Koncept apstraktnog tipa podataka – klase i objekti

U prethodnoj sekciji opisan je apstraktni objekt brojač. Realizirali smo ga pomoću jednog programskog modula.

Problem te realizacije je što u jednom programu možemo imati samo jedan takovi brojač. Ako bi željeli imati više brojača morali bi napisati više modula s različito označenim funkcijama kojima se manipulira s brojačima.

Mnogo elegantnije rješenje bilo bi da se brojač realizira kao programski objekt koji se deklarira i inicijalizira nekim tipom podatka, na sličan način kao što se primjerice cjelobrojne varijable deklariraju tipom int.

Na taj način bi se moglo u jednom programu imati više objekata brojača s kojima se manipulira na jedinstven način.

Prije nego se izvrši implementacija tipova i objekta u nekom programskom jeziku često je korisno da se zapis karakteristika objekta izvrši na apstraktnoj razini kao apstraktni tip podataka – ADT (abstract data type).

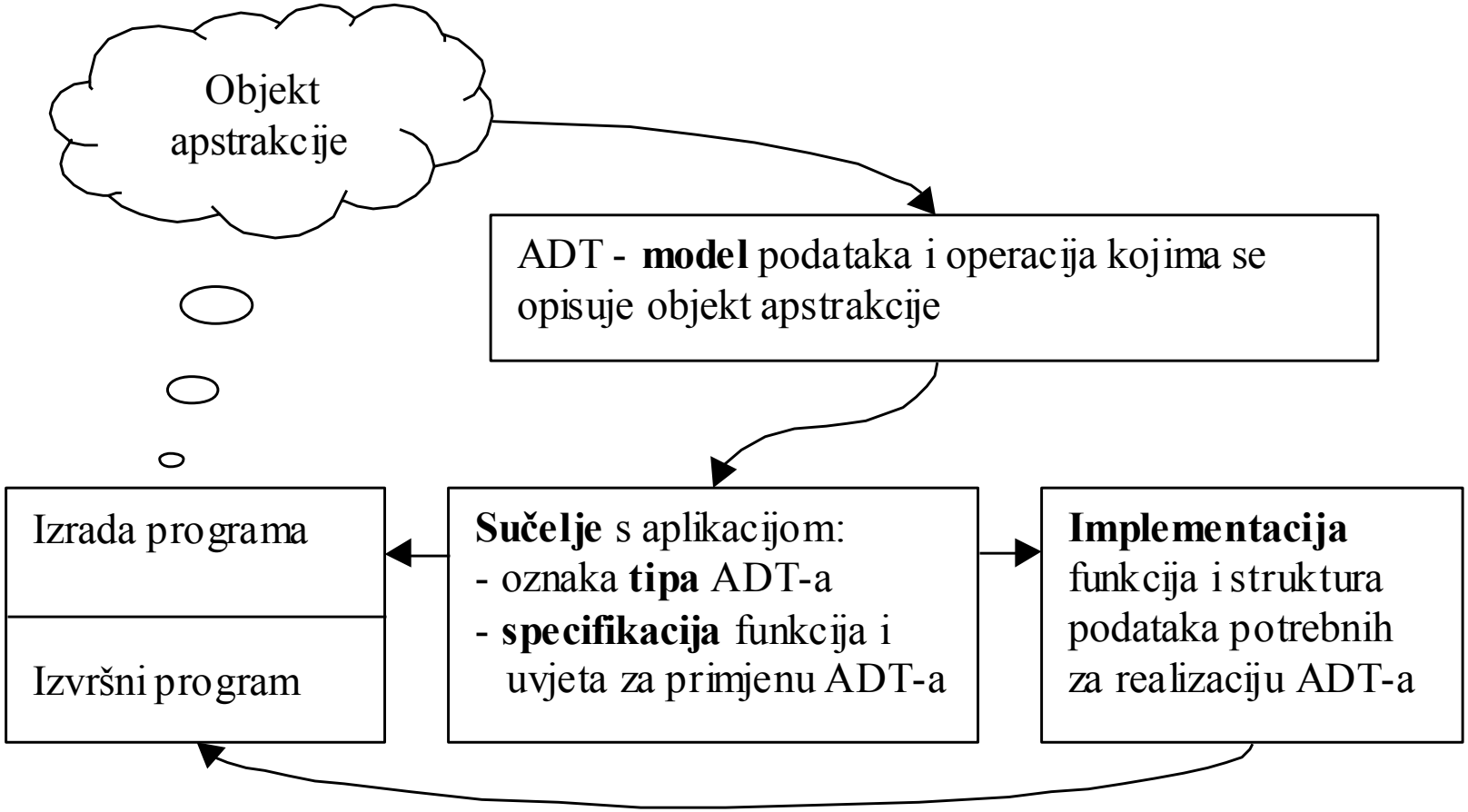
ADT opisuje stanja i ponašanja (aktivnosti) istovrsnih objekata.

- **Kada objekt mijenja stanje zbog djelovanja vanjske okoline tada kažemo da objekt prima poruke - mutatore.**
- **Kada objekt izvještava o svojim stanjima takove poruke zovemo pristupnicima.**

U programskim jezicima i u matematici stanja objekta možemo zapisivati varijablama, a aktivnost objekta zapisujemo funkcijama.

Vijek trajanja svakog objekta je ograničen.

- **Postupak kojim nastaje neki objekt zvat ćemo konstruktor objekta**
- **Postupak kojim se uništava neki objekt destruktor objekta.**



Konceptualni model ADT-a

Model formiranja i korištenja ADT-a u C++

1. Model

Model se opisuje algoritamskim zapisima ili matematičkom aksiomatikom operacija.

