Sveučilište u Zagrebu
PMF – Matematički odjel

Objektno programiranje (C++)

Vježbe 01 – Klase

Vinko Petričević

Klase

- 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 podataka i novi doseg

Klase

- Osnovni cilj prilikom stvaranja klase:
  - Sakriti informacije o internoj reprezentaciji i implementaciji u privatni dio klase (enkapsulacija - skrivanje)
  - 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 odvojamo, čak i kada to zahtijeva “puno više posla”

Klase

- Oznake pristupa (specifikatori pristupa):
  - `public`
    - Članovi klase koji su dostupni svim dijelovima programa (sučelje)
  - `private`
    - Članovi klase koji nisu dostupni dijelovima programa koji samo koriste klasu (implementacija)
  - `protected`
    - ...kasnije...
- Ne moraju svi tipovi biti apstraktni
  - Primjer: `pair`
    - dopušta direktan pristup članovima `first` i `second`
    - Skrivanje članova bi samo zakompliciralo upotrebu para

Definicija klase

- Deklaracija varijabli članica u klasi

```
#include <string>
class Screen {
  string _screen; // string( _height * _width )
  std::string::size_type _cursor; // current position
  short _height; // number of Screen rows
  short _width; // number of Screen columns
};
```

- Varijable članice klase ne mogu biti inicijalizirane u tijelu klase
  - Inicijalizacija se obavlja u konstruktoru
- U klasu možemo ubaciti i `typedef`

```
typedef std::string::size_type index;
index _cursor;
```

Definicija klase

- Deklaracija funkcija članica u klasi

```
class Screen {
public:
   void home();
   void move( int, int );
   char get();
   char get( int, int );
   bool checkRange( int, int );
};
```

- Funkcije članice klase razlikuju se od “običnih” funkcija po sljedećim svojstvima:
  - Imaju potpun pristup privatnom i javnom dijelu klase
  - Pripadaju dosegu klase

Definicija klase

- U tijelu klase mogu se nalaziti i definicije funkcija članica

```
class Screen {
public:
   void home() { _cursor = 0; }
   char get() { return _screen[_cursor]; }
};
```

- Funkcije članice klase mogu biti i preopterećene
  - Preopterećene funkcije su funkcije koje imaju isto ime i istu povratnu vrijednost, ali različitu listu parametara

```
class Screen {
public:
   char get() { return _screen[_cursor]; }
   char get( int, int );
};
```

Definicija klase

- Specifikatori pristupa: `public, private, protected`

```
class Screen {
public:
  void home();
  void move( int, int );
  char get();
  char get( int, int );
  bool checkRange( int, int );
private:
  short _height = 24;
  short _width = 80;
  string _screen;
  string::size_type _cursor;
};
```

Definicija klase

- Funkcije članice koje su definirane unutar klase, automatski su označene kao `inline`
  - Kompajler će pokušati cijelo tijelo funkcije zalijepiti na mjesto poziva
  - Tijelo inline funkcije mora biti vidljivo u svakoj datoteci koja koristi inline funkciju
    - Zato je dobro da inline funkcije članice definirati u zaglavlju u kojem je deklarirana i klasa

Definicija i deklaracija klase

- `class Screen; //deklaracija klase (nepotpun tip)`
- ```
  class Screen {
  public:
     short _height, _width;
     //...
  }; //definicija klase
  ```

- Nepotpun tip smijemo koristiti samo za članove koji su pokazivači ili reference
  - ```
    class LinkScreen {
      Screen window;
      LinkScreen *next;
    };
    ```

Definicija i deklaracija klase

- Zadatak: definirajte par klasa X i Y, gdje X sadrži pokazivač na Y, a Y sadrži pokazivač na X.

```
class X; //forward deklaracija klase X
class Y {
  X *px;
}; //definicija klase Y
class X {
  Y *py;
}; //definicija klase X
```

Objekti klase

- Kada definiramo klasu, definirali smo “samo” tip podataka, ne i objekt danog tipa
- Definicija klase ne uzrokuje alokaciju memorije
- Alokacija se događa kada definiramo objekt neke klase
  - ```
    class Screen {
      // ...
    }; //definiranje novog tipa; nema alokacije memorije
    ```
  - ```
    Screen s; //objekt s tipa Screen; alocira se memorija
    ```

Implicitni `this` pokazivač

- Funkcije članice klase imaju implicitno jedan dodatni "skriveni" parametar: `this` pokazivač
- `this` pokazivač je pokazivač na objekt koji je pozvao funkciju članicu klase
- Ne moramo (ne smijemo) sami definirati this pokazivač – kompajler ga implicitno automatski dodaje u svaku funkciju članicu (ne statičku)

Implicitni `this` pokazivač

- U tijelu funkcija članica se ne moramo uvijek pozivati na this pokazivač
  - ```
    class Screen {
    public:
     short _height, _width;
       void f() {
            this->_height = 5;  // _height = 5;
       }
    }
    ```
- This pokazivač obično koristimo kada trebamo dohvatiti cijeli objekt (ili referencu ili pokazivač na objekt), a ne samo elemente objekta
    ```

