

* Gore navedena deklaracija je zapravo definicija klase, ali se iz istorijskih razloga naziva deklaracijom.
* Pravu deklaraciju klase predstavlja samo deklaracija `class S;`. Pre potpune deklaracije (zapravo definicije) mogu samo da se definišu pokazivaèi i reference na tu klasu, ali ne i objekti te klase, jer se njihova velièina ne zna.

Pokazivaè `this`

* Unutar svake funkcije èlanice postoji implicitni (podrazumevani, ugrađeni) lokalni objekat `this`. Tip ovog objekta je "konstantni pokazivaè na klasu èija je funkcija èlanica" (ako je klasa `X`, `this` je tipa `X*const`). Ovaj pokazivaè ukazuje na objekat èija je funkcija èlanica pozvana:

```
// definicija funkcije cAdd èlanice klase complex
complex complex::cAdd (complex c) {
  complex temp=*this;
            // u temp se prepisuje objekat koji je prozvan
  temp.real+=c.real;
  temp.imag+=c.imag;
  return temp;
}
```

* Pristup èlanovima objekta èija je funkcija èlanica pozvana obavlja se neposredno; implicitno je to pristup preko pokazivaèa `this` i operatora `->`. Može se i eksplicitno pristupati èlanovima preko ovog pokazivaèa unutar funkcije èlanice:

```
// nova definicija funkcije cAdd èlanice klase complex
complex complex::cAdd (complex c) {
  complex temp;
  temp.real=this->real+c.real;
  temp.imag=this->imag+c.imag;
  return temp;
}
```

* Pokazivaè `this` je, u stvari, jedan skriveni argument funkcije èlanice. Poziv `objekat.f()` prevodilac prevodi u kôd koji ima semantiku kao `f(&objekat)`.
* Pokazivaè `this` može da se iskoristi prilikom povezivanja (uspostavljanja relacije između) dva objekta. Na primer, neka klasa `X` sadrži objekat klase `Y`, pri èemu objekat klase `Y` treba da "zna" ko ga sadrži (ko mu je "nadreðeni"). Veza se inicijalno može uspostaviti pomoæu konstruktora:

```
class X {
public:
  X () : y(this) {...}
private:
  Y y;
};


class Y {
public:
  Y (X* theContainer) : myContainer(theContainer) {...}
private:
  X* myContainer;
};
```

Primerci klase

* Za svaki objekat klase formira se poseban komplet svih podataka èlanova te klase.
* Za svaku funkciju èlanicu, postoji jedinstven skup lokalnih statièkih objekata. Ovi objekti žive od prvog nailaska programa na njihovu definiciju, do kraja programa, bez obzira na broj objekata te klase. Lokalni statièki objekti funkcija èlanica imaju sva svojstva lokalnih statièkih objekata funkcija neèlanica, pa nemaju nikakve veze sa klasom i njenim objektima.
* Podrazumevano se sa objektima klase može raditi sledeæe:

1. definisati primerci (objekti) te klase i nizovi objekata klase;
2. definisati pokazivaèi na objekte i reference na objekte;
3. dodeljivati vrednosti (operator =) jednog objekta drugom;

4. uzimati adrese objekata (operator `&`) i posredno pristupati objektima preko pokazivaèa (operator `*`);
5. pristupati èlanovima i pozivati funkcije èlanice neposredno (operator `.`) ili posredno (operator `->`);
6. prenositi objekti kao argumenti funkcija i to po vrednosti ili referenci, ili prenositi pokazivaèi na objekte;
7. vraæati objekti iz funkcija po vrednosti ili referenci, ili vraæati pokazivaèi na objekte.

* Neke od ovih operacija korisnik može redefinisati preklapanjem operatora. Ostale, ovde nenavedene operacije korisnik mora definisati posebno ako su potrebne (ne podrazumevaju se).

Konstantne funkcije èlanice

* Dobra programerska praksa je da se korisnicima klase specifikuje da li neka funkcija èlanica menja unutrašnje stanje objekta ili ga samo "èita" i vraæa informaciju korisniku klase.
* Funkcije èlanice koje ne menjaju unutrašnje stanje objekta nazivaju se *inspektori* ili *selektori* (engl. *inspector*, *selector*). Da je funkcija èlanica inspektor, korisniku klase govori reè `const` iza zaglavlja funkcije. Ovakve funkcije èlanice nazivaju se u jeziku C++ *konstantnim* funkcijama èlanicama (engl. *constant member functions*).
* Funkcija èlanica koja menja stanje objekta naziva se *mutator* ili *modifikator* (engl. *mutator*, *modifier*) i posebno se ne oznaèava:

```
class X {
  public:
    int read () const      { return i; }
    int write (int j=0)    { int temp=i; i=j; return temp; }
  private:
    int i;
};
```

* Deklarisanje funkcije èlanice kao inspektora je samo notaciona pogodnost i "stvar lepog ponašanja prema korisniku". To je "obeæanje" projektanta klase korisnicima da funkcija ne menja stanje objekta, onako kako je projektant klase definisao stanje objekta. Prevodilac nema naèina da u potpunosti proveri da li inspektor menja neke podatke èlanove klase preko nekog posrednog obraæanja.
* Inspektor može da menja podatke èlanove, uz pomoæ eksplicitne konverzije, koja "probija" kontrolu konstantnosti. To je ponekad sluèaj kada inspektor treba da izraèuna podatak koji vraæa (npr. dužinu liste), pa ga onda saèuva u nekom èlanu da bi sledeæi put brže vratio odgovor.
* U konstantnoj funkciji èlanici tip pokazivaèa `this` je `const X*const`, tako da pokazuje na konstantni objekat, pa nije moguæe menjati objekat preko ovog pokazivaèa (svaki neposredni pristup èlanu je implicitni pristup preko ovog pokazivaèa). Takoðe, za konstantne objekte klase nije dozvoljeno pozivati nekonstantnu funkciju èlanicu (korektnost konstantnosti). Za prethodni primer:

```
X x;
const X cx;

x.read();  // u redu: konstantna funkcija nekonstantnog objekta;
x.write(); // u redu: nekonstantna funkcija nekonstantnog objekta;
cx.read(); // u redu: konstantna funkcija konstantnog objekta;
cx.write();// greška: nekonstantna funkcija konstantnog objekta;
```

Ugnežðivanje klasa

* Klase mogu da se deklarišu i unutar deklaracije druge klase (ugnežðivanje deklaracija klasa). Na ovaj naèin se ugnežðena klasa nalazi u oblasti važenja okružujuæe klase, pa se njenom imenu može pristupiti samo preko operatora razrešavanja oblasti važenja `::`.
* Okružujuæa klasa nema nikakva posebna prava pristupa èlanovima ugnežðene klase, niti ugnežðena klasa ima posebna prava pristupa èlanovima okružujuæe klase. Ugnežðivanje je samo stvar oblasti važenja, a ne i kontrole pristupa èlanovima.

```
int x,y;

class Spoljna {
public:
  int x;
  class Unutrasnja {
    void f(int i, Spoljna *ps) {
      x=i;      // greška: pristup Spoljna::x nije korektan!
      ::x=i;    // u redu: pristup globalnom x;
      y=i;      // u redu: pristup globalnom y;
      ps->x=i;  // u redu: pristup Spoljna::x objekta *ps;
    }
  };
};

Unutrasnja u;   // greška: Unutrasnja nije u oblasti važenja!
Spoljna::Unutrasnja u; // u redu;
```

* Unutar deklaracije klase se mogu navesti i deklaracije nabrajanja (`enum`), i `typedef` deklaracije. Ugnežđivanje se koristi kada neki tip (nabrajanje ili klasa npr.) semantièki pripada samo datoj klasi, a nije globalno važan i za druge klase. Ovakvo korišæenje poveæava èitljivost programa i smanjuje potrebu za globalnim tipovima.

Strukture

* Struktura je klasa kod koje su svi èlanovi podrazumevano javni. Može se to promeniti eksplicitnim umetanjem `public:` i `private:`

```
struct a {        isto što i:       class a {
                                    public:
  //...                               //...
private:                            private:
  //...                               //...
};                                  };
```

* Struktura se tipièno koristi za definisanje slogova podataka koji ne predstavljaju apstrakciju, odnosno nemaju ponašanje (nemaju znaèajnije operacije). Strukture tipièno poseduju samo konstruktore i eventualno destruktore kao funkcije èlanice.

Zajednièki èlanovi klasa

Zajednièki podaci èlanovi

* Pri kreiranju objekata klase, za svaki objekat se kreira poseban komplet podataka èlanova. Ipak, moguæe je definisati podatke èlanove za koje postoji samo jedan primerak za celu klasu, tj. za sve objekte klase.
* Ovakvi èlanovi nazivaju se *statièkim èlanovima*, i deklarišu se pomoæu reèi `static`:

```
class X {
public:
  //...
private:
  static int i;          // postoji samo jedan i za celu klasu
         int j;          // svaki objekat ima svoj j
  //...
};
```

* Svaki pristup statièkom èlanu iz bilo kog objeka klase znaèi pristup istom zajednièkom èlanu-objektu.

* Statièki èlan klase ima životni vek kao i globalni statièki objekat: nastaje na poèetku programa i traje do kraja programa. Uopšte, statièki èlan klase ima sva svojstva globalnog statièkog objekta, osim oblasti važenja klase i kontrole pristupa.
* Statièki èlan mora da se inicijalizuje posebnom deklaracijom van deklaracije klase. Obraæanje ovakvom èlanu van klase vrši se preko operatora `::`. Za prethodni primer:

```
int X::i=5;
```

* Statièkom èlanu može da se pristupi iz funkcije èlanice, ali i van funkcija èlanica, èak i pre formiranja ijednog objekta klase (jer statièki èlan nastaje kao i globalni objekat), naravno uz poštovanje prava pristupa. Tada mu se pristupa preko operatora `::` (`X::j`).
* Zajednièki èlanovi se uglavnom koriste kada svi primerci jedne klase treba da dele neku zajednièku informaciju, npr. kada predstavljaju neku kolekciju, odnosno kada je potrebno imati ih "sve na okupu i pod kontrolom". Na primer, svi objekti neke klase se uvezuju u listu, a glava liste je zajednièki èlan klase.
* Zajednièki èlanovi smanjuju potrebu za globalnim objektima i tako poveæavaju èitljivost programa, jer je moguæe ogranièiti pristup njima, za razliku od globalnih objekata. Zajednièki èlanovi logièki pripadaju klasi i "upakovani" su u nju.

Zajednièke funkcije èlanice

* I funkcije èlanice mogu da se deklarišu kao zajednièke za celu klasu, dodavanjem reèi `static` ispred deklaracije funkcije èlanice.
* Statièke funkcije èlanice imaju sva svojstva globalnih funkcija, osim oblasti važenja i kontrole pristupa. One ne poseduju pokazivaè `this` i ne mogu neposredno (bez pominjanja konkretnog objekta klase) koristiti nestatièke èlanove klase. Mogu neposredno koristiti samo statièke èlanove te klase.
* Statièke funkcije èlanice se mogu pozivati za konkretan objekat (što nema posebno znaèenje), ali i pre formiranja ijednog objekta klase, preko operatora `::`.
* Primer:

```
class X {
  static int x;          // statièki podatak èlan;
         int y;
public:
  static int f(X,X&);    // statièka funkcija èlanica;
         int g();
};

int X::x=5;             // definicija statièkog podatka èlana;

int X::f(X x1, X& x2){ // definicija statièke funkcije èlanice;
  int i=x;              // pristup statièkom èlanu X::x;
  int j=y;              // greška: X::y nije statièki,
                        // pa mu se ne može pristupiti neposredno!
  int k=x1.y;           // ovo može;
  return x2.x;          // i ovo može,
                        // ali se izraz "x2" ne izraèunava;
}

int X::g () {
  int i=x;              // nestatièka funkcija èlanica može da
  int j=y;              // koristi i pojedinaène i zajednièke
  return j;             // èlanove; y je ovde this->y;
}

void main () {
  X xx;
  int p=X::f(xx,xx);    // X::f može neposredno, bez objekta;
  int q=X::g();         // greška: za X::g mora konkretan objekat!
  xx.g();               // ovako može;
  p=xx.f(xx,xx);        // i ovako može,
                        // ali se izraz "xx" ne izraèunava;
}
```

* Statièke funkcije predstavljaju operacije klase, a ne svakog posebnog objekta. Pomoæu njih se definišu neke opšte usluge klase, npr. tipièno kreiranje novih, dinamièkih objekata te klase (operator `new` je implicitno definisan kao statièka funkcija klase). Na primer, na sledeæi naèin može se obezbediti da se za datu klasu mogu kreirati samo dinamièki objekti:

```
class X {
public:
  static X* create () { return new X; }
private:
  X();  // konstruktor je privatan
};
```

Prijatelji klasa

* Èesto je dobro da se klasa projektuje tako da ima i "povlašæene" korisnike, odnosno funkcije ili druge klase koje imaju pravo pristupa njenim privatnim èlanovima. Takve funkcije i klase nazivaju se *prijateljima* (enlgl. *friends*).

Prijateljske funkcije

* Prijateljske funkcije (engl. *friend functions*) su funkcije koje nisu èlanice klase, ali imaju pristup do privatnih èlanova klase. Te funkcije mogu da budu globalne funkcije ili èlanice drugih klasa.
* Da bi se neka funkcija proglasila prijateljem klase, potrebno je u deklaraciji te klase navesti deklaraciju te funkcije sa ključnom reèi `friend` ispred. Prijateljska funkcija se definiše na uobièajen naèin:

```
class X {
  friend void g (int,X&);  // prijateljska globalna funkcija
  friend void Y::h ();     // prijateljska èlanica druge klase
  int i;
public:
  void f(int ip) {i=ip;}
};

void g (int k, X &x) {
  x.i=k;             // prijateljska funkcija može da pristupa
}                    // privatnim èlanovima klase

void main () {
  X x;
  x.f(5);            // postavljanje preko èlanice
  g(6,x);            // postavljanje preko prijatelja
}
```

* Globalne funkcije koje predstavljaju usluge neke klase ili operacije nad tom klasom (najèešæe su prijatelji te klase) nazivaju se *klasnim uslugama* (engl. *class utilities*).
* Nema formalnih razloga da se koristi globalna (najèešæe prijateljska) funkcija umesto funkcije èlanice. Postoje prilike kada su globalne (prijateljske) funkcije pogodnije:

1. funkcija èlanica mora da se pozove za objekat date klase, dok globalnoj funkciji može da se dostavi i objekat drugog tipa, koji æe se konvertovati u potrebni tip;
2. kada funkcija treba da pristupa èlanovima više klasa, efikasnija je prijateljska globalna funkcija (primer u [Stroustrup91]);
3. ponekad je notaciono pogodnije da se koriste globalne funkcije (poziv je `f(x)`) nego èlanice (poziv je `x.f()`); na primer, `max(a,b)` je èitljivije od `a.max(b)`;
4. kada se preklapaju operatori, èesto je jednostavnije definisati globalne (operatorske) funkcije neko èlanice.

* "Prijateljstvo" se ne nasleđuje: ako je funkcija `f` prijatelj klasi `X`, a klasa `Y` izvedena (naslednik) iz klase `X`, funkcija `f` nije prijatelj klasi `Y`.

Prijateljske klase

* Ako je potrebno da sve funkcije èlanice klase `Y` budu prijateljske funkcije klasi `X`, onda se klasa `Y` deklariše kao prijateljska klasa (*friend class*) klasi `X`. Tada sve funkcije èlanice klase `Y` mogu da pristupaju privatnim èlanovima klase `X`, ali obratno ne važi ("prijateljstvo" nije simetrièna relacija):

```
class X {
  friend class Y;
  //...
};
```

* "Prijateljstvo" nije ni tranzitivna relacija: ako je klasa `Y` prijatelj klasi `X`, a klasa `Z` prijatelj klasi `Y`, klasa `Z` nije automatski prijatelj klasi `X`, veæ to mora eksplicitno da se naglasi (ako je potrebno).
* Prijateljske klase se tipièno koriste kada neke dve klase imaju tešnje međusobne veze. Pri tome je nepotrebno (i loše) "otkrivati" delove neke klase da bi oni bili dostupni drugoj prijateljskoj klasi, jer æe na taj naèin oni biti dostupni i ostalima (ruši se enkapsulacija). Tada se ove dve klase proglašavaju prijateljskim. Na primer, na sledeæi naèin može se obezbediti da samo klasa `Creator` može da kreira objekte klase `X`:

```
class X {
public:
  ...
private:
  friend class Creator;
  X();  // konstruktor je dostupan samo klasi Creator
  ...
};
```

Konstruktori i destruktori

Pojam konstruktora

* Funkcija èlanica koja nosi isto ime kao i klasa naziva se *konstruktor* (engl. *constructor*). Ova funkcija poziva se prilikom kreiranja objekta te klase.
* Konstruktor nema tip koji vraæa. Konstruktor može da ima argumente proizvoljnog tipa. Unutar konstruktora, èlanovima objekta pristupa se kao i u bilo kojoj drugoj funkciji èlanici.
* Konstruktor se uvek implicitno poziva pri kreiranju objekta klase, odnosno na poèetku životnog veka svakog objekta date klase.
* Konstruktor, kao i svaka funkcija èlanica, može biti preklopljen (engl. *overloaded*). Konstruktor koji se može pozvati bez stvarnih argumenata (nema formalne argumente ili ima sve argumente sa podrazumevanim vrednostima) naziva se podrazumevanim konstruktorom.

Kada se poziva konstruktor?

* Konstruktor je funkcija koja pretvara "presne" memorijske lokacije koje je sistem odvojio za novi objekat (i sve njegove podatke èlanove) u "pravi" objekat koji ima svoje èlanove i koji može da prima poruke, odnosno ima sva svojstva svoje klase i konzistentno poèetno stanje. Pre nego što se pozove konstruktor, objekat je u trenutku definisanja samo "gomila praznih bita" u memoriji raèunara. Konstruktor ima zadatak da od ovih bita napravi objekat tako što æe inicijalizovati èlanove.
* Konstruktor se poziva uvek kada se kreira objekat klase, a to je u sledeæim sluèajevima:

1. kada se izvršava definicija statièkog objekta;
2. kada se izvršava definicija automatskog (lokalnog nestatièkog) objekta unutar bloka; formalni argumenti se, pri pozivu funkcije, kreiraju kao lokalni automatski objekti;
3. kada se kreira objekat, pozivaju se konstruktori njegovih podataka èlanova;
4. kada se kreira dinamièki objekat operatorom `new`;
5. kada se kreira privremeni objekat, pri povratku iz funkcije, koji se inicijalizuje vraæenom vrednošæu funkcije.

Naèini pozivanja konstruktora

* Konstruktor se poziva kada se kreira objekat klase. Na tom mestu je moguæe navesti inicijalizatore, tj. stvarne argumente konstruktora. Poziva se onaj konstruktor koji se najbolje slaže po broju i tipovima argumenata (pravila su ista kao i kod preklapanja funkcija):

```
class X {
public:
  X ();
  X (double);
  X (char*);
  //...
};

void main () {
  double d=3.4;
  char *p="Niz znakova";
  X a(d),                   // poziva se X(double)
    b(p),                   // poziva se X(char*)
    c;                      // poziva se X()
  //...
}
```

* Pri definisanju objekta `c` sa zahtevom da se poziva podrazumevani konstruktor klase `X`, ne treba navesti `X c();` (jer je to deklaracija funkcije), veæ samo `X d;`.
* Pre izvršavanja samog tela konstruktora klase pozivaju se konstruktori èlanova. Argumenti ovih poziva mogu da se navedu iza zaglavlja definicije (ne deklaracije) konstruktora klase, iza znaka **:** (dvotaèka):

```
class YY {
public:
  YY (int j) {...}
  //...
};

class XX {
  YY  y;
  int i;
public:
  XX (int);
};

XX::XX (int k) : y(k+1) , i(k-1) {
  // y je inicijalizovan sa k+1, a i sa k-1
  // ... ostatak konstruktora
}
```

* Prvo se pozivaju konstruktori èlanova, po redosledu deklarisanja u deklaraciji klase, pa se onda izvršava telo konstruktora klase.
* Ovaj naèin ne samo da je moguæ, veæ je i jedino ispravan: navoðenje inicijalizatora u zaglavlju konstruktora predstavlja specifikaciju *inicijalizacije* èlanova (koji su ugraðenog tipa ili objekti klase), što je razlièito od *operacije dodele* koja se može jedino vršiti unutar tela konstruktora. Osim toga, kada za èlana korisniènog tipa ne postoji podrazumevani konstruktor, ili kada je èlan konstanta ili referenca, ovaj naèin je i jedini naèin inicijalizacije èlana.
* Konstruktor se može pozvati i eksplicitno u nekom izrazu. Tada se kreira privremeni objekat klase pozivom odgovarajuæeg konstruktora sa navedenim argumentima. Isto se dešava ako se u inicijalizatoru eksplicitno navede poziv konstruktora:

```
void main () {
  complex c1(1,2.4),c2;
  c2=c1+complex(3.4,-1.5);     // privremeni objekat
  complex c3=complex(0.1,5);   // opet privremeni objekat
                               // koji se kopira u c3
}
```

* Kada se kreira niz objekata neke klase, poziva se podrazumevani konstruktor za svaku komponentu niza ponaosob, po rastuæem redosledu indeksa.

Konstruktor kopije

* Kada se objekat `x1` klase `XX` inicijalizuje drugim objektom `x2` iste klase, C++ æe podrazumevano (ugrađeno) izvršiti prostu inicijalizaciju redom èlanova objekta `x1` èlanovima objekta `x2`. To ponekad nije dobro (èesto ako objekti sadrže èlanove koji su pokazivaèi ili reference), pa programer treba da ima potpunu kontrolu nad inicijalizacijom objekta drugim objektom iste klase.
* Za ovu svrhu služi tzv. konstruktor kopije (engl. *copy constructor*). To je konstruktor klase `XX` koji se može pozvati sa samo jednim stvarnim argumentom tipa `XX`. Taj konstruktor se poziva kada se objekat inicijalizuje objektom iste klase, a to je:

1. prilikom inicijalizacije objekta (pomoæu znaka = ili sa zagradama);
2. prilikom prenosa argumenata u funkciju (kreira se lokalni automatski objekat);
3. prilikom vraæanja vrednosti iz funkcije (kreira se privremeni objekat).

* Konstruktor kopije nikad ne sme imati formalni argument tipa `XX`, a može argument tipa `XX&` ili najèešæe `const XX&`.
* Primer:

```
class XX {
public:
  XX (int);
  XX (const XX&);      // konstruktor kopije
  //...
};

XX f(XX x1) {
  XX x2=x1;      // poziva se konstruktor kopije XX(XX&) za x2
  //...
  return x2;     // poziva se konstruktor kopije za
}                // privremeni objekat u koji se smešta rezultat

void g() {
  XX xa=3, xb=1;
  //...
  xa=f(xb);      // poziva se konstruktor kopije samo za
                 // formalni argument x1,
                 // a u xa se samo prepisuje privremeni objekat,
}                // ili se poziva XX::operator= ako je definisan
```

Destruktor

* Funkcija èlanica koja ima isto ime kao klasa, uz znak ~ ispred imena, naziva se *destruktor* (engl. *destructor*). Ova funkcija poziva se automatski, pri prestanku života objekta klase, za sve navedene sluèajeve (statièkih, automatskih, klasnih èlanova, dinamièkih i privremenih objekata):

```
class X {
public:
  ~X () { cout<<"Poziv destruktora klase X!\n"; }
}

void main () {
  X x;
  //...
}  // ovde se poziva destruktor objekta x
```

* Destruktor nema tip koji vraæa i ne može imati argumente. Unutar destruktora, privatnim èlanovima pristupa se kao i u bilo kojoj drugoj funkciji èlanici. Svaka klasa može da ima najviše jedan destruktor.

* Destruktor se implicitno poziva i pri uništavanju dinamièkog objekta pomoæu operatora `delete`. Za niz, destruktor se poziva za svaki element ponaosob. Redosled poziva destruktora je u svakom sluèaju obratan redosledu poziva konstruktora.
* Destruktori se uglavnom koriste kada objekat treba da dealocira memoriju ili neke sistemske resurse koje je konstruktor alocirao; to je najèešæe sluèaj kada klasa sadrži èlanove koji su pokazivaèi.
* Posle izvršavanja tela destruktora, automatski se oslobađa memorija koju je objekat zauzimao.

Zadaci:

7. Realizovati klasu koja implementira red èekanja (*queue*). Predvideti operacije stavljanja i uzimanja elementa, sve potrebne konstruktore (i konstruktor kopije) i ostale potrebne funkcije.

8. Skicirati klasu `View` koja æe predstavljati apstrakciju svih vrsta entiteta koji se mogu pojaviti na ekranu monitora u nekom korisnièkom interfejsu (prozor, meni, dijalog, itd.). Sve klase koje æe realizovati pojedine entitete interfejsa biæe izvedene iz ove klase. Ova klasa treba da ima virtuelnu funkciju `draw`, koja æe predstavljati iscrtavanje entiteta pri osvežavanju (ažuriranju) izgleda ekrana (kada se nešto na ekranu promeni), i koju ne treba realizovati. Svaki objekat ove klase æe se, pri kreiranju, "prijavljivati" u jednu listu svih objekata na ekranu. Klasa `View` treba da sadrži statièku funkciju èlanicu `refresh` koja æe prolaziti kroz tu listu, pozivajuæi funkciju `draw` svakog objekta, kako bi se izgled ekrana osvežio. Redosled objekata u listi predstavlja redosled iscrtavanja, èime se dobija efekat preklapanja na ekranu. Zbog toga klasa `View` treba da ima funkciju èlanicu `setFocus`, koja æe dati objekat postaviti na kraj liste (daje se fokus tom entitetu). Realizovati sve navedene delove klase `View`, osim funkcije `draw`.

Preklapanje operatora

Pojam preklapanja operatora

* Pretpostavimo da su nam u programu potrebni kompleksni brojevi i operacije nad njima. Treba nam struktura podataka koja æe, pomoæu osnovnih (u jezik ugraðenih) tipova, predstaviti strukturu kompleksnog broja, a takoðe i funkcije koje æe realizovati operacije nad kompleksnim brojevima.
* Kada je potrebna struktura podataka za koju detalji implementacije nisu bitni, veæe operacije koje se nad njom vrše, sve ukazuje na klasu. Klasa upravo predstavlja tip podataka za koji su definisane operacije.
* U jeziku C++, operatori za korisnièke tipove su specijalne funkcije koje nose ime `operator@`, gde je `@` neki operator ugraðen u jezik:

```
class complex {
public:
  complex(double,double);                              /* konstruktor */
  friend complex operator+(complex,complex);           /* oparator + */
  friend complex operator-(complex,complex);           /* operator - */
private:
  double real, imag;
};

complex::complex (double r, double i) : real(r), imag(i) {}

complex operator+ (complex c1, complex c2) {
  complex temp(0,0);  /* privremena promenljiva tipa complex */
  temp.real=c1.real+c2.real;
  temp.imag=c1.imag+c2.imag;
  return temp;
}

complex operator- (complex c1, complex c2) {
/* može i ovako: vratiti privremenu promenljivu
   koja se kreira konstruktorom sa odgovarajuæim argumentima */
  return complex(c1.real-c2.real,c1.imag-c2.imag);
}
```

* Operatorske funkcije se mogu koristiti u izrazima kao i operatori nad ugraðenim tipovima. Izraz `t1@t2` se tumaèi kao `t1.operator@(t2)` ili `operator@(t1,t2)`:

```
complex c1(3,5.4),c2(0,-5.4),c3(0,0);
c3=c1+c2;      /* poziva se operator+(c1,c2) */
c1=c2-c3;      /* poziva se operator-(c2,c3) */
```

Operatorske funkcije

Osnovna pravila

* U jeziku C++, pored "obiènih" funkcija koje se eksplicitno pozivaju navoðenjem identifikatora sa zagradama, postoje i operatorske funkcije.
* Operatorske funkcije su posebna vrsta funkcija koje imaju posebna imena i naèin pozivanja. Kao i obiène funkcije, i one se mogu preklopiti za operande koji pripadaju korisnièkim tipovima. Ovaj princip naziva se preklapanje operatora (engl. *operator overloading*).
* Ovaj princip omoguæava da se definišu znaèenja operatora za korisnièke tipove i formiraju izrazi sa objektima ovih tipova, na primer operacije nad kompleksnim brojevima (`ca*cb+cc-cd`), matricama (`ma*mb+mc-md`) itd.
* Ipak, postoje neka ogranièenja u preklaanju operatora:

1. ne mogu da se preklope operatori `.`, `.*`, `::`, `?:` i `sizeof`, dok svi ostali mogu;

2. ne mogu da se redefinišu znaèenja operatora za ugraðene (standardne) tipove podataka;
3. ne mogu da se uvode novi simboli za operatore;
4. ne mogu da se menjaju osobine operatora koje su ugraðene u jezik: *n*-arnost, prioriteti i asocijativnost (smer grupisanja).

* Operatorske funkcije imaju imena `operator@`, gde je `@` znak operatora. Operatorske funkcije mogu biti èlanice ili globalne funkcije (uglavnom prijatelji klasa) kod kojih je bar jedan argument tipa korisnièke klase:

```
complex operator+ (complex c, double d) {
  return complex(c.real+d,c.imag);
}  // ovo je globalna funkcija prijatelj

complex operator** (complex c, double d) {  // ovo ne može
  // hteli smo stepenovanje
}
```

* Za korisnièke tipove su unapred definisana uvek dva operatora: = (dodela vrednosti) i `&` (uzimanje adrese). Sve dok ih korisnik ne redefiniše, oni imaju podrazumevano znaèenje.
* Podrazumevano znaèenje operatora = je kopiranje objekta dodelom èlan po èlan (pozivaju se operatori = klasa kojima èlanovi pripadaju, ako su definisani). Ako objekat sadrži èlana koji je pokazivaè, kopiraæe se, naravno, samo traj pokazivaè, a ne i pokazivana vrednost. Ovo nekad nije odgovarajuæe i korisnik treba da redefiniše operator =.
* Vrednosti operatorskih funkcija mogu da budu bilo kog tipa, pa i `void`.

Boèni efekti i veze između operatora

* Boèni efekti koji postoje kod operatora za ugraðene tipove nikad se ne podrazumevaju za redefinisane operatore: ++ ne mora da menja stanje objekta, niti da znaèi sabiranje sa 1. Isto važi i za `--` i sve operatore dodele (=, +=, -=, *= itd.).
* Operator = (i ostali operatori dodele) ne mora da menja stanje objekta. Ipak, ovakve upotrebe treba strogo izbegavati: redefinisani operator treba da ima isto ponašanje kao i za ugraðene tipove.
* Veze koje postoje izmeðu operatora za ugraðene tipove se ne podrazumevaju za redefinisane operatore. Na primer, `a+=b` ne mora da automatski znaèi `a=a+b`, ako je definisan operator +, veæ operator += mora posebno da se definiše.
* Strogo se preporuèuje da operatori koje definiše korisnik imaju oèekivano znaèenje, radi èitljivosti programa. Na primer, ako su definisani i operator += i operator +, dobro je da `a+=b` ima isti efekat kao i `a=a+b`. Treba izbegavati neoèekivana znaèenja, na primer da operator - realizuje sabiranje matrica.
* Kada se definišu operatori za klasu, treba težiti da njihov skup bude kompletan. Na primer, ako su definisani operatori = i +, treba definisati i operator +=; ili, uvek treba definisati oba operatora == i `!=`, a ne samo jedan.

Operatorske funkcije kao èlanice i globalne funkcije

* Operatorske funkcije mogu da budu èlanice klasa ili (najèešæe prijateljske) globalne funkcije. Ako je `@` neki binarni operator (na primer +), on može da se realizuje kao funkcija èlanica klase `X` na sledeæi naèin (mogu se argumenti prenositi i po referenci):

```
tip operator@ (X)
```

ili kao prijateljska globalna funkcija na sledeæi naèin:

```
tip operator@ (X,X)
```

Nije dozvoljeno da se u programu nalaze obe ove funkcije.

* Poziv `a@b` se sada tumaèi kao:
  `a.operator@(b)` , za funkciju èlanicu, ili:
  `operator@(a,b)` , za globalnu funkciju.
* Primer:

```
class complex {
  double real,imag;
public:
  complex (double r=0, double i=0) : real(r), imag(i) {}
  complex operator+(coplex c)
    { return complex(real+c.real,imag+c.imag; }
};

// ili, alternativno:

class complex {
  double real,imag;
public:
  complex (double r=0, double i=0) : real(r), imag(i) {}
  friend complex operator+(complex,coplex);
};

complex operator+ (complex c1, complex c2) {
  return complex(c1.real+c2.real,c1.imag+c2.imag);
}


void main () {
  complex c1(2,3),c2(3.4);
  complex c3=c1+c2;  // poziva se c1.operator+(c2) ili
                     // operator+(c1,c2)
  //...
}
```

* Razlozi za izbor jednog ili drugog naèina (èlanica ili prijatelj) su isti kao i za druge funkcije. Ovde postoji još jedna razlika: ako za prethodni primer hoæemo da se može vršiti i operacija sabiranja realnog broja sa kompleksnim, treba definisati globalnu funkciju. Ako hoæemo da se može izvršiti `d+c`, gde je `d` tipa `double`, ne možemo definisati novu operatorsku "èlanicu klase `double`", jer ugraðeni tipovi nisu klase (C++ nije èisti OO jezik). Operatorska funkcija èlanica "ne dozvoljava promociju levog operanda", što znaèi da se neæe izvršiti konverzija operanda `d` u tip `complex`. Treba izabrati drugi navedeni postupak (sa prijateljskom operatorskom funkcijom).

