Lösningsförslag, Tentamen i C++

2020–01–15

1. 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:

```
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`:

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

2. a) As str is a pointer and i an int, the result of the operation `str + i` is a pointer to an object *i objects after str*. That is, `"testing" + 1` is a pointer to *the second character* in the string.

b) To get string concatenation, at least one of the operands must be a `std::string`. One way is to use `std::to_string` to get a string representation of the integer `i`: `return str + std::to_string(i);`.

3. A possible solution is to use two maps as shown here. Another is to use one map and use `std::find_if` to do the reverse lookup. A third is to use a `vector<pair<char,string>>` and `find_if`.

```
#include <cctype>
#include <map>
#include <fstream>
#include <string>
#include <sstream>

class Morse_code{
public:
    Morse_code(const std::string& filename);
    std::string encode(const std::string&) const;
    std::string decode(const std::string&) const;
private:
    std::map<char, std::string> encode_map;
    std::map<std::string, char> decode_map;
};

std::string Morse_code::encode(const std::string& s) const
{
    std::stringstream res;

    for(auto  c : s) {
        auto cit = encode_map.find(toupper(c));
        if(cit != encode_map.end()) {
            res << cit->second << "  ";
        }
    }
    return res.str();
}
```

```
std::string Morse_code::decode(const std::string& code) const
{
    std::stringstream src(code);
    std::stringstream res;

    for(std::string s; src >> s;){
        auto cit = decode_map.find(s);
        if(cit != decode_map.end()) {
            res << cit->second;
        } else {
            // std::cout << "found no match for " << s << "\n";
            res << "?";
        }
    }
    return res.str();
}

Morse_code::Morse_code(const std::string& filename)
{
    std::ifstream s(filename);
    if(!s){ // throw exception or exit program }
    char c;
    while(s >> c) {
        std::string code;
        s >> code;

#if 1
        encode_map.emplace(c,code);
        decode_map.emplace(code,c);
#else
        encode_map[c] = code;
        decode_map[code] = c;
#endif
    }
}
```

4. a) As `void add(Vektor<T>, Vektor<T>, Vektor<T>)` has by-value parameters, the arguments will be copied but as `Vektor` does not define a copy constructor, only the *pointer value* is copied. When the function returns, the copies will be destroyed, and thus the array pointed to by `p` will be deallocated, leaving the `Vektor` objects in the calling function in an unusable state.

b) Yes. Change to by-reference parameters:

```
template <typename T>
void add(const Vektor<T>& c1, const Vektor<T>& c2, Vektor<T>& c3)
```

c) No, if a copy-constructor is defined (which it should be) the result will be `0 0 0 0 0 0`, as the assignments in `add` are made to the copy.

d) The empty braces (`{}`) force value initialization. Thus, without them the program will have undefined behaviour for element types that are not guaranteed to be value initialized (e.g. `Vektor<int>`).

e) Solution with std::copy

```
template <typename T> void Vektor<T>::assign(const std::initializer_list<T>& l)
{
    std::copy(l.begin(), l.end(), begin());
}
```