Za primjer brojača - aksiomatika operacija može biti slijedeća:

konstruktor( )	**brojač na nuli i modul na vrijednost MAX_INT**
incr_count( )	**inkrementira se vrijednost brojača za jedan. Ako je jednak modulu postavlja se na nulu. Daje novu vrijednost brojača**
get_count( )	**daje trenutnu vrijednost brojača**
get_modul( )	**daje trenutnu vrijednost brojača**
set_modul(mod)	**postavlja vrijednost modula na vrijednost mod. Ako je mod manji od 2 modul se postavlja na MAX_INT.**
reset( )	**postavlja vrijednost brojača na nulu**

2. Specifikacija

Na temelju modela ADT-a izrađuje se specifikacija u C++ jeziku koja mora sadržavati:

- **identifikator tipa kojim se značava ADT,**
- **funkcije koje predstavljaju konstruktor i destruktor objekta,**
- **prototip funkcija kojima se realizira model operacija s apstraktnim objektom (mutatori objekta i pristupnici objektu),**
- **uz funkcije treba jasno dokumentirati koji su uvjeti za primjenu funkcije (eng. precondition) i stanje objekta nakon djelovanja funkcije (eng. postcondition).**

Specifikacija ADT-s se u C++ jeziku zapisuje blokom koji se zove klasa ili struktura.

Zaglavlje bloka ADT počinje s ključnom riječi `class` (ili `struct`) iza koje se navodi ime apstraktnog tipa. Unutar bloka zapisuje se deklaracije funkcija i varijabli, koji opisuju ponašanje i stanje objekta. Oni se nazivaju članovi klase.

```
class  Counter
{
private:           //deklaracija "privatnih" članova
   int m_count;
    .....
public:       //deklaracija javnih članova
   .....
   int increment_count();

}
```

Pomoću ključnih riječi `private, public i protected` i znaka dvotočke označavaju se:

- `public:` **članovi koji su dostupni ostalom dijelu programa**
- `private:` **članovi koji su dostupni samo članovima ove klase**
- `protected`**: članovi koji su dostupni članovima ove klase i klasama koje nasljeđuju svojstva ove klase. Koncept nasljeđivanja bit će objašnjen kasnije**

```
class Counter          // ili struct Counter
{
private:  // specifikacija privatnih članova klase Counter

   int m_count;        // trenutno stanje zbroja
   int m_mod;          // modul brojača

public:  // specifikacija javnih članova klase Counter

    Counter();         // konstruktor objekta Counter
   ~Counter();         // destruktor objekta Counter

    // pristupnici:
   int get_count();  // daje trenutno stanje brojača
   int get_modulo(); // daje vrijednost modula brojača

  // mutatori
   void set_modulo(int mod);  // postavlja modul
   int increment_count();     // inkrementira brojača
   void reset();              // vrijednost brojača na nulu

protected:
  // specifikacija zaštićenih članova klase ADT- Counter
  // zasada nisu potrebni
};
```

Funkcija konstruktora mora imati isto ime kao i definirana klasa. Funkcija destruktora započima znakom ~ i imenom klase.

Posebnost deklaracije konstruktora i destruktora klase u odnosu na ostale funkcije je da se ne deklarira tip koji funkcija vraća.

3. Implementacija

Na temelju specifikacije vrši se implementacija modela, odnosno definiranje funkcija – članova klase. Implementacija se zapisuje u jednoj ili više datoteka. Obično se specifikacija zapisuje u ".h" datotekama, a implementacija funkcija u ".cpp" datotekama. Ispred imena svake članske funkcije navodi se ime klase i dvostruka dvotočka, primjerice

```
Counter::Counter()   // konstruktor klase
{  m_count=0;        // početno stanje brojača
   m_mod = INT_MAX; // 2147483647
}

Counter::~Counter()   // destruktor klase
{  // nema funkciju o ovom primjeru  }

inline void Counter::reset() {m_count = 0;}
inline int Counter::get_count(void){ return m_count;}
inline int Counter::get_modulo(void){ return m_mod; }

void  Counter::set_modulo(int mod)
{   if(mod <= 1)   m_mod = INT_MAX;
    else           m_mod =  mod;
}

int Counter::incr_count(void)
{   m_count++;
    if(m_count >= m_mod) m_count = 0;
    return m_count;
}
```

4. Aplikacija – stvaranje objekata koji predstavljaju pojavnost (instancu) klase

Prvo se mora se deklarirati objekt koji je opisan klasom Counter. Deklaracija se zapisuje na isti način kao i za proste tipove, pa

```
Counter c;      // stvaranje objekta c
```

predstavlja deklaraciju kojom se definira objekt c koji je tipa (ili klase) Counter.
Kompilator u sklopu deklaracije skriveno poziva izvršenje konstruktor funkcije. U ovom primjeru vrši se inicijalizacija privatnih varijabli (`m_count=0 i m_mod = INT_MAX`). Na izasku iz bloka u kojem je definiran objekt c, kompilator skriveno poziva destruktor funkciju.

Djelovanje na objekt označava se zapisom od tri dijela: ime objekta, znak točka i ime varijable ili članske funkcije (s pripadnim argumentima) ili varijable. Primjerice,

```
Counter c;                  // stvaranje objekta c

c.set_modul(3);          // vrijednost modula na 3
c.incr_count();          // inkrementira se brojač
cout << c.get_count();// ispisuje se vrijednost
```

Ako bi mjesto `c.set_modul(3);` napisali

```
c.m_modul = 3;
```

kompilator bi dojavio grešku: "nije dozvoljen pristup privatnim članovima klase".

Ako bi pak u deklaraciji klase Counter umjesto riječi `private` napisali `public`, tada bi i ovaj iskaz bio dozvoljen.

