LUND INSTITUTE OF TECHNOLOGY Department of Computer Science

Suggested exam solutions, EDAF30

17–01–09

1. a) Dynamic objects are created in `insert`, but they are never deleted. They must be deleted in the destructor:

```
~NameList() {
    for (list_type::iterator it = names.begin(); it != names.end(); ++it) {
        delete *it;
    }
}
```

b) In `printSorted` `*it` is printed, but `*it` is a pointer, not a string. Solution: `cout << **it`.

c) The class `set` sorts according to the values of the set, and these are pointers. Solution: write a functor that specifies the sorting order, and use it when the set is defined:

```
struct StringPtrLess {
    bool operator()(const std::string* ps1, const std::string* ps2) const {
        return *ps1 < *ps2;
    }
};
typedef std::set<std::string*, StringPtrLess> list_type;
```

2. a) The function should return true if `x` is in the interval $[-0.5, 0.5]$, false otherwise. However, the logical expression is computed as follows: first `-0.5 <= x` is evaluated, and the result is `true` or `false`. This value shall be compared to 0.5 and is converted to 1 or 0 (`true/false`). Finally, this value is compared to 0.5. The function should be written like this:

```
bool insideLimits(double x) {
        return -0.5 <= x && x <= 0.5;
}
```

b) We have used the wrong kind of parentheses in the `new`-expression in the constructor, it should be `new int[n]`. (`new int(n)` allocates one integer with the value `n`, which also is legal, but then `v` isn't an array and we shouldn't use `delete[]` in the destructor).

In the destructor the line `if (v != nullptr)` can be deleted, since `delete nullptr` is defined to have no effect.

c) If a class contains virtual functions each object has a pointer to the virtual table, and this pointer needs storage.

d) You declare the copy constructor and the assignment operator as `private` or (in C++-11) `=delete`.

e) As an `unsigned` variable cannot be negative, the loop will never terminate. The output becomes:

```
10
9
8
7
6
5
4
3
2
1
0
4294967295
4294967294
4294967293
4294967292
4294967291
...
```

3. a)
```
template <typename T, typename Check>
class CheckedValue {
public:
    CheckedValue(const T& v) throw(std::logic_error) { set(v); }
    T get() const { return value; }
    void set(const T& v) throw(std::logic_error) {
        if (! check(v)) {
            throw std::logic_error("Value error");
        }
        value = v;
    }
private:
    T value;
    Check check;
};

struct pos {
    bool operator()(int x) const {
        return x >= 0;
    }
};

template <int n>
struct long_string {
    bool operator()(const string& s) const {
        return s.length() > n;
    }
};
```

b) The constructor `PositiveInteger(int)` defines an implicit type conversion from `int` to `PositiveInteger`. That object is then copy-assigned to `p`. The statement is equivalent to `p = PositiveInteger(3);`

c) We need to define a type conversion operator

```
operator int() const {return get();}
```

4.
```
/* time.h */
#ifndef TIME_H
#define TIME_H
#include <string>
#include <iostream>

class Time {
public:
    Time();
    Time(int hh, int mm, int ss);
    Time(const std::string& timeString);
    Time& operator+=(int seconds);
    Time& operator++();
    friend int operator-(const Time& t1, const Time& t2);
    friend std::ostream& operator<<(std::ostream& os, const Time& t);
private:
    int h; // hours (0-23)
    int m; // minutes (0-59)
    int s; // seconds (0-59)
    int toSeconds() const;   // convert hms to seconds
    void toHMS(int seconds); // convert seconds to hms
};

int operator-(const Time& t1, const Time& t2);
std::ostream& operator<<(std::ostream& os, const Time& t);
#endif
```

```
/* time.cpp */
#include "time.h"
#include <iostream>
#include <iomanip>
#include <string>
#include <sstream>
#include <ctime>
#include <cstdlib>

using namespace std;

Time::Time() {
    time_t timer = time(nullptr); // time in seconds since 1970-01-01
    tm* locTime = localtime(&timer); // broken-down time
    h = locTime->tm_hour;
    m = locTime->tm_min;
    s = locTime->tm_sec;
}

Time::Time(int hh, int mm, int ss) : h(hh), m(mm), s(ss) {}

Time::Time(const string& timeString) {
    istringstream iss(timeString);
    iss >> h;
    iss.ignore(1);
    iss >> m;
    iss.ignore(1);
    iss >> s;
}

Time& Time::operator+=(int seconds) {
    toHMS(toSeconds() + seconds);
    return *this;
}

Time& Time::operator++() {
    return *this += 1;
}

int Time::toSeconds() const {
    return h * 60 * 60 + m * 60 + s;
}

void Time::toHMS(int seconds) {
    s = seconds % 60;
    seconds = seconds / 60;
    m = seconds % 60;
    h = seconds / 60;
}

int operator-(const Time& t1, const Time& t2) {
    return t1.toSeconds() - t2.toSeconds();
}

std::ostream& operator<<(std::ostream& os, const Time& t) {
    os << std::setw(2) << std::setfill('0') << t.h << ':';
    os << std::setw(2) << std::setfill('0') << t.m << ':';
    os << std::setw(2) << std::setfill('0') << t.s;
    return os;
}
```