2023

From Templates to Concepts

Alex Dathskovsky

C++ now

About Me:

SPEEDATA

alex.dathskovsky@speedata.io

www.linkedin.com/in/alexdathskovsky

Core C++ 2023: https://corecpp.org

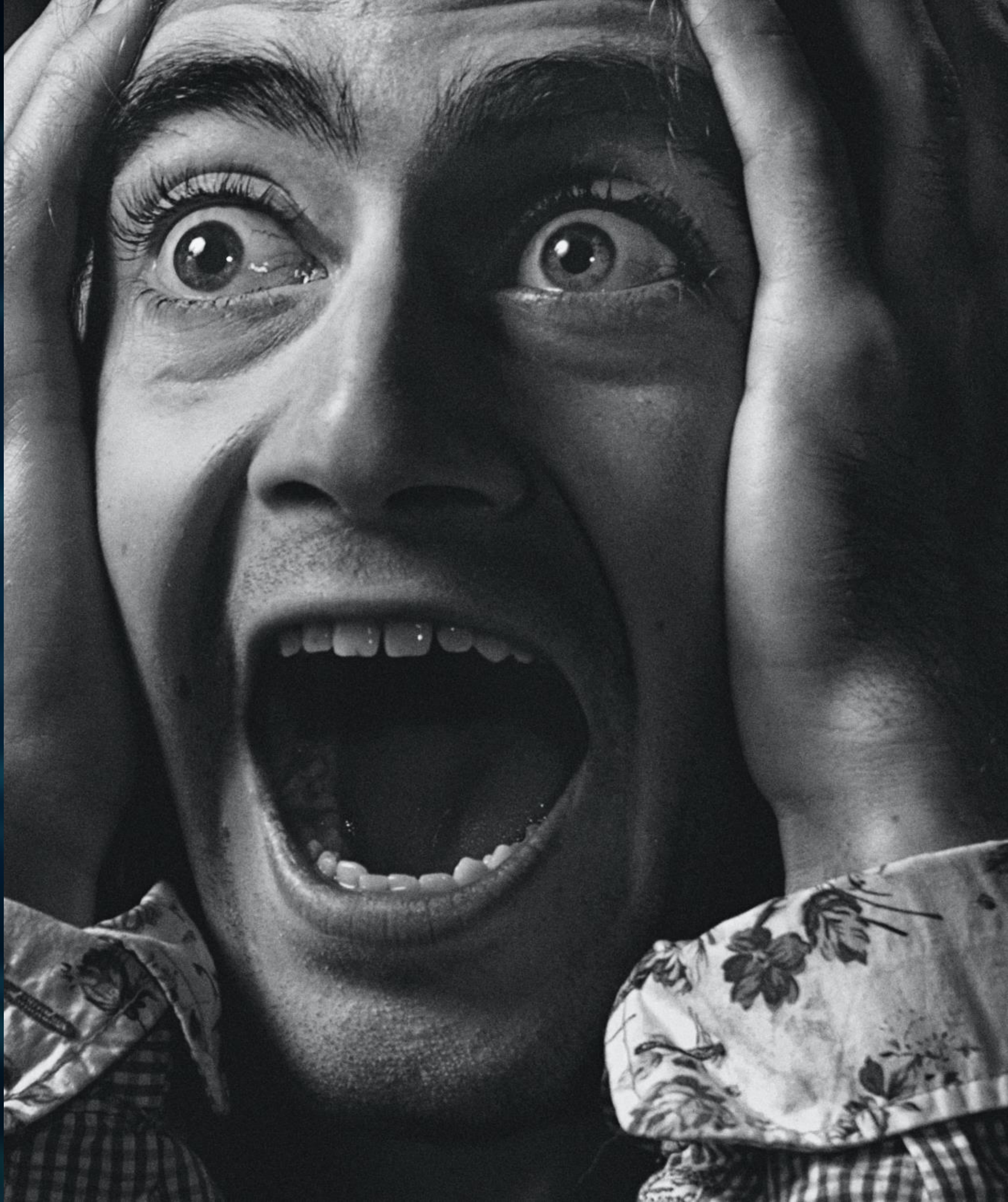
Tel-Aviv, 5th to 7th of June

Templates:

What's the first thing that comes to mind?

Templates:

What's the first thing that comes to mind?



Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

BASIC TEMPLATE RULES

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

THIS IS C++NOW THERE'S NOTHING BASIC HERE

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

QUESTION:

- What is the outcome of this code?

```cpp
template <typename>
struct Res{
};

template <typename R, typename... Args>
struct Res<R(Args...)>{
    using r_type = R;
};


template<typename M, typename T>
auto foo(M T::* pm) -> Res<M>::r_type;
```

ANSWER:

- It helps us to deduce a return type of a member function without providing the actual parameters
- With it we can create interesting traits and concepts

USAGE EXAMPLE:

```cpp
struct X{
    int add(int a, int b) {return a+b;};
};

struct Y{
    double add(double a) {return 1.+a;};
};


template <typename T, typename U>
struct is_same { static constexpr bool value = false; };

template <typename U>
struct is_same<U, U> { static inline constexpr bool value = true; };

int main(){
    static_assert(is_same<decltype(foo(&X::add)), int>::value);
    static_assert(is_same<decltype(foo(&Y::add)), double>::value);
}
```

DIGEST:

- Creating a Helper class Res

```
template <typename>
struct Res{
};

template <typename R, typename... Args>
struct Res<R(Args...)>{
    using r_type = R;
};


template<typename M, typename T>
auto foo(M T::* pm) -> Res<M>::r_type;
```

DIGEST:

- foo function

```cpp
template <typename>
struct Res{
};

template <typename R, typename... Args>
struct Res<R(Args...)>{
    using r_type = R;
};


template<typename M, typename T>
auto foo(M T::* pm) -> Res<M>::r_type;
```

SIMPLIFICATION:

- Using std::function

```
template <typename M, typename T>
auto foo(M T::* pm) -> std::function<M>::result_type;
```

SIMPLIFICATION:

- We could make it even shorter with std::mem_fn::result_type

```
static_assert(is_same<typename decltype(std::mem_fn(&X::add))::result_type, int>::value);
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

SIMPLIFICATION:

- We could make it even shorter with std::mem_fn::result_type
  - Please don't use this as result_type is deprecated since C++17