Unarni i binarni operatori

* Mnogi operatori jezika C++ (kao i jezika C) mogu da budu i unarni i binarni (unarni i binarni `-`, unarni `&`-adresa i binarni `&`-logièko I po bitovima itd.). Kako razlikovati unarne i binarne operatore prilikom preklapanja?
* Unarni operator ima samo jedan operand, pa se može realizovati kao operatorska funkcija èlanica bez argumenata (prvi operand je objekat èija je funkcija èlanica pozvana):

```
tip operator@ ()
```

ili kao globalna funkcija sa jednim argumentom:

```
tip operator@ (X x)
```

* Binarni operator ima dva argumenta, pa se može realizovati kao funkcija èlanica sa jednim argumentom (prvi operand je objekat èija je funkcija èlanica pozvana):

```
tip operator@ (X xdesni)
```

ili kao globalna funkcija sa dva argumenta:

```
tip operator@ (X xlevi, X xdesni)
```

* Primer:

```
class complex {
  double real,imag;
public:
  complex (double r=0, double i=0) : real(r), imag(i) {}
  friend complex operator+(complex,coplex);
  complex operator!()  // unarni operator!, konjugovani broj
    { return complex(real,-imag); }
};
```

Neki posebni operatori

Operatori `new` *i* `delete`

* Ponekad programer želi da preuzme kontrolu nad alokacijom dinamièkih objekata neke klase, a ne da je prepusti ugraðenom alokatoru. To je zgodno npr. kada su objekti klase mali i može se precizno kontrolisati njihova alokacija, tako da se smanje režije oko alokacije.
* Za ovakve potrebe mogu se preklopiti operatori `new` i `delete` za neku klasu. Operatorske funkcije `new` i `delete` moraju buti statièke (`static`) funkcije èlanice, jer se one pozivaju pre nego što je objekat stvarno kreiran, odnosno pošto je uništen.
* Ako je korisnik definisao ove operatorske funkcije za neku klasu, one æe se pozivati kad god se kreira dinamièki objekat te klase operatorom `new`, odnosno kada se takav objekat dealocira operatorom `delete`.
* Unutar tela ovih operatorskih funkcija ne treba eksplicitno pozivati konstruktor, odnosno destruktor. Konstruktor se implicitno poziva posle operatorske funkcije `new`, a destruktor se implicitno poziva pre operatorske funkcije `delete`. Ove operatorske funkcije služe samo da obezbede prostor za smeštanje objekta i da ga posle oslobode, a ne da od "presnih" bita naprave objekat (što rade konstruktori), odnosno pretvore ga u "presne bite" (što radi destruktor). Operator `new` treba da vrati pokazivaè na alocirani prostor.
* Ove operatorske funkcije deklarišu se na sledeæi naèun:

```
void* operator new (size_t velicina)
void operator delete (void* pokazivac)
```

Tip `size_t` je celobrojni tip definisan u `<stdlib.h>` i služi za izražavanje velièina objekata. Argument `velicina` daje veliènu potrebnog prostora koga treba alocirati za objekat. Argument `pokazivac` je pokazivaè na prostor koga treba osloboditi.

* Podrazumevani (ugraðeni) operatori `new` i `delete` mogu da se pozivaju unutar tela redefinisanih operatorskih funkcija ili eksplicitno, preko operatora `::`, ili implicitno, kada se dinamièki kreiraju objekti koji nisu tipa za koga su redefinisani ovi operatori.
* Primer:

```
#include <stdlib.h>

class XX {
  //...
public:
  void* operator new (size_t sz)
    { return new char[sz]; }  // koristi se ugraðeni new
  void operator delete (void *p)
    { delete [] p; }          // koristi se ugraðeni delete
  //...
};
```

Konstruktor kopije i operator dodele

* Inicijalizacija objekta pri kreiranju i dodela vrednosti su dve suštinski razlièite operacije.
* Inicijalizacija se vrši u svim sluèajevima kada se kreira objekat (statièki, automatski, klasni èlan, privremeni i dinamièki). Tada se poziva konstruktor, iako se inicijalizacija obavlja preko znaka =. Ako je izraz sa desne strane znaka = istog tipa kao i objekat koji se kreira, poziva se konstruktor kopije, ako je definisan. Ovaj konstruktor najèešæe kopira ceo složeni objekat, a ne samo èlanove.

* Dodelom se izvršava operatorska funkcija `operator=`. To se dešava kada se eksplicitno u nekom izrazu poziva ovaj operator. Ovaj operator najèešæe prvo uništava prethodno formirane delove objekta, pa onda formira nove, uz kopiranje delova objekta sa desne strane znaka dodele. Ova operatorska funkcija mora biti nestatièka funkcija èlanica.
* Inicijalizacija podrazumeva da objekat još ne postoji. Dodela podrazumeva da objekat sa leve strane operatora postoji.
* Ako neka klasa sadrži destruktor, konstruktor kopije ili operator dodele, sva je prilika da treba da sadrži sva tri.
* Primer - klasa koja realizuje niz znakova:

```
class String {
public:
  String(const char*);
  String(const String&);              // konstruktor kopije
  String& operator= (const String&);  // operator dodele
  //...
private:
  char *niz;
};

String::String (const String &s) {
  if (niz=new char [strlen(s.niz)+1]) strcpy(niz,s.niz);
}

String& String:operator= (const String &s) {
  if (&s!=this) {               // provera na s=s
    if (niz) delete [] niz;     // prvo oslobodi staro,
    if (niz=new char [strlen(s.niz)+1]) strcpy(niz,s.niz);
  }                             // pa onda zauzmi novo
  return *this;
}

void main () {
  String a("Hello world!"), b=a; // String(const String&);
  a=b;                           // operator=
  //...
}
```

* Posebno treba obratiti pažnju na karakteristiène sluèajeve pozivanja konstruktora kopije:
1. pri inicijalizaciji objekta izrazom istog tipa poziva se konstruktor kopije;
2. pri pozivanju funkcije, formalni argumenti se inicijalizuju stvarnim i, ako su istog tipa, poziva se konstruktor kopije;
3. pri vraæanju vrednosti iz funkcije, privremeni objekat se inicijalizuje vrednošæu koja se iz funkcije vraæa i, ako su istog tipa, poziva se konstruktor kopije.

Osnovni standardni ulazno/izlazni tokovi

Klase `istream` *i* `ostream`

* Kao i jezik C, ni C++ ne sadrži (u jezik ugrađene) ulazno/izlazne (U/I) operacije, veæ se one realizuju standardnim bibliotekama. Ipak, C++ sadrži standardne U/I biblioteke realizovane u duhu OOP-a.
* Na raspolaganju su i stare C biblioteke sa funkcijama `scanf` i `printf`, ali njihovo korišæenje nije u duhu jezika C++.
* Biblioteka èije se deklaracije nalaze u zaglavlju `<iostream.h>` sadrži dve osnovne klase, `istream` i `ostream` (ulazni i izlazni tok). Svakom primerku (objektu) klasa `ifstream` i `ofstream`, koje su redom izvedene iz navedenih klasa, može da se pridruži jedna datoteka za ulaz/izlaz, tako da se datotekama pristupa isključivo preko ovakvih objekata, odnosno funkcija èlanica ili prijatelja ovih klasa. Time je podržan princip enkapsulacije.
* U ovoj biblioteci definisana su i dva korisniku dostupna (globalna) statièka objekta:
1. objekat `cin` klase `istream` koji je pridružen standardnom ulaznom uređaju (obièno tastatura);
2. objekat `cout` klase `ostream` koji je pridružen standardnom izlaznom uređaju (obièno ekran).

* Klasa `istream` je preklopila operator >> za sve ugraðene tipove, koji služi za ulaz podataka:
  `istream& operator>> (istream &is, tip &t);`

gde je `tip` neki ugraðeni tip objekta koji se èita.

* Klasa `ostream` je preklopila operator << za sve ugraðene tipove, koji služi za izlaz podataka:
  `ostream& operator<< (ostream &os, tip x);`

gde je `tip` neki ugraðeni tip objekta koji se ispisuje.

* Ove funkcije vraæaju reference, tako da se može vršiti višestruki U/I u istoj naredbi. Osim toga, ovi operatori su asocijativni sleva, tako da se podaci ispisuju u prirodnom redosledu.
* Ove operatore treba koristiti za uobièajene, jednostavne U/I operacije:

```
#include <iostream.h>  // obavezno ako se želi U/I

void main () {
  int i;
  cin>>i;                    // uèitava se i
  cout<<"i="<<i<<'\n';       // ispisuje se npr.: i=5 i prelazi u
                             // novi red
}
```

* O detaljima klasa `istream` i `ostream` treba videti [Stroustrup91] i `<iostream.h>`.

Ulazno/izlazne operacije za korisnièke tipove

* Korisnik može da definiše znaèenja operatora >> i << za svoje tipove. To se radi definisanjem prijateljskih funkcija korisnikove klase, jer je prvi operand tipa `istream&` odnosno `ostream&`.
* Primer za klasu `complex`:

```
#include <iostream.h>

class complex {
  double real,imag;
  friend ostream& operator<< (ostream&,const complex&);
public:
  //... kao i ranije
};

//...

ostream& operator<< (ostream &os, const complex &c) {
  return os<<"("<<c.real<<","<<c.imag<<")";
}

void main () {
  complex c(0.5,0.1);
  cout<<"c="<<c<<"\n";  // ispisuje se: c=(0.5,0.1)
}
```

Zadaci:

9. Realizovati klasu `longint` koja predstavlja cele brojeve u neogranièenoj taènosti (proizvoljan broj decimalnih cifara). Obezbediti operacije sabiranja i oduzimanja, kao i sve ostale oèekivane operacije, uključujuæi i ulaz/izlaz. Skicirati glavni program koji kreira promenljive ovog tipa i vrši operacije nad njima. Uputstvo: brojeve interno predstavljati kao nizove znakova.

10. Realizovati klasu èasovnika. Èasovnik treba da ima moguænost postavljanja na poèetnu vrednost na sve oèekivane naèine (inicijalizacija, dodela, i funkcija za "navijanje"), i operacije odbrojavanja sekunde (operator ++), i poveæanja vrednosti za neki vremenski interval. Vremenski interval predstaviti posebnom, ugnežðenom strukturom podataka.

Nasleđivanje

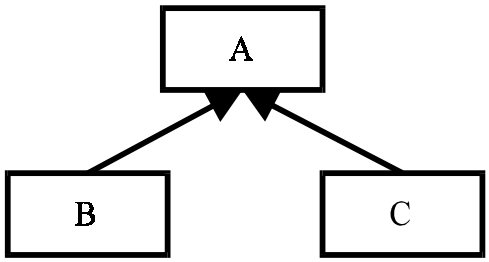
Izvedene klase

Šta je nasleđivanje i šta su izvedene klase?

* U praksi se èesto sreæe sluèaj da se jedna klasa objekata (klasa B) podvrsta neke druge klase (klasa A). To znaèi da su objekti klase B "jedna (specijalna) vrsta" ("*a-kind-of*") objekata klase A, ili da objekti klase B "imaju sve osobine klase A, i još neke, sebi svojstvene". Ovakva relacija između klasa naziva se *nasleđivanje* (engl. *inheritance*): klasa B nasleđuje klasu A.
* Primeri:

1. "Sisari" su klasa koja je okarakterisana naèinom reprodukcije. "Mesožderi" su "sisari" koji se hrane mesom. "Biljojedi" su sisari koji se hrane biljkama. Uopšte, u živom svetu odnosi "vrsta" predstavljaju relaciju nasleđivanja klasa.
2. "Geometrijske figure u ravni" su klasa koja je okarakterisana koordinatama težišta. "Krug" je figura koja je okarakterisana dužinom polupreènika. "Kvadrat" je figura koja je okarakterisana dužinom ivice.
3. "Izlazni uređaji raèunara" su klasa koja ima operacije pisanja jednog znaka. "Ekran" je izlazni uređaj koji ima moguænost i crtanja, brisanja, pomeranja kurzora itd.

* Relacija nasleđivanja se u programskom modelu definiše u odnosu na to šta želimo da klase rade, odnosno koja svojstva i servise da imaju. Primer: da li je krug jedna vrsta elipse, ili je elipsa jedna vrsta kruga, ili su i krug i elipsa podvrste ovalnih figura?
* Ako je klasa B nasledila klasu A, kaže se još da je klasa A *osnovna klasa* (engl. *base class*), a klasa B *izvedena klasa* (engl. *derived class*). Ili da je klasa A nadklasa (engl. *superclass*), a klasa B podklasa (engl. *subclass*). Ili da je klasa A roditelj (engl. *parent*), a klasa B dete (engl. *child*). Relacija nasleđivanja se najèešæe prikazuje (usmerenim acikliènim) grafom:



* Jezici koji podržavaju nasleđivanje nazivaju se *objektno orijentisanim* (engl. *Object-Oriented Programming Languages*, OOPL).

Kako se definišu izvedene klase u jeziku C++?

* Da bi se klasa izvela iz neke postojeæe klase, nije potrebno vršiti nikakve izmene postojeæe klase, pa èak ni njeno ponovno prevođenje. Izvedena klasa se deklariše navođenjem reèi `public` i naziva osnovne klase, iza znaka `:` (dvotaèka):

```
class Base {
  int i;
public:
  void f();
};

class Derived : public Base {
  int j;
public:
  void g();
};
```

* Objekti izvedene klase imaju sve èlanove osnovne klase, i svoje posebne èlanove koji su navedeni u deklaraciji izvedene klase.
* Objekti izvedene klase definišu se i koriste na uobièajen naèin:

```
void main () {
  Base b;
  Derived d;
  b.f();
  b.g();  // ovo, naravno, ne može
  d.f();  // d ima i funkciju f,
  d.g();  // i funkciju g
}
```

* Izvedena klasa ne nasleðuje funkciju èlanicu `operator=`.

Prava pristupa

* Ključna reè `public` u zaglavlju deklaracije izvedene klase znaèi da su svi javni èlanovi osnovne klase ujedno i javni èlanovi izvedene klase.
* Privatni èlanovi osnovne klase uvek to i ostaju. Funkcije èlanice izvedene klase ne mogu da pristupaju privatnim èlanovima osnovne klase. Nema naèina da se "povredi privatnost" osnovne klase (ukoliko neko nije prijatelj te klase, što je zapisano u njenoj deklaraciji), jer bi to znaèilo da postoji moguænost da se probije enkapsulacija koju je zamislio projektant osnovne klase.
* Javnim èlanovima osnovne klase se iz funkcija èlanica izvedene klase pristupa neposredno, kao i sopstvenim èlanovima:

```
class Base {
  int pb;
public:
  int jb;
  void put(int x) {pb=x;}
};

class Derived : public Base {
  int pd;
public:
  void write(int a, int b, int c) {
    pd=a;
    jb=b;
    pb=c;    // ovo ne može,
    put(c);  // veæ mora ovako
  }
};
```

* Deklaracija èlana izvedene klase sakriva istoimeni èlan osnovne klase. Sakrivenom èlanu osnovne klase može da se pristupi pomoæu operatora `::`. Na primer, `Base::jb`.

* Èesto postoji potreba da nekim èlanovima osnovne klase mogu da pristupe funkcije èlanice izvedenih klasa, ali ne i korisnici klasa. To su najèešæe funkcije èlanice koje direktno pristupaju privatnim podacima èlanovima. Èlanovi koji su dostupni samo izvedenim klasama, ali ne i korisnicima spolja, navode se iza ključne reèi `protected:` i nazivaju se *zaštiæeni èlanovi* (engl. *protected members*).
* Zaštiæeni èlanovi ostaju zaštiæeni i za sledeæe izvedene klase pri sukcesivnom nasleđivanju. Uopšte, ne može se poveæati pravo pristupa nekom èlanu koji je privatan, zaštiæen ili javni.

```
class Base {
  int pb;
protected:
  int zb;
public:
  int jb;
  //...
};

class Derived : public Base {
  //...
public:
  void write(int x) {
    jb=zb=x;  // može da pristupi javnom i zaštiæenom èlanu,
    pb=x;     // ali ne i privatnom: greška!
  }
};

void f() {
  Base b;
  b.zb=5;  // odavde ne može da se pristupa zaštiæenom èlanu
}
```

Konstruktori i destruktori izvedenih klasa

* Prilikom kreiranja objekta izvedene klase, poziva se konstruktor te klase, ali i konstruktor osnovne klase. U zaglavlju definicije konstruktora izvedene klase, u listi inicijalizatora, moguæe je navesti i inicijalizator osnovne klase (argumente poziva konstruktora osnovne klase). To se radi navođenjem imena osnovne klase i argumenata poziva konstruktora osnovne klase:

```
class Base {
  int bi;
  //...
public:
  Base(int);  // konstruktor osnovne klase
  //...
};

Base::Base (int i) : bi(i) {/*...*/}


class Derived : public Base {
  int di;
  //...
public:
  Derived(int);
  //...
};

Derived::Derived (int i) : Base(i),di(i+1) {/*...*/}
```

* Pri kreiranju objekta izvedene klase redosled poziva konstruktora je sledeæi:

1. inicijalizuje se podobjekat osnovne klase, pozivom konstruktora osnovne klase;
2. inicijalizuju se podaci èlanovi, eventualno pozivom njihovih konstruktora, po redosledu deklarisanja;

3. izvršava se telo konstruktora izvedene klase.

* Pri uništavanju objekta, redosled poziva destruktora je uvek obratan.

```
class XX {
  //...
public:
  XX() {cout<<"Konstruktor klase XX.\n";}
  ~XX() {cout<<"Destruktor klase XX.\n";}
};


class Base {
  //...
public:
  Base() {cout<<"Konstruktor osnovne klase.\n";}
  ~Base() {cout<<"Destruktor osnovne klase.\n";}
  //...
};


class Derived : public Base {
  XX xx;
  //...
public:
  Derived() {cout<<"Konstruktor izvedene klase.\n";}
  ~Derived() {cout<<"Destruktor izvedene klase.\n";}
  //...
};


void main () {
  Derived d;
}

/* Izlaz æe biti:
Konstruktor osnovne klase.
Konstruktor klase XX.
Konstruktor izvedene klase.
Destruktor izvedene klase.
Destruktor klase XX.
Destruktor osnovne klase.
*/
```

Polimorfizam

Šta je polimorfizam?

* Pretpostavimo da smo projektovali klasu geometrijskih figura sa namerom da sve figure imaju funkciju `crtaj()` kao èlanicu. Iz ove klase izveli smo klase kruga, kvadrata, trougla itd. Naravno, svaka izvedena klasa treba da realizuje funkciju crtanja na sebi svojstven naèin (krug se sasvim drugaèije crta od trougla). Sada nam je potrebno da u nekom delu programa iscrtamo sve figure koje se nalaze na našem crtežu. Ovim figurama pristupamo preko niza pokazivaèa tipa `figura*`. C++ omoguæava da figure jednostavno iscrtamo prostim navoðenjem:

```
void crtanje () {
  for (int i=0; i<broj_figura; i++)
    niz_figura[i]->crtaj();
}
```

* Iako se u ovom nizu mogu naæi razlièite figure (krugovi, trouglovi itd.), mi im jednostavno pristupamo kao figurama, jer sve vrste figura imaju zajednièku osobinu "da mogu da se nacrtaju". Ipak, svaka od figura æe svoj zadatak

ispuniti onako kako joj to i priliči, odnosno svaki objekat će "prepoznati" kojoj izvedenoj klasi pripada, bez obzira što mu se obraćamo "uopšteno", kao objektu osnovne klase. To je posledica naše pretpostavke da je i krug, i kvadrat i trougao takođe i figura.

* Svojstvo da svaki objekat izvedene klase izvršava metod tačno onako kako je to definisano u njegovoj izvedenoj klasi, kada mu se pristupa kao objektu osnovne klase, naziva se *polimorfizam* (engl. *polymorphism*).

Virtuelne funkcije

* Funkcije članice osnovne klase koje se u izvedenim klasama mogu realizovati specifično za svaku izvedenu klasu nazivaju se *virtuelne funkcije* (engl. *virtual functions*).
* Virtuelna funkcija se u osnovnoj klasi deklariše pomoću ključne reči `virtual` na početku deklaracije. Prilikom definisanja virtuelnih funkcija u izvedenim klasama ne mora se stavljati reč `virtual`.
* Prilikom poziva se odaziva ona funkcija koja pripada klasi kojoj i objekat koji prima poziv.

```
class ClanBlioteke {
  //...
protected:
  Racun r;
  //...
public:
  virtual int platiClanarinu ()  // virtuelna funkcija
    { return r-=clanarina; }
  //...
};

class PocasniClan : public ClanBiblioteke {
  //...
public:
  int platiClanarinu () { return r; }
};

void main () {
  ClanBiblioteke *clanovi[100];
  //...
  for (int i=0; i<brojClanova; i++)
    cout<<clanovi[i]->platiClanarinu();
  //...
}
```

* Virtuelna funkcija osnovne klase ne mora da se redefiniše u svakoj izvedenoj klasi. U izvedenoj klasi u kojoj virtuelna funkcija nije definisana, važi značenje te virtuelne funkcije iz osnovne klase.
* Deklaracija neke virtuelne funkcije u svakoj izvedenoj klasi mora da se u potpunosti slaže sa deklaracijom te funkcije u osnovnoj klasi (broj i tipovi argumenata, kao i tip rezultata).
* Ako se u izvedenoj klasi deklariše neka funkcija koja ima isto ime kao i virtuelna funkcija iz osnovne klase, ali različit broj i/ili tipove argumenata, onda ona sakriva (a ne redefiniše) *sve* ostale funkcije sa istim imenom iz osnovne klase. To znači da u izvedenoj klasi treba ponovo definisati sve ostale funkcije sa tim imenom. Nikako nije dobro (to je greška u projektovanju) da izvedena klasa sadrži samo neke funkcije iz osnovne klase, ali ne sve: to znači da se ne radi o pravom nasleđivanju (korisnik izvedene klase očekuje da će ona ispuniti sve zadatke koje može i osnovna klasa).
* Virtuelne funkcije moraju biti članice svojih klasa (ne globalne), a mogu biti prijatelji drugih klasa.

Dinamičko vezivanje

* Pokazivač na objekat izvedene klase se može implicitno konvertovati u pokazivač na objekat osnovne klase (pokazivaču na objekat osnovne klase se može dodeliti pokazivač na objekat izvedene klase direktno, bez eksplicitne konverzije). Isto važi i za reference. Ovo je interpretacija činjenice da se objekat izvedene klase može smatrati i objektom osnovne klase.
* Pokazivaču na objekat izvedene klase se može dodeliti pokazivač na objekat osnovne klase samo uz eksplicitnu konverziju. Ovo je interpretacija činjenice da objekat osnovne klase nema sve osobine izvedene klase.
* Objekat osnovne klase može se inicijalizovati objektom izvedene klase, i objektu osnovne klase može se dodeliti objekat izvedene klase bez eksplicitne konverzije. To se obavlja prostim "odsecanjem" članova izvedene klase koji nisu i članovi osnovne klase.

* Virtuelni mehanizam se aktivira ako se objektu pristupa preko reference ili pokazivaèa:

```
class Base {
  //...
public:
  virtual void f();
  //...
};

class Derived : public Base {
  //...
public:
  void f();
};

void g1(Base b) {
  b.f();
}

void g2(Base *pb) {
  pb->f();
}

void g3(Base &rb) {
  rb.f();
}

void main () {
  Derived d;
  g1(d);                    // poziva se Base::f
  g2(&d);                   // poziva se Derived::f
  g3(d);                    // poziva se Derived::f
  Base *pb=new Derived;
  pb->f();                  // poziva se Derived::f
  Derived &rd=d;
  rd.f();                   // poziva se Derived::f
  Base b=d;
  b.f();                    // poziva se Base::f
  delete pb;
  pb=&b;
  pb->f();                  // poziva se Base::f
}
```

* Postupak koji obezbeđuje da se funkcija koja se poziva određuje po tipu objekta, a ne po tipu pokazivaèa ili reference na taj objekat, naziva se *dinamièko vezivanje* (engl. *dynamic binding*). Razrešavanje koja æe se verzija virtuelne funkcije (osnovne ili izvedene klase) pozvati obavlja se u toku izvršavanja programa.

Virtuelni destruktor

* Destruktor je jedna "specifièna funkcija èlanica klase" koja pretvara "živi" objekat u "obiènu gomilu bita u memoriji". Zbog takvog svog znaèenja, nema razloga da i destruktor ne može da bude virtuelna funkcija.
* Virtuelni mehanizam obezbeđuje da se pozove odgovarajuæi destruktor (osnovne ili izvedene klase) kada se objektu pristupa posredno:

```
class Base {
  //...
public:
  virtual ~Base();
  //...
};

class Derived : public Base {
  //...
public:
  ~Derived();
  //...
};

void release (Base *pb) { delete pb; }

void main () {
  Base *pb=new Base;
  Derived *pd=new Derived;
  release(pb);  // poziva se ~Base
  release(pd);  // poziva se ~Derived
}
```

* Kada neka klasa ima neku virtuelnu funkciju, sva je prilika da i njen destruktor (ako ga ima) treba da bude virtuelan.
* Unutar virtuelnog destruktora izvedene klase ne treba eksplicitno pozivati destruktor osnovne klase, jer se on uvek implicitno poziva. Definisanjem destruktora kao virtuelne funkcije obezbeđuje se da se dinamièkim vezivanjem taèno određuje koji æe destruktor (osnovne ili izvedene klase) biti *prvo* pozvan; destruktor osnovne klase se *uvek* izvršava (ili kao jedini ili posle destruktora izvedene klase).
* Konstruktor je funkcija koja od "obiène gomile bita u memoriji" kreira "živi" objekat. Konstruktor se poziva pre nego što se objekat kreira, pa nema smisla da bude virtuelan, što C++ ni ne dozvoljava. Kada se definiše objekat, uvek se navodi i tip (klasa) kome pripada, pa je određen i konstruktor koji se poziva.

Nizovi i izvedene klase

* Objekat izvedene klase je jedna vrsta objekta osnovne klase. Međutim, niz objekata izvedene klase nije jedna vrsta niza objekata osnovne klase. Uopšte, neka kolekcija objekata izvedene klase nije jedna vrsta kolekcije objekata osnovne klase.
* Na primer, iako je automobil jedna vrsta vozila, parking za automobile nije i parking za (sve vrste) vozila, jer na parking za automobile ne mogu da stanu i kamioni (koji su takođe vozila). Ili, ako korisnik neke funkcije prosledi toj funkciji korpu banana (banana je vrsta voæa), ne bi valjalo da mu ta funkcija vrati korpu u kojoj je jedna šljiva (koja je takođe vrsta voæa), smatrajuæi da je korpa banana isto što i korpa bilo kakvog voæa [FAQ].
* Ako se raèuna sa nasleđivanjem, u programu ne treba koristiti nizove objekata, veæ nizove pokazivaèa na objekte. Ako se formira niz objekata izvedene klase i on prenese kao niz objekata osnovne klase (što po prethodno reèenom semantièki nije ispravno, ali je moguæe), može doæi do greške:

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

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

void f(Base *b) { cout<<b[2].bi; }

void main () {
  Derived d[5];
  d[2].bi=77;
  f(d);             // neæe se ispisati 77
}
```

* U prethodnom primeru, funkcija `f` smatra da je dobila niz objekata osnovne klase koji su kraæi (nemaju sve èlanove) od objekata izvedene klase. Kada joj se prosledi niz objekata izvedene klase (koji su duži), funkcija nema naèina da odredi da se niz sastoji samo od objekata izvedene klase. Rezultat je, u opštem sluèaju, neodreðen. Osim toga, dinamièko vezivanje važi samo za funkcije èlanice, a ne i za podatke.
* Pored navedene greške, nije fizièki moguæe u niz objekata osnovne klase smeštati direktno objekte izvedene klase, jer su oni duži, a za svaki element niza je odvojen samo prostor koji je dovoljan za smeštanje objekta osnovne klase.
* Zbog svega što je reèeno, kolekcije (nizove) objekata treba kreirati kao nizove pokazivaèa na objekte:

```
void f(Base **b, int i) { cout<<b[i]->bi; }

void main () {
  Base b1,b2;
  Derived d1,d2,d3;
  Base *b[5];                     // b se može konvertovati u tip
                                  // Base**
  b[0]=&d1; b[1]=&b1; b[2]=&d2;  // konverzije Derived* u Base*
  b[3]=&d3; b[4]=&b2;
  d2.bi=77;
  f(b,2);                         // ispisaæe se 77
}
```

* Kako je objekat izvedene klase jedna vrsta objekta osnovne klase, C++ dozvoljava implicitnu konverziju pokazivaèa `Derived*` u `Base*` (prethodni primer). Zbog logièkog pravila da niz objekata izvedene klase nije jedna vrsta niza objekata osnovne klase, a kako se nizovi ispravno realizuju pomoæu nizova pokazivaèa, C++ ne dozvoljava implicitnu konverziju pokazivaèa `Derived**` (u koji se može konvertovati tip niza pokazivaèa na objekte izvedene klase) u `Base**` (u koji se može konvertovati tip niza pokazivaèa na objekte osnovne klase). Za prethodni primer nije dozvoljeno:

```
void main () {
  Derived *d[5];  // d je tipa Derived**
  //...
  f(d,2);      // nije dozvoljena konverzija Derived** u Base**
}
```

Apstraktne klase

* Èest je sluèaj da neka osnovna klasa nema ni jedan konkretan primerak (objekat), veæ samo predstavlja generalizaciju izvedenih klasa.
* Na primer, svi izlazni, znakovno orijentisani ureðaji raèunara imaju funkciju za ispis jednog znaka, ali se u osnovnoj klasi izlaznog ureðaja ne može definisati naèin ispisa tog znaka, veæ je to specifièno za svaki ureðaj posebno.

Ili, ako iz osnovne klase osoba izvedemo dve klase muškaraca i žena, onda klasa osoba ne može imati primerke, jer ne postoji osoba koja nije ni muškog ni ženskog pola.

* Klasa koja nema instance (objekte), veæ su iz nje samo izvedene druge klase, naziva se *apstraktna klasa* (engl. *abstract class*).
* U jeziku C++, apstraktna klasa sadrži bar jednu virtuelnu funkciju èlanicu koja je u njoj samo deklarisana, ali ne i definisana. Definicije te funkcije daæe izvedene klase. Ovakva virtuelna funkcija naziva se èistom virtuelnom funkcijom. Njena deklaracija u osnovnoj klasi završava se sa `=0`:

```
class OCharDevice {
  //...
public:
  virtual int put (char) =0;     // èista virtuelna funkcija
  //...
};
```

* Apstraktna klasa je klasa koja sadrži bar jednu èistu virtuelnu funkciju. Ovakva klasa ne može imati instance, veæ se iz nje izvode druge klase. Ako se u izvedenoj klasi ne navede definicija neke èiste virtuelne funkcije iz osnovne klase, i ova izvedena klasa je takoðe apstraktna.
* Mogu da se formiraju pokazivaèi i reference na apstraktnu klasu, ali oni ukazuju na objekte izvedenih konkretnih (neapstraktnih) klasa.

Višestruko nasleðivanje

Šta je višestruko nasleðivanje?

* Nekad postoji potreba da izvedena klasa ima osobine više osnovnih klasa istovremeno. Tada se radi o *višestrukom nasleðivanju* (engl. *multiple inheritance*).
* Na primer, motocikl sa prikolicom je jedna vrsta motocikla, ali i jedna vrsta vozila sa tri toèka. Pri tom, motocikl nije vrsta vozila sa tri toèka, niti je vozilo sa tri toèka vrsta motocikla, veæ su ovo dve razlièite klase. Klasa motocikala sa prikolicom naleðuje obe ove klase.
* Klasa se deklariše kao naslednik više klasa tako što se u zaglavlju deklaracije, iza znaka `:`, navode osnovne klase razdvojene zarezima. Ispred svake osnovne klase treba da stoji reè `public`. Na primer:

```
class Derived : public Base1, public Base2, public Base3 {
//...
};
```

* Sva navedena pravila o nasleðenim èlanovima važe i ovde. Konstruktori svih osnovnih klasa se pozivaju pre konstruktora èlanova izvedene klase i konstruktora izvedene klase. Konstruktori osnovnih klasa se pozivaju po redosledu deklarisanja. Destruktori osnovnih klasa se izvršavaju na kraju, posle destruktora osnovne klase i destruktora èlanova.

Virtuelne osnovne klase

* Posmatrajmo sledeæi primer:

```
class B {/*...*/};
class X : public B {/*...*/};
class Y : public B {/*...*/};
class Z : public X, public Y {/*...*/};
```

* U ovom primeru klase `X` i `Y` nasleðuju klasu `B`, a klasa `Z` klase `X` i `Y`. Klasa `Z` ima sve što imaju `X` i `Y`. Kako svaka od klasa `X` i `Y` ima po jedan primerak èlanova klase `B`, to æe klasa `Z` imati dva skupa èlanova klase `B`. Njih je moguæe razlikovati pomoæu operatora `::` (npr. `z.X::i` ili `z.Y::i`).
* Ako ovo nije potrebno, klasu `B` treba deklarisati kao virtuelnu osnovnu klasu:

```
class B {/*...*/};
class X : virtual public B {/*...*/};
class Y : virtual public B {/*...*/};
class Z : public X, public Y {/*...*/};
```

* Sada klasa Z ima samo jedan skup èlanova klase B.
* Ako neka izvedena klasa ima virtuelne i nevirtuelne osnovne klase, onda se konstruktori virtuelnih osnovnih klasa pozivaju pre konstruktora nevirtuelnih osnovnih klasa, po redosledu deklarisanja. Svi konstruktori osnovnih klasa se, naravno, pozivaju pre konstruktora èlanova i konstruktora izvedene klase.

Privatno i zaštiæeno izvoðenje

Šta je privatno i zaštiæeno izvoðenje?

* Ključna reè public u zaglavlju deklaracije izvedene klase znaèila je da je osnovna klasa javna, odnosno da su svi javni èlanovi osnovne klase ujedno i javni èlanovi izvedene klase. Privatni èlanovi osnovne klase nisu dostupni izvedenoj klasi, a zaštiæeni èlanovi osnovne klase ostaju zaštiæeni i u izvedenoj klasi. Ovakvo izvoðenje se u jeziku C++ naziva još i javno izvoðenje.
* Moguæe je u zaglavlje deklaracije, ispred imena osnovne klase, umesto reèi public staviti reè private, što se i podrazumeva ako se ne navede ništa drugo. U ovom slučaju javni i zaštiæeni èlanovi osnovne klase postaju privatni èlanovi izvedene klase. Ovakvo izvoðenje se u jeziku C++ naziva privatno izvoðenje.
* Moguæe je u zaglavlje deklaracije ispred imena osnovne klase staviti reè protected. Tada javni i zaštiæeni èlanovi osnovne klase postaju zaštiæeni èlanovi izvedene klase. Ovakvo izvoðenje se u jeziku C++ naziva zaštiæeno izvoðenje.
* U svakom slučaju, privatni èlanovi osnovne klase nisu dostupni izvedenoj klasi. Ona može samo nadalje "sakriti" zaštiæene i javne èlanove osnovne klase izborom naèina izvoðenja.
* U slučaju privatnog i zaštiæenog izvoðenja, kada izvedena klasa smanjuje nivo prava pristupa do javnih i zaštiæenih èlanova osnovne klase, može se ovaj nivo vratiti na poèetni eksplicitnim navoðenjem deklaracije javnog ili zaštiæenog èlana osnovne klase u javnom ili zaštiæenom delu izvedene klase. U svakom slučaju, izvedena klasa ne može poveæati nivo vidljivosti èlana osnovne klase.

