LUND INSTITUTE OF TECHNOLOGY Department of Computer Science

Solutions, C++ Programming Examination

2017–03–17

1.
```
template<typename InputIterator, typename OutputIterator>
OutputIterator adjacent_difference(InputIterator first, InputIterator last,
        OutputIterator result) {
    if (first == last) {
        return result;
    }
    auto value = *first;
    *result = value;
    while (++first != last) {
        auto tmp = *first;
        *++result = tmp - value;
        value = tmp;
    }
    return ++result;
}

int main() {
    deque<int> a;
    copy(istream_iterator<int>(cin), istream_iterator<int>(), back_inserter(a));
    copy(a.begin(), a.end(), ostream_iterator<int>(cout, " "));
    cout << endl;
    while (a.size() > 1) {
        deque<int> b;
        adjacent_difference(a.begin(), a.end(), back_inserter(b));
        b.pop_front();
        a.swap(b);
        copy(a.begin(), a.end(), ostream_iterator<int>(cout, " "));
        cout << endl;
    }
}
```

2.
```
class Ruler {
friend std::ostream& operator<<(std::ostream &, const Ruler&);
friend std::istream& operator>>(std::istream &, Ruler&);
friend bool operator<(const Ruler&,const Ruler&);

public:
    Ruler() : number(0), start(0), end(0) { }
    Ruler(std::string nm, int nr, std::string co, int st, int en)
        : name(nm), number(nr), country(co), start(st), end(en) {}

private:
    std::string name;
    int number;
    std::string country;
    int start;
    int end;
};
```

```
std::ostream& operator<<(std::ostream &os, const Ruler& r) {
    os << r.name << " " << r.number << " of " << r.country << ", " << r.start << "-" << r.end;
    return os;
}

std::istream& operator>>(std::istream &is, Ruler& r) {
    is >> r.name >> r.number >> r.country >> r.start >> r.end;
    return is;
}

bool operator<(const Ruler &r1,const Ruler &r2) {
    return r1.country<r2.country || (r1.country==r2.country && r1.start<r2.start);
}

int main(int argc, char** argv) {
    std::ifstream is(argv[1]);
    std::multiset<Ruler> royalties((std::istream_iterator<Ruler>(is)),
                                          std::istream_iterator<Ruler>());
    std::copy(royalties.begin(),royalties.end(),std::ostream_iterator<Ruler>(cout,"\n"));
}
```

3.
```
template<typename C>
class back_insert_iterator {
public:
    explicit back_insert_iterator(C& c) : container(c) {}
    back_insert_iterator<C>& operator=(typename C::const_reference e) {
        container.push_back(e);
        return *this;
    }
    back_insert_iterator<C>& operator*() { return *this; }
    back_insert_iterator<C>& operator++() { return *this; }
    back_insert_iterator<C>& operator++(int) { return *this; }
private:
    C& container;                           // the container that we push-back on
};

template<typename C>
inline back_insert_iterator<C> back_inserter(C& c) {
    return back_insert_iterator<C>(c);
}
```

4.
1. Kompileringsfelet beror på att medlemsvariabeln `a` saknar initialisering. Visserligen tilldelas den ett nytt värde i konstruktorns kodkropp, men den måste ha fått ett initialt startvärde innan kodkroppen kan köras och det saknas en explicit initialisering för variabeln. Notera skillnaden mellan *initialisering* och *tilldelning*. Om klassen `A` hade haft en default-konstruktor (konstruktor utan parametrar) hade den använts, men i detta fall saknas en sådan.

   Det är också värt att notera att tilldelningen i sig (`a = i;`) är tillåten trots att `a` och `i` har olika typer. Kompilatorn inser själv att den kan göra om heltalet `i` till ett `A`-objekt genom att anropa konstruktorn i klassen `A`. Tilldelningen översätts då automatiskt till `a = A(i);`.
2. Lägg till en explicit initialisering av medlemsvariabeln `a`. Lämpligen ändras raden med kompileringsfelet till: `B(int i):a(i) {}`

5. Om pekarna hör ihop, d v s att det är ett fel att tilldela den ena men inte den andra:

```
Foo& Foo::operator=(const Foo& that)
{
    if (this != &that) {
        Bar* bar1 = 0;
        Bar* bar2 = 0;

        try {
            bar1 = new Bar(*that.fBar1);
            bar2 = new Bar(*that.fBar2);
        }
        catch (...) {
            delete bar1;
            delete bar2;
            throw;
        }

        SuperFoo::operator=(that);
        delete fBar1;
        fBar1 = bar1;
        delete fBar2;
        fBar2 = bar2;
    }
    return *this;
}
```

   Eller (om pekarna inte beror av varandra), använd `Bar::operator=`:

```
Foo& Foo::operator=(const Foo& that)
{
    SuperFoo::operator=(that);
    *fBar1 = *that.fBar1;
    *fBar2 = *that.fBar2;
    return *this;
}
```