```
static_assert(is_same<typename decltype(std::mem_fn(&X::add))::result_type, int>::value);
```

TRAIT LIBRARY

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

TRAITS

- C++11 introduced the standard type trait library

Example of useful traits:

```
integral_type<T, VALUE>
is_pointer<T>
is_abstract<T>
is_assignable<T>
is_convertible<T, U>
is_same<T, U>
…
```

TRAITS EXAMPLES: SOME ARE SIMPLE

```
template<typename T, typename U>
struct is_same : std::false_type {};

template<class T>
struct is_same<T, T> : std::true_type {};
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

TRAITS EXAMPLES: SOME ARE MORE COMPLEX

```
template<typename T>
struct is_floating_point
        : std::integral_constant<
            bool,
            std::is_same<float, typename std::remove_cv<T>::type>::value
            or std::is_same<double, typename std::remove_cv<T>::type>::value
            or std::is_same<long double, typename std::remove_cv<T>::type>::value
        > {};
```

CONSTRAINTS WITH TRAITS

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

TRAITS

- Will this always work?

```
template <typename T>
void print(T const& t){
    fmt::print("{}", t);
}
```

TRAITS

- The Answer is no. This pattern may take a pointer as well

```cpp
template <typename T>
void print(T const& t) {
    fmt::print("{}", t);
};

int main() {
    int i{1};
    print(&i);
}
```

TRAITS

- The Answer is no. This pattern may take a pointer as well

```cpp
template <typename T>
void print(T const& t) {
    fmt::print("{}", t);
};

int main() {
    int i{1};
    print(&i);
}
```

```
error: static_assert failed due to requirement 'formattable_pointer' "Formatting of non-void pointers is disallowed."
```

TRAITS

- We can fix it with traits (other implementation variants are possible):

```
template <typename T, bool>
struct printHelper {
    static void print(T const& t){fmt::print("{}", t);};
};

template <typename T>
struct printHelper<T, true> {
    static void print(T const& t){fmt::print("{}", *t);};
};

template <typename T>
void print(T const& t){
    printHelper<T, std::is_pointer<T>::value>::print(t);
}

```

TRAITS

- We can fix it with traits (other implementation variants are possible):

TRAITS

- In C++14 some of the traits got a new alias for its inner type **“trait”_t**
- In C++17 some of the traits got the **“trait”_v** aliasing

```cpp
template<typename T>
using add_pointer_t = typename add_pointer<T>::type;

template<typename T>
constexpr bool is_pointer_v = is_pointer<T>::value;
```

TRAITS

```cpp
template <typename T, bool>
struct printHelper {
    static void print(T const& t){fmt::print("{}", t);};
};

template <typename T>
struct printHelper<T, true> {
    static void print(T const& t){fmt::print("{}", *t);};
};

template <typename T>
void print(T const& t){
    printHelper<T, std::is_pointer_v<T>>::print(t);
}
```

TRAITS

- We can simplify things with Tag Dispatch
  std::is_pointer<T>::type is std::true_type or std::false_type;

```
template <typename T>
void printHelper(std::false_type, T const& t){
    fmt::print("{}", t);
}

template <typename T>
void printHelper(std::true_type, T const& t){
    fmt::print("{}", *t);
}

template <typename T>
void print(T const& t) {
    printHelper(typename std::is_pointer<T>::type{}, t);
};
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

TRAITS

- With C++17 we can simplify even further by using constexpr if

```cpp
template <typename T>
void print(T const& t){
    if constexpr (std::is_pointer_v<T>){
        fmt::print("{}", *t);
    }else{
        fmt::print("{}", t);
    }
}
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

TRAITS

- With C++20 we can use a simple Concept (more about that later). Very similar to tag dispatch, but with better readability and less code

```cpp
void print(auto& t){
    fmt::print("{}", t);
}

void print(auto* t){
    fmt::print("{}", *t);
}
```

TRAITS

- With C++20 we can use a simple Concept (more about that later). Very similar to tag dispatch, but with better readability and less code

```
void print(const auto& t){
    fmt::print("{}", t);
}

void print(const pointer auto& t){
    fmt::print("{}", *t);
}
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

CONTAINER DETECTION

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

DETECTING A CONTAINER (NAÏVE IMPLEMENTATION)

- Identifying Containers

  We want to identify containers during compile time.

  An Idea:

  All STL containers have nested::iterator type (we can use that)

```
template <typename T>
struct is_container
{ static const bool value = ???; };
```

SFINAE

SFINAE - SUBSTITUTION FAILURE IS NOT AN ERROR

Special rule for function template overload resolution:
If an overload candidate would cause a compilation error during type substitution, it is silently removed from the overload set.

ELLIPSES (...)

- Functions with variadic arguments (...) are always inferior in overload resolution

```
void print (...)  {
    fmt::print("ellipses\n");
}

void print(int) {
    fmt::print("integer\n");
}

int main(){
    print(17);
    print("17");
};
```

ELLIPSES (…)

- Functions with variadic arguments (…) are always inferior in overload resolution

```
void print (...)  {
    fmt::print("ellipses\n");
}

void print(int) {
    fmt::print("integer\n");
}

int main(){
    print(17);
    print("17");
};
```

integer
ellipses

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

DETECTING A CONTAINER (NAÏVE IMPLEMENTATION)

```
template <typename T>
struct is_container {
    template <typename S>
    static std::byte f(...);

    template <typename S>
    static std::size_t f(typename S::iterator*);

    static const bool value = (sizeof(f<T>(0)) == sizeof(std::size_t));
};
```

DETECTING A CONTAINER (NAÏVE IMPLEMENTATION)

How should we use it ?

An Idea:

DETECTING A CONTAINER (NAÏVE IMPLEMENTATION)

How should we use it ?

An Idea:

```
template <typename T>
void print (const T& t) {
    if (!is_container<T>::value) {
        fmt::print("{}", t);
    }
    else {
        for (auto const& e : t) {
            fmt::print("{}", e);
        }
    }
}
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

DETECTING A CONTAINER

- The previous example wasn't a good idea until C++17
- We will gradually get better with our approach, but until C++17 we had to do something different…

DETECTING A CONTAINER

- The previous example wasn't a good idea until C++17
- We will gradually get better with our approach, but until C++17 we had to do something different…

What can we do:

- We can delegate to a helper class
- We can delegate to a helper method
- In some cases , it’s more desirable to just write two functions and have the compiler pick the right one!

