Midterm Assignment

1. Types, Classes, Objects and their State

1.1 Object Definition

Assuming you are confronted with a class that supports the following use case:

```
DateParser date_parser;
date_parser.set_date(today);

auto day = date_parser.day_of_week();
```

What is your objection to this? How would you change the interface of class `DateParser`?

Possible improvements:

1. Konstruktoren, in denen man direkt die Zeit setzen kann
2. `set_date` stellt nicht klar welche Formate es annimmt (entweder alle oder genauer spezifizieren)
3. `day_of_week()` mit Uebergabeparametern

1.2 Standard Object Semantics

Given the following implementation of class `SemanticSurprise`:

```
class SemanticSurprise {
public:

  SemanticSurprise(int value)
  : _value(value) { }

  SemanticSurprise(const SemanticSurprise & other)
```

```
  : _value(other._value) { }

  bool operator==(const SemanticSurprise & other) const {
    return _value == other._value;
  }

  int value() {
    return _value;
  }

private:
  int _value;
};
```

- Are there compiler errors? If so: why?

  Nope: https://godbolt.org/g/ihDPeV

- Name the method/operation on `SemanticSurprise` in every line marked with `Op?`, including those that lead to compiler errors.
- Which values are returned in the lines marked with `value?` (given the line does not lead to a compiler error). Explain why these values are returned.
- How can this implementation be simplified? What is the requirement for this simplification?

  Copy Constructor can be defaulted. ???

```
SemanticSurprise s1;          // Default Constructor
s1.value();                   // Compiler Error (No matching constructor)

SemanticSurprise s2(4);       // Parameterized (Assignment) Constructor
s2.value();                   // 4 --> constructor defined

SemanticSurprise s3 = s2;     // Copy-assignment Constructor
s3.value();                   // 4 --> assigned in s2

SemanticSurprise s4(s3);      // Copy Constructor
s4 == s3;                     // True --> Copy constructor fullfill the operator==
s2 != s3;                     // Compiler Error (invailid operands to binary expression)
```

Given the implementation of `SemanticSurprise` above, the following use case leads to a compiler error:

```
void print_surprise(const SemanticSurprise & s) {
  std::cout << "SemanticSurprise.value: " << s.value() << std::endl;
```

```
}

print_surprise(SemanticSurprise(10));
```

- Explain the compiler error and how the implementation of `SemanticSurprise` has to be corrected.

  The given SemanticSurprise is marked as const but the function `value()` is not marked as const. Therefore it could potentially modify the object. Possible solutions:

  - Mark `value()` as `const`
  - Implement a second function:
    `int value() const { return _value; }`

1.3 Resource Ownership

Consider the following use case:

```
// ...
if (condition) {
        LogFileReader log_reader("performance.log");
        if (log_reader.lines().size() > 1024) { return; }
}
// ...
```

- Judging from its usage illustrated above: which rule must be satisfied in the implementation of class `LogFileReader` (or one of its members)?

  ???
  Potential memory leak when the if condition throws an error (or is it? Since lifetime of the variable is ended auf the mother if condition).
  The concept needs to fullfill the Resource Allocation is Initialization (RAII) rule.

  In addition `log_reader.lines()` should be constant since other applications can still write log information to the file (this saves us a mutex / sharable resource).

- Name a popular technique in resource management that depends on this rule, and briefly explain its principle.

  ???

2. Iterators

2.1 Algorithm Basics

The following algorithm dereferences and returns a given iterator's successor unless the successor's referenced value matches some condition. The algorithm is semantically correct but does not compile for iterators of some containers.

- Which containers? Briefly explain why the algorithm does not work for these and how it has to be changed.

```
template <typename Iterator, typename Value = typename Iterator::value_type>
Value next_value_or_default(Iterator it, Iterator end, Value default) {
  if (it == end || it+1 == end) { return default; }
  return it[1];
}
```

First it does not compile. `default` is in some compilers a reserved keyword. Renamed it to `dfault`.

The problem of this implementation is the increment of the iterator `it+1`. This is only suitable for `RandomAccessIterators`. The correct implementation should be `it++` to be also suitable for `ForwardIterator`, `BidirectionalIterator` and `InputIterator`.
Therefore all container using the `RandomAccessIterator Concept` should work with this implementation.
https://godbolt.org/g/8hrmHb

2.2 Container Wrapper

The STL's `std::vector` guarantess that its elements are stored in a contiguous memory region and is therefore compatible to C-style arrays. The member function `.data()` returns a pointer to the vector's underlying raw memory.

For many hardware-tuning techniques, data is accessed in chunks. Assuming a `std::vector<uint32_t>` and 64 bytes per cache line, for example, vector elements could be loaded in chunks of $64/(32/8) = 16$ elements.

- Write a container wrapper

  `cpppc::chunks<B, T, Container>`

that provides a sequential container interface on elements in `Container` in chunks of maximum size `B` bytes.

The template signature is derived from `std::stack`, a similar wrapper type defined in the STL:

http://en.cppreference.com/w/cpp/container/stack

You only need to implement a minimal interface like constructors and `begin()`, `end()`, however.