Implicitni `this` pokazivač

- This pokazivač obično koristimo kada trebamo dohvatiti cijeli objekt (ili referencu ili pokazivač na objekt), a ne samo elemente objekta
  - `myScreen.move(4,0); //pomakni kursor`
    `myScreen.set('#'); //zapiši '#' na poziciju kursora`
- Umjesto gornje dvije naredbe, htjeli bismo pisati
  - `myScreen.move(4,0).set('#');`
- Slična je stvar sa
  - `a=b=c;`

Implicitni `this` pokazivač

- Rješenje:
  - ```
    class Screen {
      public:
           Screen& move(index r, index c);
           Screen& set(char);
           Screen& set(index, index, char);
    };
    ```
    ```

Implicitni `this` pokazivač

- Rješenje:
  -
```
Screen& Screen::set(char c) {
    contents[cursor] = c;
    return *this;
}

Screen& Screen::move(index r, index c) {
    index row = r * width; // row location
    cursor = row + c;
    return *this;
}
```

`const` funkcije članice

```
class Screen {
public:
  char get(); //dohvati element sa pozicije kursora
}
```

- Problem:
  - ```
    const Screen blankScr;
    char c = blankScr.get(); // Greška!
    ```
  - Želimo dohvatiti znak sa pozicije kursora u “konstantnom ekranu” (ima smisla)
- Funkcije članice klase koje ne modificiraju objekt moguće je deklarirati kao const funkcije
    ```

`const` funkcije članice

```
class Screen {
public:
   char get(); //dohvati element sa pozicije kursora
}
```

- Problem:
  - ```
    const Screen blankScr;
    char c = blankScr.get(); // Greška!
    ```
  - Želimo dohvatiti znak sa pozicije kursora u “konstantnom ekranu” (ima smisla)
- Funkcije članice klase koje ne modificiraju objekt moguće je deklarirati kao const funkcije

  ```
  char get() const;
  ```
  ```

`const` funkcije članice

- Const funkcija članica ne može promijeniti članove objekta koji ju je pozvao
- Const funkcija članica može biti pozvana i od const objekta, dok ne-const funkciju članicu ne možemo pozvati iz const-objekta
  - Zato je nužno funkcije članice koje ne mjenjaju objekt označiti sa const
- Ilegalno je kao const deklarirati funkciju članicu koja modificira član klase

```
class Screen {
public:
  char greska() {_cursor = 5;};
     // dohvati element sa pozicije kursora
}
```

`const` funkcije članice

- Const funkcija članica može biti preopterećena ne-const funkcijom članicom
  - Konstantni objekti će tada pozivati konstantnu varijantu, a ne-konstantni objekti ne-konstantnu varijantu

```
class Screen {
public:
  char get(int, int);
     // dohvati element sa određene pozicije

  char get(int, int) const;
}
```

`mutable` varijable članice

```
const Screen cs(5,5);
cs.move(3,4);
char ch = cs.get();
```

- Ovaj kod predstavlja grešku jer pokušavamo pozvati ne-const funkciju move na konstantnom objektu
- Problem:
  - želimo da cs bude konstantan
  - move ne može biti const funkcija jer mijenja vrijednost varijable `_cursor`
  - Ima smisla da pomičemo kursor (move) po konstantnom objektu
- Rješenje: upotreba mutable

`mutable` varijable članice

- Da bismo dozvolili da varijabla članica bude modificirana i u slučaju const objekta, trebamo ju deklarirati kao mutable

```
mutable string::size_type _cursor;
```

- Sada move može biti deklarirana kao const (i samim time pozvana od const objekta) jer ne mijenja vrijednost niti jedne ne-mutable varijable članice

Doseg klase

- što ako u funkciji koristimo varijable istog imena kao elementi struct-a? Što ako imamo globalne varijable istog imena?

- ```
  int x=3;
  class test {
      int x;                  // negdje u programu x=5
      void ispis() {
          int x=7;
          cout << x;          // ispise 7
          cout << test::x;    // ispise 5
          cout << ::x;        // ispise 3
      }
  };
  ```
  ```

Konstruktori

- Specijalne funkcije članice koje se izvršavaju prilikom stvaranja novog objekta
- Posao konstruktora je da varijable članice klase dobiju smislene početne vrijednosti
- Kostruktori imaju isto ime kao i ime klase
- Nemaju povratni tip (pa čak ni void)
- Mogu se preopteretiti

Konstruktori