ENABLE_IF

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

ENABLE_IF

- enable_if is SFINAE – based method to force the compiler to pick an overload.

```
template<bool B, class T = void>
struct enable_if {};

template<class T>
struct enable_if<true, T> { using type = T; };

template< bool B, class T = void >
using enable_if_t = typename enable_if<B,T>::type;
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

ENABLE_IF

- enable_if is SFINAE based method to force the compiler to pick an overload.

```cpp
template <typename T>
void print (const T& t, std::enable_if_t<!is_container_v<T>, void*> = nullptr) {
    fmt::print("{}\n", t);
}

template <typename T>
void print (const T& t, std::enable_if_t<is_container_v<T>, void*> = nullptr) {
    for (auto&& e : t){
        fmt::print("{}", e);
    }
}


int main(){
    print(18);
    print(std::array<int, 3>{{1, 2, 3}});
};
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

ENABLE_IF

- enable_if is SFINAE based method to force the compiler to pick an overload.

```
template <typename T>
void print (const T& t, std::enable_if_t<!is_container_v<T>, void*> = nullptr) {
    fmt::print("{}\n", t);
}

template <typename T>
void print (const T& t, std::enable_if_t<is_container_v<T>, void*> = nullptr) {
    for (auto&& e : t){
        fmt::print("{}", e);
    }
}


int main(){
    print(18);
    print(std::array<int, 3>{{1, 2, 3}});
};
```

18
123

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

DETECTING A CONTAINER: C++17 IMPLEMENTATION

```
template <typename T>
void print(T t){
    if constexpr (!is_container_v<T>){
        fmt::print("Number: {}\n", t);
    } else {
        fmt::print("Container: ");
        for (auto&& e : t){
            fmt::print("{} ", e);
        }
    }
}

int main(){
    print(2);
    print(std::array<int, 3>{{1,2,3}});
}
```

DETECTING A CONTAINER: C++17 IMPLEMENTATION

```cpp
template <typename T>
void print(T t){
    if constexpr (!is_container_v<T>){
        fmt::print("Number: {}\n", t);
    } else {
        fmt::print("Container: ");
        for (auto&& e : t){
            fmt::print("{} ", e);
        }
    }
}

int main(){
    print(2);
    print(std::array<int, 3>{{1,2,3}});
}
```

Number: 2

Container: 1 2 3

VARIADIC TEMPLATES

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

VARIADIC TEMPLATES : C++17 FOLD-EXPRESSION EXAMPLE

- Here we will check if all types are integral

```
template <typename... T>
struct are_all_integral :
    public std::conjunction<std::is_integral<T>...>{};

template <typename... T>
void check(T... vals){
    static_assert(are_all_integral<T...>::value,
    "All vals must be integral");
}
```

VOID_T

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

VOID_T

- An extremely simple alias template that helps verify well-formedness.
- Can be used for arbitrary member/trait detection
- void_t<T> is well formed void only if T is well-formed, just like enable_if<b, T>::type

VOID_T

```cpp
template< class... >
using void_t = void;
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

VOID_T

```
template< class... >
using void_t = void;
```

Luckily for us its already provided in in type_traits since C++17

Thank You Walter.E Brown ☺

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

CONCEPTS

CONCEPTS

- We have already seen examples of concepts :
  - naïve is_container
  - are_all_integral
  - auto as function parameter

CONCEPTS: IS_CONTAINER

- Let's create a better is_container
  - A container C is a type that can be iterated with range-based for loop
  - Specifically:
    1. std::begin(C&) returns begin Iterator
    2. std::end(C&) returns tail Itererator
    3. beginIter and tailIter comparable with !=
    4. std::next can be used on beginIter
    5. beginIter has * which isn't void
    6. beginIter and tailIter are copy constructible and destructible

CONCEPTS: IS_CONTAINER

- Let's create a better is_container

```cpp
template <typename C>
using TBegin = decltype(std::begin(std::declval<C&>()));

template <typename C>
using TEnd = decltype(std::end(std::declval<C&>()));

template <typename BI, typename EI>
using TNotEqual = decltype(std::declval<BI>() != std::declval<EI>());
```

CONCEPTS: IS_CONTAINER

- Let's create a better is_container

```cpp
template <typename BI>
using TInc = decltype(std::next(std::declval<BI>()));

template <typename BI>
using TDeref = decltype(*std::declval<BI>());
```

CONCEPTS: IS_CONTAINER

- Let's create a better is_container

```
template <typename C, typename = void>
struct is_container : std::false_type {};
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

CONCEPTS: IS_CONTAINER

- Let's create a better is_container

```cpp
template <typename C>
struct is_container<C, std::void_t<
TBegin<C>,
TEnd<C>,
TInc<TBegin<C>>,
TNotEqual<TBegin<C>, TEnd<C>>,
TDeref<TBegin<C>>>> :
std::integral_constant<bool,
std::is_convertible_v<TNotEqual<TBegin<C>, TEnd<C>>, bool>
and not std::is_void_v<TDeref<TBegin<C>>>
and std::is_destructible_v<TBegin<C>>
and std::is_copy_constructible_v<TBegin<C>>
and std::is_destructible_v<TEnd<C>>
and std::is_copy_constructible_v<TEnd<C>>> {};
```

CONCEPTS: IS_CONTAINER

- Usage examples:

