Sveučilište u Zagrebu
PMF – Matematički odjel

Objektno programiranje (C++)

Vježbe 04 – Kontrola kopiranja

Vinko Petričević

Klase

- Prilikom kreiranja klase, kompajler sam definira neke operacije na tom objektu
  - copy-konstruktor
  - destruktor
  - operator pridruživanja
- Često će nam defaultno ponašanje ovih operacija prouzročiti greške (npr. ukoliko naša klasa dinamičku alokaciju memorije)
- U daljnjem će nam C označavati klasu/struktoru koju kreiramo

Klase – copy-kostruktor

- copy-construktor je specijalni konstruktor koji prima jedan parametar tipa const C&
- eksplicitno se koristi kada definiramo novi objekt i inicijaliziramo ga objektom istog tipa
- implicitno se koristi kada
  - funkciji (operaciji) šaljemo objekt tipa C
  - funkcija vraća objekt tipa C
- defaultno se pozove copy-konstruktor svih elemenata klase

Klase – copy-kostruktor

```
class C {
  public:
        C(const C& x);
  };
C a;
```

- eksplicitno korištenje:

```
C b(a);
C x = a;
```

- implicitno korištenje:
  - `void f(C x);`

```
...
f(a);
```

  - `C g(...) {`

```
  C ret
  ...
  return ret;
}
```

Klase – copy-kostruktor

```
class D {
  C x;
public:
  D(...) {}; // poziva se x()
  D(...) : x(a) {};

  D(const D& d) : x(d.x) {}
      // ovo kompajler defaultno
      // napravi
};
```

Klase – destruktor

- destruktor se automatski poziva
  - na završetku dosega u kojem je kreiran
  - kada pozovemo delete na objekt koji je dinamički kreiran
- destruktor bi trebao osloboditi resurse koje je varijabla zauzela (prilikom kreiranja ili za vrijeme života)
- nakon završetka izvršavanja destruktora, kompajler automatski izvrši destruktore svih nestatičkih elemenata klase

Klase – destruktor

```
class C {
  public:
      ~C();
};
```

- `{ C x`
  `...`
  `}`
- `void f(C x){`
  `}`
- `C *pC = new C;`
  `...`
  `delete pC;`

Klase – destruktor

```
class C {
      D a,b
  public:
      ~C(){
          ...
      } // tu se pozovu automatski
        // pozovu destruktori
        // od b i a
};
```

Klase – destruktor

```
C *pC = new C[10];
...
delete[] pC;
  // destruktor se poziva na svaki
  // element polja
```

Klase – operator pridruživanja

- operator pridruživanja bi trebao osloboditi zauzete resurse, te zauzeti nove koji su jednaki onima kojima ih pridružujemo
- operator pridruživanja može imati više preopterećenih oblika
- kompajler automatski kreira operator pridruživanja koji s desne strane ima objekt tipa const C& (ako ga sami ne kreiramo) koji izvrši operator= na svaki element klase

Klase – operator pridruživanja

```
class C {
  public:
      C& operator=(const C& x);
};
```

- ```
  C a,b;
  a = b;
  ```
  ```

Klase – operator pridruživanja

```
class C {
      D a,b;
  public:
      C& operator=(const C& x) {
          a = x.a;
          b = x.b;
          reutrn *this;
      } // ovakvu stvar kompajler po
        // defaultu napravi
};
```

Klase

- sve dobro radi ako ne korisitmo dinamičku alokaciju memorije (objekte ne kreiramo dinamički)
- ukoliko u klasi dinamički kreiramo objekte, trebamo paziti na svaku od ove tri operacije
  - ako ne koristimo destruktore, resursi neće biti oslobođeni
  - ako ne pazimo na copy-konstruktor/pridruživanje, ti nam operatori možda neće raditi dobro, ili će izazvati greške zbog pokušaja dvostruke destrukcije
- ponekad ćemo možda i željeti da se klasa ponaša “normalno” – npr. smart-pointeri

Primjer

- klasa String koja se ponaša malo pametnije od char*

