Examination
EDAF50 C++ Programming

2020-03-19, 14:00–19:00

Aid at the exam: You may use any written material available to you, printed or online. You *may not* in any way ask questions about or discuss the exam with another person (other than the course coordinator).
Assessment (preliminary): the questions give $32 + 6 + 12 = 50$ points. You need 25 points for a passing grade (3/25, 4/33, 5/42). Please note that as this is an extraordinary exam, both the scoring and grading are subject to change.
You must show that you know C++ and that you can use the C++ standard library. "C solutions", and idiomatic C++ solutions that for no good reason 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. This also applies to notes and comments in the code.
The expected solution to problem 1 is complete source code that compiles and runs the provided tests without warnings, errors of failures. Partial credit will be given for an incomplete solution together with a discussion of the problems and a discussion or sketch of possible solutions that you didn't have enough time to implement.

Submission:

- For problem 1, you are expected to hand in an archive (`.tar`, `.tar.gz`, or `.zip`) with a complete solution; source code and a `Makefile` that successfully builds and runs the provided test program. Any notes or comments can be placed either as comments in the code or in the document with the solutions to problems 2 and 3.

  If you submit a partial solution, please remove the test cases for the subproblems that you haven't solved. You may not change the test cases for the subproblems you do submit solutions for. Non-working, partial solutions should be clearly marked as such and placed separately so that the working parts do compile and run cleanly.

- For problems 2 and 3, solutions shall be handed in as a PDF document (or a plain text file if you for some reason cannot create a PDF).

- *Do not* hand in word processor files (e.g., `.doc`, `.docx`, etc.).

- Only use the prescribed archive formats. If we cannot unpack your archive, it will not be marked.

- Do not include any generated files (executables or object files) in your source code archive.

- Write your name, personal number, and STIL ID in all files you submit.

  Email your solution to `sven.robertz@cs.lth.se` *no later than 19.00.* The timestamp from the receiving LU mail server will be used as submission time. Please submit your solution on time. The subject should be *EDAF50 exam submission*, the message should contain your name, STIL ID and personal number, and your solutions attached as a PDF (or text) and a source code archive, named

  – EDAF50-studentid.pdf

  – EDAF50-studentid.tar (or possibly .tar.gz or .zip)

  where `studentid` is your STIL ID. Please don't use any spaces, swedish or special characters in the file names (that also applies to the files within the archive.) If your email provider does not allow attaching archives, first make sure that there are no executable files or object files in the archive. As a backup option you may attach the source files and Makefile separately.

1. Associative containers like `std::map` is a very useful tool for efficiently looking up a value associated with a key. Sometimes, one needs to also do the reverse lookup: find the key for a certain value. This can be done in two ways, either do a linear search of the key-value pairs for the value, or first create an inverse map and then do the lookup in that. Which is better depends on the application. For instance, if the map is frequently modified, or linear time complexity is acceptable but the map is too large to copy, the linear search alternative may be better. On the other hand, if the map is rarely modified and lookup must be efficient in both directions, creating and maintaining two maps may be preferable.
   In this problem, you will implement both of these options.

   For subproblems a) and b) you may assume that the values in the map are unique. I.e., no two values in the map are equal and no handling of duplicate values is required in your solution. In subproblem c) you must handle duplicate values, so that reverse lookup may return more than one key.

   In all subproblems, you may assume that the key type is not the same as the value type (as we will use overloading of `at` and `find`).

   Please note that you may need *include guards* in your header files.

   a) Write a function template that, given a `map<K,V>` creates the inverse `map<V,K>`, where each value is mapped to the corresponding key. The following unit test should pass:

```
bool test_invert()
{
    cout << "test_invert:\n";
    std::map<int, std::string> m{{1,"kalle"}, {2,"anka"}};

    assert(m[1] == "kalle");
    assert(m[2] == "anka");

    auto inv = invert_map(m);

    auto res = true;

    if(inv["kalle"] != 1) {res = false; cout << "wrong value for kalle\n";}
    if(inv["anka"] != 2) {res = false; cout << "wrong value for anka\n";}

    std::map<int, char> n{{1,'a'}, {2,'b'}};

    assert(n[1] == 'a');
    assert(n[2] == 'b');

    auto inv2 = invert_map(n);

    if(inv2['a'] != 1) {res = false; cout << "wrong value for a\n";}
    if(inv2['b'] != 2) {res = false; cout << "wrong value for b\n";}

    cout << "test_invert done\n";
    return res;
}
```

b) Write a class template `bidir_map<K,V>` that works just as a `std::map` but adds an overload of the member function `at` that supports lookup of the key for a given value.