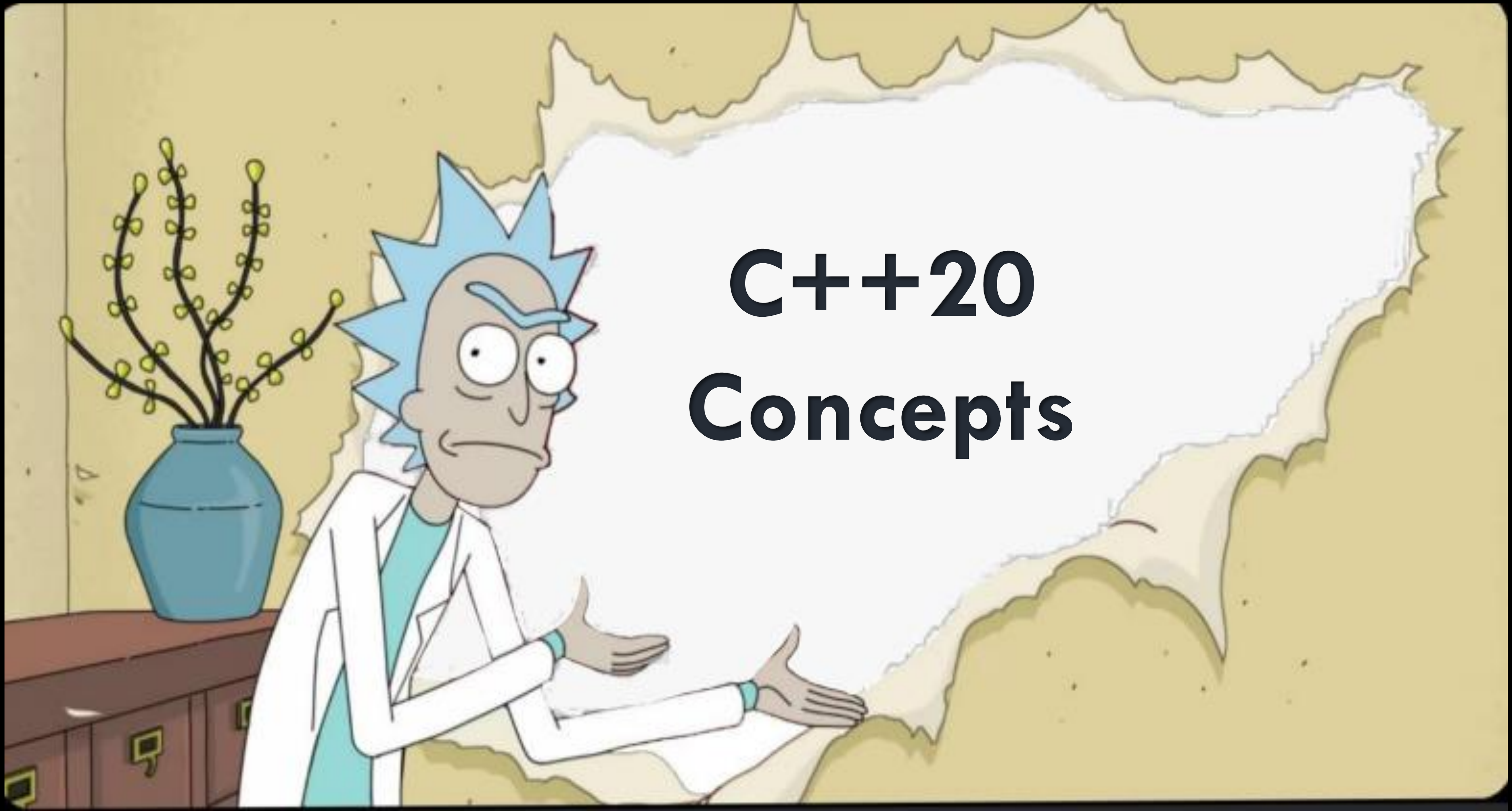
```
template <typename C>
constexpr bool isContainer(const C& c){
    return is_container<C>::value;
}

template <typename C>
constexpr std::enable_if_t<is_container<C>::value, typename C::value_type>
getFirst1(const C& c){
    return *c.begin();
}

template <typename C, std::enable_if_t<is_container<C>::value, bool> = true>
constexpr auto getFirst2(const C& c){
    return *c.begin();
}
```

CONCEPTS: IS_CONTAINER

- Problems ?
  - Its hard to develop new concepts
  - Error messages can be extremely daunting when a concept isn't met
  - enable_if or void_t aren't readable for many people

CONCEPT

- Prol
  - Its
  - Er g when a c
  - en any peo


C++20
Concepts

CONCEPTS: C++20 CONCEPT LIBRARY

- C++20 introduced the standard concepts trait library

  Example of useful concepts:

  same_as<T, U>
  integral<T>
  destructible<T>
  assignable_from<LHS, RHS>
  convertible_to<T, U>
  equality_comparable_with<T, U>
  …

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

CONCEPTS: C++20 CONCEPT LIBRARY

- Example:
  - As we mentioned, auto is the weakest concept that accepts everything:

```
bool foo(auto a, auto b){
    return a == b;
}
```

CONCEPTS: C++20 CONCEPT LIBRARY

- Example:
  - As we mentioned, auto is the weakest concept that accepts everything:

```
foo(1, 1);
```

CONCEPTS: C++20 CONCEPT LIBRARY

- Example:
  - As we mentioned, auto is the weakest concept that accepts everything:

```
foo(1, 1);
```

This is ok and will yield true

CONCEPTS: C++20 CONCEPT LIBRARY

- Example:
  - As we mentioned, auto is the weakest concept that accepts everything:

```
117    foo(1, 1.0);
```

CONCEPTS: C++20 CONCEPT LIBRARY

- Example:
  - As we mentioned, auto is the weakest concept that accepts everything:

```
foo(1, 1.0);
```

This is ok and will yield true but probably not what the writer intended

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

CONCEPTS: C++20 CONCEPT LIBRARY

- Example:
  - As we mentioned, auto is the weakest concept that accepts everything:

```
foo(1, std::vector<int>{});
```

CONCEPTS: C++20 CONCEPT LIBRARY

- Example:
  - As we mentioned, auto is the weakest concept that accepts everything:

```
120     foo(1, std::vector<int>{});
```

This is ok, the function that will be called but
It will fail because these types cannot be compared

CONCEPTS: C++20 CONCEPT LIBRARY

- Example:
  - As we mentioned, auto is the weakest concept that accepts everything:

```
120     foo(1, std::vector<int>{});
```

This is ok, the function that will be called but
It will fail because these types cannot be compared

```
<source>:108:14: error: invalid operands to binary expression ('int' and 'std::vector<int>')
    return a == b;
           ~ ^  ~
```

CONCEPTS: C++20 CONCEPT LIBRARY

- Example:
  - We can avoid this problems by constraining the auto parameter

```
bool foo2(std::integral auto a, std::integral auto b){
    return a == b;
}
```

CONCEPTS: C++20 CONCEPT LIBRARY

- Example:
  - We can avoid this problems by constraining the auto parameter

```
foo2(1, 1);
```

