Sveučilište u Zagrebu
PMF – Matematički odjel

Objektno programiranje (C++)

Predavanja 02 - Klase (ponavljanje iz RP1)

Vinko Petričević

Što je klasa?

- Svaka klasa u C++-u predstavlja novi tip podatka u programu.
- Instancu klase nazivamo **objekt**. Program se konstruira kao niz interakcija među objektima.
- Na razini sintakse klasa je proširenje strukture iz jezika C. U osnovi klasa je struktura koja pored varijabli članica može sadržavati i funkcije članice. U C++-u i strukture imaju gotovo iste mogućnosti kao klase, pa ću nekada i njih tako nazivati.
- Klasama implementiramo **korisničke tipove podataka** koji modeliraju objekte iz aplikacijske domene.

Cilj:

- omogućiti stvaranje vlastitih tipova sa kojima se radi intuitivno i jednako lako kao sa ugrađenim tipovima (int, char, float)
- Primjeri: string, stack, vector, razlomak, stablo,...
- Općenito, klasa definira novi tip podatka i novi doseg

Osnovni elementi klase

Klasa sadrži četiri vrste članova:

- Varijable članice
- Funkcije članice
- Tipove
- Druge klase (tj. ugniježđene klase)

Funkcije članice se dijele na 4 vrste:

- Obične funkcije članice
- Konstruktori — imaju isto ime kao i klasa i nemaju povratni tip. Konstruiraju objekt tipa klase.
- Destruktor — ima isto ime kao i klasa s tildom ispred imena i nema povratni tip. Automatski se poziva kod destrukcije objekta tipa klase.
- Operatori

Enkapsulacija

Osnovni cilj prilikom stvaranja klase:

- Sakriti informacije o internoj reprezentaciji i implementaciji u privatni dio klase (enkapsulacija - skrivanje), a javno ispoljiti skup operacija koja će se primjenjivati na instancama klase (sučelje)
- Tipična upotreba klase je definiranje apstrakcije podataka
- Apstrakcija je programska tehnika koja se temelji na odvajanju sučelja od implementacija
- Klase nas “prisiljavaju” da se striktno držimo apstrakcije, te da sučelje i implementaciju potpuno odvojimo, čak i kada to zahtijeva “puno više posla”
- Labele pristupa, *public*, *private* i *protected* definiraju prava pristupa elementima klase. Klasa može imati jednu ili više tih labela, a značenje im je sljedeće:
  - Članovi koji su definirani iza *private* labele nisu dohvatljivi kodu izvan klase. To je dio koda koji predstavlja *implementaciju* i koji je skriven od koda koji koristi klasu (default).
  - Članovi koji su definirani iza *public* labele dohvatljivi su u svim dijelovima programa. Oni čine *javno sučelje (u strukturi je ovo defaultni pristup)*.
  - Labela *protected* vezana je uz naslijeđivanje klasa i o njoj će biti više riječi kasnije.

```
class razlomak {
    int m_p, m_q;
public:
    //razlomak(int p=0, int q=1) : m_p(p), m_q(q)
    razlomak(int p, int q) : m_p(p), m_q(q) {
        if (q<0) {m_p=-m_p; m_q=-m_q; }
    }
    razlomak(int p=0) : m_p(p), m_q(1) { }
    void ispis();
};

#include <iostream>
void razlomak::ispis() {
    std::cout << m_p << "/" << m_q;
}

int main() {
    razlomak r(5,10);
    r.ispis();
```

- Funkcije i konstruktore možemo preopterećivati (imati više funkcija istog imena), ako to neće dovesti do dvosmislenosti
- Ukoliko ima bar jedan konstruktor, defaultni neće biti napravljen
- Implementacije funkcija možemo pisati odmah iza definicije funkcije, ili izvan klase (tada trebamo navesti puno ime funkcije koje je *naziv_klase::ime_funkcije*)
- Sa *friend* možemo označiti da neka funkcija ili klasa ima pristup privatnim dijelovima klase koju pišemo. Ako i pišemo odmah implementaciju takve funkcije, ona nije dio klase!

```
class razlomak {
    int m_p, m_q;
    void sredi() {
        if (!m_p) { m_q=1; return; }
        int p = abs(m_p), q = m_q;
        while (q) {
            int t=q; q=p%q; p=t;
        }
        if (p>1){ m_p/=p; m_q/=p; }
    }
public:
    razlomak(int p, int q) : m_p(p), m_q(q) {
        if (q<0) { m_p=-m_p; m_q=-m_q; }
        sredi();
    }
    razlomak(int p=0) : m_p(p),m_q(1) { }
    friend  std::ostream& operator<<(std::ostream& o, const razlomak& r){
        o << r.m_p << "/" << r.m_q;
        return o;
    }
};
...
    std::cout << r << std::endl;
```

