Računske vježbe 4

Programiranje II

Realizovati klasu `Worker` koja ima četiri podatka člana i to:

- koeficijent za platu (cijeli broj),
- identifikacioni broj radnika (pokazivač na cijeli broj),
- ime radnika (pokazivač na niz karaktera),
- javni statički podatak koji će služiti za brojanje ukupnog broja radnika (objekata date klase).

Klasa posjeduje konstruktor, destruktor i konstruktor kopije, kao i funkcije članice za pristup podacima članovima radi očitavanja i izmjene. Potrebno je realizovati i funkciju koja od dva radnika vraća ime radnika sa većim koeficijentom za platu.

```
#include <iostream>
#include <cstring>

using namespace std;

class Worker
{
private:
    int coeff;
    int *id;
    char *name;
public:
    Worker() // podrazumijevani konstruktor
    {
        id = 0;
        name = 0;
        number++;
    }
    Worker(int, int, char *);
    Worker(const Worker &);
    ~Worker();

    int getCoeff() const {return coeff;}
    int getId() const {return *id;}
    char * getName() const {return name;}

    void setCoeff(int a) {coeff = a;}
    void setId(int a)
    {
        if (id == 0) id = new int(a);
        else *id = a;
    }
    void setName(char *_name)
```

```
34      {
35          /*
36          Memoriju oslobadjamo ukoliko je zauzeta kako ne bi doslo do njenog curenja,
37          a to ce se desiti jer bismo u suprotnom naredbom new alocirali novu memoriju
38          te izgubili adresu onog dijela memorije na koji je pokazivac name prije toga
39          pokazivao. Ne moramo provjeravati da li je name==0 prije brisanja jer to
40          radi delete za nas.
41          */
42          delete [] name;
43          name = new char[strlen(_name) + 1];
44          strcpy(name, _name);
45      }
46
47      char * higherCoeff(Worker);
48
49      static int number;
50  };
51
52  int Worker::number = 0;
53
54  Worker::Worker(int _coeff, int _id, char *_name) : coeff(_coeff)
55  {
56      id = new int(_id); //*id=_id;
57      name = new char[strlen(_name) + 1];
58      strcpy(name, _name);
59      number++;
60  }
61  Worker::Worker(const Worker &worker) : coeff(worker.coeff)
62  {
63      id = new int(*worker.id); //*id=*worker.id;
64      name = new char[strlen(worker.name) + 1];
65      strcpy(name, worker.name);
66      number++;
67  }
68  Worker::~Worker()
69  {
70      delete id; // oslobadjamo dinamicki zauzetu memoriju
71      id = 0; // pokazivac sada ni na sta ne pokazuje
72      delete [] name; // niz karaktera, koristimo []!!
73      name = 0;
74      number--;
75  }
76  char * Worker::higherCoeff(Worker worker)
77  {
78      if(coeff >= worker.coeff)
79          return name;
80      else
81          return worker.name;
82  }
83  int main()
84  {
85      int coeff, id;
86      char name[20];
87
88      cout << "Unesite podatke za prvog radnika" << endl;
89      cin>> coeff >> id >> name;
90      Worker worker1(coeff, id, name);
91
92      cout << "Unesite podatke za drugog radnika" << endl;
```

```
    cin >> coeff >> id >> name;
    Worker worker2(coeff, id, name);

    //poziv konstruktora kopije
    Worker worker3(worker2);
    cout << "Sada radnik 3 ima ime " << worker3.getName() << endl;
    cout << "Vise je placen radnik " << worker1.higherCoeff(worker2) << endl;
    cout << "Ukupno je kreirano " << Worker::number << " radnika." << endl;
}
```

U slučaju kada klasa ima pokazivač kao podatak član, prilikom destrukcije objekta doći će do dealokacije podatka člana, ali ne i onoga na šta on pokazuje. Mi smo preko tog pokazivača recimo mogli da zauzmemo memoriju za cijeli broj ili niz cijelih brojeva koja neće biti oslobođena. Zbog toga je, kada god radimo sa podacima članovima koji su pokazivači, neophodno da realizujemo destruktor kako bismo oslobodili memoriju. Takođe, neophodan nam je i podrazumijevani konstruktor koji postavlja pokazivač da ni na šta ne pokazuje, odnosno da pokazuje na `0`. Još jednom naglašavamo, programer brine o dinamički zauzetoj memoriji, a OS o statički zauzetoj memoriji. Drugim riječima, upotrebom naredbe `delete` mi oslobađamo onu memoriju na koju pokazuje pokazivač, dok memoriju za sam pokazivač implicitno oslobađa destruktor bez naše kontrole. U kodu:

```
class X
{
private:
    int *p;
public:
    X()
    {
        p = 0; // ni na sta ne pokazuje
    }
    ~X() // sa ~ naglasavamo da se radi o destruktoru
    {
        delete p; // brisemo ono na sta pokazivac pokazuje
        p = 0; // ni na sta ne pokazuje
    }
};
```

U ovom zadatku neophodno je realizovati i konstruktor kopije. Namjena konstruktora kopije jeste da novostvoreni objekat inicijalizuje kopijom sadržaja drugog objekta istog tipa. Parametar konstruktora kopije mora da bude **referenca** na primjerak istog tipa zato što konstruktor ne može da ima parametar tipa svoje klase. Kao i za podrazumijevani konstruktor, tako i za konstruktor kopije postoji implicitna definicija. Ovako definisani konstruktor kopira sva polja izvorišnog objekta u novostvoreni objekat. Ukoliko su neka od polja tipa (drugih) klasa, za njihovo kopiranje pozivaće se kopirajući konstruktori tih klasa. Ako su neka od polja pokazivači, kopiraće se samo vrijednosti tih pokazivača, a neće se praviti kopije pokazivanih podobjekata. Ovakva kopija naziva se **plitka kopija** jer nije nezavisna od originala pa se obično želi izbjeći jer njome postižemo da polja dva objekta pokazuju na istu memorijsku lokaciju. Kopija koja objekte čini nezavisnim naziva se **duboka kopija**. Iz gore navedenog jasno je da implicitno definisani konstruktor kopije ne može riješiti ovaj problem pa se on mora preklopiti. Uočite kako smo, upravo radi prevazilaženja ovog problema, koristili funkciju `strcpy()`.
Takođe, uočite kako smo u konstruktoru kopije upotrijebili ključnu riječ `const`. Referenca je drugo ime za neki memorijski objekat odnosno njegov alijas. Referenca ne može da promijeni objekat na koji se odnosi, ali se pomoću nje može mijenjati sadržaj referenciranog objekta. Kako se referenca sama po sebi ne može naknadno mijenjati, konstantna referenca nam garantuje da ono na šta ona referencira postaje konstantno odnosno `read-only`. Zašto uopšte koristimo reference? Zato što kada prosljeđujemo argument po referenci ne pravi se kopija toga objekta kao u slučaju prosljeđivanja po vrijednosti. Time se u slučaju kompleksnijih klasa dobija značajna ušteda u vremenu.