Examination
EDAF50 C++ Programming

22-03-17, 14:00–19:00

Aid at the exam: one C++ book. *Not allowed:* printed copies of the lecture slides or other papers.

Assessment (preliminary): the questions give $9 + 14 + 10 + 8 + 9 = 50$ points. You need 25 points for a passing grade (3/25, 4/33, 5/42).
You must show that you know C++ and that you can use the C++ standard library. "C solutions" don't give any points, and idiomatic C++ solutions that reimplement standard library facilities may give deductions, even if they are correct.
In solutions, resource management must also be considered when relevant – your solutions must not leak memory.
Free-text anwers should be concise but complete, well motivated and written clearly, to the point, and in complete sentences. Answers may be given in swedish or english.

For all problems, you may choose to answer "I don't know", which will be worth 20% of the maximum score of that problem. If you opt to do so, the sentence "I don't know." or "Jag vet inte." must be clearly given as the only answer to that problem. (I.e., you will not get any credit for an answer "I don't know" to a subproblem.)

The last page has an appendix describing some standard algorithms appearing in the problems.

Please write on only one side of the paper and hand in your solutions with the papers numbered, sorted and facing the same way. Please make sure to write your anonymous code and personal identifier on each page, and that the papers are not folded or creased, as the solutions may be scanned for the marking.

1. Managing the lifetimes of dynamically allocated objects is important in C++. Study the following classes and factory functions:

```
#include <algorithm>
#include <memory>
#include <numeric>
#include <iostream>

struct Foo {
   Foo(int x) :val{x} {}
   virtual int compute(int x) {return val * x;}
private:
    int val;
};

struct Bar :public Foo {
    Bar(int x) :Foo(x), a(new int[x]), sz(x) {std::iota(a.get(), a.get()+x, 1);}
    virtual int compute(int);
private:
    std::unique_ptr<int[]>a{nullptr};
    int sz{0};
};

int Bar::compute(int x){
        return x * std::accumulate(a.get(), a.get()+sz, 1, std::plus<int>{});}

Foo* make_foo(int x){ return new Foo(x); }

Bar* make_bar(int x){ return new Bar(x); }
```

Study the following functions and determine if they leak memory. For each function, *state if it leaks memory* ("Yes" or "No") *and motivate* why it leaks or how the memory is reclaimed.
If you opt to answer "I don't know", give that as your sole answer to the entire question.

```
void example1() {
    Foo* f = make_foo(17);
    std::cout << "example1: " << f->compute(10) << '\n';
}

void example2() {
    std::unique_ptr<Foo> f{make_foo(17)};
    std::cout << "example2: " << f->compute(10) << '\n';
}

void example3() {
    Foo* b = make_bar(17);
    std::cout << "example3: " << b->compute(10) << '\n';
}

void example4() {
    Bar* b = make_bar(17);
    std::cout << "example4: " << b->compute(10) << '\n';
}

void example5() {
    std::unique_ptr<Foo> b{make_bar(17)};
    std::cout << "example5: " << b->compute(10) << '\n';
}

void example6() {
    std::unique_ptr<Bar> b{make_bar(17)};
    std::cout << "example6: " << b->compute(10) << '\n';
}
```

2. The task is to compute the frequencies of the words in a text read from a `std::istream`.
The words are to be represented by a class `word` with a public interface including the following:

```
class word {
  public:
    word(const std::string &s);
    int get_freq() const;
    const std::string& get_word() const;
};
```

You shall implement the class `word` and a function `read_words` which reads words from a stream and returns a `std::vector<word>` containing all words read (sorted alphabetically, see example below) and their frequencies:

```
std::vector<word> read_words(std::istream &s);
```

`read_words` must build the sorted vector by inserting each word at the right place when first encountering it. You must use `std::lower_bound` to efficiently find a word (or where to insert it).

Then, you shall implement two functions, `sort_by_frequency`, and `sort_alphabetically` which sort a `std::vector<word>`. These functions may use `std::sort`.