```
class Base {
  int bpriv;
protected:
  int bprot;
public:
  int bpub;
};

class PrivDerived : Base {  // privatno izvoðenje
protected:
  Base::bprot;              // vraæanje na nivo protected
public:
  PrivDerived () {
    bprot=2; bpub=3;        // može se pristupiti
  }
};

class ProtDerived : protected Base {  // zaštiæeno izvoðenje
public: //...
};

void main () {
  PrivDerived pd;
  pd.bpub=0;                // greška: bpub nije javni èlan
}
```

* Pokazivaè na izvedenu klasu može se implicitno konvertovati u pokazivaè na javnu osnovnu klasu. Pokazivaè na izvedenu klasu može se implicitno konvertovati u pokazivaè na privatnu osnovnu klasu samo unutar izvedene klase, jer samo se unutar nje zna da je ona izvedena. Isto važi i za reference.

Semantièka razlika između privatnog i javnog izvođenja

* Javno izvođenje realizuje koncept nasleđivanja, koji je iskazan relacijom "B je jedna vrsta A" (*a-kind-of*). Ova relacija podrazumeva da izvedena klasa ima sve što i osnovna, što znaèi da je sve što je dostupno korisniku osnovne klase, dostupno i korisniku izvedene klase. U jeziku C++ to znaèi da javni èlanovi osnovne klase treba da budu javni i u izvedenoj klasi.
* Privatno izvođenje ne odslikava ovu relaciju, jer korisnik izvedene klase ne može da pristupi onome èemu je mogao pristupiti u osnovnoj klasi. Javnim èlanovima osnovne klase mogu pristupiti samo funkcije èlanice izvedene klase, što znaèi da izvedena klasa u sebi sakriva osnovnu klasu. Zato privatno izvođenje realizuje jednu sasvim drugu relaciju, relaciju "A je deo od B" (*a-part-of*). Ovo je suštinski različito od relacije nasleđivanja.
* Pošto privatno izvođenje realizuje relaciju "A je deo od B", ono je semantièki ekvivalentno sa implementacijom kada klasa B sadrži èlana koji je tipa A.
* Prilikom projektovanja, treba strogo voditi raèuna o tome u kojoj su od ove dve relacije neke dve uoèene klase. U zavisnosti od toga treba izabrati naèin izvođenja.
* Ako je relacija između dve klase "A je deo od B", izbor između privatnog izvođenja i èlanstva zavisi od manje važnih detalja: da li je potrebno redefinisati virtuelne funkcije klase A, da li je unutar klase B potrebno konvertovati pokazivaèe, da li klasa B treba da sadrži jedan ili više primeraka klase A i slièno [FAQ].

Zadaci:

11. U klasu èasovnika iz zadatka 10, dodati jednu èistu virtuelnu funkciju `getTime` koja vraæa niz znakova. Iz ove apstraktne klase, izvesti klase koje predstavljaju èasovnike koji prikazuju vreme u jednom od formata (23:50, 23.50, 11:50 pm, 11:50), tako što æe imati definisanu funkciju `getTime`, koja vraæa niz znakova koji predstavlja tekuæe vreme u odgovarajuæem formatu. Definisati i globalnu funkciju za ispis vremena èasovnika na standardni izlaz (`cout`), koja koristi virtuelnu funkciju `getTime`. U glavnom programu kreirati nekoliko objekata pojedinih vrsta èasovnika, postavljati njihova vremena, i ispisivati ih.

12. Skicirati klasu `Screen` koja ima operacije za brisanje ekrana, ispis jednog znaka na odgovarajuæu poziciju na ekranu, i ispis niza znakova u oblast definisanu pravougaonikom na ekranu (redom red po red). Ukoliko postoji moguænost, realizovati ovu klasu za postojeæe tekstualno okruženje. Koristeæi ovu klasu, realizovati klasu `Window` koja predstavlja prozor, kao izvedenu klasu klase `View` iz zadatka 8. Prozor treba da ima moguænosti pomeranja, promene veliène, minimizacije, maksimizacije, kao i kreiranja (otvaranja) i uništavanja (zatvaranja). Definisati i virtuelnu funkciju `draw`. Svi prozori treba da budu uvezani u listu, po redosledu kreiranja. U glavnom programu kreirati nekoliko prozora, i vršiti operacije nad njima.

13. Realizovati klasu za jednostavnu kontrolu tastature. Ova klasa treba da ima jednu funkciju `run` koja æe neprekidno izvršavati petlju, sve dok se ne pritisne taster za izlaz Alt-X. Ukoliko se pritisne taster F3, ova funkcija treba da kreira jedan dinamièki objekat klase `Window` iz prethodnog zadatka. Ukoliko se pritisne taster F6, ova funkcija treba da prebaci fokus na sledeæi prozor u listi prozora po redosledu kreiranja. Ukoliko se pritisne taster Alt-F3, prozor koji ima fokus treba da se zatvori. Ukoliko se pritisne neki drugi taster, ova funkcija treba da pozove èistu virtuelnu funkciju `handleEvent` klase `View`, koju treba dodati. Ovoj funkciji se prosleđuje posebna struktura podataka koja u sebi sadrži informaciju da se radi o događaju pritiska na taster, i kôd tastera koji je pritisnut. U klasi `Window` treba definisati funkciju `handleEvent`, tako da odgovara na tastere F5 (smanjenje prozora na minimum) i Shift-F5 (maksimizacija prozora). Glavni program treba da formira jedan objekat klase za kontrolu tastature, i da pozove njegovu funkciju `run`.

14. Skicirati klasu koja æe predstavljati apstrakciju svih izveštaja koji se mogu javiti u nekoj aplikaciji. Izveštaj treba da bude interno predstavljen kao niz objekata klase `Entity`. Klasa `Entity` æe predstavljati apstrakciju svih entiteta koji se mogu naæi u nekom izveštaju (tekst, slika, kontrolni znaci, fajl, okviri, itd.). Ova klasa `Entity` ima èistu virtuelnu funkciju `draw` za iscrtavanje na ekranu, na odgovarajuæoj poziciji, funkciju `print` za izlaz na štampaè, i funkciju koja daje dimenzije entiteta u nekim jedinicama. Klasa izveštaja treba da ima funkciju `draw` za iscrtavanje izveštaja na ekranu, na poziciji koja je tekuæa, funkciju za zadavanje tekuæe pozicije izveštaja na ekranu, i funkciju `print` za štampanje izveštaja. Klasa `Entity` ima i funkciju `doubleClick` koja treba služi kao akcija na dupli klik miša, kojim korisnik "ulazi" u dati entitet. Klasa izveštaja ima funkciju `doubleClick`, sa argumetima koji daju koordinate duplog klika. Ova funkcija treba da pronađe entitet na koji se odnosi klik i da pozove njegovu funkciju `doubleClick`. Skicirati klasu `Entity`, a u klasi izveštaja realizovati funkcije `draw`, `print`, `doubleClick`, i funkciju za zadavanje tekuæe pozicije, kao i sve potrebne ostale pomoæne funkcije (konstruktore i destruktor).

Osnovi objektnog modelovanja

Apstraktni tipovi podataka

* Apstraktni tipovi podataka su realizacije struktura podataka sa pridruženim protokolima (operacijama i definisanim naèinom i redosledom pozivanja tih operacija). Na primer, *red* (engl. *queue*) je struktura elemenata koja ima operacije stavljanja i uzimanja elemenata u strukturu, pri èemu se elementi uzimaju po istom redosledu po kom su stavljeni.
* Kada se realizuju strukture podataka (apstraktni tipovi podataka), najèešæe nije bitno koji je tip elementa strukture, veæ samo skup operacija. Naèini realizacija tih operacija ne zavise od tipa elementa, veæ samo od tipa strukture.
* Za realizaciju apstraktnih tipova podataka kod kojih tip nije bitan, u jeziku C++ postoje *šabloni* (engl. *templates*). Šablon klase predstavlja definiciju èitavog skupa klasa koje se razlikuju samo po tipu elementa i eventualno po dimenzijama. Šabloni klasa se ponekad nazivaju i generièkim klasama.
* Konkretna klasa generisana iz šablona dobija se navoðenjem stvarnog tipa elementa.
* Formalni argumenti šablona zadaju se u zaglavlju šablona:

```
template <class T>
class Queue {
public:
  Queue ();
 ~Queue ();

  void put (const T&);
  T    get ();

//...
};
```

* Konkretna generisana klasa dobija se samo navoðenjem imena šablona, uz definisanje stvarnih argumenata šablona. Stvarni argumenti šablona su tipovi i eventualno celobrojne dimenzije. Konkretna klasa se generiše na mestu navoðenja, u fazi prevoðenja. Na primer, red dogaðaja može se kreirati na sledeæi naèin:

```
class Event;
Queue<Event*> que;

que.put(e);
if (que.get()->isUrgent()) ...
```

* Generisanje je samo stvar automatskog generisanja parametrizovanog koda istog oblika, a nema nikakve veze sa izvršavanjem. Generisane klase su kao i obiène klase i nemaju nikakve meðusobne veze.

Projektovanje apstraktnih tipova podataka

* Apstraktni tipovi podataka (strukture podataka) su veoma èesto korišæeni elementi svakog programa. Projektovanje biblioteke klasa koje realizuju standardne strukture podataka je veoma delikatan posao. Ovde æe biti prikazana konstrukcija dve èeste linearne strukture podataka:

1. Kolekcija (engl. *collection*) je linearna, neureðena struktura elemenata koja ima samo operacije stavljanja elementa i izbacivanja datog elementa iz strukture. Redosled elemenata nije bitan, a elementi se mogu i ponavljati.
2. Red (engl. *queue*) je linearna, ureðena struktura elemenata. Elementi su ureðeni po redosledu stavljanja. Operacija uzimanja vraæa element koji je najdavnije stavljen u strukturu.

* Važan koncept pridružen strukturama je pojam *iteratora* (engl. *iterator*). Iterator je objekat pridružen linearnoj strukturi koji služi za pristup redom elementima strukture. Iterator ima operacije za postavljanje na poèetak strukture, za pomeranje na sledeæi element strukture, za pristup do tekuæeg elementa na koji ukazuje i operaciju za ispitivanje da li je došao do kraja strukture. Za svaku strukturu može se kreirati proizvoljno mnogo objekata-iteratora i svaki od njih pamti svoju poziciju.
* Kod realizacije biblioteke klasa za strukture podataka bitno je razlikovati *protokol* strukture koji definiše njenu semantiku, od njene *implementacije*.
* Protokol strukture ureðuje znaèenje njenih operacija, potreban naèin ili redosled pozivanja itd.

* Implementacija se odnosi na naèin smeštanja elemenata u memoriju, organizaciju njihove veze itd. Važan element implementacije je da li je ona ogranièena ili nije. Ogranièena realizacija se oslanja na statièki dimenzionisani niz elemenata, dok se neogranièena realizacija odnosi na dinamièku strukturu (tipièno listu).
* Na primer, protokol reda izgleda otprilike ovako:

```
template <class T>
class Queue {
public:

  virtual IteratorQueue<T>* createIterator() const =0;

  virtual void put    (const T&) =0;
  virtual T    get    () =0;
  virtual void clear  () =0;

  virtual const T& first    () const =0;
  virtual int      isEmpty  () const =0;
  virtual int      isFull   () const =0;
  virtual int      length   () const =0;
  virtual int      location (const T&) const =0;

};
```

* Da bi se korisniku obezbedile obe realizacije (ogranièena i neogranièena), postoje dve izvedene klase iz navedene apstraktne klase koja definiše interfejs reda. Jedna od njih podrazumeva ogranièenu (engl. *bounded*) realizaciju, a druga neogranièenu (engl. *unbounded*). Na primer:

```
template <class T, int N>
class QueueB : public Queue<T> {
public:

  QueueB () {}
  QueueB (const Queue<T>&);
  virtual ~QueueB () {}

  Queue<T>& operator= (const Queue<T>&);

  virtual IteratorQueue<T>* createIterator() const;

  virtual void put    (const T& t);
  virtual T    get    ()          ;
  virtual void clear  ()          ;

  virtual const T& first    () const;
  virtual int      isEmpty  () const;
  virtual int      isFull   () const;
  virtual int      length   () const;
  virtual int      location (const T& t) const;

};
```

* Treba obratiti pažnju na naèin kreiranja iteratora. Korisniku je dovoljan samo opšti, zajednièki interfejs iteratora. Korisnik ne treba da zna ništa o specifiènostima realizacije iteratora i njegovoj vezi sa konkretnom izvedenom klasom reda. Zato je definisana osnovna apstraktna klasa iteratora, iz koje su izvedene klase za iteratore vezane za dve posebne realizacije reda:

```
template <class T>
class IteratorQueue {
public:

  virtual ~IteratorQueue () {}

  virtual void reset() =0;
  virtual int  next () =0;

  virtual int  isDone() const =0;
  virtual const T* currentItem() const =0;

};
```

* Izvedene klase reda kreiraæe posebne, njima specifiène iteratore koji se uklapaju u zajednièki interfejs iteratora:

```
template<class T, int N>
IteratorQueue<T>* QueueB<T,N>::createIterator() const {
  return new IteratorQueueB<T,N>(this);
}
```

* Sama realizacija ogranièene i neogranièene strukture oslanja se na dve klase koje imaju sledeæe interfejse:

```
template <class T>
class Unbounded {
public:

  Unbounded ();
  Unbounded (const Unbounded<T>&);
 ~Unbounded ();

  Unbounded<T>& operator= (const Unbounded<T>&);

  void append (const T&);
  void insert (const T&, int at=0);
  void remove (const T&);
  void remove (int at=0);
  void clear  ();

  int      isEmpty  () const;
  int      isFull   () const;
  int      length   () const;
  const T& first    () const;
  const T& last     () const;
  const T& itemAt   (int at) const;
  T&       itemAt   (int at);
  int      location (const T&) const;

};
```

```
template <class T, int N>
class Bounded {
public:

  Bounded ();
  Bounded (const Bounded<T,N>&);
 ~Bounded ();

  Bounded<T,N>& operator= (const Bounded<T,N>&);

  void append (const T&);
  void insert (const T&, int at=0);
  void remove (const T&);
  void remove (int at=0);
  void clear  ();

  int      isEmpty  () const;
  int      isFull   () const;
  int      length   () const;
  const T& first    () const;
  const T& last     () const;
  const T& itemAt   (int at) const;
  T&       itemAt   (int at);
  int      location (const T&) const;

};
```

* Definisana kolekcija se može koristiti na primer na sledeæi naèin:

```
class Event {
  //...
};

typedef QueueB<Event*,MAXEV> EventQueue;
typedef IteratorQueue<Event*> Iterator;

//...
EventQueue que;
que.put(e);

Iterator* it=que.createIterator();
for (; !it->isDone(); it->next())
   (*it->currentItem())->handle();
delete it;
```

* Promena orijentacije na ogranièeni red je veoma jednostavna. Ako se želi neogranièeni red, dovoljno je promeniti samo:

```
typedef QueueU<Event*> EventQueue;
```

* Kompletan kôd izgleda ovako:

* Datoteka `unbound.h`:

```
// Project:  Real-Time Programming
// Subject:  Abstract Data Structures
// Module:   Unbound
// File:     unbound.h
// Date:     3.11.1996.
// Author:   Dragan Milicev
// Contents:
//      Class templates: Unbounded
//                       IteratorUnbounded


////////////////////////////////////////////////////////////////////
// class template Unbounded
////////////////////////////////////////////////////////////////////

template <class T>
class Unbounded {
public:

  Unbounded ();
  Unbounded (const Unbounded<T>&);
 ~Unbounded ();

  Unbounded<T>& operator= (const Unbounded<T>&);

  void append (const T&);
  void insert (const T&, int at=0);
  void remove (const T&);
  void remove (int at=0);
  void clear  ();

  int      isEmpty  () const;
  int      isFull   () const;
  int      length   () const;
  const T& first    () const;
  const T& last     () const;
  const T& itemAt   (int at) const;
  T&       itemAt   (int at);
  int      location (const T&) const;

protected:

  void copy (const Unbounded<T>&);

private:

  friend class  IteratorUnbounded<T>;
  friend struct Element<T>;

  void remove (Element<T>*);

  Element<T>* head;
  int size;

};
```

```
template <class T>
struct Element {
  T t;
  Element<T> *prev, *next;
  Element (const T&);
  Element (const T&, Element<T>* next);
  Element (const T&, Element<T>* prev, Element<T>* next);
};



template<class T>
Element<T>::Element (const T& e) : t(e), prev(0), next(0) {}


template<class T>
Element<T>::Element (const T& e, Element<T> *n)
    : t(e), prev(0), next(n) {
  if (n!=0) n->prev=this;
}


template<class T>
Element<T>::Element (const T& e, Element<T> *p, Element<T> *n)
    : t(e), prev(p), next(n) {
  if (n!=0) n->prev=this;
  if (p!=0) p->next=this;
}
```

```
template<class T>
void Unbounded<T>::remove (Element<T>* e) {
  if (e==0) return;
  if (e->next!=0) e->next->prev=e->prev;
  if (e->prev!=0) e->prev->next=e->next;
  else head=e->next;
  delete e;
  size--;
}



template<class T>
void Unbounded<T>::copy (const Unbounded<T>& r) {
  size=0;
  for (Element<T>* cur=r.head; cur!=0; cur=cur->next) append(cur->t);
}


template<class T>
void Unbounded<T>::clear () {
  for (Element<T> *cur=head, *temp=0; cur!=0; cur=temp) {
    temp=cur->next;
    delete cur;
  }
  head=0;
  size=0;
}
```

```
template<class T>
int Unbounded<T>::isEmpty () const { return size==0; }


template<class T>
int Unbounded<T>::isFull () const { return 0; }



template<class T>
int Unbounded<T>::length () const { return size; }


template<class T>
const T& Unbounded<T>::first () const { return itemAt(0); }


template<class T>
const T& Unbounded<T>::last () const { return itemAt(length()-1); }
```

```
template<class T>
const T& Unbounded<T>::itemAt (int at) const {
  static T except;
  if (isEmpty()) return except;  // Exception!
  if (at>=length()) at=length()-1;
  if (at<0) at=0;
  int i=0;
  for (Element<T> *cur=head; i<at; cur=cur->next, i++);
  return cur->t;
}


template<class T>
T& Unbounded<T>::itemAt (int at) {
  static T except;
  if (isEmpty()) return except;  // Exception!
  if (at>=length()) at=length()-1;
  if (at<0) at=0;
  int i=0;
  for (Element<T> *cur=head; i<at; cur=cur->next, i++);
  return cur->t;
}


template<class T>
int Unbounded<T>::location (const T& e) const {
  int i=0;
  for (Element<T> *cur=head; cur!=0; cur=cur->next, i++)
    if (cur->t==e) return i;
  return -1;
}
```

```
template<class T>
void Unbounded<T>::append (const T& t) {
  if (head==0) head=new Element<T>(t);
  else {
    for (Element<T> *cur=head; cur->next!=0; cur=cur->next);
    new Element<T>(t,cur,0);
  }
  size++;
}


template<class T>
void Unbounded<T>::insert (const T& t, int at) {
  if ((at>size)||(at<0)) return;
  if (at==0) head=new Element<T>(t,head);
  else if (at==size) { append(t); return; }
  else {
    int i=0;
    for (Element<T> *cur=head; i<at; cur=cur->next, i++);
    new Element<T>(t,cur->prev,cur);
  }
  size++;
}


template<class T>
void Unbounded<T>::remove (int at) {
  if ((at>=size)||(at<0)) return;
  int i=0;
  for (Element<T> *cur=head; i<at; cur=cur->next, i++);
  remove(cur);
}


template<class T>
void Unbounded<T>::remove (const T& t) {
  remove(location(t));
}
```

```
template<class T>
Unbounded<T>::Unbounded () : size(0), head(0) {}


template<class T>
Unbounded<T>::Unbounded (const Unbounded<T>& r) : size(0), head(0) {
  copy(r);
}


template<class T>
Unbounded<T>& Unbounded<T>::operator= (const Unbounded<T>& r) {
  clear();
  copy(r);
  return *this;
}


template<class T>
Unbounded<T>::~Unbounded () { clear(); }
```

```
////////////////////////////////////////////////////////////////////
// class template IteratorUnbounded
////////////////////////////////////////////////////////////////////

template <class T>
class IteratorUnbounded {
public:

  IteratorUnbounded (const Unbounded<T>*);
  IteratorUnbounded (const IteratorUnbounded<T>&);
 ~IteratorUnbounded ();

  IteratorUnbounded<T>& operator= (const IteratorUnbounded<T>&);
  int operator== (const IteratorUnbounded<T>&);
  int operator!= (const IteratorUnbounded<T>&);

  void reset();
  int  next ();

  int      isDone() const;
  const T* currentItem() const;

private:

  const Unbounded<T>* theSupplier;
  Element<T>* cur;

};
```

```
template <class T>
IteratorUnbounded<T>::IteratorUnbounded (const Unbounded<T>* ub)
  : theSupplier(ub), cur(theSupplier->head) {}


template <class T>
IteratorUnbounded<T>::IteratorUnbounded (const IteratorUnbounded<T>& r)
  : theSupplier(r.theSupplier), cur(r.cur) {}


template <class T>
IteratorUnbounded<T>& IteratorUnbounded<T>::operator= (const
IteratorUnbounded<T>& r) {
  theSupplier=r.theSupplier;
  cur=r.cur;
  return *this;
}


template <class T>
IteratorUnbounded<T>::~IteratorUnbounded () {}
```

```
template <class T>
int IteratorUnbounded<T>::operator== (const IteratorUnbounded<T>& r) {
  return (theSupplier==r.theSupplier)&&(cur==r.cur);
}


template <class T>
int IteratorUnbounded<T>::operator!= (const IteratorUnbounded<T>& r) {
  return !(*this==r);
}
```

```
template <class T>
void IteratorUnbounded<T>::reset () {
  cur=theSupplier->head;
}


template <class T>
int IteratorUnbounded<T>::next () {
  if (cur!=0) cur=cur->next;
  return !isDone();
}


template <class T>
int IteratorUnbounded<T>::isDone () const {
  return (cur==0);
}


template <class T>
const T* IteratorUnbounded<T>::currentItem () const {
  if (isDone()) return 0;
  else return &(cur->t);
}
```

* Datoteka `bound.h`:

```
// Project:  Real-Time Programming
// Subject:  Abstract Data Structures
// Module:   Bound
// File:     bound.h
// Date:     3.11.1996.
// Author:   Dragan Milicev
// Contents:
//      Class templates: Bounded
//                       IteratorBounded


////////////////////////////////////////////////////////////////////
// class template Bounded
////////////////////////////////////////////////////////////////////

template <class T, int N>
class Bounded {
public:

  Bounded ();
  Bounded (const Bounded<T,N>&);
 ~Bounded ();

  Bounded<T,N>& operator= (const Bounded<T,N>&);

  void append (const T&);
  void insert (const T&, int at=0);
  void remove (const T&);
  void remove (int at=0);
  void clear  ();

  int      isEmpty  () const;
  int      isFull   () const;
  int      length   () const;
  const T& first    () const;
  const T& last     () const;
  const T& itemAt   (int at) const;
  T&       itemAt   (int at);
  int      location (const T&) const;

protected:

  void copy (const Bounded<T,N>&);

private:

  T dep[N];
  int size;

};
```

```
template<class T, int N>
void Bounded<T,N>::copy (const Bounded<T,N>& r) {
  size=0;
  for (int i=0; i<r.size; i++) append(r.itemAt(i));
}


template<class T, int N>
void Bounded<T,N>::clear () {
  size=0;
}


template<class T, int N>
int Bounded<T,N>::isEmpty () const { return size==0; }


template<class T, int N>
int Bounded<T,N>::isFull () const { return size==N; }


template<class T, int N>
int Bounded<T,N>::length () const { return size; }
```

```
template<class T, int N>
const T& Bounded<T,N>::first () const { return itemAt(0); }


template<class T, int N>
const T& Bounded<T,N>::last () const { return itemAt(length()-1); }


template<class T, int N>
const T& Bounded<T,N>::itemAt (int at) const {
  static T except;
  if (isEmpty()) return except;  // Exception!
  if (at>=length()) at=length()-1;
  if (at<0) at=0;
  return dep[at];
}


template<class T, int N>
T& Bounded<T,N>::itemAt (int at) {
  static T except;
  if (isEmpty()) return except;  // Exception!
  if (at>=length()) at=length()-1;
  if (at<0) at=0;
  return dep[at];
}


template<class T, int N>
int Bounded<T,N>::location (const T& e) const {
  for (int i=0; i<size; i++) if (dep[i]==e) return i;
  return -1;
}
```

```
template<class T, int N>
void Bounded<T,N>::append (const T& t) {
  if (isFull()) return;
  dep[size++]=t;
}


template<class T, int N>
void Bounded<T,N>::insert (const T& t, int at) {
  if (isFull()) return;
  if ((at>size)||(at<0)) return;
  for (int i=size-1; i>=at; i--) dep[i+1]=dep[i];
  dep[at]=t;
  size++;
}


template<class T, int N>
void Bounded<T,N>::remove (int at) {
  if ((at>=size)||(at<0)) return;
  for (int i=at+1; i<size; i++) dep[i-1]=dep[i];
  size--;
}


template<class T, int N>
void Bounded<T,N>::remove (const T& t) {
  remove(location(t));
}
```

```
template<class T, int N>
Bounded<T,N>::Bounded () : size(0) {}


template<class T, int N>
Bounded<T,N>::Bounded (const Bounded<T,N>& r) : size(0) {
  copy(r);
}


template<class T, int N>
Bounded<T,N>& Bounded<T,N>::operator= (const Bounded<T,N>& r) {
  clear();
  copy(r);
  return *this;
}


template<class T, int N>
Bounded<T,N>::~Bounded () { clear(); }
```

```
////////////////////////////////////////////////////////////////////
// class template IteratorBounded
////////////////////////////////////////////////////////////////////

template <class T, int N>
class IteratorBounded {
public:

  IteratorBounded (const Bounded<T,N>*);
  IteratorBounded (const IteratorBounded<T,N>&);
 ~IteratorBounded ();

  IteratorBounded<T,N>& operator= (const IteratorBounded<T,N>&);
  int operator== (const IteratorBounded<T,N>&);
  int operator!= (const IteratorBounded<T,N>&);

  void reset();
  int  next ();

  int      isDone() const;
  const T* currentItem() const;

private:

  const Bounded<T,N>* theSupplier;
  int cur;

};
```

```
template <class T, int N>
IteratorBounded<T,N>::IteratorBounded (const Bounded<T,N>* b)
  : theSupplier(b), cur(0) {}


template <class T, int N>
IteratorBounded<T,N>::IteratorBounded (const IteratorBounded<T,N>& r)
  : theSupplier(r.theSupplier), cur(r.cur) {}


template <class T, int N>
IteratorBounded<T,N>& IteratorBounded<T,N>::operator= (const
IteratorBounded<T,N>& r) {
  theSupplier=r.theSupplier;
  cur=r.cur;
  return *this;
}


template <class T, int N>
IteratorBounded<T,N>::~IteratorBounded () {}
```

```
template <class T, int N>
int IteratorBounded<T,N>::operator== (const IteratorBounded<T,N>& r) {
  return (theSupplier==r.theSupplier)&&(cur==r.cur);
}


template <class T, int N>
int IteratorBounded<T,N>::operator!= (const IteratorBounded<T,N>& r) {
  return !(*this==r);
}
```

```
template <class T, int N>
void IteratorBounded<T,N>::reset () {
  cur=0;
}


template <class T, int N>
int IteratorBounded<T,N>::next () {
  if (!isDone()) cur++;
  return !isDone();
}


template <class T, int N>
int IteratorBounded<T,N>::isDone () const {
  return (cur>=theSupplier->length());
}


template <class T, int N>
const T* IteratorBounded<T,N>::currentItem () const {
  if (isDone()) return 0;
  else return &theSupplier->itemAt(cur);
}
```

* Datoteka `collect.h`:

```
// Project:  Real-Time Programming
// Subject:  Abstract Data Structures
// Module:   Collection
// File:     collect.h
// Date:     5.11.1996.
// Author:   Dragan Milicev
// Contents:
//      Class templates: Collection
//                       CollectionB
//                       CollectionU
//                       IteratorCollection
//                       IteratorCollectionB
//                       IteratorCollectionU


#include "bound.h"
#include "unbound.h"


////////////////////////////////////////////////////////////////////////
// class template Collection
////////////////////////////////////////////////////////////////////////

template <class T>
class Collection {
public:

  virtual ~Collection () {}

  Collection<T>& operator= (const Collection<T>&);

  virtual IteratorCollection<T>* createIterator() const =0;

  virtual void add    (const T&) =0;
  virtual void remove (const T&) =0;
  virtual void remove (int at) =0;
  virtual void clear  () =0;

  virtual int      isEmpty  () const =0;
  virtual int      isFull   () const =0;
  virtual int      length   () const =0;
  virtual int      location (const T&) const =0;

protected:

  void copy (const Collection<T>&);

};
```

```
////////////////////////////////////////////////////////////////////
// class template IteratorCollection
////////////////////////////////////////////////////////////////////

template <class T>
class IteratorCollection {
public:

  virtual ~IteratorCollection () {}

  virtual void reset() =0;
  virtual int  next () =0;

  virtual int  isDone() const =0;
  virtual const T* currentItem() const =0;

};
```

```
template<class T>
void Collection<T>::copy (const Collection<T>& r) {
  for (IteratorCollection<T>* it=r.createIterator();
       !it->isDone(); it->next())
       if (!isFull()) add(*it->currentItem());
  delete it;
}

template<class T>
Collection<T>& Collection<T>::operator= (const Collection<T>& r) {
  clear();
  copy(r);
  return *this;
}
```

```
////////////////////////////////////////////////////////////////////
// class template CollectionB
////////////////////////////////////////////////////////////////////

template <class T, int N>
class CollectionB : public Collection<T> {
public:

  CollectionB () {}
  CollectionB (const Collection<T>&);
  virtual ~CollectionB () {}

  Collection<T>& operator= (const Collection<T>&);

  virtual IteratorCollection<T>* createIterator() const;

  virtual void add    (const T& t)             { rep.append(t); }
  virtual void remove (const T& t)             { rep.remove(t); }
  virtual void remove (int at)                 { rep.remove(at); }
  virtual void clear  ()                       { rep.clear(); }

  virtual int      isEmpty  () const           { return rep.isEmpty(); }
  virtual int      isFull   () const           { return rep.isFull(); }
  virtual int      length   () const           { return rep.length(); }
  virtual int      location (const T& t) const { return rep.location(t); }

private:
  friend class IteratorCollectionB<T,N>;
  Bounded<T,N> rep;
};
```

```
////////////////////////////////////////////////////////////////////
// class template IteratorCollectionB
////////////////////////////////////////////////////////////////////

template <class T, int N>
class IteratorCollectionB : public IteratorCollection<T>,
                     private IteratorBounded<T,N> {
public:

  IteratorCollectionB (const CollectionB<T,N>* c)
                       : IteratorBounded<T,N>(&c->rep) {}
  virtual ~IteratorCollectionB () {}

  virtual void reset() { IteratorBounded<T,N>::reset(); }
  virtual int  next () { return IteratorBounded<T,N>::next(); }

  virtual int  isDone() const { return IteratorBounded<T,N>::isDone(); }
  virtual const T* currentItem() const
                       { return IteratorBounded<T,N>::currentItem(); }

};
```

```
template<class T, int N>
CollectionB<T,N>::CollectionB (const Collection<T>& r) {
  copy(r);
}

template<class T, int N>
Collection<T>& CollectionB<T,N>::operator= (const Collection<T>& r) {
  return Collection<T>::operator=(r);
}

template<class T, int N>
IteratorCollection<T>* CollectionB<T,N>::createIterator() const {
  return new IteratorCollectionB<T,N>(this);
}
```

```
//////////////////////////////////////////////////////////////////////
// class template CollectionU
//////////////////////////////////////////////////////////////////////

template <class T>
class CollectionU : public Collection<T> {
public:

  CollectionU () {}
  CollectionU (const Collection<T>&);
  virtual ~CollectionU () {}

  Collection<T>& operator= (const Collection<T>&);

  virtual IteratorCollection<T>* createIterator() const;

  virtual void add    (const T& t)             { rep.append(t); }
  virtual void remove (const T& t)             { rep.remove(t); }
  virtual void remove (int at)                 { rep.remove(at); }
  virtual void clear  ()                       { rep.clear(); }

  virtual int      isEmpty  () const           { return rep.isEmpty(); }
  virtual int      isFull   () const           { return rep.isFull(); }
  virtual int      length   () const           { return rep.length(); }
  virtual int      location (const T& t) const { return rep.location(t); }

private:
  friend class IteratorCollectionU<T>;
  Unbounded<T> rep;
};
```

```
////////////////////////////////////////////////////////////////////
// class template IteratorCollectionU
////////////////////////////////////////////////////////////////////

template <class T>
class IteratorCollectionU : public IteratorCollection<T>,
                      private IteratorUnbounded<T> {
public:

  IteratorCollectionU (const CollectionU<T>* c)
                        : IteratorUnbounded<T>(&c->rep) {}
  virtual ~IteratorCollectionU () {}

  virtual void reset() { IteratorUnbounded<T>::reset(); }
  virtual int  next () { return IteratorUnbounded<T>::next(); }

  virtual int  isDone() const { return IteratorUnbounded<T>::isDone(); }
  virtual const T* currentItem() const
                       { return IteratorUnbounded<T>::currentItem(); }

