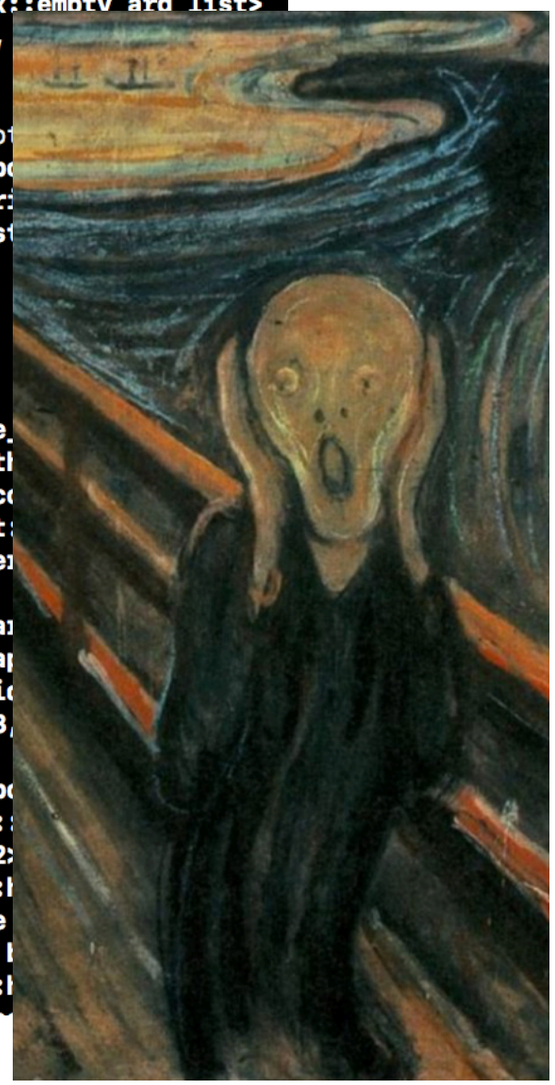
A short tour of

CONCEPTS IN C++20

Felix Kutzner (QPR Technologies)
C++ User Group Karlsruhe

Times of Old & Evil

```
, A5>::push(const value_type&) [with T = int; A0 = boost::heap::mutable_<true>; A1 = boost::heap::arity<2
>; A2 = boost::heap::compare<int>; A3 = boost::parameter::void_; A4 = boost::parameter::void_; A5 = boost
::parameter::void_; typename boost::mpl::if_c<boost::heap::d_ary_heap<T, A0, A1, A2, A3, A4, A5>::is_muta
ble, typename boost::heap::d_ary_heap<T, A0, A1, A2, A3, A4, A5>::implementation_defined::handle_type, vo
id>::type = boost::heap::detail::priority_queue_mutable_wrapper<boost::heap::detail::d_ary_heap<int, boos
t::parameter::aux::arg_list<boost::heap::compare<int>, boost::parameter::aux::arg_list<boost::heap::arity
<2>, boost::parameter::aux::arg_list<boost::heap::mutable_<true>, boost::parameter::aux::empty_arg_list>
> >, boost::heap::detail::nop_index_updater> >::handle_type; boost::heap::d_ary_heap<T,
4, A5>::value_type = int]'
Horror.cpp:14:12:   required from here
/usr/local/include/boost/heap/detail/mutable_heap.hpp:124:45: error: 'operator()' is not
st::heap::detail::priority_queue_mutable_wrapper<boost::heap::detail::d_ary_heap<int, bo
ux::arg_list<boost::heap::compare<int>, boost::parameter::aux::arg_list<boost::heap::ari
ameter::aux::arg_list<boost::heap::mutable_<true>, boost::parameter::aux::empty_arg_list
p::detail::nop_index_updater> >::value_compare' {aka 'int'}
            return value_compare::operator()(lhs->first, rhs->first);
                   ~~~~~~~~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~
In file included from /usr/local/include/boost/heap/d_ary_heap.hpp:22,
                 from Horror.cpp:1:
/usr/local/include/boost/heap/detail/mutable_heap.hpp: In instantiation of 'const value_
eap::detail::priority_queue_mutable_wrapper<PriorityQueueType>::value_comp() const [with
e = boost::heap::detail::d_ary_heap<int, boost::parameter::aux::arg_list<boost::heap::co
::parameter::aux::arg_list<boost::heap::arity<2>, boost::parameter::aux::arg_list<boost:
rue>, boost::parameter::aux::empty_arg_list> > >, boost::heap::detail::nop_index_updater
etail::priority_queue_mutable_wrapper<PriorityQueueType>::value_compare = int]':
/usr/local/include/boost/heap/d_ary_heap.hpp:815:35:   required from 'const value_compar
_ary_heap<T, A0, A1, A2, A3, A4, A5>::value_comp() const [with T = int; A0 = boost::heap
 A1 = boost::heap::arity<2>; A2 = boost::heap::compare<int>; A3 = boost::parameter::void
rameter::void_; A5 = boost::parameter::void_; boost::heap::d_ary_heap<T, A0, A1, A2, A3,
ompare = int]'
/usr/local/include/boost/heap/detail/heap_comparison.hpp:44:43:   required from 'bool bo
::value_compare(const Heap1&, const Heap2&, typename Heap1::value_type, typename Heap2::
 Heap1 = boost::heap::d_ary_heap<int, boost::heap::mutable_<true>, boost::heap::arity<2>
mpare<int> >; Heap2 = boost::heap::d_ary_heap<int, boost::heap::mutable_<true>, boost::h
ost::heap::compare<int> >; typename Heap1::value_type = int; typename Heap2::value_type
/usr/local/include/boost/heap/detail/heap_comparison.hpp:169:30:   required from 'bool h
1::heap_compare_iteration::operator()(const Heap1&, const Heap2&) [with Heap1 = boost::h
```



