Tentamen
EDAF30 Programmering i C++

2016–01–11, 8.00–13.00

Hjälpmedel: En valfri C++-bok. OH-bilderna från föreläsningarna är *inte* tillåtna.

Du ska i dina lösningar visa att du behärskar C++ och även att du kan använda C++ standardklasser och algoritmer. Rena ”C-lösningar” på problem som med fördel kan lösas enligt mera objektorienterade principer, eller egen implementering av sådant som finns i standardbiblioteket, kan därför ge poängavdrag, även om de är korrekta. I bedömningen av en lösning kan även minneshantering beaktas där det är relevant. Om du använder saker ur standardbiblioteket i din kod, var tydlig med att du gör det. Använd fullt kvalificerade namn eller `using`-direktiv för de namn du använder.

Uppgifterna ger preliminärt $9+10+7+15+9=50$ poäng. För godkänt krävs preliminärt 25 poäng (betygsgränser 3/25, 4/33, 5/42). Observera att ordningsföljden eller poängantalet för uppgifterna inte nödvändigtvis avspeglar deras svårighet.

---

1. Klassen `Foo` har konstruktorer som är `private`, och i stället ”factory-funktioner” som är `public` och `static`:

```
class Foo {
public:
  static Foo* create()               { return new Foo();   }
  static Foo* create(int i)          { return new Foo(i);  }
  static Foo* create(const Foo& foo) { return new Foo(foo);}
  int compute() const;
  // ...
private:
  Foo();
  Foo(int i);
  Foo(const Foo& foo);
  // ...
};
```

   a) Detta görs för att begränsa hur objekt av klassen `Foo` kan allokeras i minnet.
   Vilken är begränsningen?

   b) Vi vill använda klassen `Foo` i en klass `Bar` enligt följande:

```
class Bar {
public:
    Bar(int i=0) :f{i} {}
    int compute() const {return f.compute();}
private:
    Foo f;
};
```

   men detta går inte, utan man får ett kompileringsfel.

   Vilket *tillägg* behöver man göra till definitionen av klassen `Foo` för att klassen `Bar` ska fungera? Du ska inte göra några ändringar av de givna deklarationerna.

   c) Antag att vi inte kan ändra klassen `Foo`. Ändra klassen `Bar` (från uppgift b) så att den fungerar med den ursprungliga klassen `Foo`. Den ändrade klassen `Bar` ska ha samma gränssnitt utåt, och det ska fortfarande gälla att ett `Bar`-objekt *har* ett `Foo`-objekt som initieras i konstruktorn `Bar::Bar(int)`, och att `Bar::compute` anropar `compute` på detta `Foo`-objekt.
   Svara med den modifierade klassen `Bar`.

---

2. Följande program är ett exempel på användning av en polymorf klass och dynamisk bindning.

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

using std::cout;
using std::endl;

struct Foo {
    Foo(int i) :x{i} {}
    ~Foo() =default;
    virtual int getX() const {return x;}
    virtual void print() const;
private:
    int x;
};

struct Bar :Foo {
    Bar(int i, const char* c);
    ~Bar() {delete[] s;}
    const char* getS() const {return s;}
    void print() const override;
private:
    char* s;
};

void Foo::print() const
{
    cout << "Foo(" << getX() << ")" << endl;
}

Bar::Bar(int i, const char* c) :Foo(i)
{
    auto len = strlen(c)+1;
    s = new char[len];
    strncpy(s, c, len);
}
void Bar::print() const
{
    cout << "Bar(" << getX() << ", " << getS() << ")" << endl;
}

void test()
{
    std::vector<Foo*> v;

    v.push_back(new Foo(12));
    v.push_back(new Bar(16, "Kalle"));
    v.push_back(new Bar(22, "Anka"));
    v.push_back(new Foo(17));

    for(auto e : v) {
        e->print();
    }

    for(auto e : v) {
        delete e;
    }
}
```

Fortsättning på nästa sida...

Programmet fungerar som förväntat; om man kör funktionen `test()` får man följande utskrift:

