CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Concept Based Design

Case Study

The Concept of Cursor

```
// models cursor
class some_cursor {
public:
 bool done() const;
 int const &get() const;
 void next();
};

// requires cursor
void dump(auto cur) {
 for (; !cur.done(); cur.next())
   std::cout << cur.get() << std::endl;
}
```

The Concept of Cursor (C++20)

```
template <class T> concept is_cref = std::is_lvalue_reference_v<T> && std::is_const_v<std::remove_reference_t<T>>;

template <class T> concept Cursor = std::move_constructible<T> && requires(T const &cval, T &val) {
   {cval.done()} -> std::convertible_to<bool>;
   {cval.get()} -> is_cref;
   val.next();
};

class some_cursor {
public:
 bool done() const;
 int const &get() const;
 void next();
};
static_assert(Cursor<some_cursor>);

void dump(Cursor auto cur) {
 for (; !cur.done(); cur.next())
   std::cout << cur.get() << std::endl;
}
```

Concept Modelers

```
class file_cur {
    std::unique_ptr<FILE, decltype(&fclose)> strm_;
    std::optional<char> val_;
public:
    file_cur(char const* name) : strm_(fopen(name, "r"), &fclose) { next(); }
    bool done() const { return !val_; }
    char const& get() const { return *val_; }
    void next() {
        if (int c = fgetc(strm_.get()); c == EOF)
            val_ = {};
        else
            val_ = c;
    }
};
```

Concept Modelers

```
template <std::integral T> struct numbers_from {
 T val_;
 T const &get() const { return val_; }
 void next() { ++val_; }
 bool done() const { return false; }
};

dump(numbers_from{42});
```

Concept Modelers

```
template <std::integral T> struct numbers_from {
 T val_;
 T const &get() const { return val_; }
 void next() { ++val_; }
 bool done() const { return false; }
};

dump(take(10, numbers_from{42}));
```

take

```cpp
template <Cursor In> struct take {
 int count_;
 In in_;

 take(int n, In in) : count_(in.done() ? 0 : n), in_(std::move(in)) {}
 bool done() const { return count_ == 0; }
 decltype(auto) get() const { return in_.get(); }
 void next() {
   in_.next();
   if (in_.done())
     count_ = 0;
   else
     --count_;
 }
};
```

take

```
template <Cursor In> struct take_cur {
 int count_;
 In in_;

 take_cur(int n, In in) : count_(in.done() ? 0 : n), in_(std::move(in)) {}
 bool done() const { return count_ == 0; }
 decltype(auto) get() const { return in_.get(); }
 void next() {
   in_.next();
   if (in_.done())
     count_ = 0;
   else
     --count_;
 }
};

inline constexpr auto take(int n) {
   return [n](Cursor auto in) -> Cursor auto { return take_cur(n, std::move(in)); };
}

dump(take(10)(numbers_from{42}));
```

Yet Another Concept: Cursor Algorithm

Unary functional object that takes a cursor and returns a cursor.

Examples:

- filter

```
filter(is_prime)(numbers_from(1));
// produces: 1, 2, 3, 7, 11, 13 ...
```

- transform

```
transform(square)(numbers_from(1));
// produces: 1, 2, 4, 9, 25, 36 ...
```

- moving average
- ...

Composability

```
auto smth_useful(Foo foo, Bar bar) {
  return [foo, bar] (Cursor auto in) -> Cursor auto {
    return filter(foo.bla_bla)(
      do_that(bar)(
        take(bar.n)(
          do_this(foo)(
            filter(bar.baz)(std::move(in))))));
  };
}

auto out = smth_useful(foo, bar)(std::move(in));
```

Composability

```
auto smth_useful(Foo foo, Bar bar) {
  return compose(
    filter(foo.bla_bla),
    do_that(bar),
    take(bar.n),
    do_this(foo),
    filter(bar.baz));
}

auto out = smth_useful(foo, bar)(std::move(in));
```

compose

```
template <class F>
constexpr F&& compose(F&& f) { return std::forward<F>(f); }

constexpr auto compose(auto f, auto... gs) {
    return [f=std::move(f), g=compose(std::move(gs)...)]<class... Args>(Args&&... args) {
        return f(g(std::forward<Args>(args)...));
    };
}
```

Syntax Sugar: pipes

```
using namespace pipes;

// dump(take(10)(numbers_from(42)));

numbers_from(42) | take(10) | dump;
```

Syntax Sugar: pipes

```
namespace pipes {
  // TODO:
  // if f(x) is a valid expression then
  //   x|f  evaluates to f(x)
  // else
  //   g|f evaluates to [f,g](auto... x) { return f(g(x...)); }
}
```