Towards Modernity

C++11: static assertions and type traits

```
static_assert(is_foo<ParmT>::value,
              “ParmT must satisfy foo, but does not”);
```

C++20: named sets of requirements (“concepts”)

Example Problem

clause

$$F = (\neg a \ \vee \ \neg b \ \vee \ o) \ \wedge \ (a \vee \neg o) \ \wedge \ (b \vee \neg o)$$

literal

Let’s define a concept for literals!

Defining Concepts

```
template<typename Lit> concept CNFLiteral =
  requires(Lit l) {                    // l: no lifetime, no linkage
    typename Lit::sign_type;
  {l.get_sign()} noexcept -> typename Lit::sign_type;
  {!l} noexcept -> Lit
  // nested requires clauses also possible
}
&& std::StrictTotallyOrdered<Lit>          // from <concepts>
&& std::Regular<Lit>                       // from <concepts>
&& is_sign_v<typename Lit::sign_type>;     // boolean-valued
```

Using Concepts

```
template<typename T> requires CNFLiteral<T>
class Clause<T> {
  [...]
};
```

Using Concepts

```
template<typename T> requires SomeConcept<T> [...]

template<typename T> requires SomeConcept<T> &&
  (etc_property_v<T> || AndSoOnConstraints<T>) [...]

template<SomeConcept T> [...]

template<SomeConcept auto X> [...]

template<SomeConcept... T> [...]
```

Type Deduction

```
ConceptA auto x = [...];

ConceptA auto foo();

ConceptA decltype(auto) foo();

ConceptA auto foo(ConceptB auto x);

ConceptA auto foo(ConceptB auto&& x);
```

Overload Resolution & SFINAE

```
template<typename T> requires ConceptB<T>
void foo(T x);
```

```
template<typename T> requires ConceptB<T> && More<T>
void foo(T x);

void bar(int x) {
  foo(x);
}
```

END

Extras

Core Guidelines on Concepts

- **T.20:** Avoid "concepts" without meaningful semantics
- **T.21:** Require a complete set of operations for a concept
- **T.22:** Specify axioms for concepts
- **T.23:** Differentiate a refined concept from its more general case by adding new use patterns

Core Guidelines on Concepts

- **T.24:** Use tag classes or traits to differentiate concepts that differ only in semantics.

- **T.25:** Avoid complementary constraints

- **T.26:** Prefer to define concepts in terms of use-patterns rather than simple syntax