“inline” funkcije

Poziv funkcije i povrat iz funkcije su posebne naredbe koje traju izvjesno vrijeme. Kada se definira funkcija, kompilator generira skup strojnih naredbi koje se smještaju u memoriju. Pri pozivu funkcije vrši se skok na te naredbe. One se zatim sekvencijalno izvršavaju, i kada se dođe do naredbe za povrat iz funkcija, izvršenje se vraća natrag na naredbu koja slijedi iza poziva funkcije. Ako se funkcija poziva 10 puta, to znači da će se opisani postupak ponoviti 10 puta, jer postoji samo jedan memorijski skup strojnih naredbi koji opisuje izvršenje funkcije.

U C++ jeziku poziv funkcije se može izvršiti i na drugi način, a to je da se za svaki poziv funkcije generira odvojeni skup naredbi za izvršenje te funkcije, odnosno da se na mjesto poziva umeće tijelo pozvane funkcije. To znači da će za deset poziva funkcije biti deset puta generiran kod za njeno izvršenja. Ovaj način poziva funkcije zove se “inline” i može se ostvariti tako da se u deklaraciji ili definiciji funkcije dopiše prefiks inline,

Primjer: inline definicija za proračun maksimalne vrijednosti:

```
inline abs(int x)
{
   return  x>0 ? x : -x
}
```

tada je programski odsječak

```
int a,b, x=6, y=-6
a=abs(x);
b=abs(y);
```

ekvivalentan s :

```
int x=6, y=-6;
a = x>0 ? x : -x ;
b = y>0 ? y : -y ;
```

Inline je poruka kompilatoru da se programska implementacija funkcije izvrši tako da se njen kod umetne na svako mjesto gdje se ona poziva. Dozvoljeno je da kompilatori ignoriraju ovu direktivu ako procijene da to nije od koristi, primjerice ako se funkcija sastoji od većeg broja naredbi.

Definiranje klase brojača s inline funkcijama

Dozvoljeno je da se definiranje funkcija izvrši i unutar tog bloka.
Pravilo je da se tada te funkcije smatraju inline funkcijama.

```
class Counter {            // specifikacija članova klase
    int m_count;           // stanje brojača
    int m_mod;             // modul izbroja

public:
    // konstruktor
    Counter(): m_count(0), m_mod(INT_MAX) {}

     // pristupnici - automatski "inline"
    void reset() {m_count= 0;}
    int get_count(void){ return m_count;}
    int get_modulo(void) {return m_mod;}

     // mutatori
     int incr_count(void);
     void  set_modulo(int mod)

};
```

```
// Izvan tijela klase definiramo funkcije
// koje imaju više linija koda
```

```
void  Counter::set_modulo(int mod)
{
    if(mod <= 1)    m_mod = INT_MAX;
     else            m_mod =  mod;
}

int Counter::incr_count(void)
{   m_count++;
    if(m_count >= m_mod) m_count = 0;
    return m_count;
}
```

```
#include <iostream>

using namespace std;

int main() // testiranje klase Counter
{
       Counter c1, c2; // deklariraju se dva brojača
       c1.set_modulo(4);
       c2.set_modulo(3);

       for(int i=0; i < 10; i++)     {
          c1.incr_count();
          c2.incr_count();
          cout << "c1=" << c1.get_count() <<'\t'
               << "c2=" << c2.get_count() <<'\n';
       }

       return 0;
}
```

Potrebno je uočiti još neke razlike od prvog primjera:

<u>Prvo</u>, ispred deklaracije članskih varijabli nije napisan atribut private. Koristi se pravilo da svi članovi klase koji se definiraju na početku tijela klase tretiraji se kao privatni članovi. Ako se za specifikaciju ADT umjesto ključne riječi `class` koristi ključna riječ `struct` tada vrijedi obrnuto, tj. članovi bez oznake atributa tretiraju se kao javni članovi. Ovo je jedina razlika između ova dva oblika specifikacije ADT-a, a postoji zbog kompatibilnosti s C jezikom.

```
class C {
    int x;
}


class D {
public:
   int y;
}
```

ekvivalenti zapis:

```
struct C {
private:
   int x;
}

struct D {
   int y;
}
```

Drugo, definicija konstruktora je iskazana na drukčiji način nego u prvom primjeru, tj. zapis

```
Counter ()  {
    m_count=0;        // početno stanje brojača
    m_mod = INT_MAX; // 2147483647
}
```

je zamijenjen s

```
Counter(): m_count(0), m_mod(INT_MAX) {}
```

Ovaj drugi način se preporučuje za iniciranje članskih varijabli, a vrši se prema pravilu da se između zaglavlja konstruktora i tijela konstruktora napiše znak dvotočke i lista inicijalnih vrijednosti članskih varijabli. Inicijalna vrijednost se zapisuje u zagradama iz imena članske varijable.

Treće, nije zapisana definicija destruktora. Pravilo je da ne treba pisati funkciju destruktora ako se s njime ne vrši nikakova radnja.

Konstruktor s parametrima

U prethodnom primjeru

```
Counter c1, c2;
c1.set_modulo(4);
c2.set_modulo(3)
```

nakon deklaracije dva brojača postavljena je vrijednost modula. Konstruktor se može tretirati kao funkcija koja može imati proizvoljan broj parametara, s tim da neki od tih parametara mogu imati predodređene vrijednosti. Primjerice, ako bi konstruktor definirali na slijedeći način

```
Counter (int mod =INT_MAX): m_count(0), m_mod(mod){}
```

tada se inicijalizacija oba brojača mogla provesti u samoj deklaraciji:

```
Counter c1(4), c2(3);
```

Ako bi pak želili da se drugi brojač inicira na mod INT_MAX, tada bi pisali

```
Counter c1(4), c2;
```

Ova tehnika se ne može koristiti kod destruktora, jer destruktor ne smije imati parametre.