};



template<class T>
CollectionU<T>::CollectionU (const Collection<T>& r) {
  copy(r);
}

template<class T>
Collection<T>& CollectionU<T>::operator= (const Collection<T>& r) {
  return Collection<T>::operator=(r);
}

template<class T>
IteratorCollection<T>* CollectionU<T>::createIterator() const {
  return new IteratorCollectionU<T>(this);
}
```

* Datoteka `queue.h`:

```
// Project:  Real-Time Programming
// Subject:  Abstract Data Structures
// Module:   Queue
// File:     queue.h
// Date:     5.11.1996.
// Author:   Dragan Milicev
// Contents:
//      Class templates: Queue
//                       QueueB
//                       QueueU
//                       IteratorQueue
//                       IteratorQueueB
//                       IteratorQueueU


#include "bound.h"
#include "unbound.h"


////////////////////////////////////////////////////////////////////////
// class template Queue
////////////////////////////////////////////////////////////////////////

template <class T>
class Queue {
public:

  virtual ~Queue () {}

  Queue<T>& operator= (const Queue<T>&);

  virtual IteratorQueue<T>* createIterator() const =0;

  virtual void put    (const T&) =0;
  virtual T    get    () =0;
  virtual void clear  () =0;

  virtual const T& first    () const =0;
  virtual int      isEmpty  () const =0;
  virtual int      isFull   () const =0;
  virtual int      length   () const =0;
  virtual int      location (const T&) const =0;

protected:

  void copy (const Queue<T>&);

};
```

```
////////////////////////////////////////////////////////////////////
// class template IteratorQueue
////////////////////////////////////////////////////////////////////

template <class T>
class IteratorQueue {
public:

  virtual ~IteratorQueue () {}

  virtual void reset() =0;
  virtual int  next () =0;

  virtual int  isDone() const =0;
  virtual const T* currentItem() const =0;

};




template<class T>
void Queue<T>::copy (const Queue<T>& r) {
  for (IteratorQueue<T>* it=r.createIterator();
       !it->isDone(); it->next())
       if (!isFull()) put(*it->currentItem());
  delete it;
}

template<class T>
Queue<T>& Queue<T>::operator= (const Queue<T>& r) {
  clear();
  copy(r);
  return *this;
}
```

```
//////////////////////////////////////////////////////////////////////
// class template QueueB
//////////////////////////////////////////////////////////////////////

template <class T, int N>
class QueueB : public Queue<T> {
public:

  QueueB () {}
  QueueB (const Queue<T>&);
  virtual ~QueueB () {}

  Queue<T>& operator= (const Queue<T>&);

  virtual IteratorQueue<T>* createIterator() const;

  virtual void put    (const T& t){ rep.append(t); }
  virtual T    get    ()          { T t=rep.first(); rep.remove(0); return t; }
  virtual void clear  ()          { rep.clear(); }

  virtual const T& first    () const           { return rep.first(); }
  virtual int      isEmpty  () const           { return rep.isEmpty(); }
  virtual int      isFull   () const           { return rep.isFull(); }
  virtual int      length   () const           { return rep.length(); }
  virtual int      location (const T& t) const { return rep.location(t); }

private:
  friend class IteratorQueueB<T,N>;
  Bounded<T,N> rep;
};
```

```
////////////////////////////////////////////////////////////////////
// class template IteratorQueueB
////////////////////////////////////////////////////////////////////

template <class T, int N>
class IteratorQueueB : public IteratorQueue<T>,
                   private IteratorBounded<T,N> {
public:

  IteratorQueueB (const QueueB<T,N>* c)
                        : IteratorBounded<T,N>(&c->rep) {}
  virtual ~IteratorQueueB () {}

  virtual void reset() { IteratorBounded<T,N>::reset(); }
  virtual int  next () { return IteratorBounded<T,N>::next(); }

  virtual int  isDone() const { return IteratorBounded<T,N>::isDone(); }
  virtual const T* currentItem() const
                       { return IteratorBounded<T,N>::currentItem(); }

};



template<class T, int N>
QueueB<T,N>::QueueB (const Queue<T>& r) {
  copy(r);
}

template<class T, int N>
Queue<T>& QueueB<T,N>::operator= (const Queue<T>& r) {
  return Queue<T>::operator=(r);
}

template<class T, int N>
IteratorQueue<T>* QueueB<T,N>::createIterator() const {
  return new IteratorQueueB<T,N>(this);
}
```

```
//////////////////////////////////////////////////////////////////////
// class template QueueU
//////////////////////////////////////////////////////////////////////

template <class T>
class QueueU : public Queue<T> {
public:

  QueueU () {}
  QueueU (const Queue<T>&);
  virtual ~QueueU () {}

  Queue<T>& operator= (const Queue<T>&);

  virtual IteratorQueue<T>* createIterator() const;

  virtual void put    (const T& t){ rep.append(t); }
  virtual T    get    ()          { T t=rep.first(); rep.remove(0); return t; }
  virtual void clear  ()          { rep.clear(); }

  virtual const T& first    () const           { return rep.first(); }
  virtual int      isEmpty  () const           { return rep.isEmpty(); }
  virtual int      isFull   () const           { return rep.isFull(); }
  virtual int      length   () const           { return rep.length(); }
  virtual int      location (const T& t) const { return rep.location(t); }

private:
  friend class IteratorQueueU<T>;
  Unbounded<T> rep;
};
```

```
////////////////////////////////////////////////////////////////////
// class template IteratorQueueU
////////////////////////////////////////////////////////////////////

template <class T>
class IteratorQueueU : public IteratorQueue<T>,
                   private IteratorUnbounded<T> {
public:

  IteratorQueueU (const QueueU<T>* c)
                         : IteratorUnbounded<T>(&c->rep) {}
  virtual ~IteratorQueueU () {}

  virtual void reset() { IteratorUnbounded<T>::reset(); }
  virtual int  next () { return IteratorUnbounded<T>::next(); }

  virtual int  isDone() const { return IteratorUnbounded<T>::isDone(); }
  virtual const T* currentItem() const
                        { return IteratorUnbounded<T>::currentItem(); }

};



template<class T>
QueueU<T>::QueueU (const Queue<T>& r) {
  copy(r);
}

template<class T>
Queue<T>& QueueU<T>::operator= (const Queue<T>& r) {
  return Queue<T>::operator=(r);
}

template<class T>
IteratorQueue<T>* QueueU<T>::createIterator() const {
  return new IteratorQueueU<T>(this);
}
```

Relacije između klasa

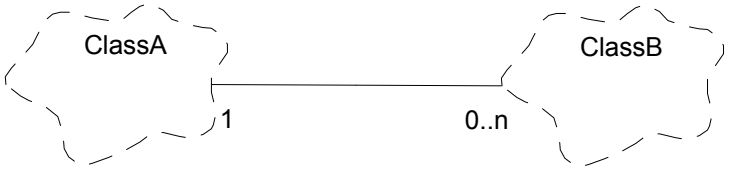
* Klasa nikada nije izolovana. Klasa dobija svoj smisao samo u okruženju drugih klasa sa kojima je u relaciji.
* Relacije između klasa su: asocijacija, zavisnost i nasleđivanje.

Asocijacija

* Asocijacija (pridruživanje, engl. *association*) je relacija između klasa èiji su objekti na neki naèin strukturno povezani. Ta veza postoji određeno duže vreme, a nije samo proceduralna zavisnost. Instanca asocijacije naziva se *vezom* (engl. *link*) i postoji između objekata datih klasa.
* Asocijacija se predstavlja punom linijom koja povezuje dve klase. Asocijacija može imati ime koje opisuje njeno znaèenje. Svaka strana u asocijaciji ima svoju *ulogu* (engl. *role*) koja se može naznaèiti na strani date klase.
* Na svakoj strani asocijacije može se specifikovati kardinalnost pomoæu sledeæih oznaka:

1 taèno 1
N proizvoljno mnogo
0..N 0 i više
1..N 1 i više
3..7 zadati opseg
i sliène.

* Primer:



* Druga posebna karakteristika svake strane asocijacije je *navigabilnost* (engl. *navigability*): sposobnost da se sa te strane (iz te klase) dospe do druge strane asocijacije. Prema ovom svojstvu, asocijacija može biti simetrièna ili asimetrièna.
* Primer: prikazana asocijacija se na strani klase A realizuje na sledeæi naèin, ukoliko postoji moguænost navigacije prema klasi B:

```
class B;

class A {
public:
  //...

  // Funkcije za uspostavljanje asocijacije:
  void insert (B* b) { classB.add(b); }
  void remove (B* b) { classB.remove(b); }

private:
  CollectionB<B*,MAXNB> classB;
};
```

* Na strani klase B, ukoliko postoji moguænost navigacije prema klasi A, ova asocijacija se može uspostaviti na više naèina: pomoæu konstruktora klase B, ili posebnom funkcijom za uspostavljanje veze:

```
class A;

class B {
public:
  B (A* a) { classA=a; }

  void link (A* a) { classA=a; }
  void unlink ()   { classA=0; }
  //...
private:
  A* classA;
};
```

* Posebna vrsta asocijacije je relacija *sadržavanja* (engl. *aggregation*, *has*): ako klasa A sadrži klasu B, to znaèi da je životni vek objekta klase B vezan za životni vek objekta klase A. Kardinalnost na strani klase A je uvek 1, što znaèi da je objekat klase A jedini potpuno odgovoran i nadležan za objekat klase B; kardinalnost na strani klase B može biti proizvoljna. Prema navigabilnosti, od klase A se uvek može pristupiti objektu klase B; obrnuto može, ali ne mora biti sluèaj. Oznaka:

* Implementacija sadržavanja može biti dvojaka: po vrednosti ili po referenci. Na strani sadržane klase, ovaj detalj oznaèava se sa:
- ☐ sadržavanje po referenci (podrazumeva se)
- ■ sadržavanje po vrednosti.

* Implementacija prethodno prikazane relacije na strani agregata realizuje se na sledeæi naèin:

```
class Supplier;

class Client {
public:
  Client ();
 ~Client ();
  //...
private:
  Supplier *supplier;
};


// Implementacija:
Client::Client () : supplier (new Supplier()) {
  //...
}

Client::~Client () { delete supplier; }
```

* Kod navedene realizacije, potrebno je obratiti pažnju na sledeæe. U deklaraciji interfejsa klase `Client` nije potrebna potpuna definicija klase `Supplier`, veæ samo prosta deklaracija `class Supplier;`, jer klasa `Client` sadrži objekat klase `Supplier` po referenci (pokazivaè). Samo u implementaciji konstruktora i destruktora potrebna je potpuna definicija klase `Supplier`. Kako se implementacije ovih funkcija nalaze u modulu `client.cpp`, samo ovaj modul zavisi od modula sa interfejsom klase `Supplier`, dok modul `client.h` ne zavisi. Na ovaj naèin se drastièno smanjuju meðuzavisnosti izmeðu modula i vreme prevoðenja.

Zavisnost

* Relacija zavisnosti (engl. *dependence*) ili *korišæenja* (engl. *uses*) postoji ako klasa A na neki naèin koristi usluge klase B. To može biti npr. odnos klijent-server (klasa A poziva funkcije klase B) ili odnos instancijalizacije (klasa A kreira objekte klase B).
* Za realizaciju ove relacije izmeðu klase A i B potrebno je da interfejs ili implementacija klase A "zna" za definiciju klase B. Tako je klasa A zavisna od klase B.
* Oznaka:

* Znaèenje relacije može da se navede kao ime (oznaka) relacije na dijagramu, npr. "calls" ili "instantiates".
* Ako klasa `Client` koristi usluge klase `Supplier` tako što poziva operacije objekata ove klase (odnos klijent-server), onda ona tipièno "vidi" ove objekte kao argumente svojih funkcija èlanica. U ovom sluèaju, kao i za sadržavanje, interfejsu klase `Client` nije potrebna definicija klase `Supplier`, veæ samo njenoj implementaciji:

```
class Supplier;

class Client {
public:
  //...
  void aFunction (Supplier*);
};


// Implementacija:
void Client::aFunction (Supplier* s) {
  //...
  s->doSomething();
}
```

* Ako je potrebno dobiti notaciju prenosa po vrednosti, a zadržati navedenu pogodnost slabije zavisnosti između modula, onda se argument prenosi preko reference na konstantu:

```
void Client::aFunction (const Supplier& s){
  s.doSomething();
}
```

* Ako klasa `Client` instancijalizuje klasu `Supplier`, onda je realizacija nalik na:

```
Supplier* Client::createSupplier (/*some_arguments*/) {
  return new Supplier(/*some_arguments*/);
}
```

Nasleđivanje

* Nasleđivanje (engl. *inheritance*) predstavlja relaciju generalizacije, odnosno specijalizacije, zavisno u kom smeru se posmatra. Oznaka:

* Realizacija:

```
class Derived : public Base //...
```

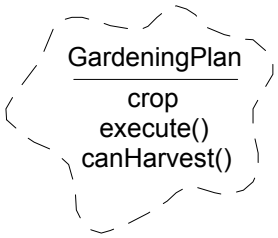
* Istom oznakom predstavlja se i privatno izvođenje, uz dve popreène crte na liniji relacije, uz izvedenu klasu.

Objektni model

* Model objektno orijentisane analize i projektovanja obezbeđuje višestruki pogled na sistem koji se razvija. Sastoji se iz:
- Logièkog modela: struktura klasa i struktura objekata;
- Fizièkog modela: arhitektura modula.

Dijagrami klasa

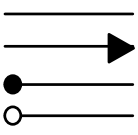
* Dijagram klasa (engl. *class diagram*) prikazuje klase u sistemu i njihove relacije.
* Klasa se prikazuje sledeæim simbolom:



* Obavezno je navesti ime klase. Opciono se navode atributi i operacije. Formati za navoðenje atributa i operacija su sledeæi:

A	samo ime atributa
: C	samo tip atributa
A : C	ime i tip atributa
A : C = E	ime, tip i poèetna vrednost atributa
N ()	samo ime operacije
R N (Arg)	ime, argumenti i povratni tip operacije

* Oznaka za apstraktnu klasu je slovo A unutar trougla-markera klase.
* Relacije izmeðu klasa:



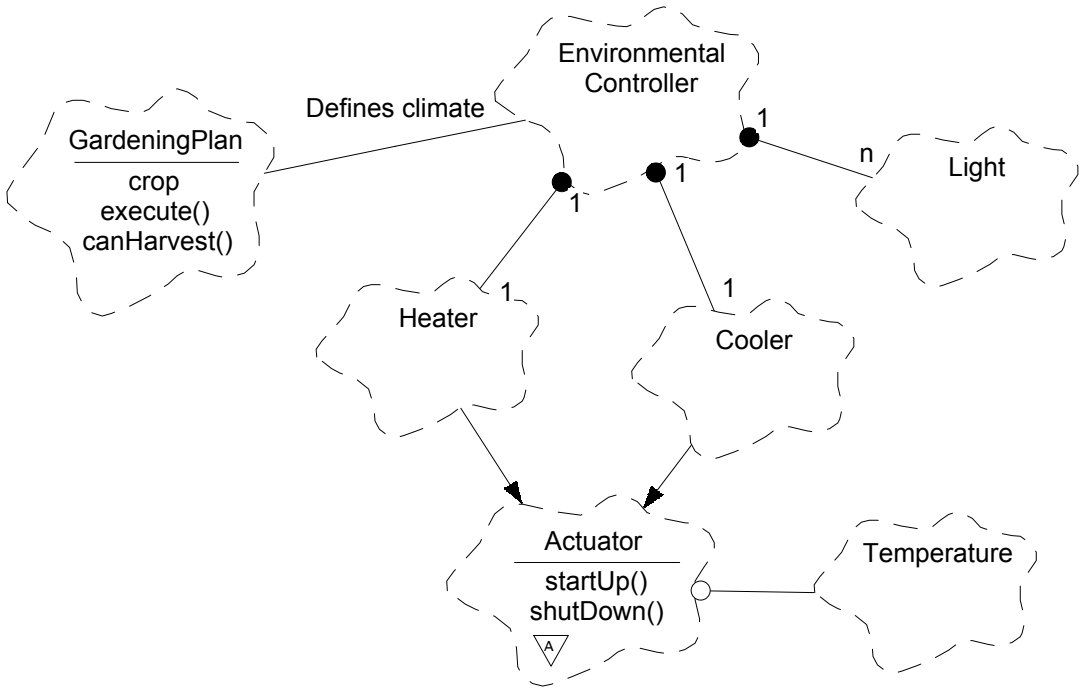
Pridruživanje (asocijacija, engl. *association*)
Nasleðivanje
Sadržavanje (engl. *has*)
Korišæenje (engl. *uses*)

* Relacija se može oznaèiti nazivom koji opisuje njenu semantiku. Sa svake strane relacije pridruživanja i na strani sadržane klase može da stoji oznaka kardinalnosti:

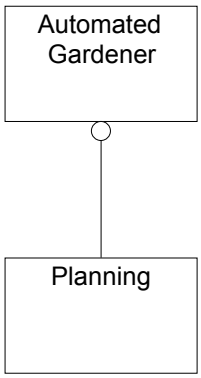
1	taèno 1
N	proizvoljno mnogo
0..N	0 i više
1..N	1 i više
3..7	zadati opseg

i slièno.

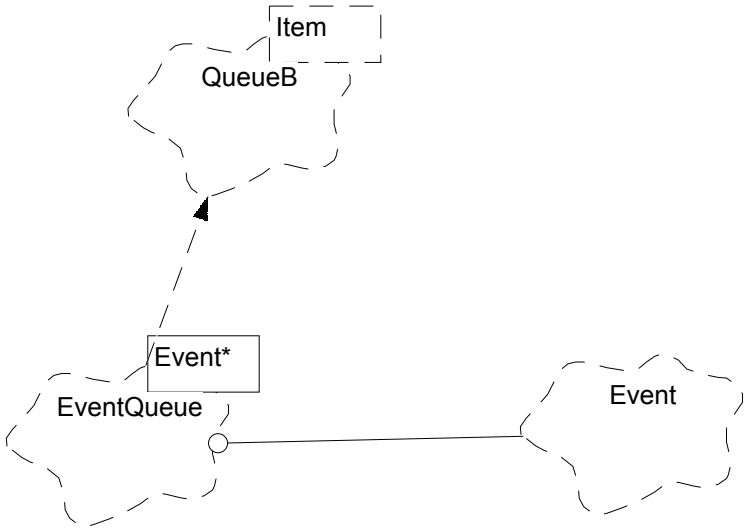
* Primer:



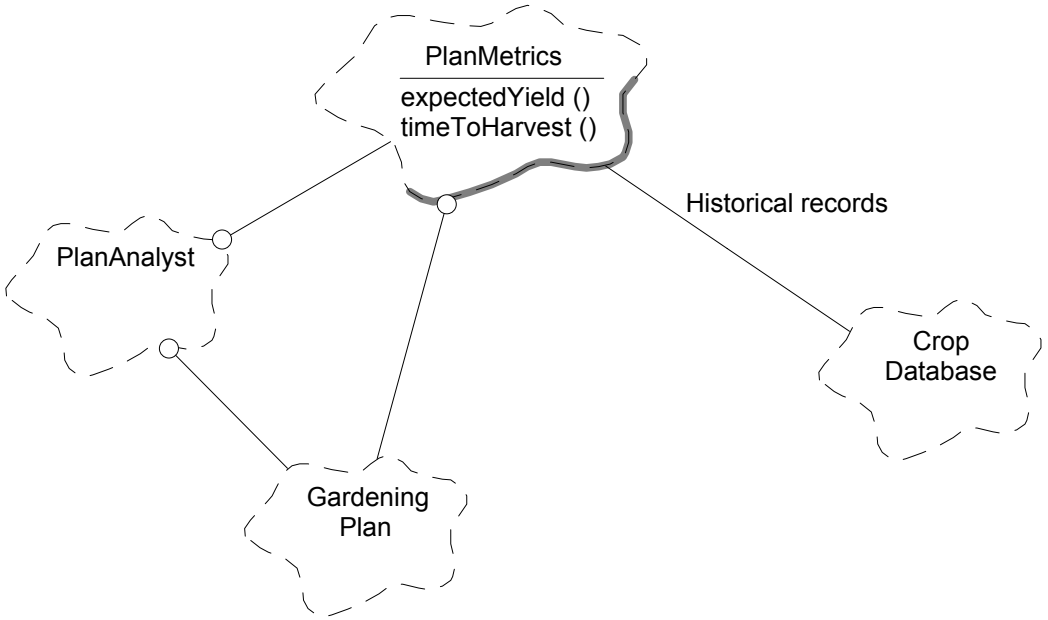
* Statièka struktura sistema predstavlja se skupom dijagrama klasa. Dijagrami klasa se organizuju u veæe semantièke celine - kategorije.
* Kategorije su jedinice organizovanja logièke, statièke strukture. Kategorije se organizuju hijerarhijski. Svaka kategorija sadrži proizvoljno mnogo dijagrama klasa. U dijagramima klasa mogu se nalaziti i potkategorije.
* Oznaka za kategoriju je pravougaonik sa imenom kategorije. Relacija između kategorija je relacija korišæenja (zavisnosti): kategorija A koristi kategoriju (zavisi od kategorije) B:



* Parametrizovane klase (šabloni) prikazuju se simbolom za klasu, uz pravougaoni, isprekidanom linijom iscrtani dodatak u kome se navode formalni argumenti. Generisane klase predstavljaju se simbolom za klasu, uz pravougaoni, punom linijom iscrtani dodatak sa stvarnim argumentima i relacijom instancijalizacije prema parametrizovanoj klasi:



* Skupovi srodnih globalnih funkcija neèlanica koje predstavljaju servise neke klase nazivaju se *uslugama klase* (engl. *class utilities*) i predstavljaju se simbolom za klasu sa senkom, uz obavezno ime skupa i spiskom funkcija:



* Relacija ili atribut se može oznaèiti prema pravu pristupa sledeæim oznakama:

bez oznake javno (*public*)
| zaštiæeno (*protected*)
|| privatno (*private*)

||| implementacija (klasa A koristi klau B u implementaciji neke svoje funkcije)

* Relacija sadržavanja se može oznaèiti na strani sadržane klase na sledeæi naèin:

□ sadržavanje po referenci

■ sadržavanje po vrednosti.

* Za svaki element dijagrama može se vezati komentar.

* Svakom elementu dijagrama ili samom dijagramu u modelu pridružena je specifikacija. Specifikacija sadrži definicije svih svojstava elementa kome je pridružena. Mnoga svojstva sadržana su u samom dijagramu, pa se generišu automatski. Ostala svojstva se mogu definisati u specifikaciji.

Dijagrami prelaza stanja

* Svakoj klasi èija se implementacija predstavlja konaènim automatom pridružuje se dijagram prelaza stanja (engl. *state transition diagram*).

* Konaèni automat se definiše pomoæu stanja i prelaza. Stanje definiše pasivan period između događaja koji pokreæu prelaze.

* Događaji mogu biti:

- simbolièko ime (konstanta) koja se dostavlja kao argument jedinstvene operacije klase koja odgovara na događaje;
- objekat neke klase ili
- ime operacije koja se poziva spolja da bi se automat (objekat) pobudio.

* Događaj uzrokuje prelaz iz stanja u stanje. Prelaz je definisan:

- događajem na koji se pokreæe;
- startnim i ciljnim stanjem i
- akcijom koja se preduzima.

* Akcija koja se preduzima kada se vrši prelaz može biti:

- upuæivanje događaja (konstante ili objekta) nekom drugom automatu ili
- poziv operacije nekog drugog objekta.

* Startno i ciljno stanje automata posebno se oznaèavaju crnim krugovima.

* Stanja se mogu ugnežđivati. Automat se u svakom trenutku nalazi u samo jednom, najdublje ugnežđenom stanju. Ako to stanje nema definisan prelaz za pristigli događaj, vrši se prelaz koji je definisan za nadređeno stanje itd. Ako prelaz ide do stanja koje ima ugnežđena stanja, vrši se ulazak u ugneđženo stanje koje je oznaèeno kao poèetno.

* Ako je neko stanje koje ima ugnežđena stanja oznaèeno sa H zaokruženo, onda se pri prelazu koje ide do njega ulazi u ono ugnežđeno stanje u kome se automat poslednji put nalazio (ulaženje po istoriji, engl. *history*).

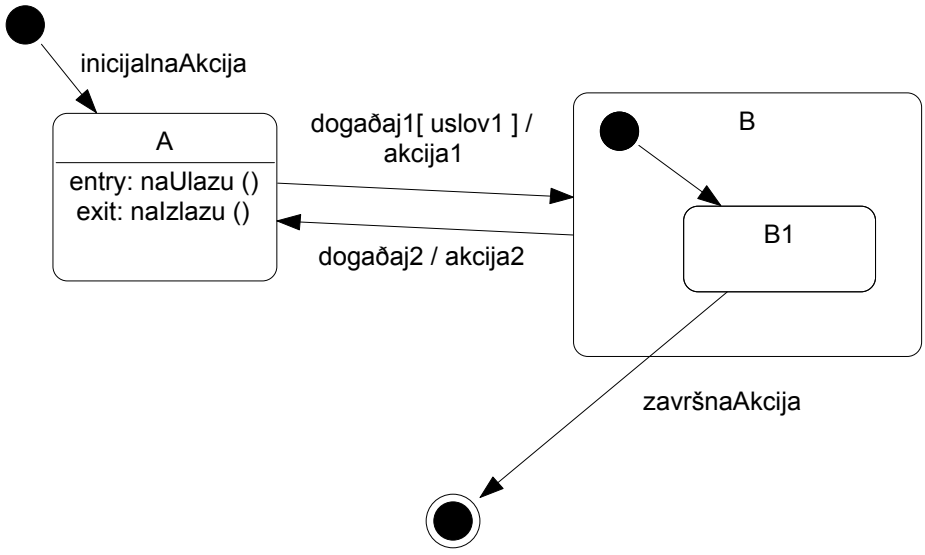
* Za svako stanje mogu se definisati akcije:

entry: akcija koja se vrši pri svakom ulasku u stanje;

exit: akcija koja se vrši pri svakom izlasku iz stanja.

* Prelaz može biti i uslovni. Uslov se zadaje između zagrada [ ]. Ako uslov nije zadovoljen kada je pristigao događaj, dati prelaz se ne vrši.

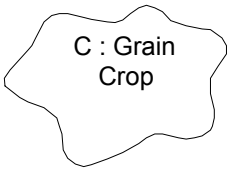
* Primer:



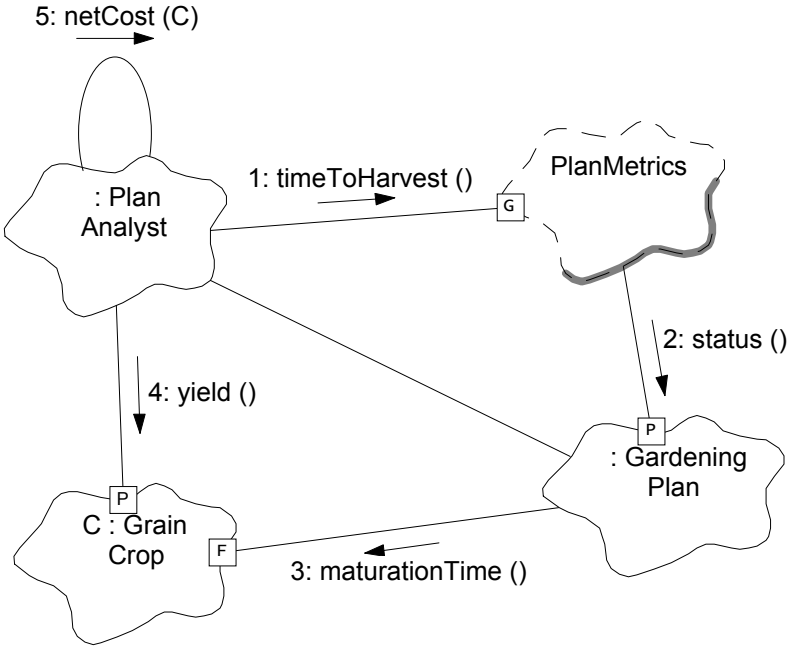
Dijagrami scenarija

* Dijagrami scenarija prikazuju logièku, dinamièku strukturu sistema. Dijagram scenarija pridružen je nekoj kategoriji.

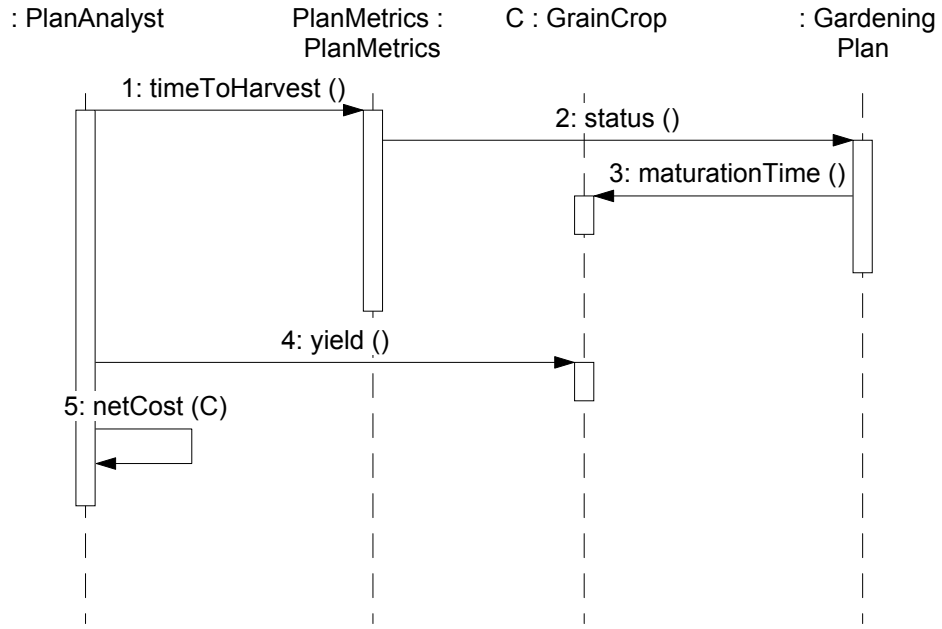
* Scenario prikazuje određene objekte i njihovu saradnju (kolaboraciju, razmenu poruka). Scenarijom se opisuje važan mehanizam ili deo mehanizma koji postoji u sistemu.
* Postoje dve vrste dijagrama scenarija. To su dva različita pogleda na istu stvar (isti scenarijo):
- dijagram objekata ili dijagram kolaboracija (engl. *object message diagram*, *collaboration diagram*) i
- dijagram poruka ili dijagram sekvence (engl. *message trace diagram*, *sequence diagram*).
* Na dijagramu objekata prikazani su objekti, njihove veze (engl. *links*) i njihova kolaboracija (razmena poruka). Objekti se prikazuju sledećom ikonom:



Oznaka objekta može da ima sledeći format:
A samo ime objekta
: C samo ime klase
A : C ime objekta i ime klase
* Veze se označavaju linijama između objekata. Vidljivost objekta klijenta označava se na njegovoj strani sledećim oznakama:
G globalan
F atribut (engl. *field*)
P parametar (argument operacije)
* Poruke (pozivi operacija) označene su celim brojevima pomoću kojih se definiše njihov redosled. Na primer:

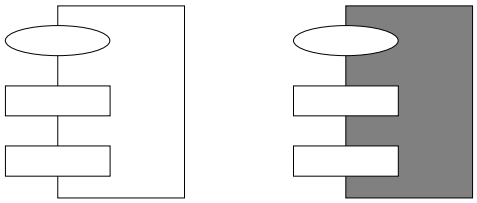


* Dijagram poruka prikazuje isti scenario ali na drugačiji način. Objekti su predstavljeni uspravnim linijama, a pozivi operacija horizontalnim linijama poređanim po redosledu. Na dijagramu poruka može se videti i kontekst poziva pomoću zadebljanih vertikalnih linija:



Dijagrami modula

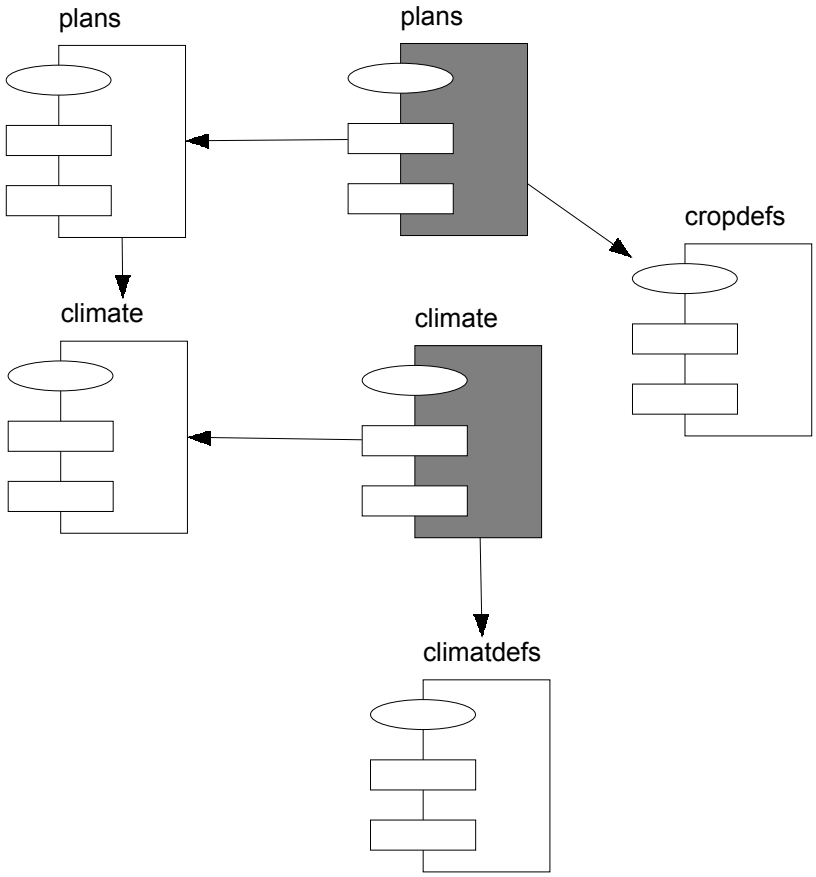
* Dijagrami modula prikazuju statièku, fizièku strukturu sistema. Na dijagramu modula prikazani su fizièki moduli na koje je izvorni program podeljen (datoteke u jeziku C++) i njihove zavisnosti.
* Modul je fizièka celina programa koja ima jasno definisan interfejs prema ostatku sistema i poseduje sakrivenu implementaciju. Jedan modul može sadržati samo jednu, ali i više povezanih klasa. Treba voditi raèuna da je u specifikaciji modula (datoteka .h) samo njegov interfejs, a nikako sve klase koje èine implementaciju tog modula.
* Simboli za module su sledeæi:



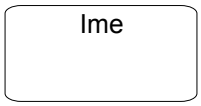
Specifikacija (.h) Telo (.cpp)

* Zavisnosti predstavljaju kompilacione zavisnosti - `#include` relacije u jeziku C++. Prilikom projektovanja fizièkog modela treba voditi raèuna da ove zavisnosti budu što slabije. To znaèi primenu sledeæih pravila:

- Eleminisati tranzitivne zavisnosti jer æe one svakako biti ispunjene.
- Ukoliko modul A zavisi od specifikacije modula B (b.h), treba videti da li to zavisi njegova specifikacija ili samo telo. Ako zavisi samo telo, a specifikaciji modula A nije potrebna specifikacija B, onda relaciju zavisnosti treba baš tako predstaviti. Ovo je sluèaj kada je specifikacija B potrebna samo implementaciji, ali ne i interfejsu modula A. Na ovaj naèin se drastièno smanjuju veze izmeðu modula, a time i vreme prevoðenja.



* Fizièki model se organizuje hijerarhijski. Moduli se organizuju u veæe celine - podsisteme (engl. *subsystem*). Svakom podsistemu može biti pridruženo nekoliko dijagrama modula. Svaki dijagram modula prikazuje module i ugnežðene podsisteme. Relacija izmeðu podsistema je takoðe relacija zavisnosti. Simbol za podsistem je:



Deo II
Programiranje u realnom vremenu

Jezgro višeprocesnog operativnog sistema

* Proces predstavlja deo programskog koda zajedno sa strukturama podataka koje omoguæuju uporedno (konkurentno, engl. *concurrent*) izvršavanje tog programskog koda sa ostalim procesima. Koncept procesa omoguæuje izvršavanje dela programskog koda tako da su svi podaci koji su deklarisani kao lokalni za taj deo programskog koda zapravo lokalni za jedno izvršavanje tog koda, i da se njihove instance razlikuju od instanci istih podataka istih delova tog koda, ali različitih procesa. Ova lokalnost podataka procesa pridruženih jednom izvršavanju datog koda opisuje se kao izvršavanje datog dela koda u *kontekstu* nekog procesa.
* U terminologiji konkurentnog programiranja razlikuju se dve vrste procesa:

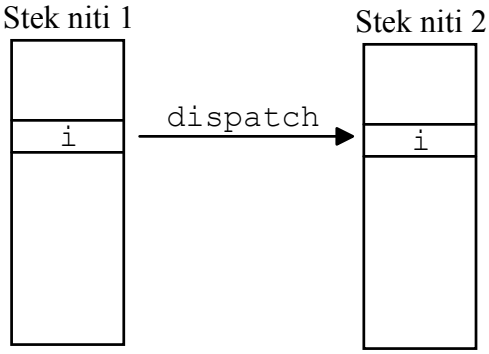
1. Proces na nivou operativnog sistema (engl. *process*). Ovakvi procesi nazivaju se ponekad "teškim" (engl. *heavy-weight*) procesima. Ovakav proces kreira se nad celim programom, ili ponekad nad delom programa. Pri tome svaki proces ima sopstvene (lokalne) instance svih vrsta podataka u programu: statičkih (globalnih), automatskih (lokalnih za potprograme) i dinamičkih.
2. Proces u okviru jednog programa. Ovakvi procesi nazivaju se "lakim" (engl. *light-weight*) ili *nitima* (engl. *thread*). Niti se kreiraju nad delovima jednog programa, najèešæe kao tok izvršavanja koji polazi od jednog potprograma. Svi dalji ugnežðeni pozivi ostalih potprograma izvršavaju se u kontekstu date niti. To znaèi da sve niti unutar jednog programa dele statièke (globalne) i dinamièke podatke. Ono što ih razlikuje je lokalnost automatskih podataka: *svaka nit poseduje svoj kontrolni stek* na kome se kreiraju automatski objekti (alokacioni blokovi potprograma). Kaže se zato da sve niti poseduju *zajednièki adresni prostor*, ali različite *tokove kontrole*.

* Termin *zadatak* (engl. *task*) se upotrebljava u različitim znaèenjima, u nekim operativnim sistemima kao "teški" proces, a u nekim jezicima kao "laki" proces. Zbog toga ovaj termin ovde neæe biti upotrebljavan.
* U ovom kursu biæe prikazana realizacija jednog jezgra višeprocesnog sistema sa nitima (engl. *multithreaded kernel*). Ovakav sistem se može iskoristiti za ugraðene (engl. *embedded*) sisteme za rad u realnom vremenu. Kod ovog sistema viši, aplikativni sloj treba da se poveže zajedno sa kodom jezgra da bi se dobio kompletan izvršni program koji ne zahteva nikakvu softversku podlogu. Prema tome, veza između višeg sloja softvera i jezgra je na nivou izvornog koda i zajedničkog povezivanja, a ne kao kod složenih operativnih sistema, gde se sistemski pozivi rešavaju u vreme izvršavanja, najèešæe preko softverskih prekida.
* Ovo jezgro biæe krajnje jednostavno, sa jednostavnom *round-robbin* raspodelom i bez moguænosti preuzimanja (engl. *preemption*). Ovi parametri se mogu jednostavno izmeniti, što se ostavlja èitaocu.
* Na nivou aplikativnog sloja softvera, želja je da se postigne sledeæa semantika: nit je aktivan objekat koji u sebi sadrži sopstveni tok kontrole (sopstveni stek poziva). Nit se može kreirati nad nekom globalnom funkcijom. Pri tome se svi ugnežðeni pozivi, zajedno sa svojim automatskim objektima, dešavaju u sopstvenom kontekstu te niti. Na primer, korisnièki program može da izgleda ovako:

