LUND INSTITUTE OF TECHNOLOGY Department of Computer Science

Solutions, C++ Programming examination

2020-03-19

1. a) The function template can be implemented either by using `std::transform` or with a simple range-for loop. In this case, the range-for option is arguably cleaner. Instead of a range-for, `std::for_each` is a third option.

```
#include <map>
#include <algorithm>

template <typename K, typename V>
std::map<V,K> invert_map(const std::map<K,V>& m)
{
    std::map<V,K> res;
#ifdef USE_STD_ALGORITHM
    std::transform(m.begin(), m.end(), std::inserter(res, res.end()),
                   [](const std::pair<K,V>& e) {return make_pair(e.second, e.first);});
#else
    std::map<V,K> res;
    for(const auto& e : m) {
        res.emplace(e.second, e.first);
    }
#endif
    return res;
}
```

b) The class template. Note that we need to pull `map::at` into the scope of this class to make it accessible for overload resolution. We also need to make the constructors from `map` accessible. Note that here it is OK to inherit from `std::map` (which does not have a virtual destructor), as `bidir_map` does not have any data members and is not used polymorphically. But, outside an exam problem, it is arguably better to make the reverse find a free function.

```
#include <map>
#include <algorithm>
#include <iterator>
#include <utility>

template <typename K, typename V>
class bidir_map :public std::map<K,V>{
public:
    using std::map<K,V>::map;
    using std::map<K,V>::at;

    const K& at(const V& v) const {
        auto it = this->lookup(this->cbegin(), v);
        if(it != this->cend()) return it->first;
        throw std::out_of_range("value not found");
    }
private:
    using map_t = std::map<K,V>;
    using iter = typename map_t::const_iterator;
    iter lookup(iter first, const V& v) const {
        using vals = typename map_t::value_type;
        return std::find_if(first, this->cend(), [&](const vals& x){return x.second == v;});
    }
};
```

c) For reverse find, the following is added:

```
public:
    using std::map<K,V>::find;
    std::pair<key_iter,key_iter> find(const V& v) const {
        return {key_iter(*this, v), key_iter(*this)};
    }

private:
    struct key_iter : std::iterator<std::forward_iterator_tag, K>{
        key_iter(const bidir_map& mm, const V& v) :m(mm),
                                                   pos(mm.lookup(mm.cbegin(), v)),
                                                   val(v) {}
        key_iter(const bidir_map& mm) :m(mm),pos(mm.cend()){}
        bool operator!=(key_iter it) const {return pos != it.pos;}
        const K& operator*() const {return pos->first;}
        key_iter& operator++() {pos=m.lookup(++pos, val); return *this;}
        key_iter operator++(int) {auto ret = *this; pos=m.lookup(++pos, *val); return ret;}

        const bidir_map& m;
        iter pos;
        const V val{};
    };
```

2. As `Base::foo() const` and `Derived::foo()` is differ in `const`ness, `Derived::foo()` does not override the virtual `Base::foo`, but `Derived::foo()` hides `Base::foo() const` in `Derived`. Therefore, when called through a `Base*`, `Base::foo() const` is called, but when called through a `Derived*`, only `Derived::foo()` is visible, so `Derived::foo()` is called.

3. a) Taking the collection parameter by value is fundamentally wrong, it should be an output parameter and must therefore be a reference (or pointer). It appears to mostly work by having undefined behaviour: As `generate2()` is *called by value* a copy of the `Vektor` object will be made and as `Vektor` does not have a user-defined copy constructor, an implicitly defined will be used. Thus, a shallow copy will be made, copying the *value* of the pointer `e`, so the copy will point to the same array as the "original" (the variable `a` in `test()`. When the function returns and the destructor is called on the copy, the pointer in `a` will become a *dangling pointer* and the program has *undefined behaviour*. The segmentation fault is due to the double delete when the destructor of `a` does a second `delete` of the same object.

b) No. The class can be changed to avoid the *undefined behaviour* by adding a *copy-constructor* (which it should according to the "rule of three"), but then the assignment will be done to the elements of the copy (which is a local variable in `generate2()` and the program will not have the expected behaviour as the argument to `generate2` is not modified.

c) Yes, by changing to call by reference.

d) No. `Counter` works as intended.

e) No, the member initializer list of the constructor initializes all members.