Examination
EDAF30 Programming i C++

2019–01–09, 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 $16 + 6 + 7 + 7 + 14 = 50$ 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. Answers may be given in swedish or english.

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 a inefficient for read-only use, as they require temporary strings to be constructed or copies to be made.

   For instance, comparisons like `operator==(const std::string&, const std::string&)`, also work 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 of a contiguous sequence of characters that is cheap to copy. 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).

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;
    const char& operator[](size_type idx) const;
    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.
- 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 contains a non-owning pointer to an array of characters and a length, 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 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 (the `main` function is on the next page):

```
#include <cstring>
#include <stdexcept>
#include <memory>

class label{
public:
    virtual const char* cstr() const =0;
};

class fixed_label :public label {
public:
    static constexpr size_t max_size = 7; // max size of actual string
    const char* cstr() const {return s;}
    static std::unique_ptr<fixed_label> make_label(const char*);
private:
    fixed_label(const char* str);
    char s[max_size+1];                    // add one for terminating null
};

class dynamic_label :public label {
public:
    const char* cstr() const {return s.get();}
    static std::unique_ptr<dynamic_label> make_label(const char*);
private:
    dynamic_label(const char* c);
    std::unique_ptr<char[]> s{nullptr};
};

std::ostream& operator<<(std::ostream& os, const label& l)
{
    return os << l.cstr();
}

fixed_label::fixed_label(const char* str)
{
    auto len = std::strlen(str);
    if(len > max_size) throw std::length_error("string too long");
    std::strcpy(s,str);
}

std::unique_ptr<fixed_label> fixed_label::make_label(const char* c)
{
    return std::unique_ptr<fixed_label>(new fixed_label(c));
}

dynamic_label::dynamic_label(const char* c)
{
    auto len = std::strlen(c)+1;
    auto tmp = new char[len];
    std::strncpy(tmp, c, len);
    s = std::unique_ptr<char[]>(tmp);
}

std::unique_ptr<dynamic_label> dynamic_label::make_label(const char* c)
{
    return std::unique_ptr<dynamic_label>(new dynamic_label(c));
}
```

Continues on the next page...

```
std::unique_ptr<label> make_label(const char* s)
{
    auto len = std::strlen(s);
    if(len > fixed_label::max_size) {
        return dynamic_label::make_label(s);
    } else {
        return fixed_label::make_label(s);
    }
}

#include <iostream>
#include <vector>

int main()
{
    std::vector< std::unique_ptr<label> > v;

    v.push_back(make_label("A label"));
    v.push_back(make_label("Longer label"));
    v.push_back(make_label("Another long label"));
    v.push_back(make_label("Hello"));

    for(const auto& e : v) {
        std::cout << *e << "\n";
    }
}
```

When the program is executed, it produces the expected output:

```
A label
Longer label
Another long label
Hello
```

Unfortunately the program has a serious error: it leaks memory. When executed with a memory error detection tool (*valgrind*) the following result is produced:

```
== LEAK SUMMARY:
==    definitely lost: 32 bytes in 2 blocks
==    indirectly lost: 0 bytes in 0 blocks
==      possibly lost: 0 bytes in 0 blocks
```

a) Explain why the program leaks memory. All dynamically allocated objects are owned by a `unique_ptr`, so that they should be automatically destroyed when the `unique_ptr` is destroyed. Why aren't they?

b) The classes do not have any public constructors, but instead provide static member functions for creating objects. This is done to restrict how the objects may be allocated. What is the restriction?

*An added benefit of the factory functions is that they make it straight-forward to add the pointers to the vector. In the general case, one would need to use `std::move` to `push_back` a `std::unique_ptr` as they cannot be copied, but as the return value from a function is a temporary (*rvalue*) it is moved automatically. This is, however, not relevant to the question.*

4. For full score on these questions, detailed and well motivated answers are required.

   a) The following class contains an error. Explain what the problem is and show how to correct it. Also, one line in the program can be removed without affecting the behaviour of the class. Which line is unnecessary? Why?

```
1   class X {
2   public:
3       X() : v(nullptr) {}
4       X(int n) : v(new int(n)) {}
5       int& operator[](int n)       {return v[n];}
6       int  operator[](int n) const {return v[n];}
7       ~X() {
8           if (v != nullptr)
9               delete[] v;
10      }
11  private:
12      int* v;
13  };
```

   b) What happens when the following program is executed?

```
#include<iostream>
int main()
{
    for(unsigned u = 10; u >= 0; --u) {
        std::cout << u << std::endl;
    }
}
```

   Answer with what is printed to the terminal or explain what is happening.

   c) A programmer studying C++ has learned that objects that should have a lifetime longer than the scope they are created in must be allocated on the heap, with `new`.
   As an experiment, the following program is written:

```
#include <iostream>
int main()
{
    {
        int *x = new int;
        *x = 17;
    }
    std::cout << "*x is " << *x << "\n";
}
```

   The program, however, doesn't compile and the programmer is confused by the error:

```
  error: 'x' was not declared in this scope
     std::cout << "*x is " << *x << "\n";
                                 ^
```

   The programmer thought that since the object was allocated on the heap it should live outside the block it was declared in. Explain the compiler error.

5. This question studies an algorithm with the following declaration:

```
template <typename InputIt, typename OutputIt, typename Pred>
std::pair<InputIt, OutputIt>
copy_while(InputIt first, InputIt last, OutputIt out, Pred p)
```

It takes an iterator range, an output iterator, and a unary predicate, and copies elements from the input range to the output until the first element for which the predicate returns `false`.
The return value is one past the last elements copied, in the respective range. I.e., the InputIt refers to the first element not satisfying the predicate (or `last` if the entire range was copied) and the OutputIt is one past the last element written.

a) First, we will use `copy_while`. Write a function

```
std::vector<int> take_while_sum_less_than(std::vector<int>& v, int n)
```

that uses `copy_while` to move[1] the first k elements from `v` to a new `std::vector`, such that the sum of the moved elements is less than n, and k is as large as possible.

For example, with `v = {1,2,3,4,5,6,7,8,9}`, and $n = 15$, the result should contain `{1,2,3,4}` and after the call, `v = {5,6,7,8,9}`.

With `v={10,11,12}` and $n = 10$, the returned vector should be empty and v unchanged.

In code,

```
std::vector<int> v{1,2,3,4,5,6,7,8,9};
auto w = take_while_sum_less_than(v, 15);
```

should result in the two vectors containing the following elements:

```
v: 5 6 7 8 9
w: 1 2 3 4
```

Hint: The predicate can in this case be a stateful function, keeping track of the sum of the copied elements.

b) Implement `copy_while`. An example use of it is

```
void example()
{
    std::vector<int> v{1,3,5,2,4,6};
    std::ostream_iterator<int> out(std::cout, " ");

    std::cout << "The first sequence of odd numbers is: ";
    auto res = copy_while(begin(v), end(v), out, [](int x){return x%2;});
    std::cout << "\n";
    std::cout << "The first even number is " << *res.first << std::endl;
}
```

which should output

```
The first sequence of odd numbers is: 1 3 5
The first even number is 2
```

[1] Here `move` has nothing to do with move semantics, it simply means copy to the destination and remove from the source.