```
#include "kernel.h"  // uključivanje deklaracija Jezgra
#include <iostream.h>

void threadBody () {
  for (int i=0; i<3; i++) {
    cout<<i<<"\n";
    dispatch();
  }
}


void userMain () {
  Thread* t1=new Thread(threadBody);
  Thread* t2=new Thread(threadBody);
  t1->start();
  t2->start();
  dispatch();
}
```

* Funkcija `threadBody()` predstavlja telo (programski kod) niti. Funkcija `dispatch()` predstavlja eksplicitni zahtev za preuzimanje (dodelu procesora drugoj niti), naravno bez blokiranja tekuæe niti. Ovo je potrebno zato što ne postoji implicitno preuzimanje, pa time ni vremenska podela procesora (engl. *time sharing*). Funkcija `userMain()` predstavlja poèetnu nit aplikativnog, korisnièkog dela programa. Funkcija `main()` nalazi se u nadležnosti Jezgra, pa korisniku nije dostupna. Jezgro inicijalno kreira jednu nit nad obaveznom funkcijom `userMain()`.
* U ovom primeru obe niti imaju isti kod, ali svaka poseduje svoj stek poziva, na kome se kreira automatski objekat `i`. Kada doðe do preuzimanja u funkciji `dispatch()`, Jezgro obezbeðuje pamæenje konteksta tekuæe niti i

povratak konteksta niti koja je izabrana za tekuæu, što znaèi da se dalje izvršavanje odvija na steku nove tekuæe niti. Ovo prikazuje sledeæa slika:



Preuzimanje

* Preuzimanje (engl. *preemption*) predstavlja dodelu procesora drugom procesu. Preuzimanje može da se dogodi u sledeæim sluèajevima:

1. Kada nit eksplicitno traži preuzimanje, tj. "dobrovoljno" se odriče procesora, pozivom funkcije `dispatch()`.
2. Kada se nit blokira na nekom sinhronizacionom elementu, npr. semaforu.
3. Kada Jezgro dobije kontrolu u nekom, bilo kom sistemskom pozivu. To može biti neblokirajuæa operacija nekog sinhronizacionog elementa (npr. *signal* semafora), ili operacija koja je potencijalno blokirajuæa (npr. *wait* semafora), nit se ne blokira jer nisu zadovoljeni uslovi za to, ali Jezgro ipak implicitno vrši preuzimanje.
4. Kada istekne vreme dodeljeno datom procesu, ako postoji mehanizam raspodele vremena (engl. *time sharing*). Ovo je poseban sluèaj implicitnog preuzimanja.

* Prva dva sluèaja predstavljaju *eksplicitno* preuzimanje, jer nit sasvim "svesno" predaje procesor drugome. Druga dva sluèaja predstavljaju *implicitno* preuzimanje, jer nit nije u stanju da "zna" kada æe izgubiti procesor. Sistem koji podržava ovo implicitno preuzimanje (sluèajevi 3 i 4) naziva se sistem sa preuzimanjem (engl. *preemptive scheduling*). Specijalno, ako postoji implicitno preuzimanje kao posledica isteka vremenskog kvanta dodeljenog procesu, naziva se sistem sa raspodelom vremena (engl. *time sharing*).
* Jezgro realizovano ovde podržava samo eksplicitno preuzimanje (nije *preemptive*), ali se implicitno preuzimanje može jednostavno dograditi, što se ostavlja èitaocu.
* Tipièno se operativni sistemi konstruišu tako da postoje dva režima rada, koja su obièno podržana i od strane procesora: sistemski i korisnièki. U sistemskom režimu dozvoljeno je izvršavanje raznih sistemskih operacija, kao što je pristup do nekih podruèja memorije koji su zaštiæeni od korisnièkih programa. Osim toga, kada postoji *time sharing*, potrebno je da se sistemski delovi programa izvršavaju neprekidivo, kako ne bi došlo do poremeæaja sistemskih delova podataka. U realizaciji ovog Jezgra, naznaèena su mesta prelaska u sistemski i korisnièki režim, èime je ostavljena moguænost za ugradnju *time sharing* režima. Prelaz na sistemski režim obavlja funkcija `lock()`, a na korisnièki funkcija `unlock()`. Sve kritiène sistemske sekcije uokvirene su u par poziva ovih funkcija. Njihova realizacija je zavisna od platforme i za sada je prazna:

```
void lock   () {}  // Switch to kernel mode
void unlcok () {}  // Switch to user mode
```

* Kada dolazi do preuzimanja, u najjednostavnijem sluèaju eksplicitnog pomoæu funkcije `dispatch()`, Jezgro treba da uradi sledeæe:

1. Saèuva kontekst niti koja je bila tekuæa (koja se izvršavala, engl. *running*).
2. Smesti nit koja je bila tekuæa u red niti koje su spremne (engl. *ready*).
3. Izabere nit koja æe sledeæa biti tekuæa iz reda niti koje su spremne.
4. Povrati kontekst novoizabrane niti i nastavi izvršavanje.

* Èuvanje konteksta niti znaèi sledeæe: vrednosti svih relevantnih registara procesora èuvaju se u nekoj strukturi podataka da bi se kasnije mogle povratiti. Ova struktura naziva se najèešæe PCB (engl. *process control block*). Povratak konteksta znaèi smeštanje saèuvanih vrednosti registara iz PCB u same registre procesora.
* U registre spada i pokazivaè steka (engl. *stack pointer*, SP), koji je najvažniji za kontekst izvršavanja niti. Kada se u SP povrati vrednost saèuvana u PCB, dalje izvršavanje koristiæe upravo stek na koji ukazuje taj SP, èime se postiže najvažnije svojstvo konkurentnosti niti: lokalnost automatskih podataka, odnosno sopstveni tok kontrole.
* Koncept sopstvenih stekova niti koje su kreirane nad potprogramima, uz eksplicitno preuzimanje, ali bez sinhronizacionih elemenata, najstariji je koncept konkurentnog programiranja i naziva se *korutinom* (engl. *coroutine*). U standardnoj biblioteci jezika C (pa time i C++) definisane su dve funkcije koje obezbeđuju koncept korutina. Ove

funkcije "sakrivaju" neposredno baratanje samim registrima procesora, pa se njihovim korišæenjem može dobiti potpuno prenosiv program.

* Deklaracije ovih funkcija nalaze se u `<setjmp.h>` i izgledaju ovako:

```
int  setjmp  (jmp_buf context);
void longjmp (jmp_buf context, int value);
```

* Tip `jmp_buf` deklarisan je u istom zaglavlju i predstavlja zapravo PCB. To je struktura koja èuva sve relevantne registre èije su vrednosti bitne za izvršavanje C programa prevedenog pomoæu datog prevodioca na datom procesoru.
* Funkcija `setjmp()` vrši smeštanje vrednosti registara u strukturu `jmp_buf`. Pri tom smeštanju ova funkcija vraæa rezultat 0. Funkcija `longjmp()` vrši povratak konteksta saèuvanog u `jmp_buf`, što znaèi da izvršavanje vraæa na poziciju steka koja je saèuvana pomoæu odgovarajuæeg `setjmp()`. Pri tome se izvršavanje nastavlja sa onog mesta gde je pozvana `setjmp()`, s tim da sada `setjmp()` vraæa onu vrednost koju je dostavljena pozivu `longjmp()` (to mora biti vrednost razlièita od 0).
* Prema tome, pri èuvanju konteksta, `setjmp()` vraæa 0. Kada se kontekst povrati iz `longjmp()`, dobija se efekat da odgovarajuæi `setjmp()` vraæa vrednost razlièitu od 0. Veoma je važno da se pazi na sledeæe: od trenutka èuvanja konteksta pomoæu `setjmp()`, do trenutka povratka pomoæu `longjmp()`, izvršavanje u kome je `setjmp()` **ne sme** da se vrati iz funkcije koja neposredno okružuje poziv `setjmp()`, jer bi se time stek narušio, pa povratak pomoæu `longjmp()` dovodi do kraha sistema.
* Tipièna upotreba ovih funkcija za potrebe realizacije korutina može da bude ovakva:

```
if (setjmp(running.context)==0) {

  // Saèuvan je kontekst.
  // Može da se preðe na neki drugi,
  // i da se njegov kontekst povrati sa:
  longjmp(running.context,1)

} else {
  // Ovde je povraæen kontekst onoga koji je saèuvan u setjmp()
}
```

* U realizaciji Jezgra ovi pozivi su "upakovani" u OO okvire. Nit je predstavljena klasom `Thread` koja poseduje atribut tipa `jmp_buf` (kontekst). Funkcija èlanica `resume()` vrši povratak konteksta jednostavnim pozivom `longjmp()`. Funkcija èlanica `setContext()` èuva kontekst pozivom `setjmp()`. Kako se iz ove funkcije ne sme vratiti pre povratka konteksta, ova funkcija je samo logièki okvir i **mora** biti prava *inline* funkcija, kako prevodilac ne bi generisao kod za poziv i povrtak iz ove funkcije `setContext()`:

```
// WARNING: This function MUST be truely inline!
inline int Thread::setContext () {
  return setjmp(myContext);
}


void Thread::resume () {
  longjmp(myContext,1);
}
```

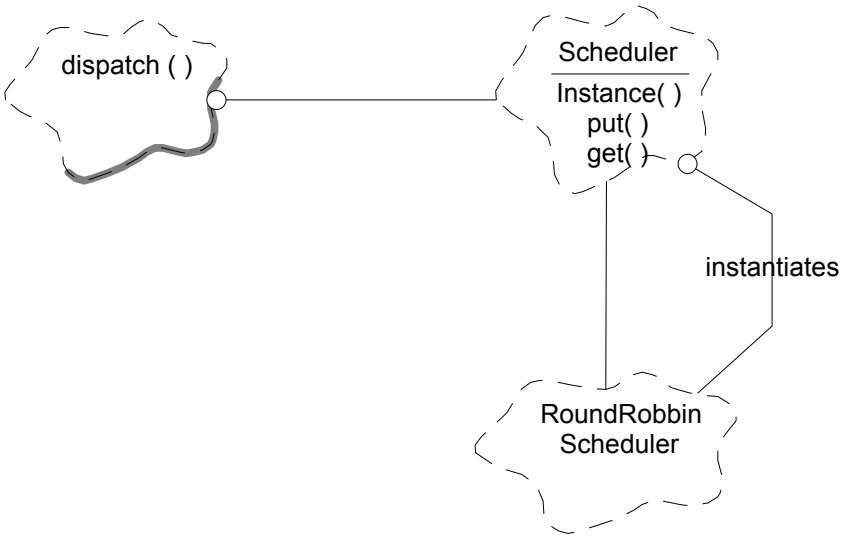
* Klasa `Scheduler` realizuje rasporeðivanje. U njoj se nalazi red spremnih niti (engl. *ready*), kao i protokol rasporeðivanja. Funkcija `get()` ove klase vraæa nit koja je na redu za izvršavanje, a funkcija `put()` stavlja novu spremnu nit u red.
* Klasa `Scheduler` poseduje samo jedan jedini objekat u sistemu (engl. *Singleton*). Ovaj jedini objekat sakriven je unutar klase kao statièki objekat. Otkrivena je samo statièka funkcija `Instance()` koja vraæa pokazivaè na ovaj objekat. Na ovaj naèin korisnici klase `Scheduler` ne mogu kreirati objekte ove klase, veæ je to u nadležnosti same te klase, èime se garantuje jedinstvenost objekta. Osim toga, korisnici ove klase ne moraju da znaju ime tog jedinog objekta, veæ im je dovoljan interfejs same klase i pristup do statièke funkcije `Instance()`. Ovakav projektni šablon (engl. *design pattern*) naziva se *Singleton*.
* Najzad, funkcija `dispatch()` izgleda jednostavno:

```
void dispatch () {
  lock ();
  if (running->setContext()==0) {

    // Context switch:
    Scheduler::Instance()->put(running);
    running=(Thread*)Scheduler::Instance()->get();
    running->resume();

  } else {
    unlock ();
    return;
  }
}
```

* Treba primetiti sledeæe: deo funkcije `dispatch()` iza poziva `setContext()`, a pre poziva `resume()`, radi i dalje na steku prethodno tekuæe niti (pozivi funkcija klase `Scheduler`). Tek od poziva `resume()` prelazi se na stek nove tekuæe niti. Ovo nije nikakav problem, jer taj deo predstavlja "ðubre" na steku iznad granice koja je zapamæena u `setContext()`. Prilikom povratka konteksta prethodne niti, izvršavanje æe se nastaviti od zapamæene granice steka, ispod ovog "ðubreta".

Rasporeðivanje

* Kao što je opisano, klasa `Scheduler` realizuje apstrakciju koja obavlja skladištenje spremnih niti, kao i rasporeðivanje. Pod rasporeðivanjem se smatra izbor one niti koja je na redu za izvršavanje. Ovo obavlja funkcija èlanica `get()`. Funkcija `put()` smešta novu nit u red spremnih.
* Klasa `Scheduler` je apstraktna klasa i realizuje samo interfejs (funkcije `put()` i `get()`) prema korisnicima (npr. funkciji `dispatch()`). Konkretne, izvedene klase realizuju sam protokol rasporeðivanja. Funkcije `put()` i `get()` su èiste virtuelne funkcije, a izvedene klase daju njihovu realizaciju. Kao što se vidi u kodu, u datoteci-zaglavlju je navedena samo deklaracija klase `Scheduler`, dok su izvedene klase kompletno sakrivene u .cpp datoteci. Na ovaj naèin se veze između komponenti (veza sa korisnicima) drastièno smanjuju i time softver èini fleksibilnijim.
* U ovoj realizaciji obezbeðen je samo jednostavan *round-robbin* rasporeðivaè realizovan odgovarajuæom klasom. Relacije između klasa predstavljene su sledeæim klasnim dijagramom:



* Klasa `Scheduler` je realizovana kao *Singleton*, što znaèi da ima samo jedan objekat. Taj objekat je primerak neke konkretne izvedene klase. Ovaj objekat je zapravo statièki lokalni objekat funkcije `Instance()`. Izbor konkretne vrste rasporeðivanja vrši se prostom definicijom odgovarajuæeg makroa:

```
Scheduler* Scheduler::Instance () {
  #ifdef _RoundRobinScheduler
    static RoundRobinScheduler instance;
  #endif
  return &instance;
}
```

* Sama klasa `RoundRobbinScheduler` je realizovana jednostavno, korišæenjem obièng reda.
* Kao što se vidi, implementacija drugog režima rasporeðivanja svodi se na sledeæe: 1) definisanje nove izvedene klase koja realizuje protokol; 2) definisanje novog makroa i 3) definisanje instance u funkciji `Instance()`. Treba primetiti da se ostali delovi programa uopšte ne menjaju. Ostavlja se èitaocu da realizuje neke druge režime rasporeðivanja.

Kreiranje niti

* Nit je predstavljena klasom `Thread`. Kao što je pokazano, korisnik kreira nit kreiranjem objekta ove klase. U tradicionalnom pristupu nit se kreira nad nekom globalnom funkcijom programa. Meðutim, ovaj pristup nije dovoljno fleksibilan. Naime, èesto je potpuno beskorisno kreirati više niti nad istom funkcijom ako one ne mogu da se meðusobno razlikuju, npr. pomoæu argumenata funkcije. Zbog toga se u ovakvim tradicionalnim sistemima èesto omoguæuje da korisnièka funkcija nad kojom se kreira nit dobije neki argument prilikom kreiranja niti. Ipak, broj i tipovi ovih argumenata su fiksni, definisanim samim sistemom, pa ovakav pristup nije u duhu jezika C++.
* U realizaciji ovog Jezgra, pored navedenog tradicionalnog pristupa, omoguæen je i OO pristup u kome se nit može definisati kao aktivan objekat. Taj objekat je objekat neke klase izvedene iz klase `Thread` koju definiše korisnik. Nit se kreira nad virtuelnom funkcijom `run()` klase `Thread` koju korisnik može da redefiniše u izvedenoj klasi. Na ovaj naèin svaki aktivni objekat iste klase poseduje sopstvene atribute, pa na taj naèin mogu da se razlikuju aktivni objekti iste klase (niti nad istom funkcijom). Suština je zapravo u tome da je jedini (doduše skriveni) argument funkcije `run()` nad kojom se kreira nit zapravo pokazivaè `this`, koji ukazuje na èitavu strukturu atributa objekta.
* Prema tome, interfejs klase `Thread` prema korisnicima izgleda ovako:

```
class Thread {
public:

  Thread ();
  Thread (void (*body)());
  void start ();

protected:

  virtual void run () {}

};
```

* Konstruktor bez argumenata kreira OO nit nad virtuelnom funkcijom `run()`. Drugi konstruktor kreira nit nad globalnom funkcijom na koju ukazuje pokazivaè-argument. Funkcija `run()` ima podrazumevano prazno telo, tako da se i ne mora redefinisati, pa klasa `Thread` nije apstraktna.
* Funkcija `start()` služi za eksplicitno pokretanje niti. Implicitno pokretanje moglo je da se obezbedi tako što se nit pokreæe odmah po kreiranju, što bi se realizovalo unutar konstruktora osnovne klase `Thread`. Meðutim, ovakav pristup nije dobar, jer se konstruktor osnovne klase izvršava pre konstruktora izvedene klase i njenih èlanova, pa se može dogoditi da novokreirana nit poène izvršavanje pre nego što je kompletan objekat izvedene klase kreiran. Kako nit izvršava redefinisanu funkciju `run()`, a unutar ove funkcije može da se pristupa atributima, moglo bi da doðe do konflikta.
* Treba primetiti da se konstruktor klase `Thread`, odnosno kreiranje nove niti, izvršava u kontekstu one niti koja poziva taj konstruktor, odnosno u kontekstu niti koja kreira novu nit.
* Prilikom kreiranja nove niti ključne i kritične su dve stvari: 1) kreirati novi stek za novu nit i 2) kreirati poèetni kontekst te niti, kako bi ona mogla da se pokrene kada doðe na red.
* Kreiranje novog steka vrši se prostom alokacijom niza bajtova u slobodnoj memoriji, unutar konstruktora klase `Thread`:

```
Thread::Thread ()
  : myStack(new char[StackSize]), //...
```

* Obezbeđenje poèetnog konteksta je mnogo teži problem. Najvažnije je obezbediti trenutak "cepanja" steka: poèetak izvršavanja nove niti na njenom novokreiranom steku. Ova radnja se može izvršiti direktnim smeštanjem vrednosti u SP. Pri tom je veoma važno sledeæe. Prvo, ta radnja se ne može obaviti unutar neke funkcije, jer se promenom vrednosti SP više iz te funkcije ne bi moglo vratiti. Zato je ova radnja u programu realizovana pomoæu makroa (jednostavne tekstualne zamene), da bi ipak obezbedila lokalnost i fleksibilnost. Drugo, kod procesora i8086 SP se sastoji iz dva registra (SS i SP), pa se ova radnja vrši pomoæu dve asemblerske instrukcije. Prilikom ove radnje vrednost koja se smešta u SP ne može biti automatski podatak, jer se on uzima sa steka èiji se položaj menja jer se menja i SP. Zato su ove vrednosti statièke. Ovaj deo programa je ujedno i jedini mašinski zavisan deo Jezgra i izgleda ovako:

```
#define splitStack(p)                \
  static unsigned int sss, ssp;      \  // FP_SEG() vraæa segmentni, a FP_OFF()
  sss=FP_SEG(p); ssp=FP_OFF(p);      \  // ofsetni deo pokazivaèa;
  asm {                              \  // neposredno ugrađivanje asemblerskih
    mov ss,sss;                      \  // instrukcija u kod;
    mov sp,ssp;                      \
    mov bp,sp;                       \  // ovo nije neophodno;
    add bp,8                         \  // ovo nije neophodno;
  }
```

* Poèetni kontekst nije lako obezbediti na mašinski nezavisan naèin. U ovoj realizaciji to je uraðeno na sledeæi naèin. Kada se kreira, nit se oznaèi kao "zapoèinjuæa" atributom `isBeginning`. Kada dobije procesor unutar funkcije `resume()`, nit najpre ispituje da li zapoèinje rad. Ako tek zapoèinje rad (što se dešava samo pri prvom dobijanju procesora), poziva se globalna funkcija `wrapper()` koja predstavlja "omotaè" korisnièke niti:

```
void Thread::resume () {
  if (isBeginning) {
    isBeginning=0;
    wrapper();
  } else
    longjmp(myContext,1);
}
```

* Prema tome, prvi poziv `resume()` i poziv `wrapper()` funkcija dešava se opet na steku prethodno tekuæe niti, što ostavlja malo "ðubre" na ovom steku, ali iznad granice zapamæene unutar `dispatch()`.
* Unutar funkcije `wrapper()` vrši se konaèno "cepanje" steka, odnosno prelazak na stek novokreirane niti:

```
void wrapper () {
    void* p=running->getStackPointer();   // vrati svoj SP
    splitStack(p);                        // cepanje steka

    unlock ();
    running->run();                       // korisnièka nit
    lock ();

    running->markOver();                  // nit je gotova,
    running=(Thread*)Scheduler::Instance()->get(); // predaje se procesor drugom
    running->resume();
}
```

* Takoðe je jako važno obratiti pažnju na to da ne sme da se izvrši povratak iz funkcije `wrapper()`, jer se unutar nje prešlo na novi stek, pa na steku ne postoji povratna adresa. Zbog toga se iz ove funkcije nikad i ne vraæa, veæ se po završetku korisnièke funkcije `run()` eksplicitno predaje procesor drugoj niti.
* Zbog ovakve logike, neophodno je da u sistemu uvek postoji bar jedna spremna nit. Uopšte, u sistemima se to najèešæe rešava kreiranjem jednog "praznog", bezposlenog (engl. *idle*) procesa, ili nekog procesa koji vodi raèuna o sistemskim resursima i koji se nikad ne može blokirati, pa je uvek u redu spremnih. U ovoj realizaciji to æe biti nit koja briše gotove niti i opisana je u narednom odeljku.

* Na ovaj naèin, startovanje niti predstavlja samo njeno upisivanje u listu spremnih, posle oznaèavanja kao "zapoèinjuæe":

```
void Thread::start () {
  //...
  fork(this);
}


void fork (Thread* aNew) {
  lock();
  Scheduler::Instance()->put(aNew);
  unlock();
}
```

Ukidanje niti

* Ukidanje niti je sledeæi veæi problem u konstrukciji Jezgra. Gledano sa strane korisnika, jedan moguæi pristup je da se omoguæi eksplicitno ukidanje kreiranog procesa pomoæu njegovog destruktora. Pri tome se poziv destruktora opet izvršava u kontekstu onoga ko uništava proces. Za to vreme sam proces može da bude završen ili još uvek aktivan. Zbog toga je potrebno obezbediti odgovarajuæu sinhronizaciju između ova dva procesa, što komplikuje realizaciju. Osim toga, ovakav pristup nosi i neke druge probleme, pa je on ovde odbaèen, iako je opštiji i fleksibilniji.
* U ovoj realizaciji opredeljenje je da niti budu zapravo aktivni objekti, koji se eksplicitno kreiraju, a implicino uništavaju. To znaèi da se nit kreira u kontekstu neke druge niti, a da zatim živi sve dok se ne završi funkcija `run()`. Tada se nit "sama" implicitno briše, taènije njeno brisanje obezbeðuje Jezgro.
* Brisanje same niti ne sme da se izvrši unutar funkcije `wrapper()`, po završetku funkcije `run()`, jer bi to znaèilo "seèenje grane na kojoj se sedi": brisanje niti znaèi i dealokaciju steka na kome se izvršava sama funkcija `wrapper()`.
* Zbog ovoga je primenjen sledeæi postupak: kada se nit završi, funkcija `wrapper()` samo oznaèi nit kao "završenu" atributom `isOver`. Poseban aktivni objekat (nit) klase `ThreadCollector` vrši brisanje niti koje su oznaèene kao završene. Ovaj objekat je nit kao i svaka druga, pa ona ne može doæi do procesora sve dok se ne završi funkcija `wrapper()`, jer završni deo ove funkcije izvršava u sistemskom režimu.
* Klasa `ThreadCollector` je takoðe *Singleton*. Kada se pokrene, svaka nit se "prijavi" u kolekciju ovog objekta, što je obezbeðeno unutar konstruktora klase `Thread`. Kada dobije procesor, ovaj aktivni objekat prolazi kroz svoju kolekciju i jednostavno briše sve niti koje su oznaèene kao završene. Prema tome, ova klasa je zadužena taèno za brisanje niti:

```
void Thread::start () {
  ThreadCollector::Instance()->put(this);
  fork(this);
}
```

```
class ThreadCollector : public Thread {
public:

  static ThreadCollector* Instance ();

  void put (Thread*);
  int  count () const;

protected:

  virtual void run ();

private:

  ThreadCollector ();
 ~ThreadCollector ();

  CollectionU<Thread*> rep;
  IteratorCollection<Thread*>* it;

  static ThreadCollector* instance;

};
```

```
void ThreadCollector::run () {
  while (1) {

    int i=0;

    for (i=0,it->reset(); !it->isDone(); it->next(),i++)
      if ((*it->currentItem())->isOver) {
      delete *it->currentItem();
      rep.remove(i);
      it->reset(); i=0;
      }

    dispatch();
  }
}
```

Pokretanje i gašenje programa

* Poslednji veæi problem pri konstrukciji Jezgra je obezbeđenje ispravnog pokretanja programa i povratka iz programa. Problem povratka ne postoji kod ugrađenih (engl. *embedded*) sistema jer oni rade neprekidno i ne oslanjaju se na operativni sistem. U okruženju operativnog sistema kao što je PC DOS, ovaj problem treba rešiti jer je želja da se ovo Jezgro koristi za eksperimentisanje na PC raèunaru.
* Program se pokreæe pozivom funkcije `main()` od strane operativnog sistema, na steku koji je odvojen od strane prevodioca i sistema. Ovaj stek nazivaæemo glavnim. Jezgro æe unutar funkcije `main()` kreirati nit klase `ThreadCollector` (ugrađeni proces) i nit nad korisnièkom funkcijom `userMain()`. Zatim æe zapamtiti kontekst glavnog programa, kako bi po završetku svih korisnièkih niti taj kontekst mogao da se povrati i program regularno završi:

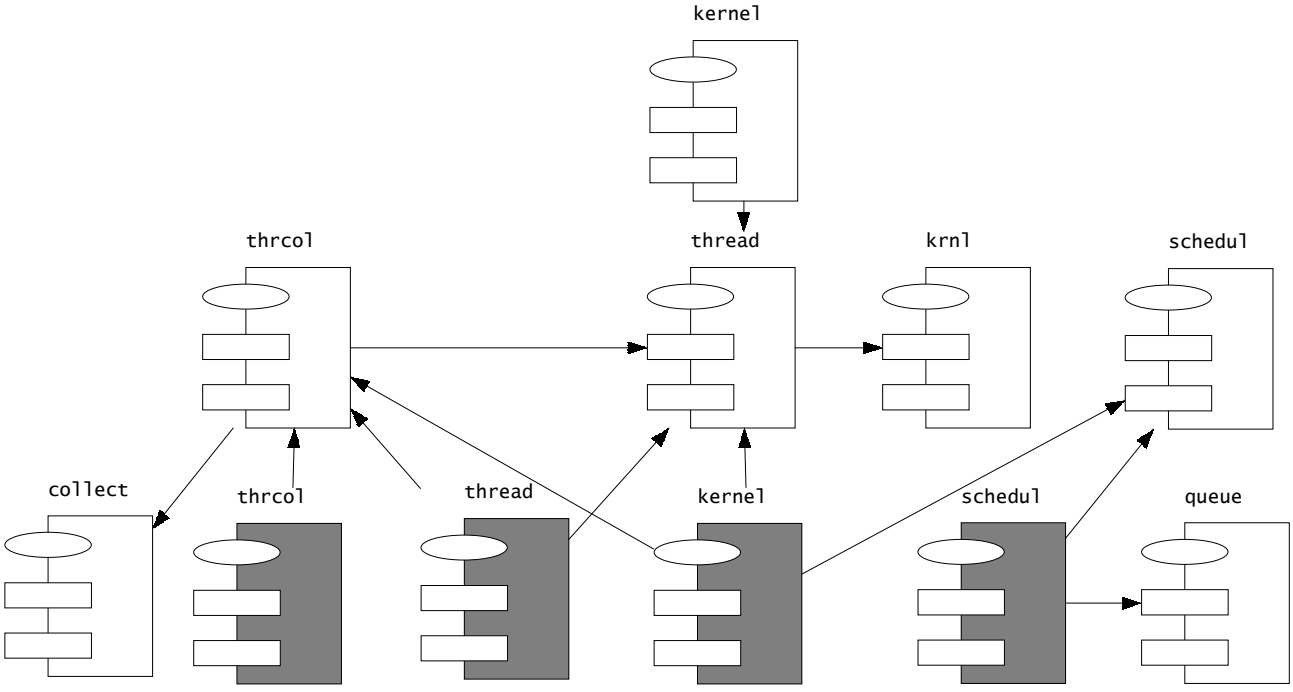
```
void main () {
  ThreadCollector::create();
  ThreadCollector::Instance()->start();

  running=new Thread(userMain);
  ThreadCollector::Instance()->put(running);

  if (setjmp(mainContext)==0) {
    unlock();
    running->resume();