The following example should work:

```
template <typename C>
void print_seq(const C& xs)
{
    for(const auto& x : xs) std::cout << x << "\n";
}

void example(std::istream& is)
{
    auto ws = read_words(is);
    print_seq(ws);
    std::cout << "by decreasing frequency:\n";
    sort_by_frequency(ws);
    print_seq(ws);
    std::cout << "alphabetically:\n";
    sort_alphabetically(ws);
    print_seq(ws);
}
```

An example `main` function and its expected output is given below:

```
int main()
{
    std::istringstream is{"aa bb aa dd aa dd cc bb dd"};
    example(is);
}
```

```
aa: 3
bb: 2
cc: 1
dd: 3
by decreasing frequency:
aa: 3
dd: 3
bb: 2
cc: 1
alphabetically:
aa: 3
bb: 2
cc: 1
dd: 3
```

Answer with the class `word` (add operations as needed by the given example) and the functions `read_words`, `sort_by_frequency`, and `sort_alphabetically`.

3. A class representing a person is defined in `user.h` as follows:

```
#ifndef USER_H
#define USER_H

#include <iostream>
#include <string>
#include <tuple>

class User {
  public:
    User() = default;
    User(const char* fname, const char* lname)
        : fn{new std::string(fname)}, ln{new std::string(lname)}
    {}
    ~User()
    {
        delete fn;
        delete ln;
    }

    friend std::ostream& operator<<(std::ostream& os, User u)
    {
        return os << *u.fn << " " << *u.ln;
    }

    bool operator==(User u) const { return *fn == *u.fn && *ln == *u.ln; }
    bool operator!=(User u) const { return !(*this == u); }
    bool operator<(User u) const
    {
        return std::tie(*ln, *fn) < std::tie(*u.ln, *u.fn);
    }

  private:
    std::string* fn{nullptr};
    std::string* ln{nullptr};
};
#endif
```

The following test program is written to test the comparison operators:

```
#include "user.h"

#include <iomanip>
#include <iostream>
#include <string>

using std::cout;

template <typename T, typename C>
bool test_cond(const std::string& msg, const T& t, const T& u, C c, bool exp)
{
    auto res = c(t, u) == exp;
    if (!res) {
        cout << "condition failed: [" << msg << "] for " << t << " and " << u;
        endl(cout);
    }
    return res;
}
```

continued on the next page...

```
int main()
{
    User u("Test", "Testsson");
    User v("Tim", "Plate");

    bool res{true};
    res &= test_cond("Equality", u, v, std::equal_to<User>(), false);
    res &= test_cond("Inquality", u, v, std::not_equal_to<User>(), true);
    res &= test_cond("Ordering (<)", u, v, std::less<User>(), false);
    res &= test_cond("Ordering (<)", v, u, std::less<User>(), true);

    if (res)
        cout << "Success.\n";
}
```

When the test program is run, the program crashes with the following output

```
user_test(36608,0x102b7c580) malloc: *** error for object 0x600000e75140: pointer being freed was not allocated
user_test(36608,0x102b7c580) malloc: *** set a breakpoint in malloc_error_break to debug
[1]    36608 abort      ./user_test
```

