Examination
EDAF50 C++ Programming

2019–03–21, 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 $20 + 6 + 19 + 5 = 50$ 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.
Free-text anwers should be concise but complete, well motivated and written clearly, to the point, and in complete sentences.

Please write on only one side of the paper and hand in your solutions with the papers sorted and facing the same way, as the solutions may be scanned for the marking.

---

1. In programs that handle strings, it is common to do various read-only operations like searching and also to work on or pass around substrings of the original string. That is well supported by `std::string`, but unfortunately many of the operations are inefficient for read-only use, as they require temporary strings to be constructed or copies to be made.

   For instance, a function like `int compare(const std::string&, const std::string&)`, also works for C-style strings (null terminated `const char*`), but then a temporary std::string object is created (as the `const char*` is implicitly converted to `std::string` in the call).

   The function `std::string::substr` returns the substring by value, which also causes a copy. That is reasonable, as `substr` should not modify the string. However, if the user only needs to look at the substring that is inefficient.

   As a solution to these problems the class `string_view` has been added in C++17. A `string_view` is basically a read-only view, that is cheap to copy, of a contiguous sequence of characters. The interface of a `string_view` is similar to that of a `const std::string`, but its representation is in principle only a pointer and a length. String views do not take part in the management of the lifetime of the underlying string, so they are cheap. As the representation is simply a pointer and a length, they also work directly for C-style strings.

   The only drawback is that they must not outlive the underlying string (and if they do and are used after the underlying string is destroyed, it has undefined behaviour).

   Your task is to implement a simplified variant of `string_view` as defined on the following page. Implement the member functions (including constructors) of `string_view`. Also implement the operators

   ```
   bool operator==(const string_view&, const string_view&);
   std::ostream& operator<<(std::ostream&, const string_view&);
   ```

   Implement error handling where possible for the functions where the user passes an index or a length, and document that behaviour (unless already given in the specification), or – for errors that cannot be detected – specify that it has undefined behaviour.

continues on the next page...

```
class string_view{
public:
    using size_type = std::string::size_type;
    using const_iterator = const char*;
    using iterator = const_iterator;

    constexpr static size_type npos = std::string::npos;

    string_view();
    ~string_view() =default;
    string_view(const string_view&) =default;
    string_view(const std::string&);
    string_view(const std::string&, size_type pos, size_type len);
    string_view(const char*, size_type len);
    string_view(const char*);
    bool empty() const;
    size_type size() const;
    const_iterator begin() const;
    const_iterator end() const;
    char operator[](size_type idx) const;
    char at(size_type idx) const;
    string_view substr(size_type pos) const;
    string_view substr(size_type pos, size_type len) const;
    size_type find(char ch, size_type pos=0) const;
    size_type find(string_view s, size_type pos=0) const;
private:
    const char* str;
    size_type sz;
};
```

The member functions should behave just like the corresponding function in `std::string`. For the non-obvious functions, the specifications are given below:

```
/** constructs a string view of a substring of the given std::string
 *  starting at index pos, and with a length of len characters. */
string_view(const std::string&, size_type pos, size_type len);

/** constructs a string view of a null-terminated character array.
 *  If the array contains no null, behaviour is undefined. */
string_view(const char*);

/** returns a string_view starting at pos and to the end. */
string_view substr(size_type pos) const;

/** returns a string_view starting at pos and
 *  with the size of (at most) len. If pos+len is larger
 *  than the total size, a view from  pos to the end is returned. */
string_view substr(size_type pos, size_type len) const;

/** searches, starting at position pos, for the character ch.
 *  returns the index of the first ch
 *  returns npos if not found. */
size_type find(char ch, size_type pos=0) const;

/** searches, starting at position pos, for the character sequence s
 *  returns the index of the first occurrence of s.
 *  returns npos if not found.
 *  Example: with string_view sv("hello, world");
 *           a call sv.find("world") returns 7. */
size_type find(string_view s, size_type pos=0) const;
```

continues on the next page. . .

Hints:

- To find the length (not including the terminating null) of a null-terminated char array, the function `size_t strlen(const char*)` (from `<cstring>`) can be used.
- To get a pointer to the underlying `char[]` of a `std::string`, the member function `c_str()` can be used.
- `std::find` searches for a single element in an iterator range . To find a string in another string (or a sequence of values in a collection), the standard algorithm `std::search` can be used. It has the following declaration

  ```
  template< class ForwardIt1, class ForwardIt2 >
  ForwardIt1 search( ForwardIt1 first, ForwardIt1 last,
                     ForwardIt2 s_first, ForwardIt2 s_last );
  ```

  It searches for the first occurrence of the sequence of elements `[s_first, s_last)` in the range `[first, last)`.
  The return value is an iterator to the beginning of first occurrence of the sequence `[s_first, s_last)` in the range `[first, last)`. If no such occurrence is found, `last` is returned. If `[s_first, s_last)` is empty, `first` is returned.
- To find the distance between two `char*`, you can simply subtract them.