```
class String {
        char* m_chars;
  public:
        String(const char* c = 0) {
              int len = 0;
              if (c) len = strlen(c);
              if (len) {
                    m_chars = new char[len+1];
                    strcpy(m_chars, c);
              } else m_chars=0;
        }
        const char* data() {
              return m_chars;
        }
};
```

Primjer

- kompajler je automatski dodao:

```
class String {
  public:
      ~String() {}
      String(const String& s) :
          m_chars(s.m_chars) {
// m_chars = s.m_chars
      }
      String& operator=(const String& s) {
          m_chars = s.m_chars;
          return *this;
      }
};
```

Primjer

```
{
        String s("123");
}
```

- međutim, klasa ne oslobodi zauzetu memoriju, pa napišimo destruktor

```
class String {
  public:
        ...
        ~String() {
              if (m_chars) delete[] m_chars;
        }
};
```

Primjer

- sada se memorija oslobađa dobro, ali neće dobro raditi kopiranje, ni pridruživanje
  - `String a("OP"), b(a); // b.data()[1]='5';`
  - `String c = a; c.data()[1]='6';`
  - `String d; d = a; d.data()[1]='7';`
  - `void f(String x) { d.data()[1]='8'; } f(a); // ovdje imamo 2 greške`
  - `String g() { String ret("123");return ret; } a = g();`

`} // ovdje ćemo dobiti više grešaka`

Primjer

- sređivanje copy-konstruktora:

```
class String {
  public:
        String(const String& s)
// : m_chars(0) // ovo kompajler stavi
  {
          int len = 0;
          if (s.m_chars)
             len = strlen(s.m_chars);
          if (len) {
            m_chars = new char[len+1];
            strcpy(m_chars, s.m_chars);
          }
          else m_chars=0;
        }
  };
```

Primjer

- sređivanje operator=:

```
class String {
  public:
      String& operator=(const String& s) {
        if (&s == this) return s;
        if (m_chars) delete[] m_chars;
        int len = 0;
        if (s.m_chars)
           len = strlen(s.m_chars);
        if (len) {
          m_chars = new char[len+1];
          strcpy(m_chars, s.m_chars);
        }
        else m_chars=0;
        return *this;
      }
};
```

Klase

- copy-konstruktor dakle treba kopirati sadržaj memorije koju zauzima parametar u trenutni objekt
- operator pridruživanja treba osloboditi zauzetu memoriju, te napraviti slično kao i copy-konstruktor
- zbog toga se razlikuje:
  - ```
    C a;
    a=b;
    ```
  - ```
    C a=b; // isto kao i C a(b);
    ```

Klase – dinamička alokacija

- new
- svakom new operatoru mora odgovarati točno jedan delete operator
- moguće greške
  - ako zaboravimo delete (exception, return, ...) resursi će ostati zauzeti (memorija, file-handle...)
  - ako više puta pokušamo pozvati delete, dobit ćemo greške

Klase – auto_ptr

- predložak `auto_ptr` nam može pojednostaviti korištenje dinamički alociranih klasa
- ponaša se isto kao i obični pointer

```
#include <memory>
  ...
{
  auto_ptr<klasa> pi ( new klasa(...) );
  *pi
  pi->...
}       // ne treba pisati delete pi;
```

- `operator=` i copy-constructor ne radi baš najbolje (stara klasa/desni operand su poslije neupotrebljivi)
- nije ga dobro koristiti kao element nekog kontejnera

Klase – auto_ptr