To investigate further, the programmer builds with `-fsantize=address`, and then the output is

```
=================================================================
==37443==ERROR: AddressSanitizer: heap-use-after-free on address 0x000108901fd7 at pc 0x0001047105dc bp 0
    ↪ x00016b6f2660 sp 0x00016b6f2658
READ of size 1 at 0x000108901fd7 thread T0
    #0 0x1047105d8 in std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char>
        ↪ >::__is_long() const+0x64 (a.out:arm64+0x1000045d8)
    #1 0x1047112c0 in std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char>
        ↪ >::size() const+0x18 (a.out:arm64+0x1000052c0)
    #2 0x104711120 in bool std::__1::operator==<std::__1::allocator<char> >(std::__1::basic_string<char,
        ↪ std::__1::char_traits<char>, std::__1::allocator<char> > const&, std::__1::basic_string<char,
        ↪  std::__1::char_traits<char>, std::__1::allocator<char> > const&)+0x3c (a.out:arm64+0
        ↪ x100005120)
    #3 0x104711058 in User::operator==(User) const+0x7c (a.out:arm64+0x100005058)
    #4 0x10471175c in User::operator!=(User) const+0x118 (a.out:arm64+0x10000575c)
    #5 0x10471153c in std::__1::not_equal_to<User>::operator()(User const&, User const&) const+0x11c (a.
        ↪ out:arm64+0x10000553c)
    #6 0x10470e778 in bool test_cond<User, std::__1::not_equal_to<User> >(std::__1::basic_string<char, std
        ↪ ::__1::char_traits<char>, std::__1::allocator<char> > const&, User const&, User const&, std::
        ↪ __1::not_equal_to<User>, bool)+0x138 (a.out:arm64+0x100002778)
    #7 0x10470de3c in main+0x284 (a.out:arm64+0x100001e3c)
    #8 0x1048890f0 in start+0x204 (dyld:arm64e+0x50f0)

0x000108901fd7 is located 23 bytes inside of 24-byte region [0x000108901fc0,0x000108901fd8)
freed by thread T0 here:
    #0 0x104c7aacc in wrap__ZdlPv+0x74 (libclang_rt.asan_osx_dynamic.dylib:arm64e+0x4aacc)
    #1 0x10470eebc in User::~User()+0x60 (a.out:arm64+0x100002ebc)
    #2 0x10470ed48 in User::~User()+0x18 (a.out:arm64+0x100002d48)
    #3 0x104710d3c in std::__1::equal_to<User>::operator()(User const&, User const&) const+0x12c (a.out:
        ↪ arm64+0x100004d3c)
    #4 0x10470e3f0 in bool test_cond<User, std::__1::equal_to<User> >(std::__1::basic_string<char, std::
        ↪ __1::char_traits<char>, std::__1::allocator<char> > const&, User const&, User const&, std::
        ↪ __1::equal_to<User>, bool)+0x138 (a.out:arm64+0x1000023f0)
    #5 0x10470ddb8 in main+0x200 (a.out:arm64+0x100001db8)
    #6 0x1048890f0 in start+0x204 (dyld:arm64e+0x50f0)

previously allocated by thread T0 here:
    #0 0x104c7a6b4 in wrap__Znwm+0x74 (libclang_rt.asan_osx_dynamic.dylib:arm64e+0x4a6b4)
    #1 0x10470ed90 in User::User(char const*, char const*)+0x34 (a.out:arm64+0x100002d90)
    #2 0x10470e2a4 in User::User(char const*, char const*)+0x28 (a.out:arm64+0x1000022a4)
    #3 0x10470dd74 in main+0x1bc (a.out:arm64+0x100001d74)
    #4 0x1048890f0 in start+0x204 (dyld:arm64e+0x50f0)

SUMMARY: AddressSanitizer: heap-use-after-free (a.out:arm64+0x1000045d8) in std::__1::basic_string<char,
    ↪ std::__1::char_traits<char>, std::__1::allocator<char> >::__is_long() const+0x64
```

Explain what the problem is; where in the execution does the program crash, and what in the code causes the problem? Explain both the fatal error which directly causes the crash, and the underlying problem in the code that leads to this error.

Also show how to change the class `User` so that the class works as expected, and this kind of error cannot happen.

4. Java has two operations for comparing objects for equality: the equality operator == and the method `Object.equals(Object)`. The following Java method shows the common semantics

```
void compareObjects(Object a, Object b) {
  if (a == b) {
    System.out.println("a and b refer to the same object");
  }

  if (a != null && a.equals(b)) {
    System.out.println("the values of a and b are equal");
  }
}
```

This problem is about how the corresponding comparisons are implemented in C++.

a) Show how one, in a C++ function, can determine if two parameters, `a` and `b`, are *the same object*. (I.e., give an expression that corresponds to `a == b` in Java[1].)

b) Show how one, in a C++ function, can determine if two parameters, `a` and `b`, have *the same value*. (I.e., give an expression that corresponds to `a.equals(b)` in Java.) You may assume that the comparison is possible for the type of `a` and `b`.

c) Give a C++ implementation that corresponds to the Java method `compareObjects`. You may limit the implementation to what is necessary to make the following program work, but state clearly what assumptions and limitations you make and what is required of the types for which your implementation of `compareObjects` works.

Main program:

```
#include <string>
#include <vector>
#include <utility>
#include "compareObjects.h"
int main()
{
  int x{10};
  int y{10};
  int& r=x;

  compareObjects("x,x", x, x);
  compareObjects("x,y", x, y);
  compareObjects("x,r", x, r);
  compareObjects("y,r", y, r);

  std::vector<int> v1{x,y};
  std::vector<int> v2{10,r};
  std::vector<int> v3{10,5};

  compareObjects("v1,v2)", v1, v2);
  compareObjects("v1,v3)", v1, v3);

  std::pair<int,std::string> p1{10,"Hello"};
  std::pair<int,std::string> p2{10,"Hello"};
  std::pair<int,std::string> p3{10, "Hej"};

  compareObjects("p1,p2", p1,p2);
  compareObjects("p1,p3", p1,p3);
}
```

Expected output:

```
compareObjects(x,x):
- the objects are equal
- the same object
compareObjects(x,y):
- the objects are equal
- different objects
compareObjects(x,r):
- the objects are equal
- the same object
compareObjects(y,r):
- the objects are equal
- different objects
compareObjects(v1,v2)):
- the objects are equal
- different objects
compareObjects(v1,v3)):
- the objects are not equal
- different objects
compareObjects(p1,p2):
- the objects are equal
- different objects
compareObjects(p1,p3):
- the objects are not equal
- different objects
```

Note: in the above example, compareObjects take an extra string argument to make the expected output clearer. You may leave this out and just give an implementation similar to the Java method.