Višestruki, predodređeni i kopirni konstruktor

U jednoj klasi može biti definirano više konstruktora, primjerice:

```
class Counter
{
public:
Counter (): m_count(0), m_mod(INT_MAX) {}
Counter (int mod): m_count(0), m_mod(mod) {}
.....
```

Prvi oblik, u kojem konstruktor nema parametara naziva se predodređeni konstruktor, ili "default" konstruktor. On ima posebni značaj, jer se aktivira ako se u deklaraciji objekta napiše ime konstruktora bez zagrada,

```
Counter c1;
```

Ponekad se koriste klase bez konstruktora, primjerice:

```
struct Position {
    int x;
    int y;
}

Position p1;          // da li postoji konstruktor?
p1.x = 5; p1.y = 6;   // sami postavimo početne
vrijednosti
Position p2 = p1;     // kako se kopira p1 u p2 ?
....
```

Ako u klasi nije definiran ni jedan konstruktor kompilator sam poziva tzv. prazni predodređeni konstruktor oblika: Position(){}, tj konstruktor kojim se ne izvodi nikakva operacija.

Kopirni konstruktor X (const X &).

Kompilator dodatno generira konstruktor oblika:

```
Position (const Position & obj)
{
    x = obj.x;
    y = obj.y;
}
```

Ovaj oblik konstruktora se naziva kopirni konstruktor.

```
X (const X &);
```

Parametar kopirnog konstruktora je referenca već postojećeg objekta. U tijelu ovog kopirnog konstruktora kopira se vrijednost svih članskih varijabli postojećeg objekta u varijable deklariranog objekta. Upravo pomoću ovog kopirnog konstruktora kompilator vrši inicijalizacija objekta p2 pomoću objekta p1 u iskazu:

```
Position p2 = p1;    // poziva se kopirni konstruktor
```

Iako to sada nije uočljivo, ponekad će biti potrebno u kopirnom konstruktoru obaviti i neke druge radnje, a na samo kopirati vrijednost članskih varijabli. U tim slučajevima korisnik će trebati sam napisati kopirni konstruktor, deklarariran u obliku: `X(const X &)`.

Još “const” osigurača

Sada još treba objasniti zašto je u deklaraciji zapisana ključna riječ `const` kao prefiks parametra funkcije.

X (const X &).

Ovakva deklaracija je poruka kompilatoru da se deklarirani parametar smije koristiti samo kao “konstantna” vrijednost. U tom slučaju kompilator nadgleda sve operacije koje se u funkciji provode s ovim objektom. Ako bi se pojavila naredba u kojoj se mijenja vrijednost članskih varijabli deklariranog objekta kompilator dojavljuje grešku. Riječ const se također može koristiti kao sufiks zaglavlja članskih funkcija klase, primjerice

```
class Counter {
public:
    int get_count(void)  const  {return m_count;}
    int get_modulo(void) const  {return m_mod;};
```

Time se osigurava da se tim funkcijama ne može mijenjati vrijednost članskih varijabli.

Pokazivači na objekte

Pokazivač na neki objekt deklarira se na isti način kao i pokazivač na skalarne tipove, jer se klasa tretira i kao oznaka tipa memorijskog objekta.

```
Counter Objekt;     // deklaracija objekta Objekt
Counter *pObjekt;   // deklaracija pokazivača pObjekt
pObjekt = &Objekt; // inicijalizacija pokazivača na
adr.Obj.
```

Pokazivač pObjekt pokazuje na objekt Objekt. Za pristup članovima objekta pomoću pokazivača, prvo je potrebno izvršiti indirekciju pokazivača, kako bi se dobila stvarna adresa objekta, a zatim pomoću točka operatora označiti člansku varijablu kojoj se pristupa ili funkciju koju se želi aktivirati. Primjerice, ako želimo pozvati člansku funkciju set_modulo(5) treba napisati:

```
(*pObjekt).set_modulo(5);
```

```
(*pObjekt).set_modulo(5);
```

Pošto je ovaj oblik označavanja prilično "nečitljiv", alternativno se može pisati

```
pObjekt->set_modulo(5);
```

dakle oznaka tipa

`(*p).` zamjenjuje se oznakom strelice `p->`

U praksi se gotovo isključivo koristi ovaj drugi oblik.

Operativni sustav	**korisnički program:** **operacijski kod**	**statički podaci**	**stog ili "stack"** **automatske var.**	**slobodna memorija** **"heap" ili hrpa**

Slika 7.2 Raspodjela memorije između operativnog sustava i korisničkog programa

Postupak, kojim se raspolaže s tom slobodnom memorijom, naziva se dinamičko alociranje memorije. Pojam alociranje memorije, asocira na činjenicu da se dio slobodne memorije dodjeljuje korisničkom programu, dakle apstraktno se mijenja njegova lokacija. Postupak kojim se memorija vraća operativnom sustavu naziva se dealociranje memorije. Praksa je pokazala da se dinamičkim alociranjem može znatno efikasnije iskoristiti memorijske resurse računala nego je to moguće korištenjem statičkih i automatskih varijabli. Zbog toga je u C++ jezik je ugrađen permanentan mehanizam za alociranja memorije. On se osvaruje pomoću dva operatora: `new i delete.`

Sintaksa izraza kojim se alocira memorija je:

new *ime_tipa*;
new *ime_tipa* (*argumenti konstruktora*);

Vrijednost izraza je adresa alocirane memorije, a veličina alocirane memorije iznosi sizeof(*ime_tipa*).

```
Counter *p= new Counter(5);
```

alocira se memorija za objekt klase Counter i pridjeljuje pokazivaču p. Također, poziva se konstruktor koji inicira modul brojača na vrijednost 5.

