Lösningsförslag, Tentamen i C++

2019–01–09

1.

```
string_view::string_view() :str{nullptr}, sz{0};
string_view::string_view(const std::string& s):str{s.c_str()}, sz{s.size()}{}
string_view::string_view(const std::string& s, size_type pos, size_type len)
{
    if(pos+len > s.size()) throw std::out_of_range("string view");
    str = s.data() + pos;
    sz = len;
}
string_view::string_view(const char* s, size_type len ) :str{s}, sz{len} {}
string_view::string_view(const char* s ) :str{s}, sz{strlen(s)} {}
inline string_view::size_type string_view::size() const {return sz;}
inline bool string_view::empty() const {return sz == 0;}
inline string_view::const_iterator string_view::begin() const {return str;}
inline string_view::const_iterator string_view::end() const {return str+sz;}
inline const char& string_view::operator[](size_type idx) const {return str[idx];}

const char& string_view::at(size_type idx) const {
    if(idx >= sz) throw std::out_of_range("string view::at");
    return (*this)[idx];
}
string_view string_view::substr(size_type pos) const {
   return substr(pos, sz);
}
string_view string_view::substr(size_type pos, size_type len) const {
    if(pos >= sz) throw std::out_of_range("string_view: pos > size");
    const auto eend = std::min(pos+len, sz);
    return string_view(str+pos, eend-pos);
}
string_view::size_type string_view::find(char ch, size_type pos) const {
    if(pos >= sz) throw std::out_of_range("pos > size");
    auto res = std::find(begin()+pos, end(), ch) ;
    return res != end() ? res - begin(): npos;
    // or return res != end() ? std::distance(begin(),res) : npos;
}
string_view::size_type string_view::find(string_view sv, size_type pos) const {
    if(pos >= sz) throw std::out_of_range("pos > size");
    auto res = std::search(begin()+pos, end(), sv.begin(), sv.end()) ;
    return res != end() ? res - begin(): npos; // or std::distance
}
bool operator==(const string_view& a, const string_view& b) {
    if(a.size() != b.size()) return false;
    return std::equal(a.begin(), a.end(), b.begin());
}
std::ostream& operator<<(std::ostream& os, const string_view& sv) {
    for(const auto& x : sv) cout << x;
    // or std::copy(sv.begin(), sv.end(), std::ostream_iterator<char>(os));
    return os;
}
```

2. a) The string literal has static storage duration, and lives for the entire execution of the program.

The temporary std::string object created in the call only lives in that expression, and is then destroyed.

b) To move the starting point, simply add to the pointer: `string_view(b+7,5)`.

3. a) The destructor in the base class Label is not virtual. As the vector v contains pointers to Label, only the destructor in Label, but not the destructor in DynamicLabel to be executed when the std::unique_ptr<Label> objects in the vector are destroyed. Thus, the dynamically allocated character arrays owned by the DynamicLabel objects will not be deleted.

b) The restriction is that objects cannot be created on the stack, but only on the heap. Further, it guarantees that a smart pointer owns the object.

4. a) The problem is that `new int(n)` creates *a single* int object with the initial value of n. To create an array, it should be `new int[n]`.

b) As an unsigned variable cannot be negative, the loop will never terminate. The output becomes:

```
10
9
8
7
6
5
4
3
2
1
0
4294967295
4294967294
...
```

c) The dynamically allocated int survives the scope, but the `int* x` is a local variable (on the stack), so x only exists in the block where it was declared. (That also means that this program leaks memory.)

5. a)

```
#include <iterator>
#include <vector>

template <typename T>
std::vector<T> take_until_sum(std::vector<T> &v, const T& n)
{
    std::vector<T> r;
    T sum{};
    auto res = copy_while(begin(v), end(v),
                          std::back_inserter(r),
                          [&](const T& x){return (sum += x)<n;});
    v.erase(begin(v), res.first);
    return r;
}
```

Or, instead of the lambda, you can use a function object like:

```
struct sum_while {
    sum_while(int max) :max{max} {}
    int tot{0};
    int max;
    bool operator()(int x){
        if( (tot + x) < max) {
            tot += x;
            return true;
        } else return false;
    }
};
```

and use it as follows:

```
    auto res = copy_while(begin(v), end(v),
                          std::back_inserter(r), sum_while(n));
```

b) The algorithm can be implemented as

```
#include <utility>

template <typename FwdIt, typename OutputIt, typename Pred>
std::pair<FwdIt, OutputIt>
copy_while(FwdIt first, FwdIt last, OutputIt out, Pred p)
{
    while(first != last && p(*first)){
        *out = *first;
        ++first;
        ++out;
    }
    return std::make_pair(first, out);
}
```