Lösningsförslag, Tentamen i C++

2023-01-11

1. a) The problem is that the virtual function in `Foo`

```
virtual int getVal() const;
```

but the function `int getVal()` in `ScaledFoo` is missing the `const`, and therefore declaring another function shadowing the virtual `int getVal() const` instead of overriding it.

b) The solution is to add `const` to the declaration of `ScaledFoo::getVal`.

2. The solution is to cast the converted value back and check if they are equal.

```
template <typename T, typename U>
T check_cast(const U& val){
    T res{static_cast<T>(val)};
    if(static_cast<U>(res) != val) {throw std::invalid_argument("narrowing cast");}
    return res;
}
```

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.

4. `std::transform` will iterate over the input range and call its function with each of the element values, in order. Thus we need a function that returns the current nesting depth for each position. That function must be stateful so that the depth is increased when called with an opening parenthesis and decreased for a closing parenthesis.

```
struct paren_depth {

    int operator()(char c)
    {
        if (c == '(') {
            ++depth;
            return depth;
        } else if (c == ')') {
            --depth;
            return depth+1;
        }
        return depth;
    }

    int depth{};
};
```

5.
```
class word {
  public:
    word(const std::string &s) : w(s) {}
    int get_freq() const {return f;}
    const std::string& get_word() const {return w;}
    /* increase word frequency */
    word& operator++(){++f; return *this;}
  private:
    std::string w;
    int f{1};
};

bool operator<(const word& a, const word& b)
{
    return a.get_word() < b.get_word();
}

std::vector<word> read_words(std::istream &s)
{
    std::vector<word> res{};
    std::string w;
    while(s >> w){
        auto it = std::lower_bound(begin(res), end(res), w);
        if(it == end(res)){
            res.emplace_back(std::move(w));
        } else if (it->get_word() == w){
            ++*it;
        } else {
            res.emplace(it, std::move(w));
        }
    }
    return res;
}

std::ostream& operator<<(std::ostream& os, const word& w)
{
    return os << w.get_word() << ": " << w.get_freq();
}
```

```
bool cmp_freq(const word& a, const word& b)
{
    if(a.get_freq() == b.get_freq()) return a.get_word() < b.get_word();
    else return a.get_freq() > b.get_freq();
}

void sort_by_frequency(std::vector<word>& ws)
{
    std::sort(begin(ws), end(ws), cmp_freq);
}

void sort_alphabetically(std::vector<word>& ws)
{
    std::sort(begin(ws), end(ws));
}
```

6.

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

template<typename Key, typename Value>
typename std::map<Key, Value>::iterator
reverse_lookup(std::map<Key,Value>& table, const Value& val)
{
    return std::find_if(table.begin(), table.end(),
            [&](const std::pair<Key,Value>& p){
            return p.second == val;});
}
```