U slučaju da se ne može izvršiti alokacija memorije vrijednost pokazivača bit će nula (NULL). U tom je slučaju najbolje prekinuti izvršenje programa:

```
Counter *p= new Counter(5);


if (p==0) {
    cout << "greška u alokacije memorije";
    exit (1);
}
```

Kasnije ćemo pokazati napredniji postupak za prihvat ovakovih "iznimnih situacija".
Ako je alokacija izvršena ispravno, pokazivači sadrže adresu memorijskih objekata s kojima se manipulira indirekcijom pokazivača, primjerice

```
p->set_modul(3);
p->incr_count();
......
```

Kada dinamički alocirani objekti više nisu potrebni, treba izvršiti dealokaciju memorije. To se ostvaruje pomoću operatora delete:

```
delete p;
```

Djelovanje operatora `delete` se odvija u dva koraka. Prvo se vrši poziv destruktora objekta, a zatim se prethodno zauzeta memorija dealocira i označava kao slobodna za ponovno korištenje. Kažemo da je ta memorija vraćena na hrpu. Nakon izvršene dealokacije vrijednost pokazivača p1 i p2 je neodređena.

Važno je upamtiti slijedeće upute:

- **Alociranje se ne može provoditi na globalnoj razini, već samo unutar nekog bloka.**
- **Pokazivač alocirane memorije može biti globalna i lokalna varijabla.**
- **Ako je pokazivač alocirane memorije lokalna varijabla tada je nužno unutar bloka, gdje je pokazivač definiran, izvršiti i dealociranje memorije, jer po izlasku iz bloka prestaje postojati ta lokalna pokazivačka varijabla. Ako se to ne napravi, više se ne može izvršiti dealociranje memorije, pa ta memorija ostaje trajno zauzeta i nedostupna programu.**

Prijenos objekata u funkcije – moguć na tri načina

```
//1. prijenos parametara po vrijednosti (zauzima stog)

   int CompareCounters_V (Counter c1, Counter c2)
   {
      return c1.get_count() - c2.get_count();
   }

//2. prijenos parametara po referenci (na stogu adresa)

   int CompareCounters (Counter &c1, Counter &c2)
   {
      return c1.get_count() - c2.get_count();
   }

//3. prijenos parametara pomoću pokazivača (na stogu
adresa)

   int CompareCounters (Counter *pc1, Counter *pc2)
   {
      return pc1->get_count() - pc2->get_count();
   }
```

Još o "const" osiguračima

Loša strana tog načina prijenosa argumenata po adresi je da se time otvara mogućnost "bočnog djelovanja" na stanje objekata pozivne funkcije.
Dobra je praksa da se parametri funkcije, kojoj se prenosi adresa objekta, deklariraju s "const" prefiksom, ukoliko se unutar te funkcije ne namjerava mijenjati sadržaj objekta. Primjerice, s

```
int CompareCounters (const Counter &c1, const Counter &c2)
{
        return c1.get_count() - c2.get_count();
}
```

kompilator dobiva poruku da provjeri da li se u funkciji mijenja stanje objekata c1 i c2. Pošto se u tijelu funkcije poziva članska funkcija get_count(), tu provjeru se dalje vrši na način da se provjerava da li ta članska funkcija može mijenjati stanje objekta.
To znači da je funkciju trebalo biti deklarirati sa sufiksom const:

```
class Counter {
public:
int get_count() const {return m_count;}
}
```

Primjetimo još da smo drugi i treći oblik funkcije za usporedbu brojača označili istim imenom CompareCounters dok prva funkcija ima ime CompareCounters_V. Razlog tome je pravilo da više funkcija može imati isto ime ako se razlikuju po broju ili po tipu parametara. U prva dva slučaja uzima se da funkcije imaju istovrsne parametre, jer se pozivaju na isti način.

```
Counter c1, c2;
CompareCounters_V (c1, c2); //poziv po vrijednosti
CompareCounters (c1, c2);   // poziv po referenci
CompareCounters (&1, &2);   // poziv pomoću pokazivača
```

Vidimo da ako bi funkcija CompareCounters_V imala ime CompareCounters, kompilator ne bi mogao znati koju će funkciju pozvati i na koji će način izvršiti prenos argumenta u funkciju.

Prijateljske funkcije i klase – "friend" deklaracije

Funkcija CompareCounters nije članska funkcija već regularna funkcija. Međutim, pošto koristi objekte klase Counter, ona na neki način pripada području definicije same klase Counter. Za takove slučajeve predviđeno je da se deklaracija funkcija može izvršiti u tijelu definicije klase, na način da se ispred deklaracije napiše sufiks `friend` (prijatelj), primjerice:

```
class Counter
{
  friend int CompareCounters (const Counter &,
                              const Counter &);
  friend int CompareCounters (const Counter *,
                              const Counter *);
}
```

Oznaka da je funkcija prijatelj klase ujedno omogućuje da se u definiciji funkcije koriste privatni i zaštićeni članovi klase kojoj je funkcija prijatelj. Stoga se definicija funkcija CompareCounter sada može pisati u obliku:

```
int CompareCounters (Counter &c1, Counter &c2)
{
    return c1.m_count - c2.m_count;
}
```

Jedna funkcija može biti prijatelj različitih klasa.

Koncept prijateljstva vrijedi i za klase.

Pravilo je da ako se unutar neke klase izvrši deklaracija druge klase s prefiksom friend, primjerice

```
class Prva
{
....
friend class Druga;
....
}
```

time se daje dozvola da članske funkcije prijateljske klase imaju pristup zaštičenim i privatnim članovima klase.

Prijateljstvo može biti i obostrano.