CONCEPTS: C++20 CONCEPT LIBRARY

- Example:
  - We can avoid this problems by constraining the auto parameter

```
foo2(1, 1);
```

This is ok and will yield true

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

CONCEPTS: C++20 CONCEPT LIBRARY

- Example:
  - We can avoid this problems by constraining the auto parameter

```
128     foo2(1, 1.0);
```

CONCEPTS: C++20 CONCEPT LIBRARY

- Example:
  - We can avoid this problems by constraining the auto parameter

```
128     foo2(1, 1.0);
```

The function will not be called and
a nice compile-time error will be produced

CONCEPTS: C++20 CONCEPT LIBRARY

- Example:
  - We can avoid this problems by constraining the auto parameter

```
128     foo2(1, 1.0);
```

The function will not be called and
a nice compile-time error will be produced

```
<source>:111:6: note: candidate template ignored: constraints not satisfied [with a:auto = int, b:auto = double]
bool foo2(std::integral auto a, std::integral auto b){
     ^
<source>:111:33: note: because 'double' does not satisfy 'integral'
bool foo2(std::integral auto a, std::integral auto b){
                                ^
```

CONCEPTS: C++20 CONCEPT LIBRARY

- Example:
  - We can constrain more than auto parameters

```
std::integral auto val_i = 1ul;
std::floating_point auto val_f = 1.f;
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

CONCEPTS: C++20 CONCEPT LIBRARY

- Example:
  - We can constrain more than auto parameters

```
std::integral auto val_i = 1ul;
std::floating_point auto val_f = 1.f;
```

This is ok

CONCEPTS: C++20 CONCEPT LIBRARY

- Example:
  - We can constrain more than auto parameters

```
133     std::floating_point auto val_f2 = val_i;
```

CONCEPTS: C++20 CONCEPT LIBRARY

- Example:
  - We can constrain more than auto parameters

```
std::floating_point auto val_f2 = val_i;
```

This will not compile

CONCEPTS: C++20 CONCEPT LIBRARY

- Example:
  - We can constrain more than auto parameters

```
133      std::floating_point auto val_f2 = val_i;
```

This will not compile

```
<source>:133:10: error: deduced type 'unsigned long' does not satisfy 'floating_point'
    std::floating_point auto val_f2 = val_i;
         ^~~~~~~~~~~~~~~~~~~~~~~
```

CONCEPTS: C++20 CONCEPT LIBRARY

- With C++20 It's easier to create new Concepts:
  - Basic Definition (*constraint-expression*)

CONCEPTS: C++20 CONCEPT LIBRARY

- With C++20 It's easier to create new Concepts:
  - Basic Definition (*constraint-expression)*

```
template <typename T>
concept convertiable_to_int = std::convertible_to<T, int>;
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

CONCEPTS: C++20 CONCEPT LIBRARY

- With C++20 It’s easier to create new Concepts:
  - Constraints
    1. conjunction

```
template <typename T>
concept convertiable_to_int_not_double = convertiable_to_int<T>
                                          and (not std::same_as<T, double>);
```

CONCEPTS: C++20 CONCEPT LIBRARY

- With C++20 It's easier to create new Concepts:
  - Constraints
    2. disjunction

```cpp
template <typename T>
concept int_or_larger = std::same_as<T, int> or sizeof(T) > sizeof(int);
```

CONCEPTS: C++20 CONCEPT LIBRARY

- With C++20 It's easier to create new Concepts:
  - Constraints
    3. atomic

```
template <typename T>
struct SC{
    constexpr bool operator()() const { return true; }
};

template <>
struct SC<bool>{
    constexpr bool operator()() const { return false; }
};

template <typename T>
concept ac = SC<T>{}();
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

CONCEPTS: C++20 CONCEPT LIBRARY

- With C++20 It's easier to create new Concepts:
  - Requires Clause

```
template <typename T> requires (sizeof(T) > sizeof(int))
struct larger_than_int{
    static inline constexpr bool val = true;
};
```

CONCEPTS: C++20 CONCEPT LIBRARY

- With C++20 It's easier to create new Concepts:
  - Requires Clause
    - Must be assembled from primary expressions

```cpp
constexpr bool ret_true(){ return true; }

template <typename T> requires (sizeof(T) > sizeof(int)) and ret_true()
struct larger_than_int1{
    static inline constexpr bool val = true;
};
```

CONCEPTS: C++20 CONCEPT LIBRARY

- With C++20 It's easier to create new Concepts:
  - Requires Clause
    - Must be assembled from primary expressions

```cpp
constexpr bool ret_true(){ return true; }

template <typename T> requires (sizeof(T) > sizeof(int)) and (ret_true())
struct larger_than_int1{
    static inline constexpr bool val = true;
};
```

CONCEPTS: C++20 CONCEPT LIBRARY

- With C++20 It's easier to create new Concepts:
  - Requires Expression

```
template <typename BI, typename EI>
concept neq_on = requires(BI bi, EI ei){
    {bi != ei} -> std::convertible_to<bool>;
};
```

CONCEPTS: C++20 CONCEPT LIBRARY

- With C++20 It's easier to create new Concepts:
  - Requires Expression

```cpp
template <typename BI, typename EI>
concept neq_on = requires(BI bi, EI ei){
    {bi != ei} -> std::convertible_to<bool>;
};
```

CONCEPTS: C++20 CONCEPT LIBRARY

- With C++20 It’s easier to create new Concepts:
  - Requires Expression: what will happen here?

```cpp
template <typename T, typename U>
concept my_same_as = requires(T, U){
    std::same_as<T, U>;
};

void do_only_on_same(auto x, my_same_as<decltype(x)> auto y){
    static_assert(std::same_as<decltype(x), decltype(y)>);
}
```

```cpp
    do_only_on_same(1, 1ul);
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

CONCEPTS: C++20 CONCEPT LIBRARY

- With C++20 It’s easier to create new Concepts:
  - Requires Expression: what will happen here?

```cpp
157 template <typename T, typename U>
158 concept my_same_as = requires(T, U){
159     std::same_as<T, U>;
160 };
161
162 void do_only_on_same(auto x, my_same_as<decltype(x)> auto y){
163     static_assert(std::same_as<decltype(x), decltype(y)>);
164 }
```

```cpp
197     do_only_on_same(1, 1ul);
```

```
<source>:164:5: error: static assertion failed
    static_assert(std::same_as<decltype(x), decltype(y)>);
    ^             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

CONCEPTS: C++20 CONCEPT LIBRARY

- With C++20 It's easier to create new Concepts:
  - Requires Expression: what will happen here?