Example use case:

```
std::vector<uint16_t> v_us;

//                                      ,-- wrapped container
//                                      |
cpppc::chunks<128, uint16t_t, std::vector<uint16_t>> v_chunks(v_us);
//             |      |
//             |      '-- element type
//             |
//             '-- maximum size of a
//                 single chunk in bytes

// Iterate chunks:
auto first_chunk = v_chunks.begin();
auto chunk_size  = std::distance(v_chunks.begin(), v_chunks.end());
                   // --> 128/(16/8) = 64

// Iterators on elements in a chunk:
uint16_t first_chunk_elem = *first_chunk.begin();
uint16_t third_chunk_elem = first_chunk[2];

// Pointer to data in second chunk:
uint16_t chunk_1_data = v_chunks[1].data();
// Pointer to data in third chunk (= end pointer of data in second chunk):
uint16_t chunk_2_data = v_chunks[2].data();
```

Note that you need two iterator types:

- `cpppc::chunks<...>::iterator` for iterating chunks; it references `cpppc::chunks<...>::value_type` (= type of single chunks)
- and another iterator type to access chunk elements.

It might be best if you use existing STL containers like `std::vector<T>` or `std::array<T, ChunkSize>` as chunk type: these would already provide the sequence container interface to chunk elements you need.

Note

The `cpppc::chunks` container adapter can (and should) be implemented as a *view*, that is: without copying elements of the underlying vector.

2.X Bonus

If you want to implement more sophisticated type size calculations like alignment requirements, refer to the C++ type support utilities:

http://en.cppreference.com/w/cpp/types

Or make yourself familiar with the boost Align library (brace for impact):

http://www.boost.org/doc/libs/1_63_0/doc/html/align.html

3. Algorithms, Function Templates, Type Deduction

- Implement a function interface `void log10(X)` that accepts a numeric value of type `X` and:
  - prints the base 10 logarithm of the value if it is an integer
  - prints the base 10 logarithm of the value's square root if it is a floating point value
- Implement a function interface `void print_walk(T begin, T end)` that accepts a range of iterators of type `T` and prints all values in the range. In this, the iteration order depends on the iterator type:
  - for random-access iterators, the order should be (pseudo) random but every element in the range must only be printed once

Is the implementation random enough?

  - for input iterators, elements are printed in order from `begin` to `end-1`
  - for bidirectional iterators, elements are printed in reverse order from `end-1` to `begin`

4. Thread-Safety

4.1 Parallelism and STL Containers

Given the following operations on an instance of `std::vector`, consider operations in the same table row to be executed by multiple threads in parallel:

All mentioning of rules (e.g. `Rule 1.`) are based on https://en.cppreference.com/w/cpp/container#Thread_safety

```
// Shared vector instance accessed by thread A and thread B:
std::vector<int> v;

// thread A:                    | thread B:
// ============================|==============================
   std::vector<int> a;          | std::vector<int> b;
// Safe: no shared resources (Rule 1.)
// ----------------------------+------------------------------
   int xa = v[3];               | int xb = v[4];
// Safe: Rad-only -> Rule 2.
// ----------------------------+------------------------------
   v[3] = 123;                  | v[4] = 345;
// Safe: Rule 3.
// ----------------------------+------------------------------
   v[3] = 123;                  | int xb = v[3];
// Not safe: Modyfing and reading the same element in the same
// container can result in ambigious behaviour
// ----------------------------+------------------------------
   v.push_back(24);             | v.size();
// Not safe: The underlying container is modified in A and its
// properties (which change) is read in B -> ambigious behaviour
// ----------------------------+------------------------------
   v.back();                    | v.push_back(54);
// Not safe: Rule 2. is violated.
// ----------------------------+------------------------------
   v.begin();                   | v.push_back(34);
// Not 100% safe: push_back can invailidate the iteratos
// ----------------------------+------------------------------
   v.back();                    | v.pop_back();
// Not safe: The underlying container is modified and read at the
// same time & pop_back can invailidate the iterators
// ----------------------------+------------------------------
```

- For every pair of operation on the same table row, give a brief explanation on the guarantees with respect to thread-safety according to the C++ standard.

4.2 Producer-Consumer Problem

Simply put, one thread is producing goods and another thread is consuming goods. We want the consumer thread to wait **using a condition variable**, and

we want `goods.push(i)` to be mutually exclusive to `goods.pop()`.

We are letting c++ and c– be surrogates for this mutual exclusion, since we can easily check if we correctly end up with 0 in the end.

Run the code as it is, and you will see that the net value is way off:

> `Note:`
>
> I know there is an easier solution, however the condition variable approach is more efficient in many cases and not as commonly known as it should be.

```
#include <iostream>
#include <thread>
#include <condition_variable>
#include <mutex>
#include <chrono>
#include <queue>
using namespace std;

int main() {
    int c = 0;
    bool done = false;
    queue<int> goods;

    thread producer([&]() {
        for (int i = 0; i < 500; ++i) {
            goods.push(i);
            c++;
        }

        done = true;
    });

    thread consumer([&]() {
        while (!done) {
            while (!goods.empty()) {
                goods.pop();
                c--;
            }
        }
    });

    producer.join();
    consumer.join();
    cout << "Net: " << c << endl;
}
```