Forward deklaracija klase

- Ukoliko trebamo pisati dvije ili više klase, čije se implementacije i definicije međusobno isprepliću, kao i funkcije, i cijelu klasu možemo samo deklarirati, te kasnije pisati samu definiciju.
- Prije definicije klase smijemo koristiti pointere i reference na tu klasu

```
class X; //forward deklaracija klase X

class Y {
    X *px;
...
}; //definicija klase Y
void f(X& x);
class X {
    Y *py;
    // ovdje bi u kodu neke funkcije smjeli pisati f(*this);
}; //definicija klase X

void f(X& x){ ... }
```

Implicitni *this* pokazivač

- Svaka nestatička funkcija klase ustvari dodaje funkciji još jedan skriveni parametar tipa klasa* koji pokazuje na cijelu klasu, koji se može koristiti u funkciji preko rezervirane riječi **this**.

```
    razlomak(int m_p, int m_q) : m_p(m_p), m_q(m_q) {
        if (m_q<0) { this->m_p = -m_p; razlomak::m_q = -m_q; }
    }
```

- Funkcija može i vratiti referencu na samu sebe, što je pogodno ulančavanju funkcija (najčešće operatora)

```
struct X {
    X& f(){ ... return *this; }
    X& g(){ ... return *this; }
};
    X x;
    x.f().g().f();

    razlomak& operator+=(const razlomak&);
```

Reference *(ukratko)*

- Referenca na neki objekt je novo ime za taj objekt. Budući da referenca uvijek mora referirati na neki objekt, ona odmah mora biti inicijalizirana:

```
    int x;
    int &rx = x;
    int &ry; // greška
```

- Najčešće ih koristimo ako želimo da u nekoj funkciji promjene na parametru rezultiraju promjenom varijable koja je poslana kao parametar funkciji

```
void uvecaj(int &a){ ++a; }
...
    x=10; uvecaj(x); cout << x;
```

- ili da nepotrebno ne kopiramo objekt

```
double zbroj(const vector<double> &v) {
    double ret=0;
    for(auto i = v.begin(); i != v.end(); ++i)
        ret += *i;
    return ret;
}
```

Copy-konstruktor *(ukratko)*

- Ukoliko funkciji kao parametar šaljemo objekt bez reference, želimo da nam taj parametar ostane nepromijenjen nakon završetka funkcije. Da bi to omogućio, kompajler kreira novu varijablu s kojom radi u funkciji, pa kôd

```
inline void f(X x){ cout << x; }
...
    f(p);
```

ustvari bude kao

```
{
    X x(p); // ili neprecizno X x = p
    cout << x;
}
```

Točnije, izvrši se copy-konstruktor, čiji je opis X(const X&), a koji treba sadržaj parametra prekopirati u novu varijablu.

- To se događa i na returnu ukoliko je X povratni tip funkcije
- Ukoliko ne radimo nešto specijalno (pointeri, datoteke, …), compiler će sam dodati cctor i sve će biti OK
- Ako ne želimo da se klasa kože kopirati, možemo ga izbaciti sa private ili delete

const članovi klasa

- Osim običnih funkcija, u klasi možemo imati i const-funkcije. One ne smiju mijenjati sadržaj varijabli klase (osim mutable elemenata), i one smiju pozivati samo konstantne funkcije

```
class razlomak {
...
    void ispis() const;
```

- One će biti pozvane ako koristimo const razlomak.
- Možemo imati i konstantnu i nekonstantnu verziju funkcije.
- Ukoliko funkcija poziva neku funkciju kojoj šalje referencu/pointer na sebe, ta funkcija mora primati const-parametar
- Isto tako, ako funkcija treba vratiti referencu ili pokazivač na this, povratak također mora biti konstantan (doduše, greška će biti tek na return, tako da ako slučajno vraćamo neki drugi objekt, sve će biti OK, ali to pak može biti opasno za neke druge stvari)

```
void h(const X* x);
struct X {
     const X& f() const { h(this); return *this; }
```

- Možemo imati i konstatnu i nekonstantnu verziju funkcije (i const se mora pisati i ako odvajamo implementaciju od definicije funkcije)