```
{
  auto_ptr<char> pc(new char[4]);
  strcpy(&*pc, “OP");
  cout<<&*pc<<endl;

  auto_ptr<char> pc1(pc);
       // ovdje više pc ne radi

  pc = pc1;
       // sada radi pc, ali
       // ne radi pc1
} // oslobođena je memorija zauzeta sa new
```

Klase – kopiranje klasa

- copy-construktor i operator= mogu biti dosta spori ako je klasa velika
- ponekad nam nije bitno da se cijela klasa kopira
- kad god možemo, dobro je klasu slati preko (const) reference, jer se tada ne vrši kopiranje
- moguća poboljšanja su smart-pointeri – koristimo ih kao što koristimo poinere, a ponašaju se nešto pametnije.

Klase – brojanje referenci

- Da bi se ubrzalo kopiranje memorije, pametno bi bilo koristiti brojače referenci, umjesto stvarnog kopiranja
- slično radi i klasa string

```
string s(“Pozdrav");
string t = s;
// t and s pokazuju na isti buffer znakova
t += " ovdje";
// kreira se novi buffer za t,
// smanji se brojač od s,
// kopira se sadržaj od t, te se dodaje
// riječ “ ovdje”. Tako da s ostaje
// nepromijenjen kao što smo i željeli
```

Klase – smart-pointeri

```
class SmartString {
  struct StringRep {
    int m_refCount;
    char* m_data;
    StringRep(const char* data)
      : m_refCount(1) {
        m_data = new char[strlen(data)+1];
        strcpy(m_data, data);
    }
    ~StringRep() {delete m_data; }
  };
  StringRep* rep;
public:
```

Klase – smart-pointeri

```
SmartString(const char* data)
   : rep(new StringRep(data)) {}

SmartString(const SmartString& ss) {
   rep = ss.rep;
   ++rep->m_refCount;
}

~SmartString() {
   if (!--rep->m_refCount) delete rep;
}
```

Klase – smart-pointeri

- pridruživanje bi bilo:

```
SmartString& operator=(const SmartString& ss){
  ++ss.rep->m_refCount;
  if (!--rep->m_refCount) delete rep;
  rep=ss.rep;
  return *this;
}
```

- kopiranje sada radi brže, ali da bi se klasa ponašala kao pointer, trebaju nam još dva operatora:

```
char& operator*() { return *rep->m_data; }
char* operator->(){ return rep->m_data; }
```

- ovaj drugi nećemo obraditi, jer kod chara nema baš smisla

Klase – smart-pointeri

- za string bi nam dobro došao i:

```
char& operator[](int i){
    return rep->m_data[i];
}
```

- klasa radi brzo, ali ne baš najbolje:

```
int main() {
  SmartString a("OP"), b(a);
  (*b)='R'; // ili b[0]='R';
  cout<<&*a<<endl;
  cout<<&*b<<endl;
  return 0;
}
```

Klase – smart-pointeri

- treba razdvojiti operator koji su konstantni od onih koji nisu:

```
const char& operator*() const {
  return *rep->m_data;
}

char& operator*(){
   if (rep->m_refCount==1)
        return *rep->m_data;
   --rep->m_refCount;
   rep = new StringRep(rep->m_data);
   return *rep->m_data;
}
```

Klase – smart-pointeri

- treba razdvojiti operator koji su konstantni od onih koji nisu:

```
const char& operator[](int i) const {
  return rep->m_data[i];
}

char& operator[](int i){
   if (rep->m_refCount==1)
        return rep->m_data[i];
   --rep->m_refCount;
   rep = new StringRep(rep->m_data);
   return rep->m_data[i];
}
```

Klase – smart-pointeri

- const-operatori se pozivaju na objektima koji su konstantni, a ne-const na ostalima
- općenito, za funkcije koje su konstantne, ne treba ništa raditi, a one koje nisu, potrebno je napraviti kopiju objekta
- tako možemo ubrzati rad sa velikim klasama, npr. ako imamo nekoliko velikih objekata, možda će se tijekom izvršavanja neke funkcije promjene dešavati samo u nekom elementu klase, pa ju nema potrebe kopirati cijelu
- kod malih objekata to samo usporava stvar