Syntax Sugar: pipes

```cpp
namespace pipes {
template <class F, class G>
constexpr decltype(auto) operator|(F &&f, G &&g) {
 if constexpr (std::is_invocable_v<G &&, F &&>)
   return std::forward<G>(g)(std::forward<F>(f));
 else
   return [f = std::forward<F>(f), g = std::forward<G>(g)](auto &&... args) {
     return g(f(std::forward<decltype(args)>(args)...));
   };
}
}
```

Any compilation problems here?

```
void dump(Cursor auto cur) {
 for (; !cur.done(); cur.next())
   std::cout << cur.get() << std::endl;
}

using namespace pipes;
numbers_from{42} | take(10) | dump;
```

Template functions are not the first class citizens. Use functors.

```
inline constexpr auto dump = [](Cursor auto cur) {
 for (; !cur.done(); cur.next())
   std::cout << cur.get() << std::endl;
};

using namespace pipes;
numbers_from{42} | take(10) | dump;
```

Cursor API for the Universal Translator

```
inline auto translate(lang from = lang::auto_detect, lang to = lang::en) {
  return [from, to](Cursor auto in) {
     // TODO: implement
  };
}
```

Any problems with that?

Cursor API for the Universal Translator

```
inline auto translate(lang from = lang::auto_detect, lang to = lang::en) {
  return [from, to](Cursor auto in) {
     // TODO: implement
  };
}
```

Any problems with that?

It forces us to do header only library:

- implementation is exposed to the user;
- increased compilation time of the user’s code.

Cursor API for the Universal Translator. Type Erasure.

```
// translate.hpp
using translate_f = std::function<any_cursor<char>(any_cursor<char>)>;

translate_f translate(lang from = lang::auto_detect, lang to = lang::en);

// translate.cpp
translate_f translate(lang from, lang to) {
  return [from, to](any_cursor<char> in) -> any_cursor<char> {
    // TODO: implement
  };
}
```

Type Erasure

```
template <class T> class any_cursor {
 struct iface {
   virtual ~iface() {}
   virtual bool done() const = 0;
   virtual T const &get() const = 0;
   virtual void next() = 0;
 };
 template <Cursor Cur> struct impl : iface {
   Cur cur_;
   impl(Cur cur) : cur_(std::move(cur)) {}
   bool done() const override { return cur_.done(); }
   T const &get() const override { return cur_.get(); }
   void next() override { cur_.next(); }
 };
 std::unique_ptr<iface> impl_;
public:
 any_cursor(Cursor auto cur) : impl_(new impl{std::move(cur)}) {}
 bool done() const { return impl_->done(); }
 T const &get() const { return impl_->get(); }
 void next() { impl_->next(); }
};
```

Challenge: external code adaptation.

```
namespace AnotherAPI {
 template <class T> struct NumbersFrom {
   T val_;

   NumbersFrom(T val) : val_(val) {}
   T const &Get() const { return val_; }
   void Next() { ++val_; }
   bool Done() const { return false; }
 };
}
static_assert(Cursor<AnotherAPI::NumbersFrom<int>>); // FAIL
```

Challenge: external code adaptation.

```
namespace AnotherAPI {
 template <class T> struct NumbersFrom {
   T val_;

   NumbersFrom(T val) : val_(val) {}
   T const &Get() const { return val_; }
   void Next() { ++val_; }
   bool Done() const { return false; }
 };
}

template <class In> struct wrapper {
   In in_;
   wrapper(In in) : in_(std::move(in)) {}
   bool done() const { return in_.Done(); }
   decltype(auto) get() const { return in_.Get(); }
   void next() { in_.Next(); }
};
static_assert(Cursor<wrapper<AnotherAPI::NumbersFrom<int>>>);
```

ADL Based Concept Definition

```
namespace cursor {
// fallbacks
auto cursor_done(auto const &cur) -> decltype(cur.done()) { return cur.done(); }
auto cursor_get(auto const &cur) -> decltype(cur.get()) { return cur.get(); }
auto cursor_next(auto& cur) -> decltype(cur.next()) { return cur.next(); }
// API for algorithms
inline constexpr auto done = [](auto const &cur) -> decltype(cursor_done(cur)) { return cursor_done(cur); };
inline constexpr auto next = [](auto &cur) -> decltype(cursor_next(cur)) { cursor_next(cur); };
inline constexpr auto get = [](auto const &cur) -> decltype(cursor_get(cur)) { return cursor_get(cur); };
}

template <class T> concept is_cref = std::is_lvalue_reference_v<T> && std::is_const_v<std::remove_reference_t<T>>;

template <class T> concept Cursor = std::move_constructible<T> && requires (T const &cval, T &val) {
  {cursor::done(cval)} -> std::convertible_to<bool>;
  {cursor::get(cval)} -> is_cref;
  cursor::next(val);
};

inline constexpr auto dump = [](Cursor auto in) {
 for (; !cursor::done(in); cursor::next(in))
   std::cout << cursor::get(in) << std::endl;
};
```