```
Foo(12)
Bar(16, Kalle)
Bar(22, Anka)
Foo(17)
```

Tyvärr har programmet ett allvarligt fel: det läcker minne. Vid test med ett verktyg som letar efter minnesläckor fick man följande resultat:

```
 LEAK SUMMARY:
    definitely lost: 11 bytes in 2 blocks
    indirectly lost: 0 bytes in 0 blocks
      possibly lost: 0 bytes in 0 blocks
```

a) Vad innebär en minnesläcka?

b) Vad är felet i programmet och varför leder det till en minnesläcka när `test()` körs??

3. Betrakta följande klasser och funktioner:

```
struct Foo{
    virtual void print() const {std::cout << "Foo" << std::endl;}
};
struct Bar :Foo{
    virtual void print() const override {std::cout << "Bar" << std::endl;}
};
struct Qux :Bar{
    virtual void print() const override {std::cout << "Qux" << std::endl;}
};
void print1(const Foo* f)
{
    f->print();
}
void print2(const Foo& f)
{
    f.print();
}
void print3(Foo f)
{
    f.print();
}
int main()
{
    Foo* a =  new Bar;
    Bar& b = *new Qux;
    Bar  c = *new Qux;

    print1(a);
    print1(&b);
    print1(&c);
    std::cout << std::endl;
    print2(*a);
    print2(b);
    print2(c);
    std::cout << std::endl;
    print3(*a);
    print3(b);
    print3(c);
    return 0;
}
```

Programmet går att kompilera och köra utan exekveringsfel.

Vad skrivs ut när `main()` exekveras?

4. Vi har en klass som representerar personer:

```
using std::string;
class Person{
public:
  Person(string f, string l,  string d) :fname{f},lname{l},birthdate{d} {}
  Person(const Person&) =default;
  Person(Person&&) =default;
  Person& operator=(const Person&) =default;
  Person& operator=(Person&&) =default;
  const string& get_first_name() const {return fname;}
  const string& get_last_name() const {return lname;}
  const string& get_birthdate() const {return birthdate;}
private:
  string fname;
  string lname;
  string birthdate;
};
```

Vi vill kunna sortera en sekvens av personer med hjälp av `std::sort`, både i bokstavsordning (på Efternamn, Förnamn) och i åldersordning, samt sortera både *stigande* (minst först) och *fallande* (störst först). Vi har definierat en abstrakt klass, `Comparator`, som ska hantera jämförelser av `Person`-objekt. Medlemsfunktionen `comp` betyder som vanligt relationen *mindre än*, men användaren kan välja om `operator()` för ett `Comparator`-objekt ska betyda *mindre än* eller *större än*:

```
class Comparator{
public:
  Comparator(bool asc=true) :ascending{asc} {}
  bool operator()(const Person& a, const Person& b) const;
  virtual bool comp(const Person& a, const Person& b) const =0;
  virtual ~Comparator() {}
private:
  bool ascending;
};
bool Comparator::operator()(const Person& a, const Person& b) const
{
  return ascending?comp(a,b):comp(b,a);
}
```

Vi ska nu ge ett exempel på användning, i ett testprogram.

```
#include<vector>
#include<algorithm>
void check(const Person& a, const Person& b)
{
    assert(a.get_first_name() == b.get_first_name());
    assert(a.get_last_name()  == b.get_last_name());
    assert(a.get_birthdate() == b.get_birthdate());
}
void test()
{
    auto anders = Person("Anders", "Karlsson", "2001-11-30");
    auto nisse = Person("Nisse", "Nilsson", "1984-01-03");
    auto bengta = Person("Bengta", "Bengtsdotter", "1934-07-25");
    auto gustav = Person("Gustav", "Karlsson", "1928-10-11");
    std::vector<Person> v{anders,nisse,bengta,gustav};

    std::sort(v.begin(), v.end(), NameComparator{}); //namn, stigande
    check(v[0], bengta);
    check(v[1], anders);
    check(v[2], gustav);
    check(v[3], nisse);
```

programmet fortsätter på nästa sida...

