Lösningsförslag, Tentamen i C++

2022–01–13

1. `example1` leaks memory, as the object pointed to by the owning pointer is not destroyed before the pointer goes out of scope.
`example2` does not leak memory, as the object owned by the `unique_ptr` is destroyed when the pointer goes out of scope.
`example3` leaks memory, as the object pointed to by the owning pointer is not destroyed before the pointer goes out of scope.
`example4` leaks memory, as the object pointed to by the owning pointer is not destroyed before the pointer goes out of scope.
`example5` leaks memory, as even though the `unique_ptr<Foo>` destroys the object it owns, `Bar::~Bar()` is not called as `Foo::~Foo()` is not `virtual`.
`example6` does not leak memory, as the object owned by the `unique_ptr` is destroyed when the pointer goes out of scope.

2. The intended solution is to use one map and use `std::find_if` to do the reverse lookup.

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

    char c;
    while (ins >> c) { // use stringstream to skip whitespace
#ifndef NO_DEBUG_INFO
        try{
#endif
        auto x = encode_map.at(toupper(c));
        res << x << "  ";
#ifndef NO_DEBUG_INFO
        } catch(...) {std::cerr << "[error encoding char: " << c << "]\n"; throw;}
#endif
    }
    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 = std::find_if(encode_map.begin(), encode_map.end(),
                                [&s](const std::pair<char, std::string>& p) {
                                    return p.second == s;
                                });
        if (cit != encode_map.end()) {
            res << cit->first;
        } else {
            res << "?";
        }
    }
    return res.str();
}

Morse_code::Morse_code(const std::string& filename)
{
    std::ifstream s(filename);
    if (!s) {
        throw std::runtime_error("failed to open morse definitions file");
    }
    char c;
    while (s >> c) {
        std::string code;
        s >> code;

        encode_map.emplace(c, code);
    }
}
```

If we assume that the input file is correctly formatted, the reading can be done like this, without the extra step of reading linewise and creating a stringstream.

3. The problem is that the class `User` does not follow the rule of three: it has owning pointers and a destructor, but not a user-defined copy constructor. As `operator==(User)`, `operator!=(User)`, and `operator<(User)` have value parameters, the default copy-constructor is called (which does a shallow copy), and the destruction of the parameter leaves a dangling pointer in `main()`.

The solution is to change to `const User&` parameters and delete (or define) the copy special member functions. Keeping call-by-value and defining the copy special member function to make a deep copy works, but is inferior as the copy is unnecessary for the comparison.

4. a) To check if they refer to the same object, compare the addresses: `if(&a == &b)...`

b) To check if they have the same value, use `if(a == b)...`

c) As there is no common superclass to all types in C++, a function template must be used. The following function template covers the simple case where both arguments are of the same type, and `operator==` is defined for the type,
The function must have reference parameters to enable checking for reference equality.

```
template <typename A>
void compareObjects(const A& a, const A& b) {
  if( &a == &b) {
    std::cout << "a and b is the same object\n");
  }

  if( a == b) {
    std::cout << "the values of a and b are equal\n");
  }
}
```

5. a Here, we need a converting constructor `Foo(int)`, which defines an implicit convesion from `int` to `Foo`. We also need `operator int()`, which defines an implicit conversion in the reverse direction. Finally, we need a default constructor, as creating a `std::vector` with a size > 0 must default-construct its elements.

b this `std::transform` calls a unary function, with each of the elements as argument, and writes its return value to the output iterator. That function is `apply` which invokes its parameter without arguments. Here, that means that a `Foo` object should be callable without arguments and return an `int` or a `Foo` (which converts implicitly to `int`).

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <numeric>
#include <vector>

class Foo {
public:
    Foo() =default;
    Foo(int x) :val{x} {}
    operator int() const {return val;}
#ifdef EXAMPLE2
    Foo operator()() const {return 2*val;}
#endif

private:
    int val{};
};
```