Solutions, C++ Programming examination

2018–03–14

1. A possible implementation is

```cpp
template <typename InputIt, typename OutputIt, typename T, typename U>
void apply_all(InputIt f, InputIt fe, OutputIt o, const T& a, const U& b)
{
    while(f != fe) {
        *o = (*f)(a, b);
        ++o;
        ++f;
    }
}
```

Instead of the while loop, `std::for_each` can be used

```cpp
using FnType = typename std::iterator_traits<InputIt>::value_type;
std::for_each(f, fe, [&](const FnType& fn) {*o++ = fn(a, b);});
```

2. a) The problem is that the class does not follow the rule of three. As the class has an owning pointer and a destructor, the implicitly defined copy constructor does not work.

The lambda takes its parameter by value (invoking the copy constructor) and when the lambda returns, the destructor is invoked on the copy. As the two pointers in the copy and the “original” point to the same array, when the copy is destroyed, the “original” is left in an unusable state.

b) Change to call by reference in the lambda. This fixes the particular problem in `main()`, but the class still will cause undefined behaviour if copied.

c) Make the member be a `std::string` instead of a `char*` and remove the destructor (as the default destructor works for RAII members).

```cpp
class Foo{
public:
    Foo() : Foo(-1, "Foo") {}  
    Foo(int x, const char* cs) : v{x}, s{cs} {}  
    "Foo() =default;
    std::string get_s() const {return s;}
    int get_v() const {return v;}
private:
    int v;
    std::string s;
};
```

Alternatively, if we want to keep the `char*`, implement a copy constructor (and copy assignment operator, which is not required by the question but included here for completeness).

```cpp
class Foo{
public:
    Foo() : Foo(-1, "Foo") {}
    Foo(int x, const char* cs) : v{x}, s{new char[std::strlen(cs)+1]} {std::strcpy(s, cs);}
    Foo(const Foo& o) : Foo(o.v, o.s) {}
};
```
Foo& operator=(const Foo&);
'Foo() {delete[](s);} std::string get_s() const {return std::string(s);}
int get_v() const {return v;}

private:
    int v;
    char* s;
};

Foo& Foo::operator=(const Foo& o)
{
    if(this == &o) return *this;
    if(o.s) {
        auto len = strlen(o.s);
        if(len < strlen(s)){ // new string longer: we must allocate
            auto tmp = new char[len+1];
            strcpy(tmp, o.s);
            delete[] s;
            s = tmp;
        } else { // we can reuse the current array
            strcpy(s, o.s);
        }
    } else {
        s = nullptr;
    }
    v = o.v;
    return *this;
}

3. Note that both const overloads are required for operator[] and that size() should be const:

#include <cassert>
#include <iostream>
#include <memory>
#include <algorithm>

template <typename T>
class Vector{
public:
    Vector() =default;
    Vector(int size) :sz(size),elem(new T[size]) {}
    Vector(std::initializer_list<T> l) :Vector(l.size()) {
        std::copy(begin(l), end(l), elem.get());}
    bool operator==(const Vector<T>& rhs) const
    {
        if(sz != rhs.sz) return false;
        return std::equal(elem.get(), elem.get()+sz, rhs.elem.get());
    }

    int size() const {return sz;}
    const T& operator[](int i) const {return elem[i];}
    T& operator[](int i) {return elem[i];}

private:
    int sz{0};
    std::unique_ptr<T[]> elem{nullptr};
};

template <typename T>
bool Vector<T>::operator==(const Vector<T>& rhs) const
{
    if(sz != rhs.sz) return false;
    return std::equal(elem.get(), elem.get()+sz, rhs.elem.get());
}
4. A solution with the dict as a `std::vector<std::string>` is shown. An alternative is to use a `std::set<std::string>`, which will automatically provide sorting and removal of duplicates.

```cpp
#include <iostream>
#include <fstream>
#include <algorithm>
#include <vector>
#include <string>
#include <stdexcept>
#include <iterator>
#include <cctype>

std::vector<std::string> read_dict(const std::string& fname)
{
    std::ifstream in(fname);
    if(!in) throw std::runtime_error("Failed to open dict");

    auto sb = std::istream_iterator<std::string>(in);
    auto se = std::istream_iterator<std::string>();
    return std::vector<std::string>(sb, se);
}

bool reverse_less(const std::string& a, const std::string& b)
{
    return std::lexicographical_compare(a.rbegin(), a.rend(), b.rbegin(), b.rend());
}

void make_lowercase_str(std::string& w) {
    std::transform(begin(w), end(w), begin(w),
        [](unsigned char c){ return std::tolower(c); });
}

void make_lowercase(std::vector<std::string>& words) {
    std::for_each(begin(words), end(words), make_lowercase_str);
}

int main()
{
    auto words = read_dict("/usr/share/dict/words");
    make_lowercase(words);
    auto uend = std::unique(begin(words), end(words));
    std::sort(begin(words), uend, reverse_less);

    std::ofstream out("backwords_all.txt");
    if(!out) throw std::runtime_error("Failed to open output file");
    std::copy(begin(words), uend, std::ostream_iterator<std::string>(out, "\n"));
}
```

5. a) The problem is that Foo does not have a default constructor. Since Bar::a is not initialized in the constructor (it is assigned in the constructor function body) the compiler needs to default construct it by calling Foo::Foo().

b) The proper way to initialize a member variable is in the member initializer list of the constructor:

```cpp
Bar(int i) : a{i} {}  
```

And, although not strictly necessary in this case (since an int can be default constructed), do the same for Foo:

```cpp
Foo(int i) : x{i} {}  
```