To je ilustrirano u datoteci krug.cpp. Definirane su dvije klase CKrug i CKvadrat. Obje klase su prijatelj jedna drugoj. Uočite kako se formira krug unutar kvadrata i kako se formira kvadrat oko kruga.

```
class CKvadrat;   // unaprijedna deklaracija

class CKrug
{
    double m_radius;
public:
    friend class CKvadrat;
    void Radius(double r) {m_radius = r;}
    double Radius() {return m_radius;}
    double Povrsina (void) {return (3.14*m_radius*m_radius );}
    void Ukvadratu (CKvadrat k);
};

class CKvadrat
{
    double m_stranica;
public:
    friend class CKrug;
    void Stranica(double s) {m_stranica=s;}
    double Stranica() {return m_stranica;}
    double Povrsina() {return m_stranica*m_stranica;}
    void OkoKruga(CKrug k);
};

void CKrug::Ukvadratu (CKvadrat kv)
{  m_radius = kv.m_stranica /2; }  // private pristup

void CKvadrat::OkoKruga(CKrug k)
{   m_stranica = k.m_radius*2;  } // private pristup
```

“this” pokazivač

Pri iniciranju svakog objekta kompilator rezervira memoriju za taj objekt. Ako na objekte tipa Counter primijenimo operator sizeof(Counter) dobit ćemo vrijednost 8 bajta. To je veličina memorije koji zauzimaju članske varijable m_count i m_mod. Postavlja se pitanje: kako objekt pamti članske funkcije, odnosno kako objekt komunicira s članskim funkcijama?

U C++ jeziku to je riješeno tako da kompilator za svaki objekt inicira jednu pokazivačku varijablu u koju zapisuje adresu samog objekta. Ta varijabla ima status privatne varijable klase i naziva se this. Ova pokazivačka varijabla koristi se pri pozivu članskih funkcija kao skriveni argument, koji služi da funkcija dobije poruku o adresi objekta kojeg obrađuje.

Primjerice, stvarni poziv funkcije:

```
void Counter::set_modulo(int m) {m_mod=m;};
```

kompilator skriveno prevodi kao C-funkciju:

```
int Counter_set_modulo(int m, Counter *this)
{    this->m_mod = m;  }
```

kojoj se pored deklariranih parametara dodaje još jedan parametar – pokazivač na sam objekt koji se obrađuje.

Zapamti: Riječ this je ključna riječ C++ jezika i može se koristiti kao lokalna privatna varijabla klase u svim slučajevima kada trebamo raspolagati s pokazivačem na objekt.

Mogućnost korištenja this pokazivača u definiranju članskih funkcija pokazat ćemo u primjeru iz datoteke this.cpp.

```
class Tocka{
    int m_x;  int m_y;
public:
    Tocka(): m_x(0),m_y(0) {}
    Tocka(int x, int y): m_x(x), m_y(y) {}
    int x() {return this->m_x;}
    int y() {return this->m_y;}
    Tocka *adresa() {return this;}
};

int main( void)
{
    Tocka t(5,10);
    cout << "Tocka:" << t.x() << ',' << t.y() << endl;
    cout << "adresa objekta: " << &t << endl;
    cout << "this pokazivac: " << t.adresa() << endl;
    return 0;
}
```

```
Rezultat izvršenja programa je:
Tocka:5,10
adresa objekta: 0x74fdd0
this pokazivac: 0x74fdd0
```

Lokalni i statički članovi klase

Uz svaku pojavnost nekog objekta nastaju i njegove članske varijable. U svim primjerima koje smo do sada analizirali radili smo s članskim varijablama koje nastaju kada nastaje neki objekt, i koje nestaju kada nestaje neki objekt. Takove varijable se nazivaju lokalne varijable klase.

Postoje slučajevi kada ćemo trebati da svi objekti koji pripadaju nekoj klasi imaju i neke zajedničke varijable. Takove varijable moraju postojati čim se definira neka klasa, jer čim se definira prvi objekt pomoću te klase on mora raspolagati s tom varijablom. Nestankom tog objekta ne smije nestati ta zajednička varijabla jer se kasnije ponovo može inicirati jedan ili više novih objekata koji će koristiti tu varijablu. Zbog ovog svojstva takove se varijable nazivaju statičke varijable klase. Deklariraju unutar klase tako da se deklaracija započne rječju static;

```
class CL {
public:
    static int n;        // statička varijabla klase
    ...
```

U radu kompilatora statičke varijable se tretiraju se kao globalne varijable s dosegom koji je ograničen na klasu.

Statičke varijable pripadaju klasi, za razliku od lokalnih varijabli koje pripadaju objektu koji se inicira klasom.

Statičke varijable se mogu inicijalizirati na neku početnu vrijednost. Ta inicijalizacija se mora provesti na globalnoj razini, izvan tijela klase, kako bi se osiguralo da se inicijalizacija vrši samo jedan put. Pri inicijalizaciji obvezno se stavlja oznaka dosega, koja se sastoji od imena klase i dvostruke dvotočke, primjerice, s

```
CL::int n = 0;
```

inicijalizira se varijabla n, koja pripada klasi CL na vrijednost 0. Ovoj varijabi se može pristupiti, dvojako

1. **iz globalnog dosega,**

```
cout << CL::n;
```

2. **ili preko objekta koji je pojavnost klase:**

```
CL obj;
cout << obj.n;
```

Preopterećenje operatora =

U C jeziku operatori imaju jedinstveno značenje i mogu se koristiti samo u izrazima s prostim skalarnim tipovima. U C++ jeziku je moguće promijeniti smisao djelovanja nekih operatora, tako da se mogu primijeniti na objekte i između objekata u različitim izrazima.

Da bi promijenili značaj nekog operatora potrebno je deklarirati člansku funkciju u obliku

```
oznaka_tipa operator oznaka_operatora (parameteri);
```

odnosno funkciju kojoj se ime sastoji od dva dijela:

- **ključne riječi operator i**
- **oznake operatora (+, - , =, ….) kojem se mijenja značenje.**

Mi ćemo kasnije detaljno obraditi kako se vrši ovo preopterećenje operatora (operator overloading).

Jedini operator koji je podržan od kompilatora je operator pridjele vrijednost '='.

```
Counter a, b, c;
a = b;
a = b = c;
```

U izrazu a=b pridjeljuje se vrijednost članskih varijabli objekta b objektu a. Mogli bi zaključiti da bi kompilator to mogao obaviti koristeći kopirni konstruktor. Međutim, u iskazu a=b=c treba obaviti slijedeće radnje: prvo treba kopirati vrijednosti iz objekta c u objekt b, a zatim tako dobivene vrijednosti treba pridjeliti objektu a. Dakle, izraz b=c mora dati neku vrijednost koja će se pridjeliti objektu a. Pošto objekti mogu imati više članova, da bi mogli doprijeti do njih

rezultat izraza b=c mora biti referenca objekta b.

