Examination
EDAF50 C++ Programming

2024-05-31, 08:00–13:00

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

Assessment (preliminary): the questions give $13 + 9 + 9 + 5 + 9 + 5 = 50$ 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.)

The last page has an appendix describing some standard library facilities 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. Consider the following program:

```
#include <algorithm>
#include <initializer_list>
#include <iostream>

template <typename T>
class Vektor {
  public:
    Vektor(size_t sz) : size{sz}, p{new T[size]{}} {}
    Vektor(std::initializer_list<T> l) : Vektor(l.size())
    {
        std::copy(l.begin(), l.end(), p);
    }
    ~Vektor() { delete[] p; }
    T& operator[](size_t i) { return p[i]; }
    size_t length() const { return size; }
    T* begin() { return p; }
    T* end() { return p + size; }
    const T* cbegin() const { return p; }
    const T* cend() const { return p + size; }

  private:
    size_t size;
    T* p;
};

template <typename T>
void add(const Vektor<T> c1, const Vektor<T> c2, Vektor<T> c3)
{
    auto it1 = c1.cbegin();
    auto it2 = c2.cbegin();
    auto it3 = c3.begin();

    while ((it1 != c1.cend() || it2 != c2.cend()) && it3 != c3.end()) {
        T tmp1{};
        if (it1 != c1.cend()) {
            tmp1 += *it1;
            ++it1;
        }
        T tmp2{};
        if (it2 != c2.cend()) {
            tmp2 += *it2;
            ++it2;
        }
        *it3 = tmp1 + tmp2;
        ++it3;
    }
}

int main()
{
    Vektor<int> v1{1, 2, 3, 4, 5, 6};
    Vektor<int> v2{10, 20, 30, 40};
    Vektor<int> v3(v1.length());

    add(v1, v2, v3);
    for (auto e : v3) {
        std::cout << e << " ";
    }
    std::cout << std::endl;
}
```

the problem continues on the next page...

When the program is excuted it produces the output

```
923586848 38199 2043 44 5 6
```

instead of the expected:

```
11 22 33 44 5 6
```

a) Explain what happens when the program is executed.

b) Can the program be corrected so that it produces the expected output *by only changing the function template* `add`?

If yes, show a corrected function template. If no, motivate.

c) Can the program be corrected so that it produces the expected output *by only changing the class template* `Vektor`?

If yes, show a corrected class template. If no, motivate.

d) In the function template `add`, the temporary variables in the loop are declared as

```
T tmp1{};
T tmp2{};
```

Would it change the behaviour of the function if they were instead declared as

```
T tmp1;
T tmp2;
```

If yes, how? If no, motivate.

2. Your task is to create an algorithm `copy_nth`, which takes an input iterator range, an output iterator, and an integer *n*, and copies every *nth* element, starting with the first, from the input range to the output. The example

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

int main()
{
    std::vector<int> v(30);
    std::iota(begin(v), end(v), 1);

    auto out1 = std::ostream_iterator<int>(std::cout, ", ");
    copy_nth(begin(v), end(v), out1, 4);
    endl(cout);

    std::stringstream ss{"one two three four five six seven"};

    std::istream_iterator<std::string> it(ss);
    std::istream_iterator<std::string> ie{};
    auto out2 = std::ostream_iterator<std::string>(std::cout, ", ");
    copy_nth(it, ie, out2, 3);
    endl(cout);
}
```

is expected to produce the output

```
1, 5, 9, 13, 17, 21, 25, 29,
one, four, seven,
```

For full credit your solution should use the standard algorithm `std::copy_if` with a function object as a stateful predicate. Just like `std::copy_if`, your `copy_nth` shall return an iterator to one past the last element written.

Hint: you can use `std::iterator_traits<It>::value_type` to get the type referenced by an iterator type `It`.

3. One of the lab assignments was the *tag remover* that removed all HTML tags from a text. In the lab, the `TagRemover` class read from a `std::istream` and stored the processed text so that it could later be output to a `std::ostream` with the member function `print`. The main program was

```
int main() {
  TagRemover tr(std::cin); // read from cin
  tr.print(std::cout); // print on cout
}
```

It can be observed that – if we assume that the input is valid HTML with no mismatched tags or wrong special characters – the tags can be removed and special characters replaced "on the fly" without storing the text. Using standard library facilities, that can be done with the algorithm `std::copy` and an `std::istream_iterator<token>`, with a custom class `token` wich has an `operator>>` that skips tags, reads special character codes as the character they represent, and all other characters as themselves.

That is, the main program can be a single function call:

```
void remove_html(std::istream& is, std::ostream& os)
{
    std::istream_iterator<token> iit(is);
    std::istream_iterator<token> ie{};
    std::ostream_iterator<char> oi(os);
    std::copy(iit, ie, oi);
}

int main() {
  remove_html(std::cin, std::cout); // read from cin, write to cout
}
```

Your task is to implement the class `token` such that `html_remover(std::istream&, std::ostream&)` works as specified: it reads from its input stream and copies the text with all HTML removed to the output stream.

- All tags should be removed from the output. A tag starts with a < and ends with a >.
- You can assume that there are no nested tags and that all < have a matching >.
- Special characters should be translated. You can assume that they are correctly formatted, and you only need to handle the three special characters in the example given below.
- Line breaks in the input should be preserved.

Example: for the program

```
void example()
{
  std::string html =
      "<html><body><H1>Test</H1>\n"
      "<p>This is a test with a list</p>\n"
      "<ul><li>item1\n"
      "<li>item2\n"
      "<li>item3</ul>\n"
      "<p>and some text with special chars (&lt;, &amp;, and &gt;) to be translated</p>\n"
      "</body></html>";

  std::istringstream is(html);
  remove_html(is, std::cout);
}
```

the expected output is

```
Test
This is a test with a list
item1
item2
item3
and some text with special chars (<, &, and >) to be translated
```

Hint: the class `token` needs to have an implicit conversion to `char` for the output of `token` objects through an `std::ostream_iterator<char>` to work.

4. In the main funciton of the tag remover example, a stringstream variable was created and then passed to `remove_html`.