Static elementi klasa

- Osim svega što smo naveli, funkcije i varijable članice klase mogu biti označene sa static, to znači da su zajednički svim objektima toga tipa
- Static varijable moramo negdje u programu inicijalizirati
- Static funkcije nemaju this u sebi, pa ne znaju kojem konkretnom objektu pripadaju, to jest, smiju pristupati samo statičkim elementima klase
- Možemo im pristupiti direktno preko klase, ili preko nekog objekta toga tipa
- Primjer

```
class struktura {
  // ovo samo spomenuti  int tkoSam;
    string ime;
    static list<struktura*> svi;
    static int kolikoNasIma;
public:
    struktura(string ime) : ime(ime) {tkoSam = kolikoNasIma++;svi.push_back(this);}
    ~struktura() {--kolikoNasIma;svi.remove(this);}
    void ispis() { cout << tkoSam << " " << ime << endl;}

    static int koliko() { return kolikoNasIma;}
    static void ispisiSve() {
        for(list<struktura*>::iterator i = svi.begin(); i != svi.end(); ++i)
            (*i)->ispis();
    }
};

int struktura::kolikoNasIma(0); // ili = 0
list<struktura*> struktura::svi; // ili ()

int main() {
    struktura s("prva");
    {
        struktura s1("druga"); cout << struktura::koliko()<<endl;
        struktura *sp = new struktura("treca"); s1.ispisiSve();
    }
    cout << s.koliko()<<endl;     struktura::ispisiSve();
}
```

Ugniježđene klase

- Klasa unutar sebe može imati i cijelu novu klasu
- Da ne bi bilo zabune, ona nema nikakvu vezu sa podatcima klase, samo je njezino ime klasa::novaKlasa, te može pristupati i privatnim dijelovima glavne klase
- Primjer su iteratori u stl-u

- Klase mogu definirati i nove tipove

```
struct T {
    typedef int mojtip;

    struct podT {
    };
};
```

Implicitne konverzije

- Ukoliko funkcija f prima parametar tipa X, mi joj pošaljemo parametar tipa Y, a postoji konstruktor na X-u koji prima Y, konstruktor će napraviti implicitnu konverziju, tj. F(X(y))

```
struct X {
    /*explicit*/ X(int x) { cout <<"Kreiran X("<<x<<")" << endl;}
};
void f(X x){}
...
    X v = 20;
    f(1000);
```

- Ukoliko to ne želimo, trebamo napisati ključnu riječ explicit
  - npr. ako je X vector stringova, da li uistinu želimo da nam izvrši funkciju f sa vektorom od 1000 praznih stringova, koje će prije poziva funkcije kreirati, a odmah nakon završetka funkcije uništiti?
- Isto se događa i sa operatorima za konverziju (na Y imamo castanje u X)

Operatori (ukratko)

- U C++-u svaki operator je funkcija
- Možemo ih napraviti da primaju bilo što i vraćaju bilo što (bitno je samo da isti broj parametara primaju kao što se od njih očekuje)
- Možemo ih pisati kao elemente klasa (tada je obično prvi operand this),

```
razlomak operator+(const razlomak& d) const
```

a mogu biti i vanjske funkcije

```
/*friend*/ razlomak operator+(const razlomak& l, const razlomak& d)
```

- Nakon bilo koje od ove dvije izvedbe, radit će nam

```
razlomak r1,r2;
...
razlomak r = r1+r2;
```

- A nakon druge izvedbe će raditi i npr. 3+r (ukoliko na razlomku imamo construktor koji prima 3)
- Operatori castanja služe da konvertiramo klasu u neku drugu

```
/*explicit*/ operator double() const { return double(m_p)/m_q; }
```

- Operatore možemo dodati i na ugrađene tipove

```
template <typename T>
ostream& operator<<(ostream& os, const set<T>& v)
```

Templateovi (ukratko)

- U C++-u svaka funkcija, pa i klasa može biti parametrizirana nekim tipom, brojem, templateom, … ili više njih

```
template<class T> T min(T a, T b){ return (a<b?a:b);}

template<class T>
class stog {
    T data[100];
    T pop();
    void push(T x);
};
```

- Kod funkcija compiler uglavnom sam može odrediti tipove parametara, a kod klasa moramo navesti na koji tip mislimo

```
    cout << ::min(10,20) << endl;
    stog<int> ss;
```

- Ako nam se za neki tip ne sviđa implementacija, možemo napraviti specijalizaciju:

```
  /*template<>*/    const char* min(const char* a, const char* b){
        return (strcmp(a,b)<0?a:b);
}
```

Nasljeđivanje (ukratko)

- Slični tipovi mogu imati neka zajednička svojstva i ponašanje

```
struct A {
    int x, y;
    A(int x, int y) : x(x), y(y) {}
};
void ispisi(A a){ cout << a.x << " " << a.y << endl; }

struct B : /*public*/ A {
    B(int x, int y) : A(x,y){}
};
...
    B b(1,2); ispisi(b);
```

- A sa druge strane, mogu se različito ponašati

```
    virtual void A::ispisi() { cout << "A" << endl; }
void ispisi1(A& a){ a.ispisi(); }
    void B::ispisi() { cout <<  "B" << endl; }

    ispisi1(b);
```