```cpp
template <typename T, typename U>
concept my_same_as = requires(T, U){
    requires std::same_as<T, U>;
};

void do_only_on_same(auto x, my_same_as<decltype(x)> auto y){
    static_assert(std::same_as<decltype(x), decltype(y)>);
}
```

```cpp
    do_only_on_same(1, 1ul);
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

CONCEPTS: C++20 CONCEPT LIBRARY

- With C++20 It's easier to create new Concepts:
  - Requires Expression: what will happen here?

```
157 template <typename T, typename U>
158 concept my_same_as = requires(T, U){
159     requires std::same_as<T, U>;
160 };
161
162 void do_only_on_same(auto x, my_same_as<decltype(x)> auto y){
163     static_assert(std::same_as<decltype(x), decltype(y)>);
164 }
```

```
197    do_only_on_same(1, 1ul);
```

```
<source>:162:30: note: because 'my_same_as<unsigned long, decltype(x)>' evaluated to false
void do_only_on_same(auto x, my_same_as<decltype(x)> auto y){
                             ^
```

CONCEPTS: C++20 CONCEPT LIBRARY

- With C++20 It's easier to create new Concepts:
  - Requires Expression: types

```
template <typename T>
concept types_check = requires {
    typename T::Type;
    typename SC<T>;
};
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

CONCEPTS: C++20 CONCEPT LIBRARY

- With C++20 It's easier to create new Concepts:
  - AD-HOC constraint

```
template <typename T>
requires requires (T t) { not t; }
struct not_oper_possible_types_only{};
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

CONCEPTS: C++20 CONCEPT LIBRARY

- With C++20 It's easier to create new Concepts:
  - Smile

```cpp
template <typename T>
requires requires { requires std::is_pointer_v<T>;
requires requires { requires std::convertible_to<T, int>;
requires requires { requires sizeof(T) > sizeof(int)
;} ;} ;}
struct Smile{};
```

CONCEPTS: C++20 CONCEPT LIBRARY

- We may use the same concept in many different ways
  - Reminder

```cpp
template <typename BI, typename EI>
concept neq_on = requires(BI bi, EI ei){
    {bi != ei} -> std::convertible_to<bool>;
};
```

CONCEPTS: C++20 CONCEPT LIBRARY

- We may use the same concept in many different ways

```
template <typename EI, neq_on<EI> BI>
constexpr bool fun(BI bi, EI ei){
    return true;
}
```

CONCEPTS: C++20 CONCEPT LIBRARY

- We may use the same concept in many different ways

```cpp
template <typename BI, typename EI>
    requires neq_on<BI, EI>
constexpr bool fun_2(BI bi, EI ei){
    return true;
}
```

CONCEPTS: C++20 CONCEPT LIBRARY

- We can write the same constraint in many ways

```
template <typename BI, typename EI>
constexpr bool fun_3(BI bi, EI ei) requires neq_on<BI, EI>{
    return true;
}
```

CONCEPTS: C++20 CONCEPT LIBRARY

- We can write the same constraint in many ways

```
constexpr bool fun_4(auto bi, neq_on<decltype(bi)> auto ei) {
    return true;
}
```

CONCEPTS: C++20 CONCEPT LIBRARY

- Example:

```
fun(1, 1ul);
```

CONCEPTS: C++20 CONCEPT LIBRARY

- Example:

```
250        fun(1, std::vector<int>(1));
```

```
<source>:186:24: note: because 'neq_on<int, std::vector<int> >' evaluated to false
template <typename EI, neq_on<EI> BI>
                       ^
<source>:163:9: note: because 'bi != ei' would be invalid: invalid operands to binary expression ('int' and 'std::vector<int>')
    {bi != ei} -> std::convertible_to<bool>;
        ^
```

REMINDER: C++17 FOLD-EXPRESSION EXAMPLE

- C++17 style concept

```
template <typename... T>
struct are_all_integral :
    public std::conjunction<std::is_integral<T>...>{};

template <typename... T>
void check(T... vals){
    static_assert(are_all_integral<T...>::value,
    "All vals must be integral");
}
```

C++20 ALL INTEGRAL

- C++20 style concept

```cpp
template <typename... T>
bool check_integral(T...)
requires std::conjunction_v<std::is_integral<T>...> {
    return true;
}
```

C++20 ALL INTEGRAL

- C++20 style concept

```
template <typename... T>
bool check_integral(T...)
requires std::conjunction_v<std::is_integral<T>...> {
    return true;
}
```

Is it a good way?

ITS NICE BUT NOT PERFECT

```
<source>:275:5: error: no matching function for call to 'check_integral'
    check_integral(1, 1ul, 2.);
    ^~~~~~~~~~~~~~
<source>:221:6: note: candidate template ignored: constraints not satisfied [with T = <int, unsigned long, double>]
bool check_integral(T...)
     ^
<source>:222:10: note: because 'std::conjunction_v<std::is_integral<int>, std::is_integral<unsigned long>, std::is_integral<double> >' evaluated to false
requires std::conjunction_v<std::is_integral<T>...> {
         ^
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

PARTIAL ORDERING OF CONSTRAINTS

```cpp
template <typename... T>
bool check_integral_2(T...)
requires (std::integral<T> and ...) {
    return true;
}
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

PARTIAL ORDERING OF CONSTRAINTS

```
<source>:278:5: error: no matching function for call to 'check_integral_2'
    check_integral_2(1, 2, 2.);
    ^~~~~~~~~~~~~~~~
<source>:227:6: note: candidate template ignored: constraints not satisfied [with T = <int, int, double>]
bool check_integral_2(T...)
     ^
<source>:228:11: note: because 'double' does not satisfy 'integral'
requires (std::integral<T> and ...) {
          ^
```

