CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Advanced C++ course 2023

Concept based design by example

CSCS (designed by Anton Afanasyev)

Motivation

> Concepts (requirements on template arguments) are the central feature of C++ generic library design; they define the terms in which a library's generic data structures and algorithms are specified. Every working generic library is based on concepts. These concepts may be represented using specifically designed language features, in requirements tables, as comments in the code, in design documents, or simply in the heads of programmers. However, without concepts (formal or informal), no generic code could work.

(from "Design of Concept Libraries for C++" by Andrew Sutton, Bjarne Stroustrup)

> Generic Programming doesn't mean templates. It means generalizing algorithm implementations iteratively, discovering sets of requirements on their arguments and grouping the requirements into named concepts and hierarchies of concepts. It's about algorithms, not templates.

(Eric Niebler on X, Jan 8th 2020)

Writing generic libraries requires discipline!

Warm-up

```
template<class T>
T square(T v) {
    return v*v;
}
```

```
template<std::floating_point T>
T square(T v) {
    return v*v;
}
```

```
template<HasMultiplyOp T>
T square(T v) {
    return v*v;
}
```

Example Concept for today

```
struct some_cursor {
    int const& get() const;
    bool done() const;
    void next();
};
```

We will develop a library around that concept: start

Exercises

starting point for exercises https://godbolt.org/z/9x75P78bo

Summary: Cursor Concept

```
struct some_cursor {
    int const& get() const;
    bool done() const;
    void next();
};

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

static_assert(Cursor<some_cursor>);
```

Summary: Cursor Algorithms

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

struct take_impl{/* ... */};
constexpr inline auto take = [](int n) {
    return [n](Cursor auto cur) { return take_impl(std::move(cur), n); };
};

dump(take(2)(numbers_from(42)))
```

Summary: Improving Syntax: Composition and Pipe

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

Summary: Customization Points

```
namespace cursor {
// default done
auto cursor_done(...) { return false; }

// cursor fallback
auto cursor_done(auto const& cur) -> decltype(cur.done()) { return cur.done(); }
auto cursor_next(auto& cur) -> decltype(cur.next()) { cur.next(); }
auto cursor_get(auto const& cur) -> decltype(cur.get()) { return cur.get(); }

// customization point
constexpr inline auto done = [](auto const& cur) -> decltype(cursor_done(cur)) { return cursor_done(cur); };
constexpr inline auto next = [](auto& cur) -> decltype(cursor_next(cur)) { cursor_next(cur); };
constexpr inline auto get = [](auto const& cur) -> decltype(cursor_get(cur)) { return cursor_get(cur); };
}
```

Summary: Type Erasure

```
template <class T> class any_cursor {
    struct iface {
        virtual ~iface(){};
        virtual T get() const = 0;
        virtual bool done() const = 0;
        virtual void next() = 0;
    };

    template <Cursor C> struct impl : iface {
        C cur_;
        impl(C cur) : cur_(std::move(cur)) {}
        T get() const { return cursor::get(cur_); }
        bool done() const { return cursor::done(cur_); }
        void next() { cursor::next(cur_); }
    };
    std::unique_ptr<iface> impl_;

public:
    template <class C> any_cursor(C cur) : impl_{ new impl<C>(std::move(cur)) } {}
    friend auto cursor_done(any_cursor const& cur) -> decltype(auto) { return cur.impl_->done(); }
    friend auto cursor_next(any_cursor& cur) -> decltype(auto) { cur.impl_->next(); }
    friend auto cursor_get(any_cursor const& cur) -> decltype(auto) { return cur.impl_->get(); }
};
```

Summary: range-based for loop

```
struct sentinel {};

template <Cursor C>
struct iter {
    C& cur_;

    decltype(auto) operator*() const {
        return cursor::get(cur_);
    }
    void operator++() {
        cursor::next(cur_);
    }
    bool operator!=(sentinel) const {
        return !cursor::done(cur_);
    }
};

template <Cursor C>
iter<C> begin(C& cur) { return { cur }; }
template <Cursor C>
sentinel end(C const& cur) { return {}; }
```