  } else {
    ThreadCollector::destroy();
    return;
  }
}
```

* Treba još obezbediti "hvatanje" trenutka kada su sve korisnièke niti završene. To najbolje može da uradi sam `ThreadCollector`: onog trenutka kada on sadrži samo jednu jedinu evidentiranu nit u sistemu (to je on sam), sve ostale niti su završene. (On evidentira sve aktivne niti, a ne samo spremne.) Tada treba izvršiti povratak na glavni kontekst:

```
void ThreadCollector::run () {
  //...
    if (count()==1)
      longjmp(mainContext,1);  // return to main
  //...
}
```

Realizacija

* Dijagram modula Jezgra prikazan je na sledeæoj slici:



* Zaglavlje kernel.h služi samo da uključi sva zaglavlja koja predstavljaju interfejs prema korisniku. Tako korisnik može jednostavno da uključi samo ovo zaglavlje u svoj kod da bi dobio deklaracije Jezgra.
* Program je preveden na prevodiocu Borland C++ 3.1 za DOS.
* Prilikom prevoðenja u bilo kom prevodiocu treba obratiti pažnju na sledeæe opcije prevodioca:

1. Funkcije deklarisane kao *inline* moraju tako i da se prevode. U Borland C++ prevodiocu treba da bude isključena opcija Options\Compiler\C++ options\Out-of-line inline functions. Kritična je funkcija `Thread::setContext()`.
2. Program ne sme biti preveden kao *overlay* aplikacija. U Borland C++ prevodiocu treba izabrati opciju Options\Application\DOS Standard.
3. Memorijski model treba da bude takav da su svi pokazivači tipa *far*. U Borland C++ prevodiocu treba izabrati opciju Options\Compiler\Code generation\Compact ili Large ili Huge.
4. Mora da bude isključena opcija provere ograničenja steka. U Borland C++ prevodiocu treba da bude isključena opcija Options\Compiler\Entry/Exit code\Test stack overflow.

* Sledi kompletan izvorni kod opisanog Jezgra.
* Datoteka `kernel.h`:

```
// Project:  Real-Time Programming
// Subject:  Multithreaded Kernel
// Module:   Kernel
// File:     kernel.h
// Date:     23.11.1996.
// Author:   Dragan Milicev
// Contents: Kernel Interface

extern void dispatch ();

#include "thread.h"
#include "semaphor.h"
#include "msgque.h"
#include "timer.h"
```

* Datoteka `krnl.h`:

```
// Project:  Real-Time Programming
// Subject:  Multithreaded Kernel
// Module:   Kernel
// File:     krnl.h
// Date:     23.11.1996.
// Author:   Dragan Milicev
// Contents: Kernel module interface
//      Class utilities:  dispatch
//                        fork
//      Helper functions:
//                        lock
//                        unlock
//      Objects:          running
//                        mainContext

#ifndef _KRNL_
#define _KRNL_

#include <setjmp.h>

void dispatch ();

class Thread;
void fork (Thread*);

void lock ();    // Switch to kernel mode
void unlock ();  // Switch to user mode

extern Thread* running;
extern jmp_buf mainContext;

#endif
```

* Datoteka `thread.h`:

```
// Project:  Real-Time Programming
// Subject:  Multithreaded Kernel
// Module:   Thread
// File:     thread.h
// Date:     23.11.1996.
// Author:   Dragan Milicev
// Contents: Thread class declaration
//      Class: Thread


#ifndef _THREAD_
#define _THREAD_

#include <setjmp.h>
#include "krnl.h"

////////////////////////////////////////////////////////////////////////
// class Thread
////////////////////////////////////////////////////////////////////////

class Thread {
public:

  Thread ();
  Thread (void (*body)());
  void start ();

protected:

  friend void wrapper ();
  void markOver ();
  virtual void run ();

  friend class ThreadCollector;
  virtual ~Thread ();

  friend void dispatch ();
  friend void main ();
  friend class Semaphore;

  inline int setContext ();
  void resume ();
  char* getStackPointer () const;
```

```
private:

  void (*myBody)();
  char* myStack;

  jmp_buf myContext;

  int isBeginning;
  int isOver;

};


// WARNING: This function MUST be truely inline!
inline int Thread::setContext () {
  return setjmp(myContext);
}


#endif
```

* Datoteka `schedul.h`:

```
// Project:  Real-Time Programming
// Subject:  Multithreaded Kernel
// Module:   Scheduler
// File:     schedul.h
// Date:     16.11.1996.
// Author:   Dragan Milicev
// Contents:
//      Class: Scheduler

#ifndef _SCHEDUL_
#define _SCHEDUL_

////////////////////////////////////////////////////////////////////////
// class Scheduler
////////////////////////////////////////////////////////////////////////

class Thread;

class Scheduler {
public:

  static Scheduler* Instance ();

  virtual void          put (const Thread*) = 0;
  virtual const Thread* get () = 0;

protected:
           Scheduler () {}
  virtual ~Scheduler () {}
};


#endif
```

* Datoteka `thrcol.h`:

```
// Project:  Real-Time Programming
// Subject:  Multithreaded Kernel
// Module:   Thread Collector
// File:     thrcol.h
// Date:     17.11.1996.
// Author:   Dragan Milicev
// Contents: Thread Collector responsible for thread deletion
//      Class:  ThreadCollector

#ifndef _THRCOL_
#define _THRCOL_

#include "collect.h"
#include "thread.h"

////////////////////////////////////////////////////////////////////
// class ThreadCollector
////////////////////////////////////////////////////////////////////

class ThreadCollector : public Thread {
public:

  static ThreadCollector* Instance ();

  void put (Thread*);
  int  count () const;

protected:

  friend void main ();
  static void create ();
  static void destroy ();

  virtual void run ();

private:

  ThreadCollector ();
 ~ThreadCollector ();

  CollectionU<Thread*> rep;
  IteratorCollection<Thread*>* it;

  static ThreadCollector* instance;

};


#endif
```

* Datoteka `kernel.cpp`:

```
// Project:  Real-Time Programming
// Subject:  Multithreaded Kernel
// Module:   Kernel
// File:     kernel.cpp
// Date:     16.11.1996.
// Author:   Dragan Milicev
// Contents: Kernel dispatching and context switching functions
//      Class utilities:  dispatch
//                        fork
//      Helper functions: wrapper
//                        lock
//                        unlock
//      Object:           running
//      Function:         main


#include <dos.h>
#include "thread.h"
#include "schedul.h"
#include "thrcol.h"


////////////////////////////////////////////////////////////////////////
// Helper functions lock () and unlock ()
////////////////////////////////////////////////////////////////////////

void lock () {}    // Switch to Kernel mode
void unlock () {}  // Switch to User mode


////////////////////////////////////////////////////////////////////////
// Global declarations
////////////////////////////////////////////////////////////////////////

Thread* running = 0;
jmp_buf mainContext;
```

```
//////////////////////////////////////////////////////////////////////
// Utility dispatch ()
//////////////////////////////////////////////////////////////////////

void dispatch () {
  lock ();
  if (running->setContext()==0) {

    // Context switch:
    Scheduler::Instance()->put(running);
    running=(Thread*)Scheduler::Instance()->get();
    running->resume();       // context switch

  } else {
    unlock ();
    return;
  }
}


//////////////////////////////////////////////////////////////////////
// Utility fork()
//////////////////////////////////////////////////////////////////////

void fork (Thread* aNew) {
  lock();
  Scheduler::Instance()->put(aNew);
  unlock();
}
```

```
//////////////////////////////////////////////////////////////////////
// Warning: Hardware/OS Dependent!
//////////////////////////////////////////////////////////////////////

// Borland C++: Compact, Large, or Huge memory Model needed!
#if defined(__TINY__) || defined(__SMALL__) || defined(__MEDIUM__)
  #error Compact, Large, or Huge memory model needed
#endif

#define splitStack(p)            \
  static unsigned int sss, ssp;  \
  sss=FP_SEG(p); ssp=FP_OFF(p);  \
  asm {                          \
    mov ss,sss;                  \
    mov sp,ssp;                  \
    mov bp,sp;                   \
    add bp,8                     \
  }
```

```
//////////////////////////////////////////////////////////////////////
// Helper function: wrapper ()
//////////////////////////////////////////////////////////////////////

void wrapper () {
    void* p=running->getStackPointer();
    splitStack(p);

    unlock ();
    running->run();
    lock ();

    running->markOver();
    running=(Thread*)Scheduler::Instance()->get();
    running->resume();
}


//////////////////////////////////////////////////////////////////////
// Function: main ()
//////////////////////////////////////////////////////////////////////

extern void userMain ();  // User's main function

void main () {
  ThreadCollector::create();
  ThreadCollector::Instance()->start();

  running=new Thread(userMain);
  ThreadCollector::Instance()->put(running);

  if (setjmp(mainContext)==0) {
    unlock();
    running->resume();

  } else {
    ThreadCollector::destroy();
    return;
  }
}
```

* Datoteka `thread.cpp`:

```
// Project:  Real-Time Programming
// Subject:  Multithreaded Kernel
// Module:   Thread
// File:     thread.cpp
// Date:     17.11.1996.
// Author:   Dragan Milicev
// Contents: Thread class definition
//      Class: Thread


#include "thread.h"
#include "thrcol.h"

////////////////////////////////////////////////////////////////////////
// class Thread
////////////////////////////////////////////////////////////////////////

const int StackSize = 4096;

Thread::Thread ()
  : myBody(0), myStack(new char[StackSize]),
  isBeginning(1), isOver(0) {}


Thread::Thread (void (*body)())
  : myBody(body), myStack(new char[StackSize]),
  isBeginning(1), isOver(0) {}


void Thread::markOver () {
  isOver=1;
}
```

```
void Thread::run () {
  if (myBody!=0) myBody();
}


void Thread::start () {
  ThreadCollector::Instance()->put(this);
  fork(this);
}


Thread::~Thread () {
  delete [] myStack;
}


void Thread::resume () {
  if (isBeginning) {
    isBeginning=0;
    wrapper();
  } else
    longjmp(myContext,1);
}


char* Thread::getStackPointer () const {
  // WARNING: Hardware\OS dependent!
  // PC Stack grows downwards:
  return myStack+StackSize-10;
}
```

* Datoteka `schedul.cpp`:

```
// Project:  Real-Time Programming
// Subject:  Multithreaded Kernel
// Module:   Scheduler
// File:     schedul.cpp
// Date:     16.11.1996.
// Author:   Dragan Milicev
// Contents:
//      Classes: Scheduler
//               RoundRobinScheduler

#define _RoundRobinScheduler

#include "schedul.h"
#include "queue.h"


////////////////////////////////////////////////////////////////////////
// class RoundRobinScheduler
////////////////////////////////////////////////////////////////////////

class RoundRobinScheduler : public Scheduler {
public:

  virtual void          put (const Thread* t) { rep.put(t); }
  virtual const Thread* get ()                { return rep.get(); }

private:
  QueueU<const Thread*> rep;
};




////////////////////////////////////////////////////////////////////////
// class Scheduler
////////////////////////////////////////////////////////////////////////

Scheduler* Scheduler::Instance () {
  #ifdef _RoundRobinScheduler
    static RoundRobinScheduler instance;
  #endif
  return &instance;
}
```

* Datoteka `thrcol.cpp`:

```
// Project:  Real-Time Programming
// Subject:  Multithreaded Kernel
// Module:   Thread Collector
// File:     thrcol.cpp
// Date:     23.11.1996.
// Author:   Dragan Milicev
// Contents: Thread Collector responsible for thread deletion
//      Class:  ThreadCollector

#include "thrcol.h"

////////////////////////////////////////////////////////////////////
// class ThreadCollector
////////////////////////////////////////////////////////////////////

ThreadCollector* ThreadCollector::instance = 0;


ThreadCollector* ThreadCollector::Instance () {
  if (instance==0) create();
  return instance;
}


ThreadCollector::ThreadCollector ()
 : it(rep.createIterator()) {}


ThreadCollector::~ThreadCollector () {
  delete it;
}
```

```
void ThreadCollector::create () {
  instance=new ThreadCollector;
}


void ThreadCollector::destroy () {
  delete instance;
}


void ThreadCollector::put (Thread* t) {
  rep.add(t);
}


int  ThreadCollector::count () const {
  return rep.length();
}
```

```
void ThreadCollector::run () {
  while (1) {

    int i=0;

    for (i=0,it->reset(); !it->isDone(); it->next(),i++)
      if ((*it->currentItem())->isOver) {
      delete *it->currentItem();
      rep.remove(i);
      it->reset(); i=0;
      }

    if (count()==1)
      longjmp(mainContext,1);  // return to main

    dispatch();
  }
}
```

Elementi za sinhronizaciju i komunikaciju

* Kao elementi za sinhronizaciju procesa realizovani u prikazanom Jezgru izabrani su:
1. Semafor (engl. *semaphore*),
2. Događaj (engl. *event*) i
3. Prekid (engl. *interrupt*).
* Kao element za komunikaciju između procesa izabran je samo ograničeni bafer poruka (engl. *message buffer*). Poruka može biti pri tom bilo kog korisničkog tipa.
* Navedeni elementi se jednostavno realizuju kao viši sloj Jezgra, oslanjanjem na opisane funkcije Jezgra.

Semafor

* Realizovan je standardni Dijkstra semafor sa operacijama P() (*wait*) i V() (*signal*).
* Semafor je predstavljen odgovarajućom klasom `Semaphore`. Interno, semafor sadrži red blokiranih niti na semaforu i jednu celobrojnu promenljivu `val` koja ima sledeće značenje:
1) `val>0`: još `val` niti može da izvrši operaciju *wait* a da se ne blokira;
2) `val=0`: nema blokiranih na semaforu, ali će se nit koja naredna izvrši *wait* blokirati;
3) `val<0`: ima `-val` blokiranih niti, a *wait* izaziva blokiranje.
* Operacija `signalWait(s1,s2)` izvršava neprekidivu sekvencu operacija `s1.signal()` i `s2.wait()`. Ova operacija je pogodna za jednostavnu realizaciju reda poruka koji je kasnije opisan.
* Izvorni kod za interfejs klase `Semaphore` izgleda ovako (kompletna realizacija data je kasnije):

```
class Semaphore {
public:
  Semaphore (int initValue=1);
 ~Semaphore ();

  void wait   ();
  void signal ();

  friend void signalWait (Semaphore& s, Semaphore& w);

  int  value () const;

protected:

  void block ();
  void deblock ();

  int val;

private:

  Queue<Thread*>* blocked;

};
```

Događaj

* Događaj se ovde definiše kao jedna vrsta binarnog semafora: njegova vrednost ne može da bude veća od 1. Operacija *wait* blokira proces, ukoliko vrednost događaja nije 1, a postavlja vrednost događaja na 0, ako je njegova vrednost bila 1. Operacija *signal* deblokira proces koji je blokiran, ako ga ima, odnosno postavlja vrednost događaja na 1, ako blokiranog procesa nema.
* Na događaj po pravilu čeka samo jedan proces, pa je semantika događaja nedefinisana ako postoji više blokiranih procesa. Zato se u nekim sistemima događaj proglašava kao vlasništvo nekog procesa, i jedino taj proces može izvršiti operaciju *wait*, dok operaciju *signal* može vršiti svako.
* Operacija *signal* je po pravilu takva da se u njenom izvršavanju *ne gubi procesor* (nije *preemptive*). Ovo je bitno jer se događaj često upotrebljava za slanje elementarnog signala nekom procesu da se nešto dogodilo, gde pošiljalac može biti i prekidna rutina. Ovakva realizacija omogućuje brzu i kratku dojavu signala (događaja) procesu, bez promene konteksta.

* U mnogim sistemima postoje složene operacije èekanja na više događaja, po kriterijumu "i" i "ili". Ovi koncepti su izuzetno korisni u praksi. Ovde je realizovana samo najjednostavnija varijanta prostog èekanja.
* Izvorni kod za klasu `Event` izgleda ovako:

```
class Event : private Semaphore {
public:
  Event ();

  void wait   ();
  void signal ();

};
```

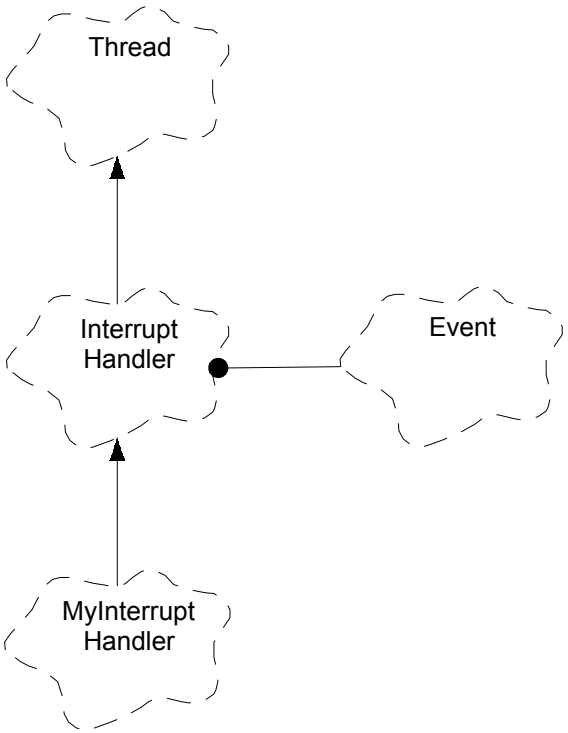
```
Event::Event () : Semaphore(0) {}

void Event::wait () {
  lock();
  if (--val<0)
    block();
  unlock();
}


void Event::signal () {
  lock();
  if (++val<=0)
    deblock();
  else
    val=1;
  unlock();
}
```

Prekid

* Prekidi predstavljaju važan elemenat svih programa u realnom vremenu. Međutim, u konkurentnom okruženju, prekidi donose sledeæe probleme.
* Posao koji se obavlja kao posledica prekida logièki nikako ne pripada niti koja je prekinuta, jer se u opštem sluèaju i ne zna koja je nit prekinuta: prekid je za softver signal nekog asinhronog spoljnjeg događaja. Zato posao koji se obavlja kao posledica prekida treba da ima sopstveni kontekst. Nikako ne valja da se dogodi da se u prekidnoj rutini, koja se izvršava u kontekstu niti koja je prekinuta, vrši neka operacija koja može da blokira pozivajuæu nit.
* Drugo, znaèajno je da se u prekidnoj rutini vodi raèuna kako dolazi do preuzimanja, ako je to potrebno.
* Treæe, u svakom sluèaju, prekidna rutina treba da završi svoje izvršavanje što je moguæe kraæe, kako ne bi zadržavala ostale prekide.
* Prema tome, jako je opasno u prekidnoj rutini pozivati bilo kakve operacije drugih objekata, jer one potencijalno nose opasnost od navedenih problema. Ovaj problem rešava se ako se na suštinu prekida posmatra na sledeæi naèin.
* Prekid zapravo predstavlja obaveštenje (signal) softveru da se neki događaj dogodio. Pri tome, signal o tom događaju ne nosi nikakve druge informacije, jer prekidne rutine nemaju argumente. Sve što softver može da sazna o događaju svodi se na softversko èitanje podataka (eventualno nekih registara hardvera). Prema tome, prekid je *signal događaja*.
* Navedeni problemi rešavaju se tako što se obezbedi jedan događaj koji æe prekidna rutina da signalizira, i jedan proces koji æe na taj događaj da èeka. Na ovaj naèin su konteksti potpuno razdvojeni, prekidna rutina je kratka jer samo obavlja signal događaja, a prekidni proces može da obavlja proizvoljne operacije posla koji se vrši kao posledica prekida.
* Ukoliko operativni sistem treba da odmah odgovori na prekid, onda operacija signaliziranja događaja iz prekidne rutine treba da bude sa preuzimanjem (engl. *preemptive*), pri èemu treba voditi raèuna kako se to preuzimanje vrši na konkretnoj platformi (maskiranje prekida, pamæenje konteksta u prekidnoj rutini i slièno).
* Treba primetiti da eventualno slanje poruke unutar prekidne rutine u neki bafer ne dolazi u obzir, jer je bafer tipièno složena struktura koja zahteva međusobno isključeènje, pa time i potencijalno blokiranje. Događaj, kako je

opisano, predstavlja pravi koncept za ovaj problem, jer je njegova operacija *signal* potpuno "bezazlena" (u svakom sluèaju neblokirajuæa).

* Sa druge strane, kod ovog rešenja postoji potencijalna opasnost da se dogodi više poziva prekidne rutine, pa time i operacije *signal*, pre nego što prekidni proces to stigne da obradi. Ovo nije problem opisanog rešenja, veæ samog softvera, jer u ovom sluèaju on nije u stanju da odgovori na spoljašnje pobude u realnom vremenu. Isti problem se može dogoditi kod svakog softverskog rešenja, ako softver ne obradi prekid, a pristigne više hardverskih prekida istovremeno. Kao što hardver nije u stanju da višestruko baferiše zahteve za prekid, ni softver ne mora to da radi.
* Ovde je opisano rešenje "upakovano" u OO koncepte, kako bi se obezbedilo jednostavno korišæenje i smanjila moguænost od greške. Klasa `InterruptHandler` sadrži opisani događaj i jednu nit. Korisnik iz ove klase treba da izvede sopstvenu klasu za svaku vrstu prekida koji se koristi. Korisnièka klasa treba da bude *Singleton*, a prekidna rutina definiše se kao statièka funkcija te klase (jer ne može imati argumente). Relacije između klasa prikazane su na sledeæem klasnom dijagramu:



* Korisnièka prekidna rutina treba samo da pozove funkciju jedinog objekta `InterruptHandler::interruptHandler()`. Ova funkcija æe izvršiti operaciju *signal* na događaju i time se prekidna rutina završava. Dalje, korisnik treba da redefiniše virtuelnu funkciju `handle()`. Ovu funkciju æe pozvati prekidni proces kada primi signal, pa u njoj korisnik može da navede proizvoljan kod.
* Osim navedene uloge, klasa `InterruptHandler` obezbeđuje i implicitnu inicijalizaciju interapt vektor tabele: konstruktor ove klase zahteva broj prekida i pokazivaè na prekidnu rutinu. Na ovaj naèin ne može da se dogodi da programer zaboravi inicijalizaciju, a ta inicijalizacija je lokalizovana, pa su zavisnosti od platforme svedene na minimum.
* Izvorni kod klase `InterruptHandler` izgleda ovako:

```
typedef unsigned int IntNo;  // Interrupt Number

class InterruptHandler : public Thread {
protected:

  InterruptHandler (IntNo num, void (*intHandler)());

  virtual void run ();

  virtual int handle () { return 0; }
  void interruptHandler ();

private:

  Event ev;

};
```

```
void initIVT (IntNo, void (*)() ) {
  // Init IVT entry by the given vector
}


InterruptHandler::InterruptHandler (IntNo num, void (*intHandler)()) {
  // Init IVT entry num by intHandler vector:
  initIVT(num,intHandler);

  // Start the thread:
  start();
}


void InterruptHandler::run () {
  for(;;) {
    ev.wait();
    if (handle()==0) return;
  }
}


void InterruptHandler::interruptHandler () {
  ev.signal();
}
```

* Primer upotrebe ove klase je sledeæi:

```
// Timer interrupt entry:
const int TimerIntNo = 0;


class TimerInterrupt : public InterruptHandler {
protected:

  TimerInterrupt () : InterruptHandler(TimerIntNo,timerInterrupt) {}

  static void timerInterrupt () { instance.interruptHandler(); }
  virtual int handle () { TimerController::Instance()->tick(); return 1; }

private:

  static TimerInterrupt instance;

};


TimerInterrupt TimerInterrupt::instance;
```

Međusobno isključenje

* Kritična sekcija je oblast programskog koda koja treba da bude zaštićena od konkurentnog pristupa dva ili više procesa. Ova zaštita naziva se *međusobno isključenje* (engl. *mutual exclusion*).
* Često korišćen koncept međusobnog isključenja su klase, odnosno objekti čije su sve operacije međusobno isključive. Ovakve klase i objekti nazivaju se *sinhronizovanim* (engl. *synchronized*), ili *monitorima* (engl. *monitor*).
* Međusobno isključenje neke operacije (funkcije članice) može da se obezbedi na jednostavan način pomoću semafora:

```
class Monitor {
public:
  Monitor () : sem(1) {}
  void criticalSection ();
private:
  Semaphore sem;
};


void Monitor::criticalSection () {
  sem.wait();
  //... telo kritične sekcije
  sem.signal();
}
```

* Međutim, opisano rešenje ne garantuje ispravan rad u svim slučajevima. Na primer, ako funkcija vraća rezultat nekog izraza iza naredbe `return`, ne može se tačno kontrolisati trenutak oslobađanja kritične sekcije, odnosno poziva operacije *signal*. Drugi, teži slučaj je izlaz i potprograma u slučaju izuzetka (engl. *exception*, vidi [Miliæev95]). Na primer:

```
int Monitor::criticalSection () {
  sem.wait();
  return f()+2/x;  //  gde pozvati signal()?
}
```

* Opisani problem se jednostavno rešava na sledeći način. Potrebno je unutar funkcije koja predstavlja kritičnu sekciju, na samom početku, definisati lokalni automatski objekat koji će u svom konstruktoru imati poziv operacije *wait*, a u destruktoru poziv operacije *signal*. Semantika jezika C++ obezbeđuje da se uvek destruktor ovog objekta pozove tačno na izlasku iz funkcije, pri svakom načinu izlaska (izraz iza `return` ili izuzetak).
* Jednostavna klasa `Mutex` obezbeđuje ovakvu semantiku:

```
class Mutex {
public:

  Mutex (Semaphore* s) : sem(s) { sem->wait(); }
 ~Mutex ()                      { sem->signal(); }

private:
  Semaphore *sem;
};
```

* Upotreba ove klase je takoðe veoma jednostavna: ime samog lokalnog objekta nije uopšte bitno, jer se on i ne koristi eksplicitno.

```
void Monitor::criticalSection () {
  Mutex dummy(&sem);
  //... telo kritiène sekcije
}
```

* Konaèno, kompletan izvorni kod za datoteke `semaphor.h` i `semaphor.cpp` izgleda ovako:

```
// Project:  Real-Time Programming
// Subject:  Multithreaded Kernel
// Module:   Semaphore
// File:     semaphor.h
// Date:     29.11.1996.
// Author:   Dragan Milicev
// Contents: Synchronization concepts: Semaphore, Event, Mutex,
//           and InterruptHandler
//      Classes: Semaphore
//               Event
//               Mutex
//               InterruptHandler

#ifndef _SEMAPHOR_
#define _SEMAPHOR_

#include "queue.h"
#include "thread.h"

////////////////////////////////////////////////////////////////////////
// class Semaphore
////////////////////////////////////////////////////////////////////////

class Semaphore {
public:
  Semaphore (int initValue=1);
 ~Semaphore ();

  void wait   ();
  void signal ();

  friend void signalWait (Semaphore& s, Semaphore& w);

  int  value () const;

protected:

  void block ();
  void deblock ();

  int val;

private:

  Queue<Thread*>* blocked;

};
```

```
//////////////////////////////////////////////////////////////////////
// class Event
//////////////////////////////////////////////////////////////////////

class Event : private Semaphore {
public:
  Event ();

  void wait   ();
  void signal ();

};




//////////////////////////////////////////////////////////////////////
// class Mutex
//////////////////////////////////////////////////////////////////////

class Mutex {
public:

  Mutex (Semaphore* s) : sem(s) { sem->wait(); }
 ~Mutex ()                      { sem->signal(); }

private:
  Semaphore *sem;
};
```

```
//////////////////////////////////////////////////////////////////////
// class InterruptHandler
//////////////////////////////////////////////////////////////////////

typedef unsigned int IntNo;  // Interrupt Number

class InterruptHandler : public Thread {
protected:

  InterruptHandler (IntNo num, void (*intHandler)());

  virtual void run ();

  virtual int  handle () { return 0; }
  void interruptHandler ();

private:

  Event ev;

};


#endif
```

```
// Project:  Real-Time Programming
// Subject:  Multithreaded Kernel
// Module:   Semaphore
// File:     semaphor.cpp
// Date:     29.11.1996.
// Author:   Dragan Milicev
// Contents: Synchronization concepts: Semaphore, Event, Mutex,
//           and InterruptHandler
//      Classes: Semaphore
//               Event
//               Mutex
//               InterruptHandler

#include "semaphor.h"
#include "schedul.h"

////////////////////////////////////////////////////////////////////
// class Semaphore
////////////////////////////////////////////////////////////////////

Semaphore::Semaphore (int init) : val(init),
                          blocked(new QueueU<Thread*>) {}


Semaphore::~Semaphore () {
  lock();
  for (IteratorQueue<Thread*>* it=blocked->createIterator();
       !it->isDone(); it->next())
       Scheduler::Instance()->put(*it->currentItem());
  delete it;
  delete blocked;
  unlock();
}


void Semaphore::block () {
  if (running->setContext()==0) {
    // Blocking:
    blocked->put(running);
    running=(Thread*)Scheduler::Instance()->get();
    running->resume();       // context switch
  } else return;
}


void Semaphore::deblock () {
  // Deblocking:
  Thread* t=blocked->get();
  Scheduler::Instance()->put(t);
}
```

```
void Semaphore::wait () {
  lock();
  if (--val<0)
    block();
  unlock();
}


void Semaphore::signal () {
  lock();
  if (val++<0)
    deblock();
  unlock();
}


void signalWait (Semaphore& s, Semaphore& w) {
  lock();
  if (s.val++<0) s.deblock();
  if (--w.val<0) w.block();
  unlock();
}


int Semaphore::value () const {
  return val;
}
```

```
//////////////////////////////////////////////////////////////////////
// class Event
//////////////////////////////////////////////////////////////////////

Event::Event () : Semaphore(0) {}

void Event::wait () {
  lock();
  if (--val<0)
    block();
  unlock();
}


void Event::signal () {
  lock();
  if (++val<=0)
    deblock();
  else
    val=1;
  unlock();
}
```

```
////////////////////////////////////////////////////////////////////
// class InterruptHandler
////////////////////////////////////////////////////////////////////

void initIVT (IntNo, void (*)() ) {
  // Init IVT entry by the given vector
}


InterruptHandler::InterruptHandler (IntNo num, void (*intHandler)()) {
  // Init IVT entry num by intHandler vector:
  initIVT(num,intHandler);

  // Start the thread:
  start();
}


void InterruptHandler::run () {
  for(;;) {
    ev.wait();
    if (handle()==0) return;
  }
}


void InterruptHandler::interruptHandler () {
  ev.signal();
}
```

Bafer poruka

* Koncept koji se èesto primenjuje za razmenu poruka između procesa (engl. *message passing*, ili *interprocess communication*, IPC), je *bafer poruka* (engl. *message buffer*). Bafer poruka je objekat koji poseduje sledeæe operacije:
1) *send*, slanje poruke u bafer; ako je bafer ogranièenog kapaciteta i ako je trenutno pun, poziv ove operacije blokira pozivajuæi proces sve dok se mesto za poruku ne oslobodi;
2) *receive*, prijem poruke iz bafera; ako je bafer prazan, poziv ove operacije blokira pozivajuæi proces sve dok se u baferu ne pojavi poruka.
* Ovde æe bafer poruka biti realizovan kao *ogranièeni bafer* (engl. *bounded buffer*), što znaèi da operacija *send* blokira pozivajuæi proces ako je bafer pun.
* Kako operacije *send* i *receive* zahtevaju po pravilu složenije manipulacije internim podacima bafera, ove operacije moraju da budu međusobno isključive, pa je ogranièeni bafer zapravo monitor.
* Operacija *send* treba da ima sledeæu semantiku: ako je bafer pun, pozivajuæi proces se blokira, ali pre toga oslobađa pristup baferu. Inaèe, poruka se smešta u bafer i deblokira se eventualno blokirani proces koji èeka na poruku, ako je bafer bio prazan.
* Operacija *receive* treba da ima sledeæu semantiku: ako je bafer prazan, pozivajuæi proces se blokira, ali pre toga oslobađa pristup baferu. Inaèe, poruka se uzima iz bafera i deblokira se eventualno blokirani proces koji èeka na prostor za poruku, ako je bafer bio pun.
* Za potrebe sinhronizacije u implementaciji bafera postoje tri semafora:
1) `mutex`, semafor koji obezbeđuje međusobno isključenje;
2) `notFull`, semafor koji služi za èekanje na prazan prostor i
3) `notEmpty`, semafor koji služi za èekanje na pojavu neke poruke.
* Opisana semantika može se obezbediti sledeæim kodom:

