Examination
EDAF30 Programming i C++

2023-01-11, 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 $6 + 8 + 13 + 5 + 13 + 5 = 50$ 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.)

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. When compiling the following classes:

```
class Foo {
  public:
    Foo() = default;
    Foo(int x) : val{x} {}
    virtual ~Foo() = default;
    virtual int getVal() const { return val; }

  private:
    int val{0};
};

class ScaledFoo : public Foo {
  public:
    ScaledFoo(int x, int factor = 1) : Foo(x), f{factor} {}
    virtual int getVal() override { return f * Foo::getVal(); }
  private:
    int f;
};
```

the compiler gives the error:

```
example.cc|15 col 17 error| 'getVal' marked 'override' but does not override
|| any member functions
||     virtual int getVal() override { return f * Foo::getVal(); }
||                 ^
```

a) Explain what the problem in the code is and why that causes this error to be generated.

b) Show how the class `ScaledFoo` must be changed to compile and work as intended.
The expected behaviour is shown in the following example:

```
void print_value(const Foo& f)
{
    std::cout << "value: " << f.getVal() << '\n';
}

int main()
{
    Foo f(42);
    print_value(f);

    ScaledFoo sf(17,10);
    print_value(sf);
}
```

which is expected to output:

```
value: 42
value: 170
```

2. Type casts are error-prone, and one potential problem is when they perform *narrowing conversions* which lose information. Of the named casts in modern C++, `static_cast` is the safest as it only allows casting between compatible data types, but as it does not check the values it can still lose information.

   Your task is to implement a safer alternative, `check_cast`, which performs a `static_cast` if it can be done without losing information but throws an `std::illegal_argument` if the cast is narrowing. Two examples that show the expected behaviour:

```
#include <stdexcept>
#include <iostream>
#include "check_cast.h"

void example1(double d)
{
    try {
        std::cout << d << " --> " << check_cast<int>(d) << "\n";
    } catch(std::invalid_argument &e){
        std::cout << "casting " << d << " to int: " << e.what() << "\n";
    };
}

void example2(int i)
{
    try {
        std::cout << i << " --> " << check_cast<short>(i) << "\n";
    } catch(std::invalid_argument &e){
        std::cout << "casting " << i << " to short: " << e.what() << "\n";
    };
}

int main()
{
    std::cout << "example1:\n";
    example1(7);
    example1(7.8);

    std::cout << "example2:\n";
    example2(32000);
    example2(33000);
}
```

   The expected output is:

```
7 --> 7
7.8 --> casting 7.8 to int: narrowing cast
example2:
32000 --> 32000
33000 --> casting 33000 to short: narrowing cast
```

   Implement `check_cast<T>` so that it returns the result of a `static_cast<T>` if that is equal to the original value, or throws a `std::invalid_argument` otherwise.

   Hint: Note that the function template has two type parameters (the type to cast to and the type of the original value), but the second type is not given explicitly as it can be derived from the function argument.

3. The algorithm `std::generate()` fills a collection with values by traversing it and assigning each element with the result of calling a function. A possible implementation of this algorithm is

```
template <typename FwdIt, typename Gen>
void generate(FwdIt first, FwdIt last, Gen gen)
{
    for(FwdIt i = first; i != last; ++i) {
        *i = gen();
    }
}
```

   A programmer has attempted to implement a variant of this algorithm where the container object is passed instead of an iterator range. It looks as follows:

```
template <typename Con, typename Gen>
void generate2(Con c, Gen gen)
{
    for(auto& x : c) {
        x = gen();
    }
}
```

   Unfortunately it does not seem to work, when tested with the following data structure and generator function:

```
template <typename T>
struct Vektor{
    explicit Vektor(int s) :sz{s}, e{new T[sz]} {}
    ~Vektor() {delete[] e;}
    T* begin() {return sz > 0 ? e : nullptr;}
    T* end() {return begin()+sz;}
    const T* begin() const {return sz > 0 ? e : nullptr;}
    const T* end() const {return begin()+sz;}
    int sz;
    T* e;
};

template <typename T>
struct Counter{
   Counter(T start, T stride) :s{start-stride},d{stride} {}
   T operator()() {return s+=d;}
private:
   T s;
   T d;
};

void test()
{
    Vektor<int> a(10);
    auto c = Counter<int>(100,10);
    generate2(a, c);

    for(auto x : a) {
        cout << x << " ";
    }
    cout << endl;
}
```

   An execution of `test()` produces the output:

```
23785520 0 120 130 140 150 160 170 180 190
```

   and then crashes with a segmentation fault, instead of the expected

```
100 110 120 130 140 150 160 170 180 190
```

Continues on the next page...