PARTIAL ORDERING OF CONSTRAINTS: EXAMPLE

```
template <auto K>
class DoStuff{
    public:
    explicit(false) operator std::string_view() const {
        return str_;
    }

    private:
    //turn the integer into a string
    std::string get_str() requires std::integral<decltype(K)>{
        return fmt::format("{}", K);
    }

    //if the number is larger than max devide by 1024 and set main and multiplier
    std::string get_str() requires std::integral<decltype(K)> and
    (K >= std::numeric_limits<short>::max()) {
        return fmt::format("main: {}, mult: {}", static_cast<double>(K)/1024., 1024);
    }

    //its a string lest return it
    std::string get_str() requires std::convertible_to<decltype(K), std::string_view>{
        return K;
    }

    std::string str_ = get_str();
};

int main(){
    fmt::print("doing stuff with {}\n", std::string_view(DoStuff<10>()));
    //doing stuff with 10
    fmt::print("doing stuff with {}\n",
    std::string_view(DoStuff<std::numeric_limits<short>::max()>()));
    //doing stuff with main: 31.9990234375, mult: 1024
}
```

SHORT DETOUR: POINTER CONCEPT

- Reminder:

```cpp
#include <fmt/core.h>

void print(auto const& val){
    fmt::print("This is a ref val: {}", val);
}

void print(auto const* val){
    fmt::print("This is a pointer val: {}", *val);
}

int main(){
    int i = 10;
    print(&i);
}
```

SHORT DETOUR: POINTER CONCEPT

SHORT DETOUR: POINTER CONCEPT

```
template <>
void print<int*>(int* const& val)
{
  fmt::print("This is a ref val: {}", val);
}
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

SHORT DETOUR: POINTER CONCEPT

```
template <>
void print<int*>(int* const& val)
{
  fmt::print("This is a ref val: {}", val);
}
```

```
template <>
void print<int*>(int* const* val)
{
  fmt::print("This is a pointer val: {}", *val);
}
```

SHORT DETOUR: POINTER CONCEPT

- How to fix it:

SHORT DETOUR: POINTER CONCEPT

- How to fix it:

```
print<int>(&i);
```

SHORT DETOUR: POINTER CONCEPT

- How to fix it:

```
print<int>(&i);
```

```
template <>
void print<int>(int const& val)
{
  fmt::print("This is a ref val: {}", val);
}

template <>
void print<int>(int const* val)
{
  fmt::print("This is a pointer val: {}", *val);
}
```

SHORT DETOUR: POINTER CONCEPT

- How to fix it:

SHORT DETOUR: POINTER CONCEPT

- How to fix it:

```
int const* j = &i;
```

SHORT DETOUR: POINTER CONCEPT

- How to fix it:

```
int const* j = &i;
```

```
template <>
void print<int>(int const* val)
{
   fmt::print("This is a pointer val: {}", *val);
}
```

SHORT DETOUR: POINTER CONCEPT

- How to fix it: Concepts to the rescue

```
template <typename T>
concept pointer =  std::same_as<T, void*> or
requires (T t) { *t; };
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

SHORT DETOUR: POINTER CONCEPT

SHORT DETOUR: POINTER CONCEPT

```cpp
void print(const auto& t){
    fmt::print("{}", t);
}

void print(const pointer auto& t){
    fmt::print("{}", *t);
}
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

CONCEPTS: IS_CONTAINER

```
template <typename C>
concept not_equal_begin_end = requires (C c){
    { std::begin(c) != std::end(c) } -> std::same_as<bool>;
};

template <typename C>
concept has_begin_and_end = requires (C c) {
    std::begin(c);
    std::end(c);
};
```

CONCEPTS: IS_CONTAINER

```cpp
template <typename C>
concept incrementable_begin = requires(C c){
    std::begin(c)++;
};

template <typename C>
concept dereferenciable_begin_not_void = requires(C c){
   requires not std::same_as<decltype(*std::begin(c)), void>;
};
```

CONCEPTS: IS_CONTAINER

```cpp
template <typename C>
concept begin_and_end_copy_constructible_and_destructible = requires (C c) {
    requires std::copy_constructible<decltype(std::begin(c))>;
    requires std::copy_constructible<decltype(std::end(c))>;
    requires std::destructible<decltype(std::begin(c))>;
    requires std::destructible<decltype(std::end(c))>;
};
```

CONCEPTS: IS_CONTAINER

```
template <typename C>
concept container = has_begin_and_end<C>                                             and
                    incrementable_begin<C>                                           and
                    dereferenciable_begin_not_void<C>                                and
                    begin_and_end_copy_constructible_and_destructible<C>;
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

CONCEPTS: IS_CONTAINER

Usage Examples:

```cpp
bool is_first_element_the_same(Container auto c1, Container auto c2){
    return *std::begin(c1) == *std::begin(c2);
};

int main(){
    std::vector v{1, 2, 3};
    std::array  a{1, 2, 3};
    return is_first_element_the_same(v, a) ? 0 : 1;
}
```

CONCEPTS: IS_CONTAINER

Error Example:

```cpp
int main(){
    std::vector v{1, 2, 3};
    std::tuple  a{1, 2, 3};
    return is_first_element_the_same(v, a) ? 0 : 1;
}
```

CONCEPTS: IS_CONTAINER

Error Example:

```
<source>:292:12: error: no matching function for call to 'is_first_element_the_same'
    return is_first_element_the_same(v, a) ? 0 : 1;
           ^~~~~~~~~~~~~~~~~~~~~~~~~
<source>:285:6: note: candidate template ignored: constraints not satisfied [with c1:auto = std::vector<int>, c2:auto = std::tuple<int, int, int>]
bool is_first_element_the_same(Container auto c1, Container auto c2){
     ^
<source>:285:51: note: because 'std::tuple<int, int, int>' does not satisfy 'Container'
bool is_first_element_the_same(Container auto c1, Container auto c2){
                                                  ^
<source>:279:21: note: because 'std::tuple<int, int, int>' does not satisfy 'has_begin_and_end'
concept Container = has_begin_and_end<C>                                  and
                    ^
<source>:248:5: note: because 'std::begin(c)' would be invalid: no matching function for call to 'begin'
    std::begin(c);
    ^
```

CONCLUSION

- Concepts simplify the code
- Concepts make the code More readable and maintainable
- Makes Metaprogramming easier
- Make the compiler errors much clearer