Primjer,

```
class Counter
{

  public:

        Counter& operator=(const Counter& c) {
            m_count = c. m_count;
            m_mod = c. m_count;
            return *this;
        }
```

Parametar funkcije "`operator=`" `()` je objekt s desne strane znaka =. U funkciji se pridjeljuje vrijednost lokalnih varijabli, a funkcija vraća referencu objekta klase Counter. To se ostvaruje naredbom `return *this`, jer kako smo pokazali u poglavlju 3, u radu s objektima dereferencirani pokaživač ima značaj reference.

Iskaz a=b se može zapisati i u funkcijskom obliku:

```
a.operator=(b);
```

a funkcijski oblik iskaza a=b=c je

```
a.operator=(b.operator=(c));
```

Iz ovog se iskaza jasno vidi da rezultat funkcije b.operator=(c) mora biti tipa reference jer se taj rezultat koristi kao argument funkcije a.operator=(..).

Ortodoksni kanonički oblik klase

Često se oblik definicije klase koja sadrži:

- **predodređeni konstruktor,**
- **kopirni konstruktor,**
- **destruktor i**
- **definiciju operatora =**

naziva ortodoksni kanonički oblik klase.

```
class X                          //ortodoksni kanonički oblik
{
public:
    X();                         // predodređeni konstruktor
    X(X &);                      // kopirni konstruktor
    ~X();                        // destruktor
    X& operator =(X &);          // operator =
    ...
}
```

Pokazali smo da kompilator sam generira ove članove ukoliko to nije izvršeno u definiciji klase.

Kada sami moramo definirati kopirni konstruktor i operator = ????

Uzmimo banalni slučaj da se u klasi Točka, koordinate x i y ne bilježe u običnim cjelobrojnom varijablama, već sa za to koriste pokazivači na int. Tada se klasa točka može napisati na slijedeći način:

```
class Tocka{
    int *m_px;
    int *m_py;
public:
   Tocka ()                                  // konstruktor
   {m_px = new int; m_py = new int;} // alocira mem.

    ~ Tocka ()                               // destruktor
    { delete m_px; delete m_py; }     // oslobađa mem.

    Tocka(const Tocka &);                 // kopirni kons.
    Tocka & operator =(const Tocka &); // operator=

    int x() const {return *m_px;}          // pristupnici
    int y() const {return *m_py;}
    void x(int x) {*m_px = x;}             // mutatori
    void y(int y) {*m_py = y;}
};
```

Još treba definirati kopirni konstruktor i operator =. Pogledajmo najprije kako bi to napravio kompilator kada ne bi bio definiran kopirni konstruktor. Tada se podrazumijeva konstruktor:

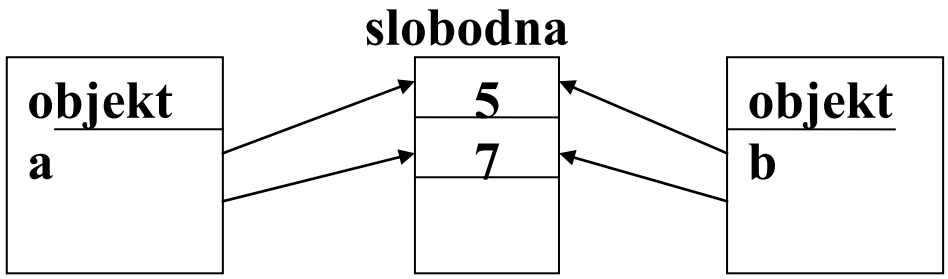
```
Tocka::Tocka(const Tocka& t)
{
    m_px = t.m_px;     // kompilatorska inteligencija
    m_py = t.m_px;     // pridjeljuje pokazivače !!!
}
```

dakle kopira se vrijednost pokazivača na vrijednost koordinata točke.

To znači da bi nakon slijedećih naredbi

```
Tocka a;
a.x(5);
a.y(7);
Tocka b = a;  // poziv kopirnog konstruktora
```

oba objekta koristila isti memorijski prostor za vrijednost koordinata točke. Ovaj način kopiranja objekata naziva s plitko kopiranje (eng. shallow copy).



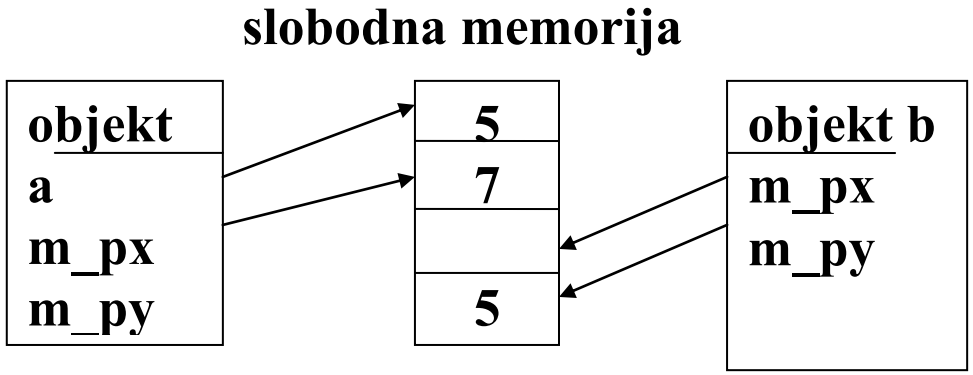
Slika 5.3 Primjer "plitkog" kopiranja alociranih objekata

Problem ovog plitkog načina kopiranja je u tome da, ako jednom objektu promijenimo vrijednost, automatski će se promijeniti vrijednost i drugog objekta. Također je problem ako jedan od objekata iziđe iz dosega, tada se poziva destruktor koji oslobađa zauzetu memoriju. Time i drugi objekt gubi pravo kontrole nad alociranom memorijom.