2. a) With classes like `string_view` in problem 1, which contain a non-owning pointer to an array of characters, the user must be careful not to run into problems related to object lifetimes. For instance, given the following function to print a `string_view`

   ```
   void print(string_view s)
   {
       cout << s << "\n";
   }
   ```

   the following two lines work as intended:

   ```
       string_view a("Hello, world!");
       print(a);
   ```

   but the following code has undefined behaviour:

   ```
       string_view b(std::string("Goodbye, world!"));
       print(b);
   ```

   Explain the difference. Why is it OK to create a `string_view` from a string literal, and why does it have undefined behaviour if you instead first convert the string literal to a `std::string`?

   b) The `string_view` has constructors taking both a `std::string` and a `const char*` and we see that the one taking a C-style string only takes a pointer and a length, where the one taking a `const std::string&` has three arguments and takes both a starting offset and a length.
   If we have a `std::string` variable,
   `std::string a{"Hello, world!"};`
   we can create a `string_view` of that variable for the substring `"world"` with the expression
   `string_view(a,7,5)`
   If we instead have the C-style string variable
   `char b[]{"Hello, world!"};`
   show how to create a `string_view` for the substring `"world"` of `b`. Answer with an expression.

3. Consider the following program:

```
#include<iostream>
#include<iomanip>
#include<initializer_list>

template <typename T>
class Vektor{
public:
    Vektor(size_t sz) :size{sz},p{new T[size]{0}} {}
    Vektor(std::initializer_list<T> l) :Vektor(l.size()) {assign(l);}
    ~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;
    void assign(const std::initializer_list<T>& l);
};

template <typename T>
void Vektor<T>::assign(const std::initializer_list<T>& l)
{
    size_t idx{0};

    for(const auto& e : l) {
        p[idx++] = e;
    }
}

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;
    }
}
```

the program continues on the next page…

```
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;
}
```

When the program is excuted it produces the output

```
15523872 0 33 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 function 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.

e) Show how the member function `assign` can be implemented using the algorithm `std::copy` instead of the hand-written loop.

f) Show how the function template `add` can be implemented using standard algorithms (e.g. `std::transform` and `std::copy`) instead of hand-written loops. Be careful to keep the semantics when the vectors have different sizes. Your implementation of `add` should give the expected result for the example program and be consistent with your answers to *b)* and *c)*.

`std::copy` and `std::transform` have the following declarations:

```
template< class InputIt, class OutputIt >
OutputIt copy( InputIt first, InputIt last, OutputIt d_first );

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 );
```

The algorithms return an iterator to the destination range, one past the last element written.

4. The following function does not compile:

```
 1  #include <string>
 2  #include <map>
 3  #include <algorithm>
 4
 5  using  MapStringInt = std::map<std::string, unsigned int>;
 6
 7  void test(MapStringInt& m, const std::string& text)
 8  {
 9      auto mit = std::find(m.begin(), m.end(), text);
10      if (mit == m.end()) m.insert(std::make_pair(text, 1));
11  }
```

An example of a use is

```
#include <iostream>
#include <cassert>

int main()
{
    MapStringInt ml;

    test(ml, "foo");
    test(ml, "bar");
    assert(ml["foo"]== 1);
    assert(ml["bar"]== 1);
    std::cout << "success!\n";
}
```

The compiler gives the error message shown below. *Explain what the problem is and show how to change the function* `test(MapStringInt&, const std::string&)` *to make it work.*

```
In file included from /usr/include/c++/4.9/bits/stl_algobase.h:71:0,
                 from /usr/include/c++/4.9/bits/char_traits.h:39,
                 from /usr/include/c++/4.9/string:40,
                 from program.cc:1:
/usr/include/c++/4.9/bits/predefined_ops.h: In instantiation of 'bool __gnu_cxx
    ::__ops::_Iter_equals_val<_Value>::operator()(_Iterator) [with _Iterator =
    std::_Rb_tree_iterator<std::pair<const std::basic_string<char>, unsigned int>
     >; _Value = const std::basic_string<char>]':
/usr/include/c++/4.9/bits/stl_algo.h:104:50:   required from '_InputIterator std
    ::__find_if(_InputIterator, _InputIterator, _Predicate, std::
    input_iterator_tag) [with _InputIterator = std::_Rb_tree_iterator<std::pair<
    const std::basic_string<char>, unsigned int> >; _Predicate = __gnu_cxx::__ops
    ::_Iter_equals_val<const std::basic_string<char> >]'
/usr/include/c++/4.9/bits/stl_algo.h:162:43:   required from '_Iterator std::
    __find_if(_Iterator, _Iterator, _Predicate) [with _Iterator = std::
    _Rb_tree_iterator<std::pair<const std::basic_string<char>, unsigned int> >;
    _Predicate = __gnu_cxx::__ops::_Iter_equals_val<const std::basic_string<char>
     >]'
/usr/include/c++/4.9/bits/stl_algo.h:3779:50:   required from '_IIter std::find(
    _IIter, _IIter, const _Tp&) [with _IIter = std::_Rb_tree_iterator<std::pair<
    const std::basic_string<char>, unsigned int> >; _Tp = std::basic_string<char
    >]'
program.cc:9:50:   required from here
/usr/include/c++/4.9/bits/predefined_ops.h:191:17: error: no match for 'operator
    ==' (operand types are 'std::pair<const std::basic_string<char>, unsigned int
    >' and 'const std::basic_string<char>')
  { return *__it == _M_value; }
                 ^
```