CONCLUSION

- Concepts simplify the code
- Concepts make the code More readable and maintainable
- Makes Metaprogramming easier
- Make the compiler errors much clearer

```
template <typename C>
struct is_container<C, std::void_t<
TBegin<C>,
TEnd<C>,
TInc<TBegin<C>>,
TNotEqual<TBegin<C>, TEnd<C>>,
TDeref<TBegin<C>>>> :
std::integral_constant<bool,
std::is_convertible_v<TNotEqual<TBegin<C>, TEnd<C>>, bool>
and not std::is_void_v<TDeref<TBegin<C>>>
and std::is_destructible_v<TBegin<C>>
and std::is_copy_constructible_v<TBegin<C>>
and std::is_destructible_v<TEnd<C>>
and std::is_copy_constructible_v<TEnd<C>>> {};
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

CONCLUSION

- Concepts simplify the code
- Concepts make the code More readable and maintainable
- Makes Metaprogramming easier
- Make the compiler errors much clearer

```
template <typename C>
struct is_container<C, std::void_t<
TBegin<C>,
TEnd<C>,
TInc<TBegin<C>>,
TNotEqual<TBegin<C>, TEnd<C>>,
TDeref<TBegin<C>>>> :
std::integral_constant<bool,
std::is_convertible_v<TNotEqual<TBegin<C>, TEnd<C>>, bool>
and not std::is_void_v<TDeref<TBegin<C>>>
and std::is_destructible_v<TBegin<C>>
and std::is_copy_constructible_v<TBegin<C>>
and std::is_destructible_v<TEnd<C>>
and std::is_copy_constructible_v<TEnd<C>>> {};
```

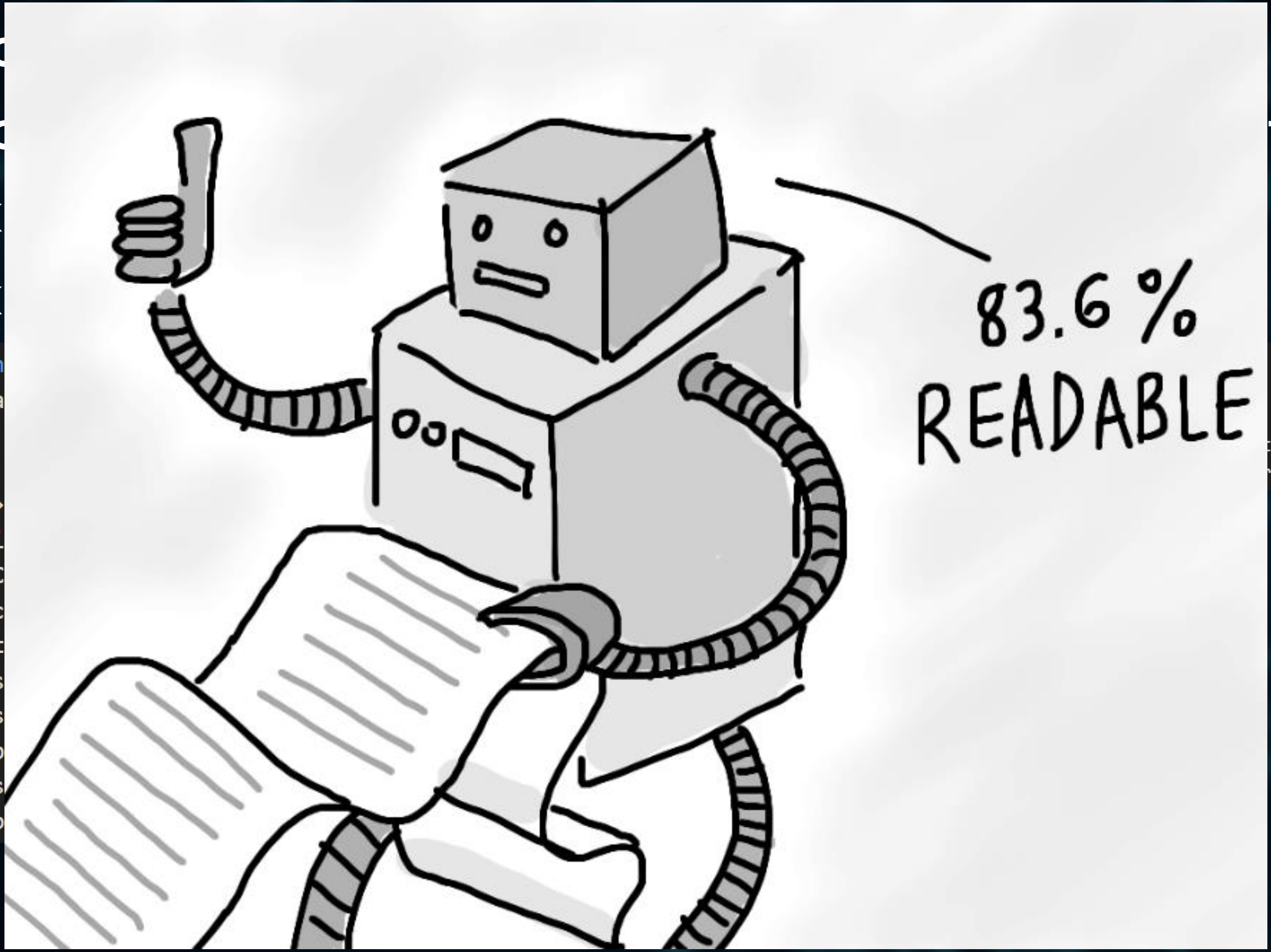
```
template <typename C>
concept container = has_begin_and_end<C>                                     and
                    incrementable_begin<C>                                   and
                    dereferenciable_begin_not_void<C>                        and
                    begin_and_end_copy_constructible_and_destructible<C>;
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

CONCLUSION

- Conc
- Conc … tainable
- Mak
- Mak

```
template <typen
struct is_conta
TBegin<C>,
TEnd<C>,
TInc<TBegin<C>>
TNotEqual<TBegi
TDeref<TBegin<C
std::integral_c
std::is_convert
and not std::is
and std::is_des
and std::is_cop
and std::is_des
and std::is_cop
```




```
                                                    and
                                                    and
_void<C>                                            and
uctible_and_destructible<C>;
```

Alex Dathshkovsky | alex.dathshkovsky@speedata.io | www.linkedin.com/in/alexdathshkovsky

QUESTIONS

?

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

THANK YOU FOR LISTENING

Alex Dathskovsky

+97254-7685001

alex.dathskovsky@speedata.io

www.linkedin.com/in/alexdathskovsky

Link to presented code: https://godbolt.org/z/W6zvzMzv7