Examination
EDAF50 C++ Programming

2023-03-17, 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 $10 + 5 + 5 + 12 + 11 + 7 = 50$ 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. Consider the following class hierarchy and functions:

```
#include <memory>
#include <iostream>

class person{
public:
    person(const std::string& fname, const std::string& lname);
    virtual ~person() =default;
    person(const person&) =delete;
    void operator=(const person&) =delete;

    virtual std::string to_string() const;
    const std::string& first_name() const {return fn;}
    const std::string& last_name() const {return ln;}
private:
    std::string fn;
    std::string ln;
};

person::person(const std::string& f, const std::string& l) :fn(f), ln(l) {}

class student :public person{
    using person::person;
    virtual std::string to_string() const override;
};

class teacher :public person{
    using person::person;
    virtual std::string to_string() const override;
};

std::string person::to_string() const
{
    return last_name() + ", " + first_name();
}

std::string student::to_string() const
{
    return person::to_string() + " (student)";
}

std::string teacher::to_string() const
{
    return person::to_string() + " (teacher)";
}

void print(const person& p)
{
    std::cout << p.to_string() << '\n';
}

void print(const std::unique_ptr<person>& p)
{
    std::cout << p->to_string() << '\n';
}
```

```
const person& make_person(const std::string& fname, const std::string& lname)
{
    return person(fname, lname);
}

const person& make_student(const std::string& fname, const std::string& lname)
{
    student res(fname, lname);
    return res;
}

std::unique_ptr<person> make_teacher(const std::string& fname, const std::string& lname)
{
    return std::unique_ptr<person>(new teacher(fname, lname));
}
```

For each of the following examples,

- state if the function works correctly or has undefined behaviour (answer "Correct" or "UB"), and
- give a brief motivation of your answer.

a)

```
void example1()
{
    const auto& p = make_person("Test", "Testsson");
    print(p);
}
```

b)

```
void example2()
{
    const auto& s = make_student("Test", "Testsson");
    print(s);
}
```

c)

```
void example3()
{
    auto t = make_teacher("Test", "Testsson");
    print(t);
}
```

d)

```
void example4()
{
    teacher t("Test", "Testsson");
    print(t);
}
```

2. A programmer is trying to understand the relationship between a pointer and a built-in array in C++, and has written a small test program:

```
#include <iostream>

int main (){
    int arr[5] = {1,2,3,4,5};

    std::cout << arr << '\n';          //Line 1
    std::cout << &arr << '\n';         //Line 2

    std::cout << sizeof(arr) << '\n';  //Line 3
    std::cout << sizeof(&arr) << '\n'; //Line 4
}
```

When executed, it printed

```
0x7ffeba483f90
0x7ffeba483f90
20
8
```

The programmer is confused: having heard things like "an array is just a pointer to the first element" and "arrays decay to pointers", and seeing that `arr` and `&arr` are the same (pointer) value (lines 1 and 2), the programmer expected lines 3 and 4 to print the same size, but they are different. Explain the numbers printed by lines 3 and 4 and why they are different.

3. Consider this small program

```
#include <iostream>
using std::cout;
using std::endl;

void print(char* a, int first, int last)
{
    for(int pos = first; pos =! last; ++pos){
        cout << a[pos];
    }
    cout << endl;
}

int main()
{
    char str[] = "Hello, world.";

    print(str, 0, 5);
}
```

The code compiles and runs, but does not produce any output. The programmer expected it to output `Hello`. When compiling with `-Wall`, the compiler gives the warning

```
example.cc: In function 'void print(char*, int, int)':
example.cc:7:33: warning: suggest parentheses around assignment used as truth value
     for(int pos = first; pos =! last; ++pos){
                          ~~~~^~~~~~
```

Explain why the compiler gives a warning, what happens when the program is executed, and why no output is produced.

4. Your task is to implement a class `Time`, which represents a time in 24 hour format (i.e., one or two digits for the hour ($0-23$) and two digits for the minutes ($0-59$)) that can be read from, and written to a stream, and supports simple arithmetic[1]. The class `Time` shall be written such that the following example function works.

```
#include <iostream>
#include <sstream>
#include <string>
#include "Time.h"

void example()
{
    std::istringstream is("13:15 1:45 lecture\n"
                          "15:00 0:15 break\n"
                          "15,15 1:45 lab\n"
                          "27:00 0:15 break\n"
                          "17:00 0:30 administration\n"
                          "18:00 0:90 commute\n");

    std::string line;
    while (std::getline(is, line)) {
        std::istringstream ss(line);
        Time s;
        Time d;
        std::string event;
        if (ss >> s && ss >> d && std::getline(ss, event)) {
            Time e = s + d;
            std::cout << s << " - " << e << ":" << event << '\n';
        } else {
            std::cout << "[malformed line ignored]\n";
        }
    }
}
```

