Solutions, C++ Programming examination

2019–03–21

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 char string_view::operator[](size_type idx) const {return str[idx];}

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(auto x : sv) os << 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) Eftersom `void add(Vektor<T>, Vektor<T>, Vektor<T>)` har värdeparametrar så kommer argumenten att kopieras. När funktionen returnerar så kommer destruktorn att köras på kopiorna, och eftersom `Vektor` inte har någon kopieringskonstruktor så kommer arrayen som `p` pekar på att avallokeras.

b) Ändra till referensanrop

```
template <typename T>
void add(Vektor<T>& c1, Vektor<T>& c2, Vektor<T>& c3)
```

c) Nej, om man implementerar en copy-konstruktor (vilket man bör göra) så kommer resulatet att bli `0 0 0 0 0 0`, eftersom tilldelningarna i `add` görs till kopian.

d) The empty braces (`{}`) force value initialization. Thus, without them the program will have undefined behaviour for element types that are not guaranteed to be value initialized (e.g. `Vektor<int>`).

e) Solution with std::copy

```
template <typename T> void Vektor<T>::assign(const std::initializer_list<T>& l)
{
    std::copy(l.begin(), l.end(), p);
}
```

f) Solution with algorithms. Note the logic required to handle the case when the iterator ranges have different lengths.,

```
template <typename T>
void add(const Vektor<T>& c1, const Vektor<T>& c2, Vektor<T>& c3)
{
    auto c1_smaller = c1.length() < c2.length();
    auto shortest = c1_smaller? c1 : c2;
    auto longest  = c1_smaller? c2 : c1;
    auto it = std::transform(shortest.begin(), shortest.end(),
                             longest.begin(),
                             c3.begin(),
                             std::plus<T>{});
    std::copy(longest.begin()+shortest.length(), longest.end(), it);

    // or, using std::distance and std::next
    // auto dist = std::distance(shortest.begin(), shortest.end());
    // std::copy(std::next(longest.begin(), dist), longest.end(), it);
}
```

4. The problem is that the types do not match when using `std::find`: the third argument should be of the value type of the collection, in this case a `std::pair<std::string, int>`, but here a `std::string` is passed:

```
auto mit = std::find(m.begin(), m.end(), text);
```

The solution is to use the member function instead:

```
auto mit = m.find(text);
```