LUND INSTITUTE OF TECHNOLOGY Department of Computer Science

Solutions, C++ Programming examination

2025-06-03

1.
```
#include <algorithm>
#include <iterator>

template <typename Iter>
class unique_iter{
private:
    Iter f;
    Iter l;
public:
    using value_type = typename std::iterator_traits<Iter>::value_type;
    // alternatives with decltype:
    // using value_type = decltype(*f);
    // using value_type = decltype(*std::declval<Iter>());
    unique_iter(Iter first, Iter last) :f(first),l(last) {}
    unique_iter& operator++() {
        f = std::find_if(f,l, [=](const value_type& x){ return x != *f;});
        return *this;
    }
    bool operator!=(const Iter& i) {return f != i;}
    value_type& operator*() {return *f;}
};

template <typename Iter>
unique_iter<Iter> unique_subseq(Iter first, Iter last)
{
    return unique_iter<Iter>(first,last);
}
```

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. 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 functions to make a deep copy works, but is inferior as the copy is unnecessary for the comparison.

4. The class needs a converting constructor and an `operator int()` to handle the conversions from and to `int`, and an `operator()(int)` for the accumulation. It also needs a default constructor. No other operators are required, as the implicit conversions to/from `int` allows using assignment, greater than, and the output operator for `int`.

```
class counter {
  public:
    counter(int val = 0) : v{val} {}
    operator int() const { return v; }
    counter& operator()(int val = 1)
    {
        v += val;
        return *this;
    }

  private:
    int v;
};
```

5. 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);`.

6.
```
#include <string>
#include <iostream>
#include <fstream>

int print_matching_lines(const std::string& phrase, const std::string& fname)
{
    std::ifstream f(fname);
    if(!f){
        std::cerr << "cannot open " << fname << '\n';
        return 1;
    } else
    {
        std::string line;
        while(std::getline(f, line)){
            if(line.find(phrase) != std::string::npos) {
                std::cout << line << "\n";
            }
        }
    }
    return 0;
}
```