Lösning, Tentamen i C++

2016–01–11

1. a) Det går bara att skapa objekt av klassen `Foo` *på heapen*, och inte på stacken, som `static`-objekt, mm.

b) Eftersom konstruktorerna är privata behöver man deklarera `friend class Bar;`

c) Gör om medlemmen till en `Foo`-pekare, och lägg till en destruktor:

```
class Bar{
public:
    Bar(int i=0) :f{Foo::create(i)} {}
    int compute() const {return f->compute();}
    ~Bar() {delete f;}
private:
    Foo* f;
};
```

2. a) En minnesläcka är att allokera minne (med `new`) som aldrig avallokeras (med `delete`).

b) Konstruktorn i klassen `Foo` är inte virtuell. Eftersom vektorn `v` innehåller pekare till `Foo`-objekt, kommer bara destruktorn i `Foo`, men inte destruktorn i `Bar` att köras när programmet gör `delete` på objekten som pekarna i `v` pekar på. Alltså kommer c-strängarna i `Bar`-objekten inte att avallokeras.

3. Variabeln `a` är pekare och `b` är en referens, så där fungerar tilldelningarna som förväntat men `c` är ett `Bar`-objekt.
De två första funktionerna referensanropas så där fås det polymorfa beteendet.
I `print3(Foo)` sker en kopiering och kopian blir ett `Foo`-objekt.

```
Bar
Qux
Bar

Bar
Qux
Bar

Foo
Foo
Foo
```

4. a) Funktionen `comp` i de båda klasserna ska jämföra rätt medlem i objekten med *mindre än*. Notera att de måste ha en konstruktor som anropar konstruktorn i `Comparator` för att ge `ascending` rätt värde.

```
struct NameComparator :Comparator{
  NameComparator(bool asc=true) :Comparator(asc) {}
  bool comp(const Person& a, const Person& b) const override;
};

bool NameComparator::comp(const Person& a, const Person& b) const
{
  if(a.get_last_name() == b.get_last_name()){
      return a.get_first_name() < b.get_first_name();
  } else {
      return a.get_last_name() < b.get_last_name();
  }
}

struct DateComparator :Comparator{
    DateComparator(bool asc=true) :Comparator(asc) {}
    bool comp(const Person& a, const Person& b) const override {
        return a.get_birthdate()  < b.get_birthdate();}
};
```

b) Överlagra `operator<` (kan även göras analogt som medlemsfunktion):

```
bool operator<(const Person& a, const Person& b)
{
    static NameComparator cmp;
    return cmp(a, b);
}
```

c) Om vi implementerar `operator==` för två `Person` kan vi utnyttja `operator==` för två `std::vector`.

```
bool operator==(const Person& a, const Person& b)
{
    return (a.get_first_name() == b.get_first_name()) &&
           (a.get_last_name()  == b.get_last_name()) &&
           (a.get_birthdate() == b.get_birthdate());
}

bool check(const vector<Person>& v, const vector<Person>& c)
{
   return v == c;
}
```

Ett alternativ om vi inte vill ha en `operator==`är att använda algoritmen `std::equal` med hjälp av predikatet:

```
bool equal(const Person& a, const Person& b)
{
    return (a.get_first_name() == b.get_first_name()) &&
           (a.get_last_name()  == b.get_last_name()) &&
           (a.get_birthdate() == b.get_birthdate());
}
```

Överlagringen av `check`:

```
bool check(const vector<Person>& v, const vector<Person>& c)
{
    return std::equal(v.begin(), v.end(), c.begin(), equal);
}
```

5. Lösningen är att returnera ett värde i stället för en pekare:

```
vector<string> collect_with_initial2(char c, std::istream& is)
{
    auto res = vector<string>{}; //eller    vector<string> res{};

    string s;
    while(is >> s){
        if(s[0] == c)
            res.push_back(s);
    }
    return res;
}
```