# Solutions, C++ Programming examination

## 22-03-17

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.
```
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));
}
```

**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{};
};
```