Problem plitkog kopiranja se može izbjeći ako se implementacija kopirnog konstruktora izvrši tako da se najprije alocira memorija za novi objekt, a zatim se u tu memoriju upišu vrijednosti iz kopirane klase. Primjerice,

```
Tocka::Tocka(const Tocka & t)
{
    m_px = new int;
    m_py = new int;
    *m_px = t.x();
    *m_py = t.x();
}
```

Time se vrši tzv. duboko kopiranje objekata (eng. deep copy).



Slika 5.4 Primjer "dubokog" kopiranja objekata

Efekti plitkog kopiranja nastat će i ako se koristi operator pridjele vrijednosti. Možemo ih izbjeći ako operator pridjele vrijednosti definiramo slijedećom funkcijom:

```
Tocka& Tocka::operator=(const Tocka &desno)
{
    *m_px = desno.x(); // pridjeli vrijednosti
    *m_py = desni.y(); // indirekcijom pokazivača
    return *this;
}
```

Ponekad alocirani objekti neće biti iste veličine. Rješenje za taj slučaju možemo simulirati slijedećom funkcijom:

```
Tocka& Tocka::operator=(const Tocka &desno)
{
    if (this == &desno)// ako se objekt pridjeljuje
                       // sam sebi
       return *this;   // vrati taj objekt, inače

    delete m_px;       // dealociraj memoriju
    delete m_py;       // članskih varijabli

// procijeni potrebnu memoriju iz kopiranog objekta

    m_px  = new int;   // alociraj
    m_py = new int;    // memoriju
    *m_px = desno.x(); // pridjeli vrijednosti
    *m_py = desno.y();
    return *this;
}
```

Prvo se uspoređuje da li se objekt pridjeljuje sam sebi izrazom `this == &desno`. Ako se radi o pridjeli različitog objekta prvo se oslobađa se memorija koju trenutno zauzima objekt s lijeve strane, a zatim se alocira memorija za članske varijable i u njih se upisuje vrijednost članskih varijabli objekta s desne strane znaka jedanakosti.

Pogled unaprijed - naslijeđivanje klasa

Često se spominju dva pojma: objektno temeljeno programiranje i objektno orijentirano programiranje.

Objektno temeljeno programiranje je metoda programiranja kojoj je temeljni princip da se klasa definira kao samostalna programska cjelina. U ovom poglavlju smo pokazali kako se to izvodi u C++ jeziku.

Pod objektno orijentiranim programiranjem (OOP) podrazumijeva se metoda programiranja kojom se definiranje neke klase vrši korištenjem svojstava postojećih klasa. Ovo svojstvo OOP se naziva naslijeđivanje.Također su razvijene tehnike kojima se djelovanja na objekte i između objekata mogu programski unificirati za različite tipove objekata. Ovo svojstvo OOP se naziva polimorfizam.

Tehniku programiranja kojom se realizira OOP detaljno ćemo upoznati u poglavlju 15. Sada ćemo, samo djelomično, pokazati kako se u C++ jeziku realizira naslijeđivanje.

Nasljeđivanje je tehnika kojom se definiranje neke klase vrši korištenjem definicije postojeće klase koja se naziva temeljna. klasa. Tako dobivena klasa se naziva izvedena klasa.
Kada se pomoću izvedene klase deklarira neki objekt, njegove članske funkcije i varijable postaju funkcije i varijable koje su deklarirane u temeljnoj klasi.

Sintaksa deklaracije izvedene klase najčešće se koristi u obliku:

```
class ime_izvedene_klase : public ime_temeljne_klase
{
  // sadrži članove koji su definirani u temeljnoj
klasi
  // definiranje dodatnih članova klase
}
```

Primjerice, neka postoji klasa imena `Temelj`:

```
class Temelj
{
public;
    Temelj() { elem0=0; }
    int elem0;
}
```

Ako pomoću nje deklariramo klasu `Izveden`

```
class Izveden : public Temelj
{
public:
    Izvedena() {elem1 = 0}
    int elem1;
}
```

tada i objekti ove klase imaju dva člana `elem0` i `elem1`.

Objektu `x`, deklariranom s:

```
Izveden x;
```

članskim varijablama pristupamo s:

```
x.elem0  = 5;
x.elem1 = 7;
```

Kažemo da je klasa `Izveden` naslijedila član klase `Temelj`, jer je `elem0` deklariran u klasi `Temelj`.

Ukoliko ne bi koristili nasljeđivanje, klasu `Izveden` bi mogli zapisati u funkcionalno ekvivalentnom obliku:

```
class Izveden // bez naslijeđivanja
{
    public:
         Izvedena() {elem0 = 0; elem1=0;}
         int elem0;
         int elem1;

}
```

U ovoj klasi smo definirali obje članske varijable i njihovu inicijalizaciju u konstruktoru klase.

U verziji s nasljeđivanjem konstruktor je zadužen za inicijalizaciju samo onih varijabli koje su deklarirane u pojedinoj klasi.

U C++ jeziku vrijedi pravilo da se konstruktor ne nasljeđuje, a pri inicijalizaciji objekta izvršavaju se svi konstruktori iz hijerarhije nasljeđivanja – najprije konstruktor temeljne klase, a zatim konstruktor izvedenih klasa. Isto vrijedi i za destruktor, jedino se poziv destruktora vrši obrnutim redoslijedom – najprije se izvršava destruktor izvedene klase, a potom destruktor temeljne klase.

Kasnije će biti detaljno obrađeni ostali aspekti i tehnika objektno orijentiranog programiranja.