The example reads agenda records (where each line has the format `start_time duration event`) from a stream, and outputs an agenda on the format `start_time - end_time: event`
When executed, `example()` is expected to print:

```
13:15 - 15:00: lecture
15:00 - 15:15: break
[malformed line ignored]
[malformed line ignored]
17:00 - 17:30: administration
[malformed line ignored]
```

Answer with an implementation of the class `Time` and its required operations. Only implement what is needed by `example()`. For this problem, you must implement the time representation yourself and not use the standard library facilities from `<chrono>`.

Note: The constructor argument to the `istringstream` in `example()` is correct and is a single string literal containing a multi-line string. In C and C++, adjacent string literals are concatenated, so `"hello " "world"` is treated as a single string literal, equivalent to `"hello world"`. This is often used to avoid long lines and make the code more readable.

[1] Note that the type `Time` is very limited to simplify the problem. It does not distinguish between points in time and durations in its semantics for the addition operation. It is also limited in that a time — even if it is intended to represent a duration — cannot be more than 23:59, or be expressed as a number of minutes > 59.

5. We want functionality that lets us iterate over all elements in an iterator range `[first,last)` that are equal to a value `val`. This is to be implemented by a function template `find_all` and a helper class template `result_iter`:

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

Example use: the program

```
#include <vector>
#include <iostream>
void example()
{
    std::vector<int> v{1,3,2,4,3,5,4,6,5,7,3};

    auto it = find_all(begin(v), end(v), 3);

    while(it != end(v)){
        *it = 42;
        ++it;
    }

    for(auto x : v) std::cout << x << " ";
    std::cout << "\n";
}
```

should output `1 42 2 4 42 5 4 6 5 7 42`

Implement the function template `find_all` and the helper class template `result_iter`. You should only implement the functionality required by the above example[2].

As defined by the use, `result_iter` should have a public interface like:

```
template <typename Iter, typename T>
class result_iter {
public:
    // add suitable constructor
    result_iter& operator++();
    T& operator*();
    bool operator!=(Iter) const;
private:
    // add members as needed
};
```

`operator++` should return an iterator to the next element in the range `[first,last)` that is equal to `val`, or – when no such element is found – an iterator that is equal to `last`.

`operator*` should return a reference to an element in the original sequence.

Note that you only need an `operator!=` that compares a `result_iter` to an instance of its underlying interator type, as it is only used in this way: `while (it != end(v))`.

Answer with implementations of the function template `find_all` and the class template `result_iter`.

Hint: Most of the work is done in `result_iter`.

[2] Note that — to simplify the problem — the `result_iter` class does not need to have all the functionality of a standard library iterator. It does not need to have an iterator tag and the type members required by the standard algorithms.

6. The following function uses a lambda expression to implement the predicate *less than* `x`, which is then used in a call to `std::copy_if`.

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

void example1()
{
    std::vector<int> v{1, 2, 3, 4, 5, 6, 7};
    int x = 4;

    auto f = [&x](int val) { return val < x; };

    std::copy_if(begin(v), end(v), std::ostream_iterator<int>(std::cout), f);
    std::cout << "\n";
    x = 7;
    std::copy_if(begin(v), end(v), std::ostream_iterator<int>(std::cout), f);
    std::cout << "\n";
}
```

The result of a lambda expression is an instance of an anonymous class. Your task is to show how the class described by the above lambda expression looks. That is, implement a class `my_less_than` so that the expression

```
auto f = my_less_than(x);
```

is equivalent to

```
auto f = [&x](int val) { return val < x; };
```

That is, the following `example2()` should have the same behaviour as `example1()` when used with your class:

```
void example2()
{
    std::vector<int> v{1, 2, 3, 4, 5, 6, 7};
    int x = 4;

    auto f = my_less_than(x);

    std::copy_if(begin(v), end(v), std::ostream_iterator<int>(std::cout), f);
    std::cout << "\n";
    x = 7;
    std::copy_if(begin(v), end(v), std::ostream_iterator<int>(std::cout), f);
    std::cout << "\n";
}
```

(The only difference between `example1()` and `example2()` is the expression used to initialize the variable `f`.) When executed, both `example1` and `example2` should output

```
123
123456
```

Answer with the definition of the class `my_less_than`.