```
class Screen {
public:
   Screen(int hi=8, int wid=40, char bg='#'); //deklaracija
}
```

- Implementacija konstruktora unutar klase:
  - ```
    Screen(int hi=8, int wid=40, char bg='#') {
       _cursor=0; _height=hi; _width=wid;
       _screen = string(hi*wid, bg);
    };
    ```
- Implementacija konstruktora unutar klase inicijalizacijskom listom:
  - ```
    Screen(int hi=8, int wid=40, char bg='#') :
       _cursor(0), _height(hi), _width(wid),
       _screen(string(hi*wid, bg))
    { };
    ```

Konstruktori

```
class Screen {
public:
   Screen(int hi=8, int wid=40, char bg='#');
   //deklaracija
}
```

- Implementacija konstruktora izvan klase:
  - ```
    Screen::Screen(int hi=8, int wid=40, char bg='#') {
       _cursor=0; _height=hi; _width=wid;
       _screen = string(hi*wid, bg);
    };
    ```
    ```

Inicijalizacijska lista

- Inicijalizacijska lista je ponekad nužna:
- ```
  class ConstRef {
    public:
       ConstRef(int ii);
    private:
       int i;
       const int ci;
       int &ri;
  };
  ```
- ```
  ConstRef::ConstRef(int ii) {
    i = ii;  // ok
    ci = ii; // const sa lijeve strane
    ri = i;  // naknadno pridruživanje referenci
  }
  ```

Default konstruktor

- Konstruktor koji može biti pozvan bez navođenja liste parametara
- Kompajler sam sintetizira default konstruktor samo ako nismo napisali niti jedan drugi konstruktor
  - Inače se pretpostavlja da trebamo sami napisati i default konstruktor

Implicitne konverzije

- Konstruktori koji imaju točno jedan parametar, mogu se koristiti za implicitnu konverziju
- Primjena konstruktora za implicitnu konverziju bit će onemogućena ako navedemo ključnu riječ explicit prije danog konstruktora

```
#include<string>
#include<iostream>
using namespace std;

class tekst {
private:
        string s;
public:
        tekst():s(string()){};
        tekst(const tekst &);
        tekst(const string &r):s(r){};
        explicit tekst(int i);
        string data(){return s;};
};

tekst::tekst(int i) {    s = "ovo je integer";}

void ispisi(tekst t) {   cout << t.data() << endl;}

int main() {
    tekst blabla("xyz"); ispisi(blabla);
    string str("string"); ispisi(str); // zašto ovo radi?
    int i=5; ispisi(i); // zašto ovo ne radi?
    system("pause");
}
```

friend

- Neke funkcije, tj. klase možemo označiti “prijateljskima”, te im dozvoliti pristup i privatnom dijelu klase

```
class X {
    friend class Y;
    friend void f() {
/* ok to define friend function in the class body */
  }

    friend int g(int, ...);
};
```

```
int g(int a, ...);
```

```
class Y { ... };
```

`static` članovi klase – varijable

- Svi objekti neke klase dijele jednu static varijablu članicu klase, tj. ta varijabla je zajednička za sve objekte toga tipa
- ne zauzimaju prostor na heapu/stacku
- moraju biti inicijalizirani izvan struktura
  - ```
    struct MyStruct {
        static int brojZivihStruktura;
        MyStruct() { brojZivihStruktura++; }
        ~MyStruct() { brojZivihStruktura--; }
    };
    int MyStruct::brojZivihStruktura = 0;
    ```
  - ```
    int main() {
        MyStruct a, b, c;
        cout << MyStruct::brojZivihStruktura;
        return 0;
    }
    ```
- možemo im pristupati i ako nemamo niti jednu varijablu toga tipa

`static` članovi klase – funkcije

- static funkcije mogu pristupati samo static članovima strukture (nemaju this)
  - ```
    struct MyStruct{
        static int brojStruktura;
        int brojac;
        MyStruct() { brojac = brojStruktura++; }
        static int broj() {
            return brojac; // greska, brojac nije static
            return brojStruktura; // OK
        }
    };
    ```
- možemo ih pozivati i direktno (bez varijable)
  - ```
    int i = MyStruct::broj();
    // MyStruct s; int i=s.broj();
    ```
- pogodne su za rekurzije, jer 'troše' manje sistemskog stacka

Ugnježđene klase

- ponekad struktura članica ima smisla samo unutar veće strukture

```
struct Automobil {
    struct Motor {
        int snaga;
        double obujam;
        Motor (int s, double o)
            { snaga = s; obujam = o; }
    };

    Motor m;
    int brojVrata;
    Automobil() : m(90, 1.4) { ... }
    Automobil(int s, double o) : m(s, o) {...}
};
```

Ugnježđene klase

- možemo svejedno deklarirati i varijable tipa Motor:
  - ```
    Automobil yugo(45, 1.2);
    Motor tdi(120, 1.9); // krivo!

    Automobil::Motor tdi(120, 1.9); // OK

    yugo.m = turboDiesel;

    cout << yugo.m.snaga;
    cout << yugo.brojVrata;
    ```
    ```