```
    std::sort(v.begin(), v.end(), NameComparator{false}); // namn, fallande
    check(v[0], nisse);
    check(v[1], gustav);
    check(v[2], anders);
    check(v[3], bengta);

    std::sort(v.begin(), v.end(), DateComparator{true}); // födelsedatum, stigande
    check(v[0], gustav);
    check(v[1], bengta);
    check(v[2], nisse);
    check(v[3], anders);

    std::sort(v.begin(), v.end(), DateComparator{false});// födelsedatum, fallande
    check(v[0], anders);
    check(v[1], nisse);
    check(v[2], bengta);
    check(v[3], gustav);
}
```

a) Implementera `NameComparator` och `DateComparator` (subklasser till `Comparator`) så att detta testprogram fungerar utan fel.

   Svara med fullständig definition av klasserna, inklusive alla medlemsfunktioner.

b) Komplettera programmet med det som krävs för att man ska kunna sortera en `std::vector<Person>` på namn (stigande) enligt ovan utan att skicka med ett komparator-objekt, d v s med anropet `std::sort(v.begin(), v.end());`
   Använd om möjligt din lösning till uppgift a) för att undvika duplicerad kod.

   Svara med de ändringar och tillägg (uttryckta som C++-kod) du behöver göra.

c) Vi vill skriva om vårt testprogram för att undvika att explicit jämföra element för element.
   I stället för

```
    std::sort(v.begin(), v.end(), NameComparator{}); //namn, stigande

-    check(v[0], bengta);
    check(v[1], anders);
    check(v[2], gustav);
    check(v[3], nisse);
```

   vill vi kunna skriva

```
    std::sort(v.begin(), v.end(), NameComparator{}); //namn, stigande

    if(!check(v, std::vector<Person>{bengta, anders, gustav, nisse}))
        std::cout << "*** fel i namn, stigande" << std::endl;
```

   Skriv en ny funktion `check`, som går att använda enligt detta exempel. Du får definiera hjälpfunktioner om du behöver men använd om möjligt funktionalitet som redan finns i standardbiblioteket.
   Svara med funktionen `check` och eventuella hjälpfunktioner.

5. Betrakta följande kod

```
using std::vector;
using std::string;
using std::cout;
using std::endl;

template <typename C>
void print_seq(const C& c)
{
    for(auto x:c)
        cout << x << " ";
    cout << endl;
}

vector<string>* collect_with_initial(char c, std::istream& is)
{
    auto res = new vector<string>{};

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

void use1()
{
    std::stringstream ss{"smurf nisse tomte sallad kakor"};

    auto strings = collect_with_initial('s', ss);

    print_seq(*strings); // använd pekarnotation

    delete strings;      // kom ihåg delete
}
```

Funktionen `collect_with_initial` läser strängar från en ström och samlar de strängar, som börjar på en given bokstav (parametern `c`), i en `std::vector<string>`. När man använder denna (t ex i `use1`), så måste man dels använda pekar-notation, och dels komma ihåg att göra `delete` på resultatet. Detta är ofta onödigt besvär.

För att förenkla för användaren vill vi ändra så att man i stället kan använda funktionen som i `use2()` (där funktionsmallen `print_seq` är samma som ovan):

```
void use2()
{
    std::stringstream ss{"smurf nisse tomte sallad kakor"};

    auto strings = collect_with_initial('s', ss);

    print_seq(strings);
}
```

Ändra funktionen `collect_with_initial` så att man i stället kan använda den som i `use2`.
(Både `use1` och `use2` ska skriva ut `smurf sallad`.)

Svara med en fullständig funktionsdefinition för den ändrade `collect_with_initial`.