```
void MsgQueue<T,N>::send (const T& t) {
  Mutex dummy(mutex);
  if (rep.isFull()) signalWait(*mutex,notFull);
  mutex->wait();
  rep.put(t);
  if (notEmpty.value()<0) notEmpty.signal();
}


template <class T, int N>
T MsgQueue<T,N>::receive () {
  Mutex dummy(mutex);
  if (rep.isEmpty()) signalWait(*mutex,notEmpty);
  mutex->wait();
  T temp=rep.get();
  if (notFull.value()<0) notFull.signal();
  return temp;
}
```

* Treba primetiti sledeæe: operacija oslobaðanja kritiène sekcije (`signal(*mutex)`) i blokiranja na semaforu za èekanje na prazan prostor (`wait(notFull)`) moraju da budu neprekidive, inaèe bi moglo da se dogodi da između ove dve operacije neki proces uzme poruku iz bafera, a prvi proces se blokira na semaforu `notFull` bez razloga. Isto važi i u operaciji *receive*. Zbog toga je upotrebljena neprekidiva sekvenca `signalWait()`.
* Bafer poruka realizovan je kao šablon, parametrizovan tipom poruka i kapacitetom. Pomoæna operacija `receive()` koja vraæa `int` je neblokirajuæa: ako je bafer prazan, ona vraæa 0, inaèe smešta jednu poruku u argument i vraæa 1. Kompletan kod izgleda ovako:

```
// Project:  Real-Time Programming
// Subject:  Multithreaded Kernel
// Module:   MessageQueue
// File:     msgque.h
// Date:     23.11.1996.
// Author:   Dragan Milicev
// Contents: Communication concept: Message Queue
//      Template: MsgQueue

#ifndef _MSGQUE_
#define _MSGQUE_

#include "queue.h"
#include "semaphor.h"

////////////////////////////////////////////////////////////////////
// Template class MsgQueue
////////////////////////////////////////////////////////////////////

template <class T, int N>
class MsgQueue {
public:

  MsgQueue ();
 ~MsgQueue ();

  void send    (const T&);
  T    receive ();    // blocking
  int  receive (T&);  // nonblocking
  void clear   ();

  const T& first    () const;
  int      isEmpty  () const;
  int      isFull   () const;
  int      length   () const;

private:

  QueueB<T,N> rep;
  Semaphore *mutex, notEmpty, notFull;

};
```

```
template <class T, int N>
MsgQueue<T,N>::MsgQueue () : mutex(new Semaphore(1)),
                        notEmpty(0), notFull(0) {}

template <class T, int N>
MsgQueue<T,N>::~MsgQueue () {
  mutex->wait();
  delete mutex;
}
```

```
template <class T, int N>
void MsgQueue<T,N>::send (const T& t) {
  Mutex dummy(mutex);
  if (rep.isFull()) signalWait(*mutex,notFull);
  mutex->wait();
  rep.put(t);
  if (notEmpty.value()<0) notEmpty.signal();
}


template <class T, int N>
T MsgQueue<T,N>::receive () {
  Mutex dummy(mutex);
  if (rep.isEmpty()) signalWait(*mutex,notEmpty);
  mutex->wait();
  T temp=rep.get();
  if (notFull.value()<0) notFull.signal();
  return temp;
}


template <class T, int N>
int MsgQueue<T,N>::receive (T& t) {
  Mutex dummy(mutex);
  if (rep.isEmpty()) return 0;
  t=rep.get();
  if (notFull.value()<0) notFull.signal();
  return 1;
}
```

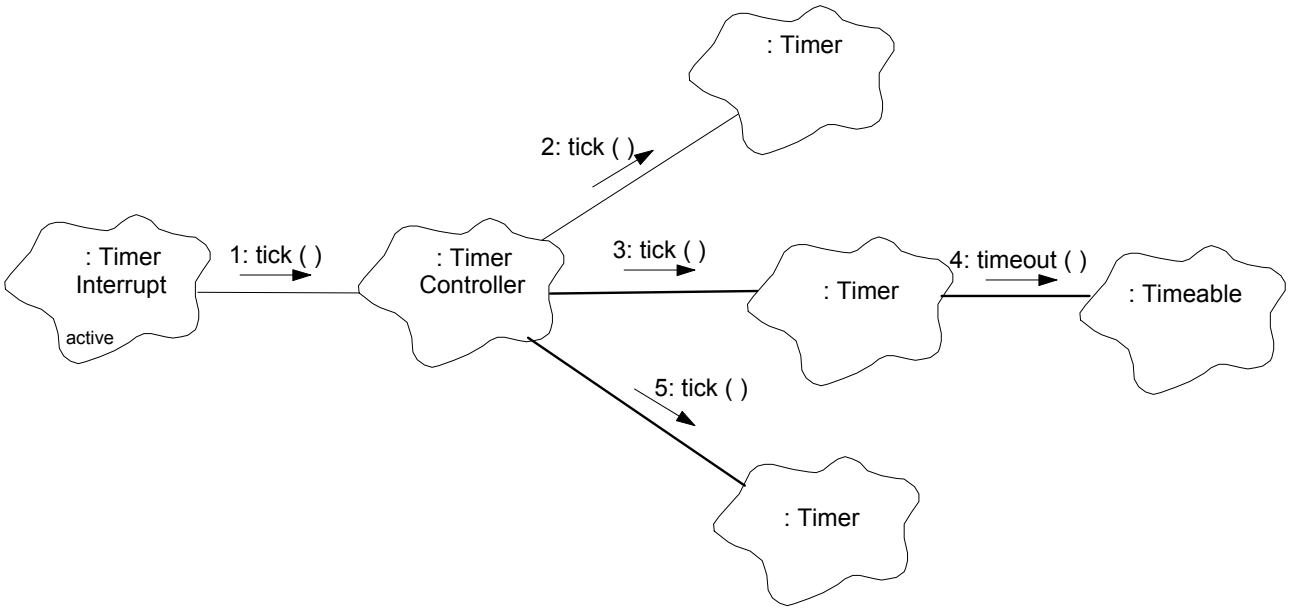
```
template <class T, int N>
void MsgQueue<T,N>::clear () {
  Mutex dummy(mutex);
  rep.clear();
}


template <class T, int N>
const T& MsgQueue<T,N>::first () const {
  Mutex dummy(mutex);
  return rep.first();
}

template <class T, int N>
int MsgQueue<T,N>::isEmpty () const {
  Mutex dummy(mutex);
  return rep.isEmpty();
}

template <class T, int N>
int MsgQueue<T,N>::isFull () const {
  Mutex dummy(mutex);
  return rep.isFull();
}

template <class T, int N>
int MsgQueue<T,N>::length () const {
  Mutex dummy(mutex);
  return rep.length();
}


#endif
```

Merenje vremena

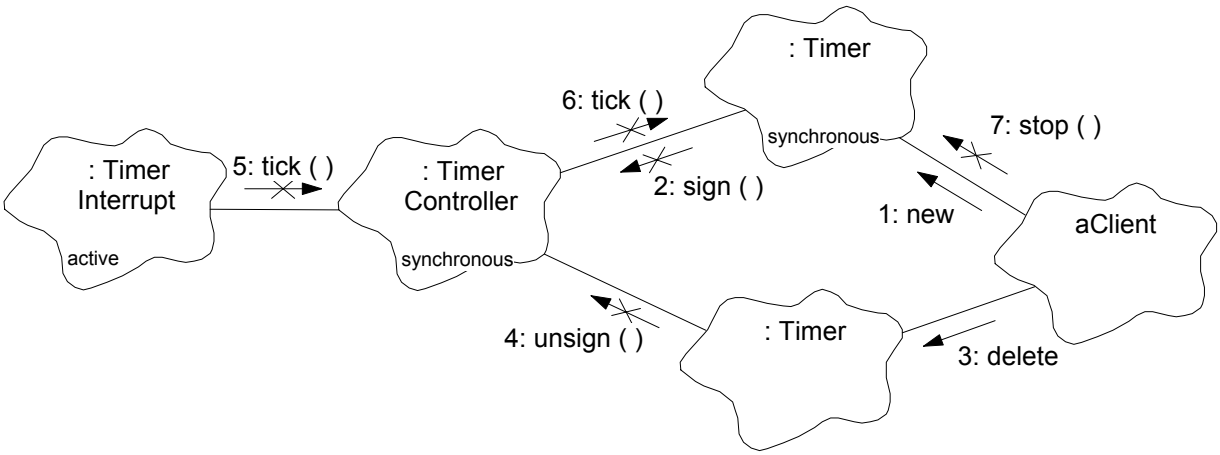
* Merenje vremena je u sistemima za rad u realnom vremenu jedna od ključnih i neizbežnih funkcija. Postoji potreba za dve funkcije merenja i kontrole vremena:
1. Merenje trajanja neke aktivnosti. Potrebno je na početku neke aktivnosti pokrenuti merenje vremena, a na kraju aktivnosti zaustaviti merenje i očitati izmereno vreme.
2. Kontrola trajanja aktivnosti (engl. *timeout*). Potrebno je po pokretanju neke aktivnosti ili stanja čekanja pokrenuti i vremensku kontrolu, tako da se po isteku vremenske kontrole signalizira ovaj istek, ukoliko aktivnost nije završena. Ukoliko je aktivnost završena pre isteka vremena, vremenska kontrola se zaustavlja. Tipično se ovakva kontrola vrši kada se čeka odgovor na neku akciju, poruku i slično.
* Opisana funkcionalnost može se obezbediti apstrakcijom `Timer`. Ova apstrakcija predstavlja vremenski brojač kome se zadaje početna vrednost i koji odbrojava po otkucajima sata realnog vremena. Brojač se može zaustaviti (operacija `stop()`) pri čemu vraća proteklo vreme.
* Ako je potrebno vršiti vremensku kontrolu, onda se korisnička klasa izvodi iz jedne jednostavne apstraktne klase `Timable` koja poseduje čistu virtuelnu funkcije `timeout()`. Ovu funkciju korisnik može da redefiniše, a poziva je `Timer` kada zadato vreme istekne. Ovakve jednostavne klase koje služe samo da obezbede interfejs prema datom delu sistema i poseduju jednostavno ponašanje bitno za taj deo sistema nazivaju se *mixin* klase.
* Opisani interfejsi izgledaju ovako:

```
class Timeable {
public:
  virtual void timeout () = 0;
};


class Timer {
public:

  Timer (Time, Timeable* =0);
 ~Timer ();

  Time stop    ();
  void restart (Time=0);

  Time elapsed () const;
  Time remained() const;

};
```

* Funkcija `restart()` ponovo pokreće brojač za novozadatim vremenom, ili sa prethodno zadatim vremenom, ako se novo vreme ne zada. Funkcije `elapsed()` i `remained()` vraćaju proteklo, odnosno preostalo vreme. Drugi argument kontstruktora predstavlja pokazivač na objekat kome treba poslati poruku `timeout()` kada zadato vreme istekne. Ako se ovaj pokazivač ne zada, brojač neće poslati ovu poruku pri isteku vremena.
* Mehanizam merenja vremena može se jednostavno realizovati na sledeći način. Hardver mora da obezbedi (što tipično postoji u svakom raèunaru) brojač (sat) realnog vremena koji periodično generiše prekid sa zadatim brojem. Ovaj prekid kontrolisaće aktivni objekat klase `TimerInterrupt`. Ovaj aktivni objekat, pri svakom otkucaju sata realnog vremena, odnosno po pozivu prekidne rutine, prosleđuje poruku `tick()` jednom centralizovanom *Singleton* objektu tipa `TimerController`, koji sadrži spisak svih kreiranih objekata tipa `Timer` u sistemu. Ovaj kontroler će proslediti poruku `tick()` svim brojačima.
* Svaki brojač tipa `Timer` se prilikom kreiranja prijavljuje u spisak kontrolera (operacija `sign()`), što se obezbeđuje unutar konstruktora klase `Timer`. Analogno, prilikom ukidanja, brojač se odjavljuje (operacija `unsign()`), što obezbeđuje destruktor klase `Timer`.
* Vremenski brojač poseduje atribut `isRunning` koji pokazuje da li je brojaæ pokrenut (odbrojava) ili ne. Kada primi poruku `tick()`, brojač će odbrojati samo ako je ovaj indikator jednak 1, inače jednostavno vraća kontrolu pozivaocu. Ako je prilikom odbrojavanja brojač stigao do 0, šalje se poruka `timeout()` objektu tipa `Timeable`.
* Opisani mehanizam prikazan je na sledećem dijagramu scenarija:

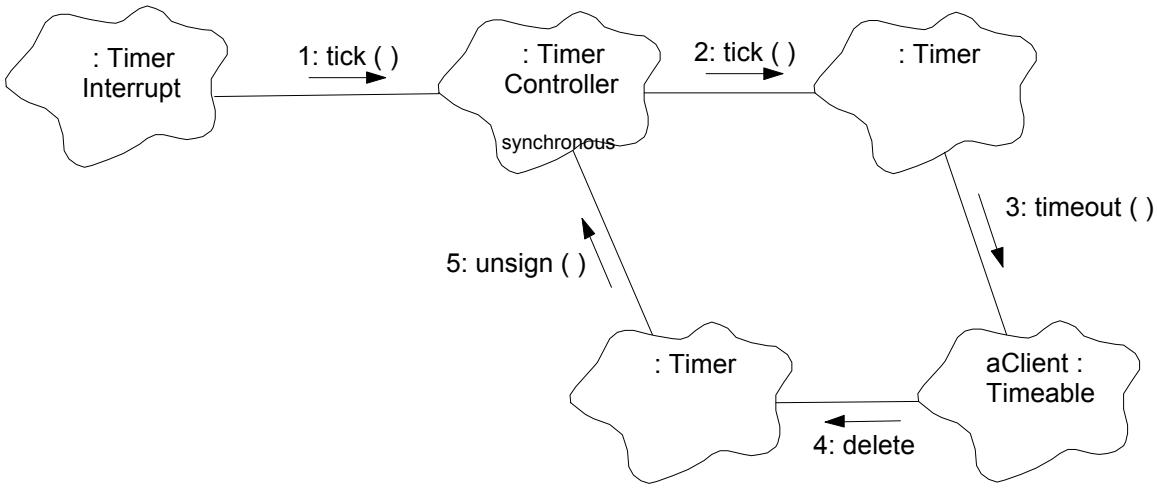


* Kako do objekta `TimerController` stižu konkurentne poruke sa dve strane, od objekta `InterruptHandler` poruka `tick()` i od objekata `Timer` poruke `sign()` i `unsign()`, ovaj objekat mora da bude sinhronizovan (monitor). Slièno važi i za objekte klase `Timer`. Ovo je prikazano na sledeæem dijagramu scenarija, pri èemu su blokirajuæi pozivi iskljuèivih operacija oznaèeni precrtanim strelicama:

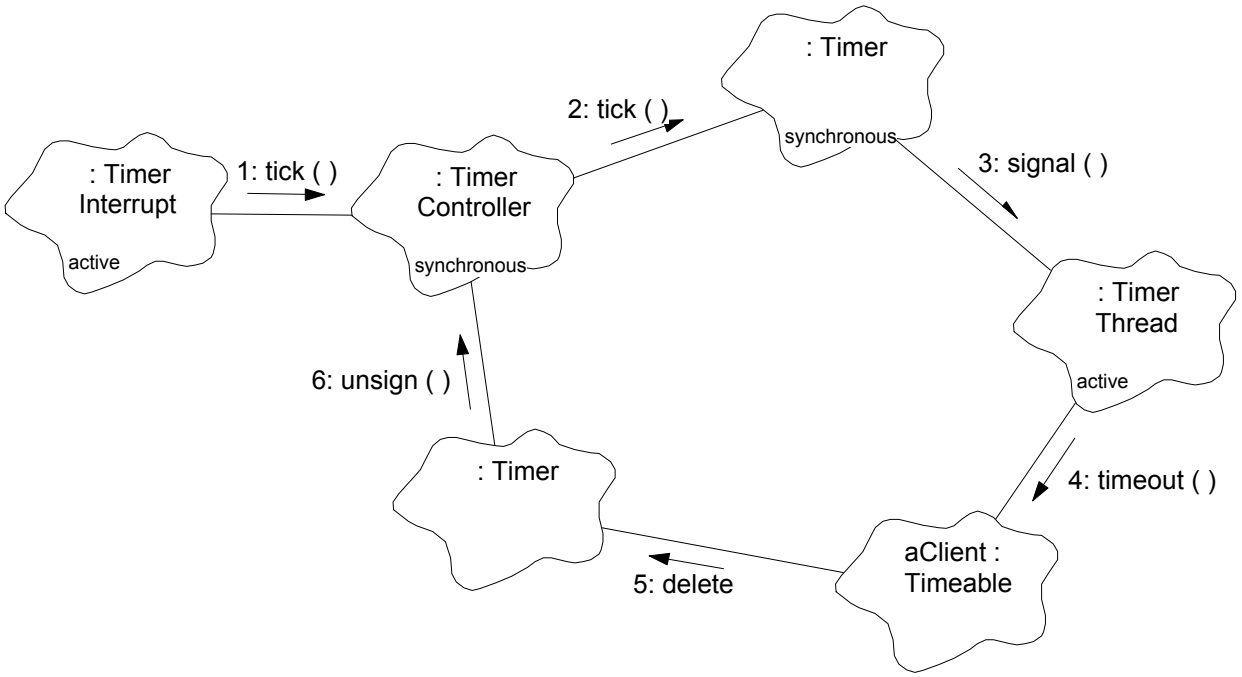


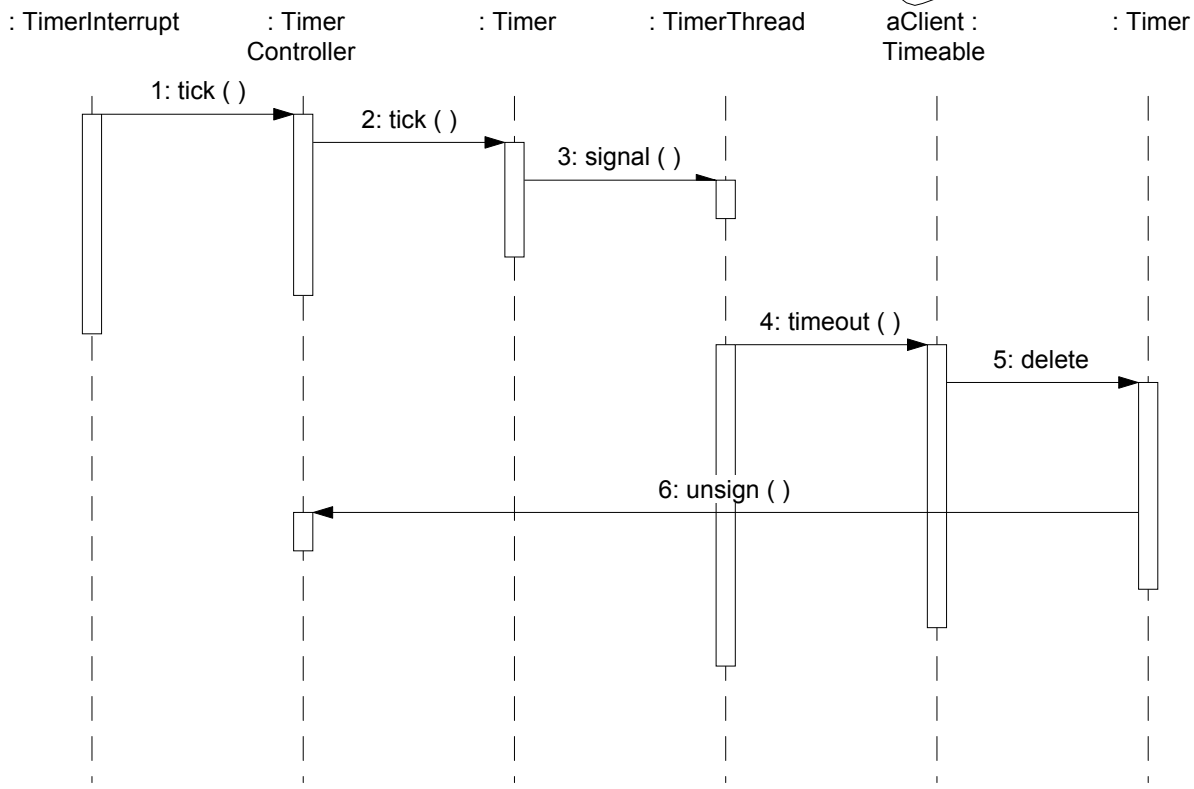
* Meðutim, ovakav mehanizam dovodi do sledeæeg problema: može se dogoditi da se unutar istog toka kontrole (niti) koji potièe od objekta `TimerInterrupt`, pozove `TimerController::tick()`, èime se ovaj objekat "zakljuèava" za nove pozive svojih operacija, zatim odatle pozove `Timer::tick()`, brojaè dobrojava do nule, poziva se `Timeable::timeot()`, a odatle neka korisnièka funkcija. Unutar ove korisnièke funkcije može se, u opštem sluèaju, kreirati ili brisati isti ili neki drugi `Timer`, sve unutar iste niti, èime se dolazi do poziva operacija objekta `TimerController`, koji je ostao zakljuèan. Na taj naèin dolazi do *kružnog blokiranja* (engl. *deadlock*) i to jedne niti same sa sobom.

* Èak i ako se ovaj problem zanemari, ostaje problem eventualno predugog zadržavanja unutar konteksta niti koja ažurira brojaèe, jer se ne može kontrolisati koliko traje izvršavanje korisnièke operacije `timeout()`. Time se neodreðeno zadržava mehanizam ažuriranja brojaèa, pa se gubi smisao samog merenja vremena. Opisani problemi prikazani su na sledeæem dijagramu scenarija:

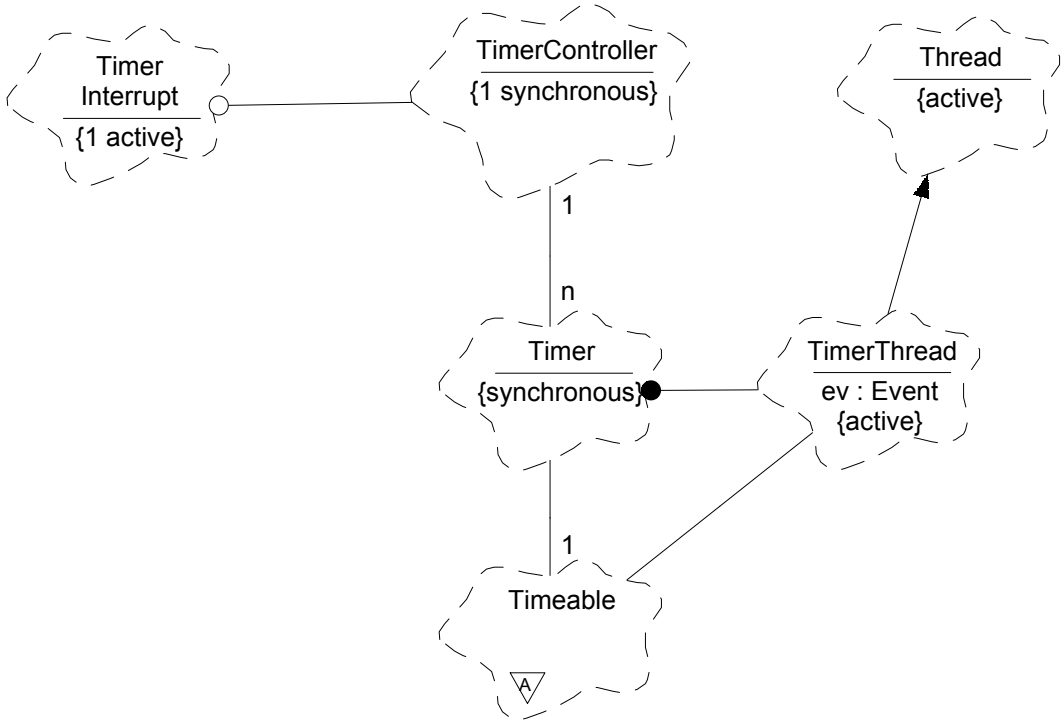


* Problem se rešava na isti naèin kao i kod obrade prekida: potrebno je na nekom mestu prekinuti kontrolu toka i razdvojiti kontekste uvoðenjem niti kojoj æe biti signaliziran dogaðaj. Ovde je to uèinjeno tako što objekat `Timer`, ukoliko poseduje pridružen objekat `Timeable`, poseduje i jedan aktivni objkekat `TimerThread` koji predstavlja nezavisan tok kontrole koji obavlja poziv operacije `timeout()`. Objekat `Timer` æe, kada vreme istekne, samo signalizirati dogaðaj pridružen objektu `TimerThread` i vratiti kontrolu objektu `TimerController`. `TimerThread` æe, kada primi signal, obaviti poziv operacije `timeout()`. Na ovaj naèin se navedeni problemi eliminišu, jer se sada korisnièka funkcija izvršava u kontekstu sopstvene niti. Mehanizam je prikazan na sledeæim dijagramima:





* Dijagram opisanih klasa izgleda ovako:


Timer
Interrupt
{1 active}
TimerController
{1 synchronous}
Thread
{active}
1
n
Timer
{synchronous}
TimerThread
ev : Event
{active}
1
Timeable
A

* U nastavku je dat kompletan izvorni kod za ovaj podsistem. Datoteka `timer.h`:

```
// Project:  Real-Time Programming
// Subject:  Multithreaded Kernel
// Module:   Timer
// File:     timer.h
// Date:     23.11.1996.
// Author:   Dragan Milicev
// Contents: Timers
//      Classes: Time (typedef)
//               Timer
//               Timeable (mixin)

#ifndef _TIMER_
#define _TIMER_

typedef unsigned int Time;

////////////////////////////////////////////////////////////////////////
// class Timeable
////////////////////////////////////////////////////////////////////////

class Timeable {
public:
  virtual void timeout () = 0;
};
```

```
//////////////////////////////////////////////////////////////////////
// class Timer
//////////////////////////////////////////////////////////////////////

class TimerThread;
class Semaphore;

class Timer {
public:

  Timer (Time, Timeable* =0);
 ~Timer ();

  Time stop    ();
  void restart (Time=0);

  Time elapsed () const;
  Time remained() const;

protected:

  friend class TimerController;
  void tick ();

private:

  Timeable* myTimeable;
  TimerThread* myThread;

  Time counter;
  Time initial;

  int isRunning;

  Semaphore* mutex;

};


#endif
```

* Datoteka `timer.cpp`:

```
// Project:  Real-Time Programming
// Subject:  Multithreaded Kernel
// Module:   Timer
// File:     timer.cpp
// Date:     23.11.1996.
// Author:   Dragan Milicev
// Contents: Timers
//      Classes:
//               Timer
//               TimerThread
//               TimerController
//               TimerInterrupt

#include "timer.h"
#include "semaphor.h"
#include "collect.h"

////////////////////////////////////////////////////////////////////////
// class TimerThread
////////////////////////////////////////////////////////////////////////

class TimerThread : public Thread {
public:

  TimerThread  (Timeable*);

  void signal  ();
  void destroy ();

protected:

  virtual void run ();

private:

  Event ev;
  Timeable* myTimeable;
  int isOver;
  Semaphore mutex;

};
```

```
TimerThread::TimerThread (Timeable* t) : myTimeable(t), isOver(0),
                                mutex(1) {}

void TimerThread::signal () {
  ev.signal();
}


void TimerThread::destroy () {
  isOver=1;
  ev.signal();
}


void TimerThread::run () {
  while (1) {
    ev.wait();
    if (isOver)
      return;
    else
      myTimeable->timeout();
  }
}
```

```
////////////////////////////////////////////////////////////////////////
// class TimerController
////////////////////////////////////////////////////////////////////////

class TimerController {
public:

  static TimerController* Instance();
 ~TimerController ();

  void tick ();

  void sign   (Timer*);
  void unsign (Timer*);

private:

  TimerController ();
  static TimerController instance;

  CollectionU<Timer*> rep;
  IteratorCollection<Timer*>* it;
  Semaphore mutex;

};
```

```
TimerController TimerController::instance;

TimerController* TimerController::Instance () {
  return &instance;
}


TimerController::TimerController () : mutex(1) {
  it=rep.createIterator();
}


TimerController::~TimerController () {
  delete it;
}


void TimerController::tick () {
  Mutex dummy(&mutex);
  for (it->reset(); !it->isDone(); it->next())
    (*it->currentItem())->tick();
}


void TimerController::sign (Timer* t) {
  Mutex dummy(&mutex);
  rep.add(t);
}


void TimerController::unsign (Timer* t) {
  Mutex dummy(&mutex);
  rep.remove(t);
}
```

```
//////////////////////////////////////////////////////////////////////
// class TimerInterrupt
//////////////////////////////////////////////////////////////////////

// Timer interrupt entry:
const int TimerIntNo = 0;


class TimerInterrupt : public InterruptHandler {
protected:

  TimerInterrupt () : InterruptHandler(TimerIntNo,timerInterrupt) {}

  static void timerInterrupt () { instance.interruptHandler(); }
  virtual int handle () { TimerController::Instance()->tick(); return 1; }

private:

  static TimerInterrupt instance;

};


TimerInterrupt TimerInterrupt::instance;
```

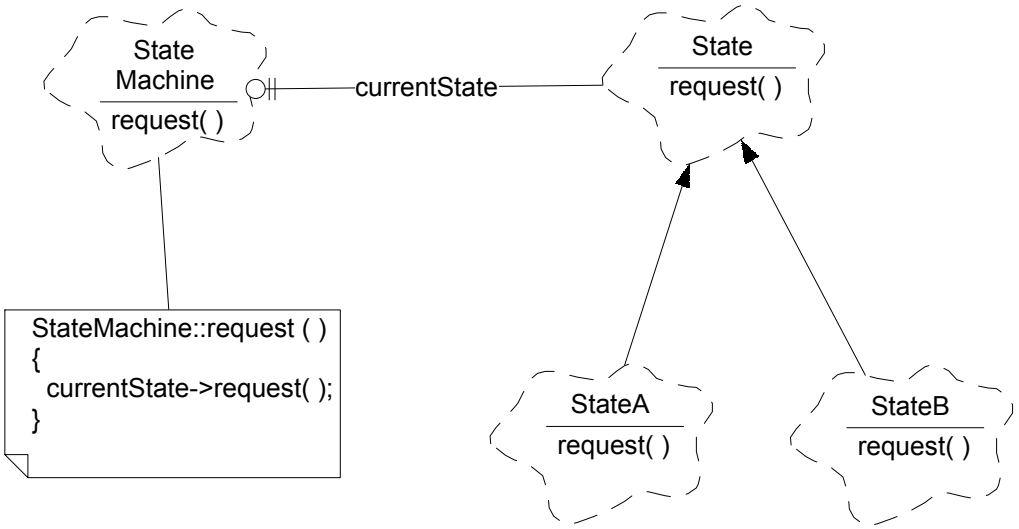
```
////////////////////////////////////////////////////////////////////
// class Timer
////////////////////////////////////////////////////////////////////

Timer::Timer (Time t, Timeable* tmbl) : myTimeable(tmbl), myThread(0),
                  counter(t), initial(t), isRunning(1),
                  mutex(new Semaphore(1)) {
  if (myTimeable!=0) {
    myThread=new TimerThread(myTimeable);
    myThread->start();
  }
  TimerController::Instance()->sign(this);
}


Timer::~Timer () {
  mutex->wait();
  TimerController::Instance()->unsign(this);
  if (myThread!=0) myThread->destroy();
  delete mutex;
}


Time Timer::stop () {
  Mutex dummy(mutex);
  isRunning=0;
  return initial-counter;
}


void Timer::restart (Time t) {
  Mutex dummy(mutex);
  if (t!=0)
    counter=initial=t;
  else
    counter=initial;
  isRunning=1;
}


Time Timer::elapsed () const {
  return initial-counter;
}


Time Timer::remained () const {
  return counter;
}


void Timer::tick () {
  Mutex dummy(mutex);
  if (!isRunning) return;
  if (--counter==0) {
    isRunning=0;
    if (myThread!=0) myThread->signal();
  }
}
```

Konaèni automati

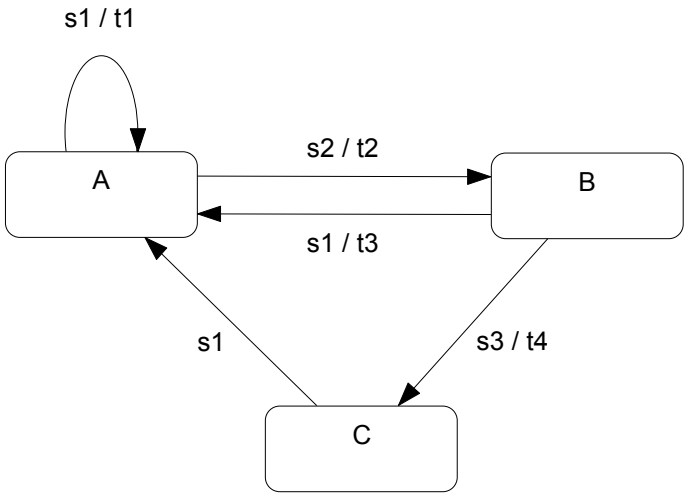
* Konaèni automati su jedan od najèešæe primenjivanih i najefikasnijih koncepata u projektovanju OO softvera za rad u realnom vremenu. Postoji mnogo naèina realizacije konaènih automata koje se u opštem sluèaju razlikuju po sledeæim najvažnijim parametrima:

1. Kontrola toka. Konaèni automat može imati sopstvenu, nezavisnu kontrolu toka (nit). U tom sluèaju automat prima poruke (signale) najèešæe preko nekog bafera, obraðuje ih jednu po jednu, a drugim automatima poruke šalje asinhrono. Kako je slanje poruka asinhrono, nema problema sinhronizacije, meðusobnog isključèenja i slièno. Sa druge strane, automat može biti i pasivan objekat, pri èemu se prelaz (obrada poruke) izvršava u kontekstu onoga ko je poruku poslao (pozivaoca).
2. Naèin prijema poruke. Poruke se mogu primati centralizovano, preko jedinstvene funkcije za prijem poruke, ili jedinstvenog bafera za poruke. U tom sluèaju sadržaj poruke određuje operaciju, odnosno prelaz automata. Sa druge strane, interfejs automata može da sadrži više operacija, i da svaka operacija predstavlja zapravo jedan dogaðaj (signal)-poruku automatu na osnovu koje se vrši prelaz.

* Ovde æe biti prikazan jedan jednostavan naèin realizacije automata. Izabran je pristup kojim se automat realizuje kao pasivan objekat, što znaèi da nema sopstvenu nit. Ako ovakav ovbjekat treba da funkcioniše u konkurentnom okruženju, onda njegove funkcije treba da budu meðusobno isključèive, što je ovde izostavljeno. Dalje, interfejs automata sadrži sve one operacije koje predstavljaju poruke (signale) na koje automat reaguje.
* Implementacija objekta-automata sadrži više podobjekata; to su stanja automata. Svi ovi podobjekti imaju zahednièki interfejs, što znaèi da su njihove klase izvedene iz osnovne klase stanja datog automata (u primeru klasa `State`). Ovaj interfejs stanja sadrži sve operacije interfejsa samog automata, s tim da je njihovo podrazumevano ponašanje prazno. Izvedene klase konkretnih stanja redefinišu ponašanje za svaku poruku za koju postoji prelaz iz datog stanja. Objekat-automat sadrži pokazivaè na tekuæe stanje, kome se obraæa preko zajednièkog interfejsa tako što poziva onu funkciju koja je pozvana spolja. Virtuelni mehanizam obezbeðuje da se izvrši prelaz svojstven tekuæem stanju. Posle prelaza, tekuæe stanje vraæa pokazivaè na odredišno, naredno tekuæe stanje.
* Na ovaj naèin dobija se efekat da objekat-automat menja ponašanje u zavisnosti od tekuæeg stanja (projektni šablon *State*), odnosno kao da "menja svoju klasu". Ovaj šablon prikazan je na sledeæem dijagramu klasa:



* Ogranièenja ovog jednostavnog koncepta su da ne postoji ugnežðivanje stanja, *entry* i *exit* akcije se vrše uvek, èak i ako je prelaz u isto stanje, nema inicijalnih prelaza ni pamæenja istorije.
* Realizacija opisanog šablona biæe prikazana na primeru sledeæeg automata:



* Izvorni kod za ovaj primer izgleda ovako:

