Quiz 6: Lexical Conversion

1 Background

When processing input and outputs, programmers often need to convert values with the same lexical representation between different types. For example, the string `"123"` may need to be converted to an integer value of `123`, and vice versa. In order to convert a C-style string (`const char*`) to an `int` or a `double` value, we can invoke the functions `atoi` (for ASCII to `int`, defined in C) and `std::stod` (for `std::string` to `double`, defined in C++11), respectively. For example,

```
// converts a C-style string value to an integer value
int x = atoi("123");
assert(x == 123);

// converts a C-style string value to a double value
double y = std::stod(std::string("1.0"));
assert(y == 1.0);
```

The downside of this approach is that programmers need to decide which conversion function they would use, depending on the types they want to convert from and to—there might be $M \times N$ conversion functions available for converting from M kinds of types to N kinds of types. Furthermore, each conversion function may handle a conversion error in different ways. For example, `atoi()` returns `0` if the input string cannot be converted, while `std::stof()` throws an exception.

In C programming, conversion from an `unsigned int` to a C-style string (`const char*`) is even more troublesome as C does not define a standard function to perform the conversion. Programmers often use `snprintf` for the conversion. For example, the following code gives an example of such a conversion.

```
#define UINT_DIGITS 10 // ceil(log10(2^32 - 1)), assuming unsigned int is 32-bits
char s[UINT_DIGITS + 1]; // Need one more byte for '\0'
snprintf(s, UINT_DIGITS + 1, "%u", 123u);
assert(strcmp(s, "123") == 0);
```

Fortunately, C++ offers another approach to lexcical conversions. Programmers could use a `std::stringstream` object as a buffer that holds the intermediate lexical representation of a value and simply use the stream insertion (`operator<<`) and extraction (`operator>>`) operators for conversions, as long as the two operators are defined. For example, the following two program fragments show the conversion from `const char*` to `int` and from `int` to `std::string`, respectively.

```
// converting from const char* to int
std::stringstream ss;
ss << "123";
int x;
ss >> x;
assert(x == 123);
```

```
// converting from int to std:string
std::stringstream ss;
ss << 123;
std::string s;
ss >> s;
assert(s == "123");
```

With the `std::stringstream` class, the number of functions that need to be implemented for converting from M kinds of types to N kinds of types can be reduced to only $M + N$.

We could even create a function template, whose prototype is shown below, to unify the interface of all lexical conversion functions for improving the programmability.

```
template<typename To, typename From>
To lexical_cast(const From&);
```

With this function template, all the aforementioned examples could then be re-written concisely as follows.

```
int         x = lexical_cast<int>("123");
double      y = lexical_cast<double>(std::string("1.0"));
std::string s = lexical_cast<std::string>(123);
```

We could also unify the error-handling process by ensuring that the function template throws an exception (of type `bad_lexical_cast`) when an conversion fails. An example of using the function template for lexical conversions is given below.

```
try {
    int x = lexical_cast<int>("abc");
} catch (const bad_lexical_cast& e) {
    // handle conversion error
}
```

2 Problem Statement

In this quiz, you are tasked to implement the aforementioned `lexical_cast` function template which (1) performs an arbitrary lexical conversion between various types and (2) thows an exception of type `bad_lexical_cast` if the conversion fails. More details about the `lexical_cast` function template and the `bad_lexical_cast` exception type are given as follows.

2.1 lexical_cast

```
template<typename To, typename From>
To lexical_cast(const From& val);
```

The function template has two template parameters: `To` and `From`. The function template performs a lexical conversion from a value (or an object) `val` of type `From` to type `To` and throws `bad_lexical_cast` if the conversion fails. You are required to use `std::stringstream` under the hood to perform lexical conversions. The conversion succeeds only when the extraction to a variable of `To` type is successful and there is no characters left in the stream buffer; otherwise, `lexical_cast` should throw a `bad_lexical_cast` exception. The extraction operation is not allowed to skip whitespaces at the beginning (See `std::noskipws`), for instance, attempting to convert the string `" 123"` into an integer should fail and throw an exception.

2.2 bad_lexical_cast

```
struct bad_lexical_cast;
```

`bad_lexical_cast` should be defined as a struct and should inherit from `std::bad_cast`, which is defined in the header `<typeinfo>`.

3 Hints

If you'd like to use `std::basic_ios::eof()` to check whether the buffer is empty after extraction, you should note that the stream extraction of `char` **does not** set the `eof` bit of the stream even if it is the last character in the stream:

```
std::stringstream ss("a");
char c;
ss >> c;
ss.eof(); // false!
```

You should use `std::basic_istream::peek()` to reliably detect EOF:

```
ss.peek() == EOF; // true
```

4 Submission

Submit your implementation of `lexical_cast` to E3. The file name should be `lexical_cast.hpp` or you will get zero credit.

5 Examples

```
#include <cassert>
#include <cstdint>
#include <complex>
#include <string>

#include "lexical_cast.hpp"

int main() {
    assert(123 == lexical_cast<int>("123"));
    assert(-123 == lexical_cast<int>("-123"));
    assert(0 == lexical_cast<int>("0"));

    assert(32.0 == lexical_cast<double>("32.0"));
    assert(9e15 == lexical_cast<double>("9e+15"));

    assert('c' == lexical_cast<char>("c"));

    assert("123" == lexical_cast<std::string>(123));
    assert("-123" == lexical_cast<std::string>(-123));
    assert("0" == lexical_cast<std::string>(0));

    assert("0" == lexical_cast<std::string>(0ull));

    assert(42 == lexical_cast<int>(42l));

    std::complex<double> i(0.0, 1.0); // 0.0 + 1.0i
    assert(i == lexical_cast<std::complex<double>>("(0.0,1.0)"));

    try {
        lexical_cast<int>("abc"); // conversion failure
        assert(false); // should not reach here
    } catch (const bad_lexical_cast&) {}
```

```
    try {
        lexical_cast<int>(""); // empty
        assert(false);
    } catch (const bad_lexical_cast&) {}

    try {
        lexical_cast<int>("123abc"); // trailing characters
        assert(false);
    } catch (const bad_lexical_cast&) {}

    try {
        lexical_cast<int>("123 "); // trailing whitespace
        assert(false);
    } catch (const bad_lexical_cast&) {}

    try {
        lexical_cast<char>("ab"); // trailing characters
        assert(false);
    } catch (const bad_lexical_cast&) {}

    try {
        lexical_cast<int>(" 123"); // leading whitespace
        assert(false);
    } catch (const bad_lexical_cast&) {}

    try {
        lexical_cast<short>("12345678901234567890"); // value too large
        assert(false);
    } catch (const bad_lexical_cast&) {}

    try {
        lexical_cast<std::uint32_t>(std::uint64_t(UINT32_MAX) + 1); // value too large
        assert(false);
    } catch (const bad_lexical_cast&) {}

    try {
        lexical_cast<int>(""); // empty
        assert(false);
    } catch (const std::bad_cast&) {} // catch by ref. to base class

    return 0;
}
```