ADL Based Concept Definition

```
namespace AnotherAPI {
 template <class T> struct NumbersFrom {
   T val_;

   NumbersFrom(T val) : val_(val) {}
   T const &Get() const { return val_; }
   void Next() { ++val_; }
   bool Done() const { return false; }
 };
}

// adaptation
namespace AnotherAPI {
   bool cursor_done(auto const &cur) { return cur.Done(); }
   decltype(auto) cursor_get(auto const &cur) { return cur.Get(); }
   void cursor_next(auto &cur) { return cur.Next(); }
}

static_assert(Cursor<AnotherAPI::NumbersFrom<int>>);
```

ADL Based Concept Definition

```
namespace cursor {
// fallbacks
inline bool cursor_done(...) { return false; }
auto cursor_done(auto const &cur) -> decltype(cur.done()) { return cur.done(); }
auto cursor_get(auto const &cur) -> decltype(cur.get()) { return cur.get(); }
auto cursor_next(auto& cur) -> decltype(cur.next()) { return cur.next(); }
// API for algorithms
inline constexpr auto done = [](auto const &cur) { return cursor_done(cur); };
inline constexpr auto next = [](auto &cur) -> decltype(cursor_next(cur)) { cursor_next(cur); };
inline constexpr auto get = [](auto const &cur) -> decltype(cursor_get(cur)) { return cursor_get(cur); };
}

struct num_from {
   int val;
   friend int const &cursor_get(num_from const& cur) { return cur.val; }
   friend void cursor_next(num_from& cur) { ++cur.val; }
};
static_assert(Cursor<num_from>);
```

ADL Based Concept Definition

Type ***Cur*** is a cursor of ***T*** iff:

- ***Cur*** has public move constructor
- if ***cursor_get*** is invocable with ***Cur const&*** the result should be ***T const&*** otherwise ***Cur*** should have ***get() const*** method that returns ***T const&***
- Either ***cursor_next*** is invocable with ***Cur&*** or ***Cur*** has ***next()*** method
- if ***cursor_done*** is invocable with ***Cur const&*** the result should be explicitly convertible to ***bool*** else if ***Cur*** has ***done() const*** method its result should be explicitly convertible to ***bool***

Range For Loop Adaptation

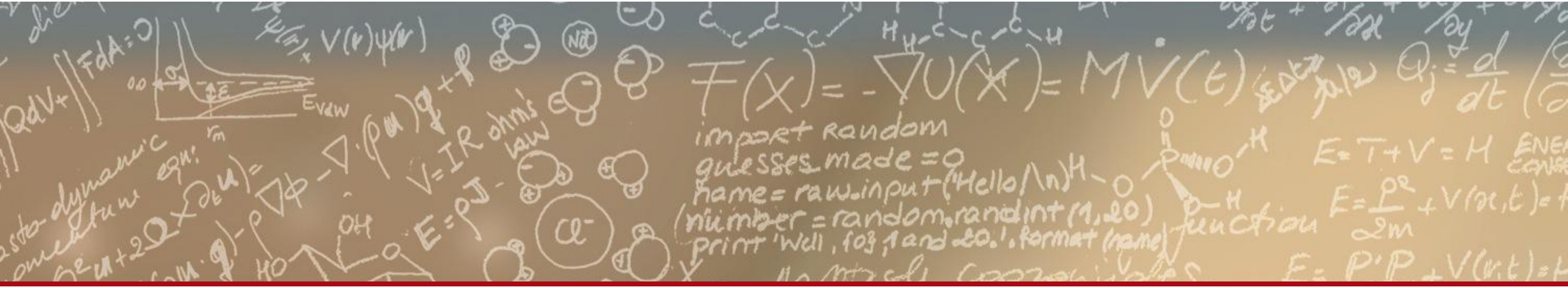
```
using namespace pipes;

for (auto &&x : numbers_from(42) | take(10))
  std::cout << x << std::endl;
```

Range For Loop Adaptation

```
using namespace pipes;

{
  auto &&rng = numbers_from(42) | take(10);
  auto it = begin(rng);
  auto e = end(rng);
  for (; it != e; ++it) {
    auto &&x = *it;
    std::cout << x << std::endl;
  }
}
```

Range For Loop Adaptation

```
struct sentinel {};

template <Cursor Cur> struct iter {
 Cur &cur_;
 decltype(auto) operator*() const { return cursor::get(cur_); }
 void operator++() { cursor::next(cur_); }
 bool operator!=(sentinel) const { return !cursor::done(cur_); }
};

auto begin(Cursor auto &cur) { return iter{cur}; }
auto end(Cursor auto const &) { return sentinel(); }
```



Questions are welcome