[1] but disregarding Java's special semantics for comparison of primitive types.

5. In C++ one can create a type that can be used just like an int or some other type. Consider the following example program that uses a user-defined type `Foo`.

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

std::vector<Foo> example1()
{
    std::cout << "example1:\n";
    constexpr auto sz = 10;
    std::vector<Foo> v(sz);
    std::iota(begin(v), end(v), 1);
    std::ostream_iterator<int> it(std::cout, " ");
    std::copy(begin(v), end(v), it);
    std::cout << '\n';

    return v;
}

template <typename F>
auto apply(const F& f) -> decltype(f())
{
    return f();
}

void example2(const std::vector<Foo>& v) {
    std::cout << "example2:\n";
    std::vector<Foo> w;
    std::transform(begin(v), end(v), back_inserter(w), apply<Foo>);

    std::ostream_iterator<int> it(std::cout, " ");
    std::copy(begin(v), end(v), it);
    std::cout << '\n';
    std::copy(begin(w), end(w), it);
    std::cout << '\n';
}

int main()
{
    auto v = example1();
    example2(v);
}
```

Your task is to implement such a class `Foo`.

a Write a class Foo such that `example1` compiles and runs correctly. The expected output is

```
example1:
1 2 3 4 5 6 7 8 9 10
```

b Make the necessary additions to `Foo` to make `example2()` work. The expected output is

```
example2:
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
```

Only implement what is necessary for the examples to work. For each member and operation, briefly comment on what it does and why it is needed.

If you answer both a) and b) you may answer with a single version of the class `Foo`, as long as you clearly indicate which parts are required by which subproblem.

Appendix

Some standard algorithms that appear in the problems:

`std::accumulate`:

```
template< class InputIt, class T >
T accumulate( InputIt first, InputIt last, T init );
```

Computes the sum of the given value init and the elements in the range `[first, last)`, using `operator+` to sum up the elements.

Returns the sum of the given value and elements in the given range.

`std::iota`:

```
template< class ForwardIt, class T >
void iota( ForwardIt first, ForwardIt last, T value );
```

Fills the range `[first, last)` with sequentially increasing values, starting with value and repetitively evaluating `++value`.

`std::transform`:

```
template< class InputIt,
          class OutputIt,
          class UnaryOperation >
OutputIt transform( InputIt first1,
                    InputIt last1,
                    OutputIt d_first,
                    UnaryOperation unary_op );

template< class InputIt1,
          class InputIt2,
          class OutputIt,
          class BinaryOperation >
OutputIt transform( InputIt1 first1,
                    InputIt1 last1,
                    InputIt2 first2,
                    OutputIt d_first,
                    BinaryOperation binary_op );
```

`std::transform` applies the given function to a range and stores the result in another range, keeping the original elements order and beginning at `d_first`.
The unary operation `unary_op` is applied to the range defined by `[first1, last1)`.
The binary operation `binary_op` is applied to pairs of elements from two ranges: one defined by `[first1, last1)` and the other beginning at `first2`.
`unary_op` and `binary_op` must not invalidate any iterators, including the end iterators, or modify any elements of the ranges involved.
The return value is an output iterator to the element past the last element transformed.

`std::lower_bound`:

```
template< class ForwardIt, class T >
ForwardIt lower_bound( ForwardIt first, ForwardIt last, const T& value );

template< class ForwardIt, class T, class Compare >
ForwardIt lower_bound( ForwardIt first, ForwardIt last, const T& value, Compare comp );
```

`std::lower_bound` returns an iterator pointing to the first element in the range $[first, last)$ that is not less than (i.e. greater or equal to) value, or last if no such element is found.

The range $[first, last)$ must be partitioned with respect to the expression `element < value` or `comp(element, value)`, i.e., all elements for which the expression is true must precede all elements for which the expression is false. A fully-sorted range meets this criterion.

The first version uses `operator<` to compare the elements, the second version uses the given comparison function `comp`.