The following unit tests should pass:

```
/* helper function: returns true iff m.at(k) == v, prints error message otherwise */
bool test_lookup(const std::map<std::string, int>& m, const std::string &k, int v);

bool test_forward(const std::map<std::string, int>& m)
{
    bool res {true};
    cout << "test_forward:\n";
    res |= test_lookup(m,"a",1);
    res |= test_lookup(m,"b",2);
    res |= test_lookup(m,"c",1);
    res |= test_lookup(m,"d",2);
    res |= test_lookup(m,"e",1);
    res |= test_lookup(m,"f",3);

    auto it = m.find("a");
    if(it == m.end() || it->second != 1) {
        cout << "wrong value for find(\"a\")\n";
        res = false;
    }
    return res;
}
bool test_reverse_at(const bidir_map<std::string, int>& m)
{
    cout << "test_reverse_at:\n";
    auto res = true;
    auto x1 = m.at(1);
    std::string ones = "ace";
    if(ones.find(x1) == std::string::npos) {
        cout << "Wrong key for 1\n";
        res = false;
    }

    auto x2 = m.at(2);
    std::string twos = "bd";
    if(twos.find(x2) == std::string::npos) {
        cout << "Wrong key for 2\n";
        res = false;
    }
    return res;
}
```

where the test functions will be run on a map created by the function

```
bidir_map<std::string,int> setup()
{
    bidir_map<std::string, int> m{ {"a",1}, {"b",2}, {"c",1},
                                   {"d",2}, {"e",1}, {"f",3}};
    return m;
}
```

The reverse `at` should have the same semantics as `std::map::at`, but be implemented using linear search of the map.

Please note that we assume that the key type and the value type are different, as we use overloading of `at` to distinguish the normal and reverse lookup.

Note that only `at` is used for the reverse lookup, and not `operator[]`, as we only care about lookup, and we don't want to default construct keys for new values.

c) Add a reverse version of the member function `find`, that returns a pair of iterators representing a range of keys that are mapped to the given value, or an empty range if the value is not in the map.

Just as for the inverse `at`, the inverse `find` should do a linear search of the map.

Dereferencing the iterator should give a reference to the actual key in the map, no copying of elements should be done.

Hint: implement a new, separate, iterator type for the result of reverse find.

The following unit test should pass:

```
bool test_reverse_find(const bidir_map<std::string, int>& m)
{
    cout << "test_reverse_find:\n";
    auto res = true;
    auto p1 = m.find(1);
    std::vector<std::string> vone;
    std::copy(p1.first, p1.second, back_inserter(vone));
    if(vone != std::vector<std::string>{"a","c","e"}) {
        cout << "Wrong keys for 1\n";
        res = false;
    }

    auto p2 = m.find(2);
    std::vector<std::string> vtwo;
    std::copy(p2.first, p2.second, back_inserter(vtwo));
    if(vtwo != std::vector<std::string>{"b","d"}){
        cout << "Wrong keys for 2\n";
        res = false;
    }

    auto pe = m.find(4);
    if(pe.first != pe.second){
        cout << "Wrong: returned key for 4\n";
        res = false;
    }
    return res;
}
```

2. Consider this code:

```
#include <iostream>

class Base
{
public:
  virtual void foo() const { std::cout << "Base's foo!" << std::endl; }
};

class Derived : public Base
{
public:
  void foo() { std::cout << "Derived's foo!" << std::endl; }
};

int main()
{
  Base* o1 = new Base();
  Base* o2 = new Derived();
  Derived* o3 = new Derived();

  o1->foo();
  o2->foo();
  o3->foo();
}
```

The code compiles without errors, but the output is:

```
Base's foo!
Base's foo!
Derived's foo!
```

where the programmer expected

```
Base's foo!
Derived's foo!
Derived's foo!
```

as `Base::foo()` is `virtual` and `o2` points to a `Derived` object.

Explain the behaviour. Why does not `o2->foo()` call `Derived::foo`? Why does `o3->foo()` call `Derived::foo`?

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.