```
  std::istringstream is(html);
  remove_html(is, std::cout);
```

A programmer tried to make the code shorter by removing the local variable and creating the `istringstream` directly as an argument to `remove_html` but this code does not compile:

```
void remove_html(std::istream& is, std::ostream& os);

void example()
{
  std::string html =
      "<html><body><H1>Test</H1>\n"
      "<p>This is a test with a list</p>\n"
      "<ul><li>item1\n"
      "<li>item2\n"
      "<li>item3</ul>\n"
      "<p>and some text with special chars (&lt;, &amp;, and &gt;) to be translated</p>\n"
      "</body></html>";

  remove_html(std::istringstream(html), std::cout);
}
```

The error message from the compiler is

```
tag_remover.cc|99 col 5 error| no matching function for call to 'remove_html'
||     remove_html(std::istringstream(html), std::cout);
||     ^~~~~~~~~~~
tag_remover.cc|75 col 6 info| candidate function not viable: expects an lvalue
||                             for 1st argument
|| void remove_html(std::istream& is, std::ostream& os)
||      ^
|| 1 error generated.
```

Explain the compiler error. Why is the first argument required to be an lvalue, and why does this change cause this error?

5. 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 are correct or not. For each function, *state if it has problems related to resource management* ("Yes" if there are problems and "No" if correct) *and motivate* why it leaks memory (or has undefined behaviour) or how the memory is reclaimed.

```
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';
}
```

6. A programmer has written a small test program to learn how to read strings with `<iostream>`.

```
#include<iostream>

using std::cin;
using std::cout;
using std::endl;

int main() {
    const int prefix_len = 8;

    int nbr;
    char prefix[prefix_len];

    cout << "Enter number of times to loop:" << endl;
    cin >> nbr;

    cout << "Enter a prefix (max length: " << prefix_len << " chars):" << endl;
    cin >> prefix;

    cout << "Looping " << nbr << " times." << endl;
    while(nbr--) {
        char txt[100];
        cout << "Enter a text:" << endl;
        cin >> txt;
        cout << prefix << " : " << txt << endl;
    }
    cout << "Program finished" << endl;
    return 0;
}
```

The following test run works as expected:

```
Enter number of times to loop:
3
Enter a prefix (max length: 8 chars):
test
Looping 3 times.
Enter a text:
aaaa
test : aaaa
Enter a text:
bb
test : bb
Enter a text:
ccccc
test : ccccc
Program finished
```

but this test run gives an unexpected result:

```
Enter number of times to loop:
5
Enter a prefix (max length: 8 chars):
abcdefgh
Looping 0 times.
Program finished
```

The programmer expected that the program should ask for a string five times, but it exits directly. Explain what happens in the second run with the unexpected result.
Hint: Think about how a C string is represented in memory.

Appendix: Some standard library facilities:

```
template< class InputIt, class OutputIt, class UnaryPred >

OutputIt copy_if( InputIt first, InputIt last,
                  OutputIt d_first, UnaryPred pred );
```

`std::copy_if` copies the elements, for which the predicate `pred` returns true, in the range, defined by `[first, last)`, to another range beginning at `d_first` (copy destination range).
This copy algorithm is stable: the relative order of the elements that are copied is preserved. If `[first, last)` and the copy destination range overlaps, the behavior is undefined.
`std::copy_if` returns an output iterator to the element in the destination range, one past the last element copied.

```
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`.

```
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.

```
template< class T,
          class CharT = char,
          class Traits = std::char_traits<CharT>,
          class Distance = std::ptrdiff_t >
class istream_iterator
    : public std::iterator<std::input_iterator_tag, T, Distance, const T*, const T&>
```

`std::istream_iterator` is a single-pass input iterator that reads successive objects of type `T` from the `std::basic_istream` object for which it was constructed, by calling the appropriate `operator>>`. The actual read operation is performed when the iterator is incremented, not when it is dereferenced. The first object is read when the iterator is constructed. Dereferencing only returns a copy of the most recently read object.

The default-constructed `std::istream_iterator` is known as the end-of-stream iterator. When a valid `std::istream_iterator` reaches the end of the underlying stream, it becomes equal to the end-of-stream iterator. Dereferencing or incrementing it further invokes undefined behavior.

```
template< class T,
          class CharT = char,
          class Traits = std::char_traits<CharT> >
class ostream_iterator
    : public std::iterator<std::output_iterator_tag, void, void, void, void>
```

`std::ostream_iterator` is a single-pass `LegacyOutputIterator` that writes successive objects of type `T` into the `std::basic_ostream` object for which it was constructed, using `operator<<`. Optional delimiter string is written to the output stream after every write operation. The write operation is performed when the iterator (whether dereferenced or not) is assigned to. Incrementing the `std::ostream_iterator` is a no-op.