To debug this, a version with `std::vector` was written:

```
void test2()
{
    std::vector<int> a(10);
    auto c = Counter<int>(100,10);
    generate2(a, c);

    print(a);
}
```

but this does not work either, now the program exits successfully but the output is:

```
0 0 0 0 0 0 0 0 0 0
```

instead of the expected

```
100 110 120 130 140 150 160 170 180 190
```

a) Explain what the problem is. Why is the output wrong and what causes the crash?

b) Can you fix the problem, so that `test()` executes without errors and produces the expected output, by only changing `Vektor`? If yes, show how. If no, motivate.

c) Can you fix the problem, so that `test()` executes without errors and produces the expected output, by only changing `generate2`? If yes, show how. If no, motivate.

d) Can you fix the problem, so that `test()` executes without errors and produces the expected output, by only changing `Counter`? If yes, show how. If no, motivate.

e) In `Counter`, the members are declared

```
T s;
T d;
```

If the declarations were changed to

```
T s{};
T d{};
```

would it change the behaviour of the class in any way? Motivate your answer.

4. The standard algorithms and function objects are very useful for expressing computations on sequences of data. In this problem, we want to compute the nesting level of parentheses for each position in an expression. Implement a class `paren_depth` such that the following program

```
#include <algorithm>
#include <iostream>
#include <vector>
void example()
{
    std::string exp{"a + (b * (c + d) / (b + 2)) + e"};

    std::vector<int> depths;
    std::transform(begin(exp), end(exp), std::back_inserter(depths), paren_depth{});

    std::cout << exp << '\n';
    for (auto d : depths) std::cout << d;
    std::cout << '\n';
}
```

gives the output

```
a + (b * (c + d) / (b + 2)) + e
0000111112222222111222222210000
```

You may assume that the input contains properly nested parentheses.

5. 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`.

6. A std::map is an associative container that lets the user efficiently find a value associated with a particular key, but it does not support reverse lookup – finding the key for a particular value. Your task is to implement a function template, `reverse_lookup`, for doing reverse lookup in a std::map:

```
template<typename Key, typename Value>
typename std::map<Key, Value>::iterator
reverse_lookup(std::map<Key,Value>&, const Value&)
```

   It should work with the following example:

```
void do_lookup(std::map<int, std::string>& m, const std::string& s)
{
    auto it = reverse_lookup(m, s);
    if(it != m.end()){
        std::cout << "found " << it->first << '\n';
    } else {
        std::cout << "found nothing\n";
    }
}

void example()
{
    std::map<int, std::string> m;
    m.emplace(10, "Test");
    m.emplace(12, "Testing");
    m.emplace(15, "something");
    m.emplace(5, "something else");

    do_lookup(m, std::string{"Testing"});
    do_lookup(m, std::string{"Foobar"});
}
```

   which is expected to output

```
found 12
found nothing
```

   Answer with the implementaion of `reverse_lookup`.

Appendix

Some standard algorithms that appear in the problems:

`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`.

`std::sort`:

```
template< class RandomIt >
void sort( RandomIt first, RandomIt last );

template< class RandomIt, class Compare >
void sort( RandomIt first, RandomIt last, Compare comp );
```

Sorts the elements in the range [first, last) in non-descending order. The order of equal elements is not guaranteed to be preserved.

In the first version, elements are compared using `operator<`. In the second version, elements are compared using the given binary comparison function comp.

A sequence is sorted with respect to a comparator `comp` if for any iterator `it` pointing to the sequence and any non-negative integer n such that `it + n` is a valid iterator pointing to an element of the sequence, `comp(*(it + n), *it)` (or `*(it + n) < *it`) evaluates to `false`.