```
// Project:  Real-Time Programming
// Subject:  Finite State Machines (FSM)
// Module:   FSM Example
// File:     fsmexmpl.cpp
// Date:     23.11.1996.
// Author:   Dragan Milicev
// Contents: State Design Pattern Example

#include <iostream.h>

////////////////////////////////////////////////////////////////////////
// class State
////////////////////////////////////////////////////////////////////////

class FSM;

class State {
public:

  State (FSM* fsm) : myFSM(fsm) {}

  virtual State* signal1 () { return this; }
  virtual State* signal2 () { return this; }
  virtual State* signal3 () { return this; }

  virtual void entry () {}
  virtual void exit  () {}

protected:

  FSM* fsm () const { return myFSM; }

private:

  FSM* myFSM;

};
```

```
//////////////////////////////////////////////////////////////////
// classes StateA, StateB, StateC
//////////////////////////////////////////////////////////////////

class StateA : public State {
public:

  StateA (FSM* fsm) : State(fsm) {}

  virtual State* signal1 ();
  virtual State* signal2 ();

  virtual void entry () { cout<<"Entry A\n"; }
  virtual void exit  () { cout<<"Exit  A\n"; }

};


class StateB : public State {
public:

  StateB (FSM* fsm) : State(fsm) {}

  virtual State* signal1 ();
  virtual State* signal3 ();

  virtual void entry () { cout<<"Entry B\n"; }
  virtual void exit  () { cout<<"Exit  B\n"; }

};


class StateC : public State {
public:

  StateC (FSM* fsm) : State(fsm) {}

  virtual State* signal1 ();

  virtual void entry () { cout<<"Entry C\n"; }
  virtual void exit  () { cout<<"Exit  C\n"; }

};
```

```
//////////////////////////////////////////////////////////////////////
// class FSM
//////////////////////////////////////////////////////////////////////

class FSM {
public:

  FSM ();

  void signal1 ();
  void signal2 ();
  void signal3 ();

protected:

  friend class StateA;
  friend class StateB;
  friend class StateC;
  void transition1 () { cout<<"Transition 1\n"; }
  void transition2 () { cout<<"Transition 2\n"; }
  void transition3 () { cout<<"Transition 3\n"; }
  void transition4 () { cout<<"Transition 4\n"; }

private:

  StateA stateA;
  StateB stateB;
  StateC stateC;

  State* currentState;

};
```

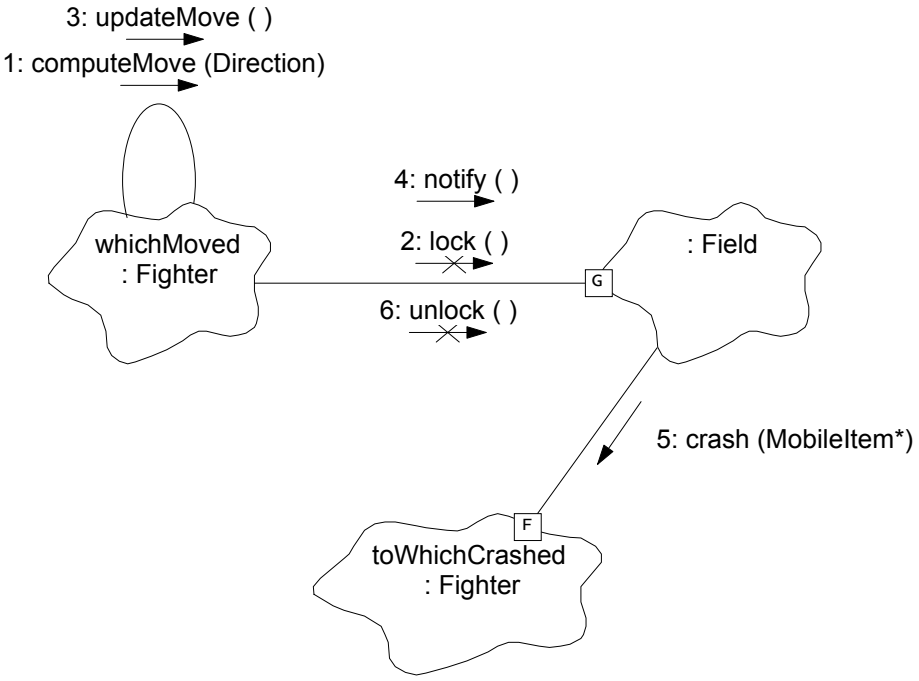
```
FSM::FSM () : stateA(this), stateB(this), stateC(this),
              currentState(&stateA) {
  currentState->entry();
}


void FSM::signal1 () {
  currentState->exit();
  currentState=currentState->signal1();
  currentState->entry();
}


void FSM::signal2 () {
  currentState->exit();
  currentState=currentState->signal2();
  currentState->entry();
}


void FSM::signal3 () {
  currentState->exit();
  currentState=currentState->signal3();
  currentState->entry();
}
```

```
//////////////////////////////////////////////////////////////////////
// Implementation
//////////////////////////////////////////////////////////////////////

State* StateA::signal1 () {
  fsm()->transition1();
  return this;
}

State* StateA::signal2 () {
  fsm()->transition2();
  return &(fsm()->stateB);
}


State* StateB::signal1 () {
  fsm()->transition3();
  return &(fsm()->stateA);
}

State* StateB::signal3 () {
  fsm()->transition4();
  return &(fsm()->stateC);
}


State* StateC::signal1 () {
  return &(fsm()->stateA);
}
```

```
//////////////////////////////////////////////////////////////////////
// Test
//////////////////////////////////////////////////////////////////////

void main () {
  cout<<"\n\n";
  FSM fsm; cout<<"\n";
  fsm.signal1(); cout<<"\n";
  fsm.signal2(); cout<<"\n";
  fsm.signal1(); cout<<"\n";
  fsm.signal3(); cout<<"\n";
  fsm.signal1(); cout<<"\n";
  fsm.signal2(); cout<<"\n";
  fsm.signal3(); cout<<"\n";
  fsm.signal2(); cout<<"\n";
  fsm.signal1(); cout<<"\n";
}
```

Primer jednostavne aplikacije

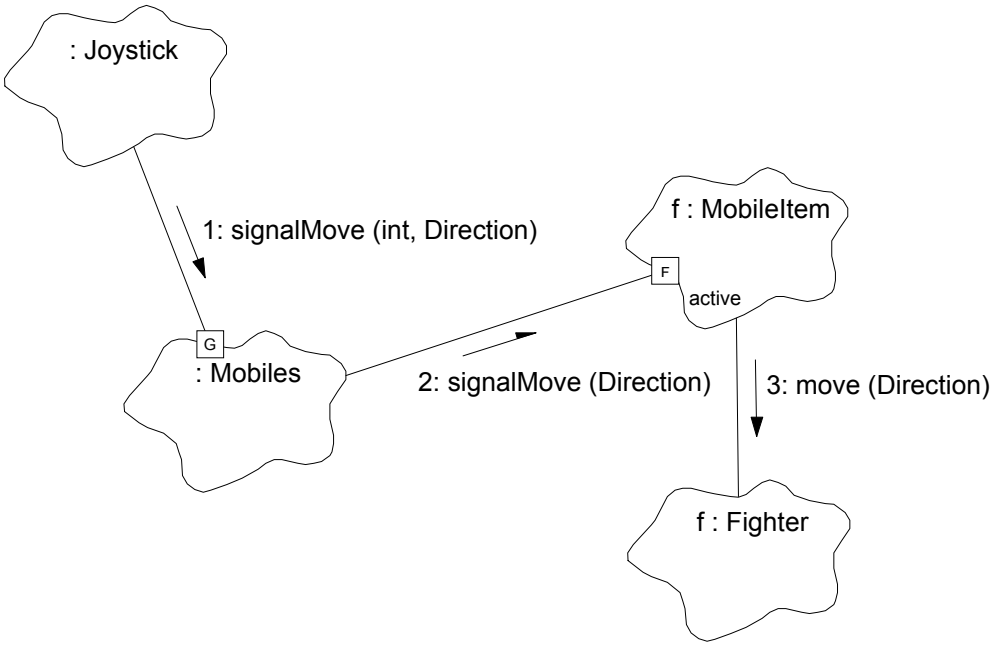
* Prikazaæemo na kraju primer jedne sasvim jednostavne ali fleksibilne aplikacije koja koristi mnoge prikazane elemente za rad u realnom vremenu. Aplikacija je projektovana tako da omoguæuje razna proširenja, èime može da postane realni simulator u relanom vremenu. Zbog toga su neki delovi napravljeni i malo složenije nego što je to bilo neophodno prema datim zahtevima. Proširenja se ostavljaju èitaocu.
* Aplikacija treba da predstavlja raèunarsku simulaciju jedne krajnje jednostavne igre koja æe biti nazivana *Fight* ("borba"). Igru igraju dva igraèa. Svaki igraè rukuje malim objektom na "bojnom polju" (ekranu) pomoæu igraèke palice (engl. *joystick*). Objekti se kreæu u polju koje predstavlja pravougaoni celobrojni koordinanti sistem sa jedinièним pomerajima. Objekti se mogu kretati u èetiri smera, sever, zapad, jug i istok. Igra traje sve dok se dva objekta ne sudare, što znaèi da se naðu u istoj taèki koordinatnog sistema. Svaki pomeraj objekta zadaje njegov smer kretanja i pomera ga za jedinièNu vrednost u tom smeru, koji se naziva tekuæim smerom. Prilikom sudara, ukoliko su tekuæi smerovi dva objekta suprotni, ishod igre je nerešen. Inaèe, pobednik je onaj igraè koji je udario u drugog, jer ga je udario sa strane ili sa leða.
* Pobuda (ulaz) aplikacije je sledeæa. Svaki pokret bilo koje igraèke palice generiše prekid. Pri tome se u jedan registar hardvera upisuje redni broj igraèa koji je pomerio palicu (0 ili 1), a u drugi oznaka smera pomeraja (0..3). Ako je prekid generisan sa vrednošæu broja igraèa 2, igra je prekinuta spolja. Interaktivni izlaz simulacije ovde ne treba realizovati, veæ samo treba navesti ishod igre.
* Poèetna kratka analiza zahteva ukazuje na sledeæa moguæa proširenja funkcionalnosti aplikacije koje treba uzeti u obzir prilikom projektovanja. Prvo, sasvim je logièno da igru može da igra i više od dva igraèa, bez suštinskog menjanja ulaza u program. Drugo, moguæe je da se na bojnom polju naðu i ostali objekti. Ti objekti mogu da budu ili statièki (fiksirani), npr. neke prepreke, "zidovi i hodnici" i slièno, ali mogu da budu i mobilni objekti koje upravlja raèunar, a ne igraèi. Treæe, objekti mogu da budu ne proste taèke, veæ složeniji oblici koji imaju svoje dve dimenzije. Takoðe, brzine kretanja objekata mogu da budu razlièite i da se menjaju. Najzad, treba predvideti i prikaz simulacije na ekranu u realnom vremenu.
* Analiza zahteva polazi od ključène funkcionalne taèke koju treba predstaviti dijagramom scenarija, a to je pomeraj jednog igraèa, uz eventualni sudar sa drugim. Tokom projektovanja scenarija, uoèavaju se ključène apstrakcije (klase) i njihove kolaboracije, èime se definiše osnovni mehanizam rada aplikacije. Ovaj scenario prikazan je na sledeæem dijagramu:



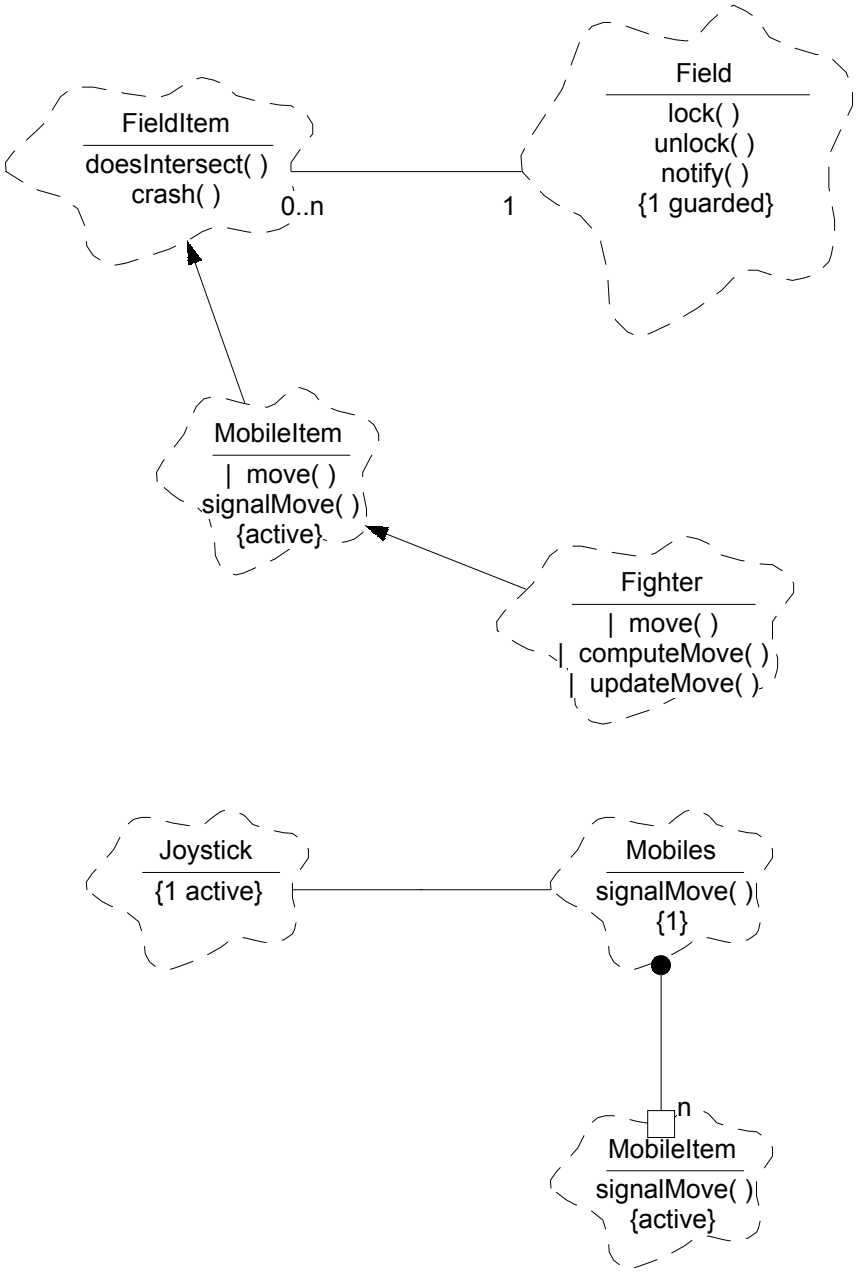
* Ključèni deo mehanizma je sledeæi. Kada se pomeri jedan igraè, treba utvrditi da li je tim pomeranjem došlo do sudara sa nekim drugim objektom na polju. Jedan centralizovani objekat klase `Field` vodiæe evidenciju o svim objektima na polju i utvrdiæe da li se dati objekat koji se pomerio sudario sa nekim drugim. Objekti koji se pomeraju su aktivni, što znaèi da postoje konkurentni procesi koji pomeraju objekte. Zbog toga se ne sme desiti da se jedan objekat pomeri, prijavi to objektu `Field`, ovaj utvrdi sudar sa nekim drugim objektom koji se u međuvremenu pomerio. Zbog toga sekvenca operacija promene položaja jednog objekta i utvrđivanje sudara sa nekim drugim mora da bude neprekidiva, tzv. *transakcija*. Kako operaciju pomeranja obavlja sam objekat, a jedino je objekat `Field` centralizovan, transakcija se obezbeđuje tako što je objekat `Field` *èuvani* (engl. *guarded*): spoljni objekat (klijent) mora da zahteva

poèetak transakcije operacijom `lock()` i da oznaèi kraj transakcije peracijom `unlock()`. Između ova dva poziva obavljaju se operacije promene položaja i obaveštavanja objekta `Field` o toj promeni (operacija `notify()`).

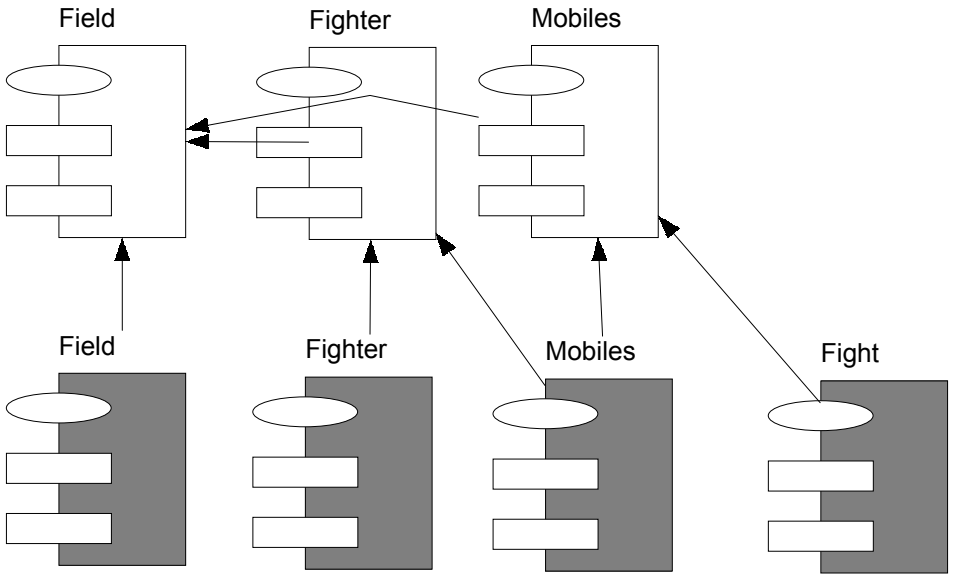
* Da opisana transakcija ne bi trajala suviše dugo, što se može dogoditi ako je pomeraj složen (postoji brzina objekta, rotacija i slièno), posebna pripremna funkcija `computeMove()` objekta izraèunava novi položaj objekta posle pomeraja, i ne izvršava se u okviru transakcije. U okviru transakcije izvršava se samo operacija `updateMove()` koja samo novoizraèunati položaj smešta u interne atribute objekta, što znaèi da stvarno obavlja (verifikuje) pomeraj. Ukoliko se u toku transakcije pomerio i neki drugi objekat, on je mogao samo da izraèuna svoje nove koordinate, ali ne i da ih verifikuje, jer nije mogao da dobije ulaz u transakciju (operacija `lock()` ga je blokirala). Na taj naèin æe korektno biti detektovan i sudar.
* Sudar se obrađuje virtuelnom operacijom `crash()` èije ponašanje zavisi od konkretnih vrsta objekata koji su se sudarili, pa se mogu lako dograditi i razlièite druge vrste objekata i sudara.
* Drugi deo mehanizma je sama pobuda pokretnih objekata pomoæu palice. Ovaj deo mehanizma prikazan je na sledeæem dijagramu:



* Objekat klase `Joystick` je nadzornik prekida. On oèitava registre broja igraèa i smera pomeraja i signalizira pomeraj objektu klase `Mobiles`. Ovaj objekat vodi raèuna o svim igraèima, èime se jednostavno može poveæati broj igraèa. Ovaj objekat predstavlja kolekciju igraèa i prosleđuje poruku onom pokretnom objektu koji se pomerio. Ovaj pokretni objekat (klasa `MobileItem`) je zapravo u našem sluèaju objekat konkretne izvedene klase `Fighter` (igraè), koji obavlja svoju virtuelnu operaciju `move()` onako kako je prikazano na prethodnom scenariju.
* Sledeæa dva dijagrama klasa prikazuju najvažnije klase u sistemu i njihove relacije:



* Najzad, dijagram modula je prikazan na sledeæoj slici. Modul `Field` sadrži klasu `Field`, osnovne klase `FieldItem` i `MobileItem`, kao i pomoæne klase `Direction` i `Coord`. Modul `Fighter` sadrži realizaciju konkretne izvede klase igraèa. Modul `Mobiles` sadrži klasu `Mobiles`. Modul `Fight` sadrži klasu `Joystick`, kao i glavni program za testiranje.



* U nastavku je dat kompletan izvorni kod aplikacije. Treba obratiti pažnju na naèin testiranja aplikacije. Naime, jedina pobuda aplikacije je preko prekida i èitanja odgovarajuæih registara, koji se ne mogu obezbediti bez odgovarajuæeg hardvera. Međutim, prekid se može lako simulirati u glavnom programu, pozivom odgovarajuæe funkcije koja predstavlja prekidnu rutinu. Vrednosti registara se opet jednostavno "podmeæu" iz glavnog programa, tako što se simuliraju jednostavne funkcije niskog nivoa koje treba da èitaju registre. Na ovaj naèin je moguæe testirati softver na PC platformi pod odgovarajuèim operativnim sistemom, pri èemu je taj softver inaèe namenjen za ugraðiavanje u specijalizovani hardver, uz jednostavne izmene.
* Datoteka `field.h`:

```
// Project:  Real-Time Programming
// Subject:  Fight Sample Application
// Module:   Field
// File:     field.h
// Date:     29.11.1996.
// Author:   Dragan Milicev
// Contents: Battle field and its contents
//      Classes:  Coord
//                Direction
//                Field
//                FieldItem
//                MobileItem

#ifndef _FIELD_
#define _FIELD_

#include "collect.h"
#include "kernel.h"

////////////////////////////////////////////////////////////////////
// Class Coord
////////////////////////////////////////////////////////////////////

class Coord {
public:

  Coord (int xx, int yy) : x(xx), y(yy) {}

  friend int operator== (Coord, Coord);
  friend int operator!= (Coord a, Coord b) { return !(a==b); }

  friend Coord  operator+  (Coord x, Coord delta);
  Coord& operator+= (Coord delta) { return *this=*this+delta; }

private:
  int x,y;
};
```

```
//////////////////////////////////////////////////////////////////////
// Class Direction
//////////////////////////////////////////////////////////////////////

enum Direction { NORTH, WEST, SOUTH, EAST };
Direction minus (Direction);


//////////////////////////////////////////////////////////////////////
// Class FieldItem
//////////////////////////////////////////////////////////////////////

class MobileItem;

class FieldItem {
public:

  FieldItem ();
 ~FieldItem ();

  virtual int  doesIntersect (const FieldItem*, Coord) const
                       { return 0; }

  virtual int  crash (MobileItem*) { return 1; }
};
```

```
//////////////////////////////////////////////////////////////////////
// Class MobileItem
//////////////////////////////////////////////////////////////////////

class MobileItem : public FieldItem, public Thread {
public:

  void signalMove (Direction);

  virtual Direction direction () const { return dir; }

protected:

  MobileItem () : dir(NORTH), toFinish(0) {}
  virtual void run  ();

  virtual int  move (Direction) { return 1; }
        void cancel ();

private:

  Event ev;
  Direction dir;
  int toFinish;

};
```

```
////////////////////////////////////////////////////////////////////
// Class Field
////////////////////////////////////////////////////////////////////

class Mutex;
class Semaphore;

class Field {
public:

  static Field* Instance ();
 ~Field ();

  void add    (FieldItem*);
  void remove (FieldItem*);

  void lock   ();
  void unlock ();

  int notify (MobileItem* whoMoved, Coord movedTo);

protected:

  Field ();

private:

  CollectionU<FieldItem*> items;
  IteratorCollection<FieldItem*>* it;

  Mutex* mutex;
  Semaphore* sem;

};


#endif
```

* Datoteka `field.cpp`:

```
// Project:  Real-Time Programming
// Subject:  Fight Sample Application
// Module:   Field
// File:     field.cpp
// Date:     29.11.1996.
// Author:   Dragan Milicev
// Contents: Battle field and its contents
//      Classes:  Coord
//                FieldItem
//                MobileItem
//                Field

#include "field.h"

////////////////////////////////////////////////////////////////////////
// Class Coord
////////////////////////////////////////////////////////////////////////

int operator==(Coord a, Coord b) {
  return (a.x==b.x)&&(a.y==b.y);
}

Coord operator+ (Coord a, Coord b) {
  return Coord(a.x+b.x,a.y+b.y);
}
```

```
////////////////////////////////////////////////////////////////////////
// Class Direction
////////////////////////////////////////////////////////////////////////

Direction minus (Direction d) {
  switch (d) {
    case NORTH: return SOUTH;
    case WEST:  return EAST;
    case SOUTH: return NORTH;
    case EAST:  return WEST;
    default:    return d;
  }
}




////////////////////////////////////////////////////////////////////////
// Class FieldItem
////////////////////////////////////////////////////////////////////////

FieldItem::FieldItem () {
  Field::Instance()->add(this);
}


FieldItem::~FieldItem () {
  Field::Instance()->remove(this);
}
```

```
////////////////////////////////////////////////////////////////////
// Class MobileItem
////////////////////////////////////////////////////////////////////

void MobileItem::signalMove (Direction d) {
  dir=d;
  ev.signal();
}

void MobileItem::cancel () {
  toFinish=1;
  ev.signal();
}


void MobileItem::run  () {
  while (1) {
    ev.wait();
    if (toFinish) return;
    if (move(dir)==0) return;
  }
}
```

```
////////////////////////////////////////////////////////////////////
// Class Field
////////////////////////////////////////////////////////////////////

Field* Field::Instance () {
  static Field instance;
  return &instance;
}


Field::Field () : it(items.createIterator()),
              sem(new Semaphore(1)), mutex(0) {}

Field::~Field () {
  delete it;
  delete mutex;
}


void Field::add (FieldItem* toInsert) {
  lock();
  items.add(toInsert);
  unlock();
}

void Field::remove (FieldItem* toRemove) {
  lock();
  items.remove(toRemove);
  unlock();
}
```

```
void Field::lock () {
  mutex=new Mutex(sem);
}

void Field::unlock () {
  delete mutex;
}


int Field::notify (MobileItem* m, Coord p) {
  for (it->reset(); !it->isDone(); it->next()) {
    FieldItem* f=*it->currentItem();
    if ( (f!=m) && (f->doesIntersect(m,p)) )
      return f->crash(m);
  }
  return 1;
}
```

* Datoteka `fighter.h`:

```
// Project:  Real-Time Programming
// Subject:  Fight Sample Application
// Module:   Fighter
// File:     fighter.h
// Date:     29.11.1996.
// Author:   Dragan Milicev
// Contents: Fighter item
//      Classes:  Fighter

#ifndef _FIGHTER_
#define _FIGHTER_

#include "field.h"

////////////////////////////////////////////////////////////////////////
// Class Fighter
////////////////////////////////////////////////////////////////////////

class Fighter : public MobileItem {
public:

  Fighter (int index, Coord position, Direction direction);

  virtual int  doesIntersect (const FieldItem*, Coord) const;
  virtual int  crash (MobileItem*);

  virtual Direction direction () const;

protected:

  virtual int  move  (Direction);

  void computeMove (Direction);
  void updateMove  ();
```

```
private:

  int index;

  Coord myPosition;
  Direction myDirection;

  Coord myNewPosition;
  Direction myNewDirection;

};


#endif
```

* Datoteka `fighter.cpp`:

```
// Project:  Real-Time Programming
// Subject:  Fight Sample Application
// Module:   Fighter
// File:     fighter.cpp
// Date:     29.11.1996.
// Author:   Dragan Milicev
// Contents: Fighter item
//      Classes:  Fighter

#include "fighter.h"
#include <iostream.h>

////////////////////////////////////////////////////////////////////////
// Class Fighter
////////////////////////////////////////////////////////////////////////

Fighter::Fighter (int i, Coord p, Direction d) : MobileItem(),
  index(i), myPosition(p), myDirection(d),
  myNewPosition(p), myNewDirection(d) {
  start();
}


int  Fighter::doesIntersect (const FieldItem*, Coord p) const {
  return (p==myPosition);
}


Direction Fighter::direction () const {
  return myDirection;
}
```

```
int Fighter::crash (MobileItem* m) {
  Direction d=m->direction();
  if (d==minus(myDirection)) {
    // Mutual crash:
    cout<<"Mutual crash.\n";
  } else {
    // m wins:
    cout<<"Crash.\n";
  }
  cancel();
  return 0;
}


int Fighter::move (Direction d) {
  computeMove(d);
  Field::Instance()->lock();
  updateMove();
  int ret=Field::Instance()->notify(this,myPosition);
  Field::Instance()->unlock();
  return ret;
}


void Fighter::computeMove (Direction d) {
  myNewDirection=d;
  int deltaX=0, deltaY=0;
  switch (d) {
    case NORTH: deltaY=1;  break;
    case WEST:  deltaX=-1; break;
    case SOUTH: deltaY=-1; break;
    case EAST:  deltaX=1;  break;
  }
  myNewPosition=myPosition+Coord(deltaX,deltaY);
}


void Fighter::updateMove () {
  myPosition=myNewPosition;
  myDirection=myNewDirection;
}
```

* Datoteka `mobiles.h`:

```
// Project:  Real-Time Programming
// Subject:  Fight Sample Application
// Module:   Mobiles
// File:     mobiles.h
// Date:     29.11.1996.
// Author:   Dragan Milicev
// Contents: Mobile items in the battle field
//      Classes:  Mobiles

#ifndef _MOBILES_
#define _MOBILES_

#include "field.h"

////////////////////////////////////////////////////////////////////
// Class Mobiles
////////////////////////////////////////////////////////////////////

const int MAXMOB = 2;

class Mobiles {
public:

  Mobiles ();
  static Mobiles* Instance();

  void signalMove (int index, Direction);

private:

  MobileItem* mobiles[MAXMOB];

};


#endif
```

* Datoteka `mobiles.cpp`:

```
// Project:  Real-Time Programming
// Subject:  Fight Sample Application
// Module:   Mobiles
// File:     mobiles.cpp
// Date:     29.11.1996.
// Author:   Dragan Milicev
// Contents: Mobile items in the battle field
//      Classes:  Mobiles

#include "mobiles.h"
#include "fighter.h"

////////////////////////////////////////////////////////////////////
// Class Mobiles
////////////////////////////////////////////////////////////////////

Mobiles* Mobiles::Instance () {
  static Mobiles instance;
  return &instance;
}


Mobiles::Mobiles () {
  for (int i=0; i<MAXMOB; i++)
    mobiles[i]=new Fighter(i,Coord(i,i),NORTH);
}


void Mobiles::signalMove (int i, Direction d) {
  mobiles[i]->signalMove(d);
}
```

* Datoteka `fight.cpp`:

```
// Project:  Real-Time Programming
// Subject:  Fight Sample Application
// Module:   Fight
// File:     fight.cpp
// Date:     29.11.1996.
// Author:   Dragan Milicev
// Contents: Main control
//      Classes:  Joystick
//      Function: userMain

#include "kernel.h"
#include "mobiles.h"

#include <iostream.h>

////////////////////////////////////////////////////////////////////////
// Class Joystick
////////////////////////////////////////////////////////////////////////

class Joystick : public InterruptHandler {
private:
  friend void userMain();
  static Joystick* instance;
  Joystick ();

  static void interruptHandler ();
  virtual int handle ();

};


Joystick* Joystick::instance=0;

Joystick::Joystick () : InterruptHandler(0,interruptHandler) {}


void Joystick::interruptHandler () {
  instance->InterruptHandler::interruptHandler();
}


int readIndex ();
Direction readDirection ();


int Joystick::handle () {
  int index=readIndex();
  if (index==MAXMOB) return 0;
  Direction dir=readDirection();
  Mobiles::Instance()->signalMove(index,dir);
  return 1;
}
```

```
//////////////////////////////////////////////////////////////////////
// Testing environment
//////////////////////////////////////////////////////////////////////

int index=0;
Direction direction=NORTH;

int readIndex () {
  return index;
}

Direction readDirection () {
  return direction;
}


void userMain() {
  Field::Instance();
  Mobiles::Instance();
  Joystick::instance=new Joystick;
  cout<<"\n\n";

  // Move 0 to NORTH:
  cout<<"Move 0 to NORTH.\n";
  index=0; direction=NORTH;
  Joystick::interruptHandler();
  dispatch();

  // Move 1 to SOUTH:
  cout<<"Move 1 to SOUTH.\n";
  index=1; direction=SOUTH;
  Joystick::interruptHandler();
  dispatch();

  // Move 1 to NORTH:
  cout<<"Move 1 to NORTH.\n";
  index=1; direction=NORTH;
  Joystick::interruptHandler();
  dispatch();

  // Move 0 to EAST:
  cout<<"Move 0 to EAST.\n";
  index=0; direction=EAST;
  Joystick::interruptHandler();
  dispatch();

  // Stop:
  cout<<"Stop.\n";
  index=2;
  Joystick::interruptHandler();
  dispatch();

}
```