LUND INSTITUTE OF TECHNOLOGY Department of Computer Science

Solutions, C++ Programming examination

2026-06-04

1. a) The problem is that `person` does not follow the rule of three, as it does not define a copy assignment operator. That causes corruption in the call to std::sort, which copy assigns vector elements when it swaps them. As no user-defined copy assignment operator exists, the default one will simply copy the `char *` value. Then, two objects will have an owning pointer to the same `char[]`, and when it is deleted in the destructor that will lead to memory corruption: use-after-free (and double delete).

Please note that a class with only a copy constructor and copy assignment operator does fulfill the requirements of `MoveConstructible` and `MoveAssignable`. They can be created and assigned from an `rvalue`, just a bit less efficiently than if they had a move constructor and assignment operator.

b) The solution is to implement a copy assignment operator.

```
person& operator=(const person& o)
{
    if(this != &o){
        new_name = new char[strlen(o.name) + 1];
        delete[] name;
        name = new_name;
        strcpy(name, o.name);
        id=o.id;
    }
    return *this;
}
```

In this particular example, it would be enough to overload `swap(person&, person&)` but following the rule of three (or five) is the general solution.

c) As name is a C string (a `char *` pointing to the first character), comparing the pointers will compare the addresses where the strings are located in memory, not the actual string values. (Comparing pointers to different objects with $<$ or $>$ actually has undefined behaviour.)

2. Both functions `do_find` and `do_binary` have a by-value parameter, which means that the vector is copied (and the copy is destroyed) on every call. The time the actual search takes is much smaller, so the time spent on copying and deleting dominates the time measured in the benchmarks. Changing the parameters to `std::vector<int>&` gives the following times:

```
-------------------------------------------------------------------
Benchmark                       Time             CPU   Iterations
-------------------------------------------------------------------
ns_bench/LinearSearch      25158 ns        25155 ns        27805
ns_bench/BinarySearch        460 ns          460 ns      1478578
```

3. A possible solution is:

```
#include <algorithm>
template <typename Iter, typename T>
class result_iter {
  public:
    result_iter(Iter first, Iter last, const T& t) : f(first), l(last), val(t) {
        next();
    }
```

```
    result_iter& operator++() {
        ++f;
        next();
        return *this;
    }
    T& operator*() { return *f; }
    bool operator!=(Iter it) const { return f != it; }

  private:
    void next() { f = std::find(f, l, val); }

    Iter f;
    Iter l;
    T val;
};

template <typename Iter, typename T>
result_iter<Iter, T>
find_all(Iter first, Iter last, const T& val)
{
    return result_iter<Iter, T>(first, last, val);
}
```

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**. The constructor does not initiate the variable `a`, it *assigns* it in the function body. As the object must be completely constructed before the function body is executed, there is an implicit call of the default constructor `A::A()`, but that does not exist.

b. The problem can be fixed in `A` by giving it a default constructor. Two examples:

```
A(int x=0) {val = x;}
```

or

```
A() = default;
A(int x) {val = x;}
```

Here, assigning in the function body will work, but you should always write constructors with initiaisation (e.g., `A(int x=0) :val{x} {}`) or add a default initiaizer (e.g., `int val{};`) when possible.

c. To solve the problem in `B`, the member `a` is initialised instead of assigned:

```
B(int x) :a{x} {}
```

or

```
B(int x) :a{A(x)} {}
```