From Templates to Concepts

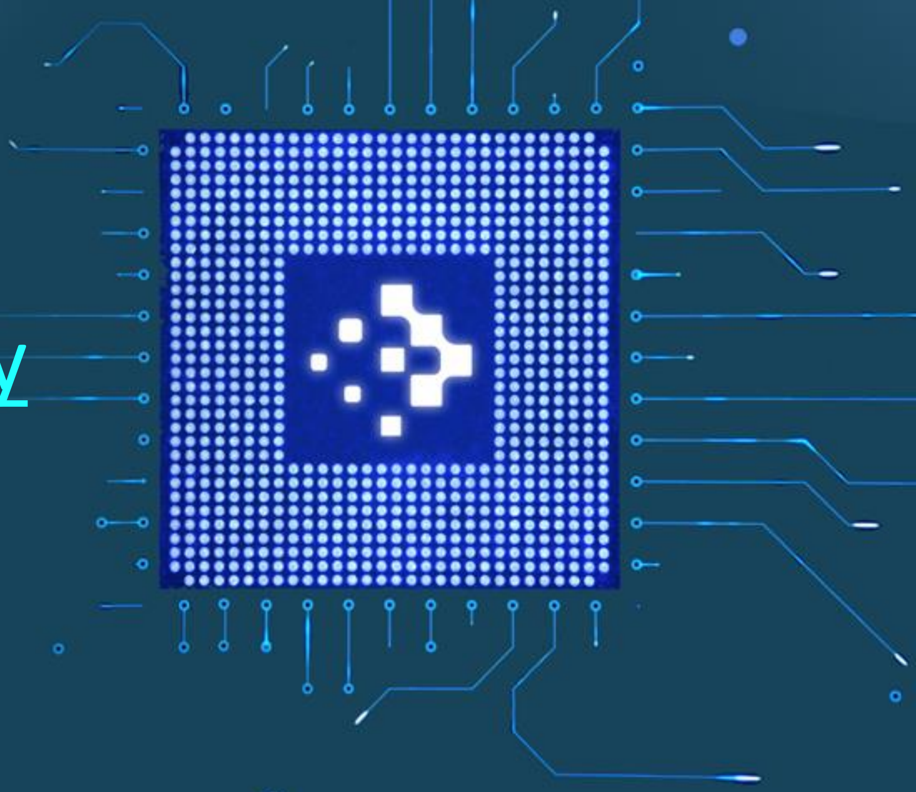
The Amazing Journey of Metaprogramming

ALEX DATHSKOVSKY

CppCon
The C++ Conference
2022
September 12th-16th

About Me:

SPEEDATA



calebxyz@gmail.com

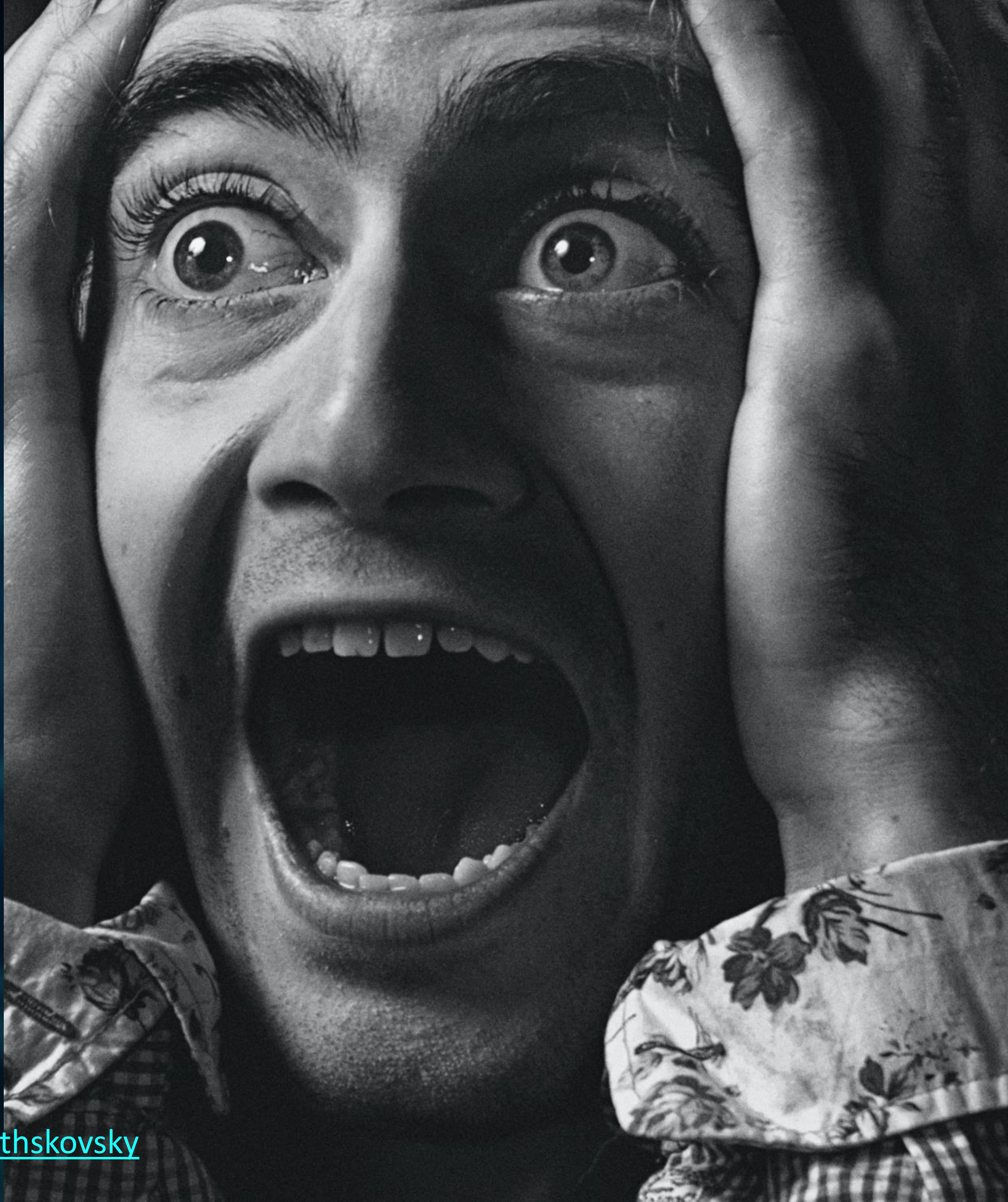
www.linkedin.com/in/alexdathskovsky

Templates:

What's the first thing that comes to mind?

Templates:

What's the first thing that comes to mind?



Alex Dathshkovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathskovsky

BASIC TEMPLATE RULES

Alex Dathskovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathskovsky

FUNCTION TEMPLATES

- Function templates can be fully specialized or overloaded
- Partial specialization is not allowed

FUNCTION TEMPLATES

- This is ok:

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

template <>
void print(int t) {fmt::print("T:Int=={}\n", t);};


int main() {
  print("1");
  print(1);
}
```

FUNCTION TEMPLATES

- This is ok:

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

template <>
void print(int t) {fmt::print("T:Int=={}\n", t);};


int main() {
    print("1");
    print(1);
}
```

T: 1

T:Int==1

Alex Dathskovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathskovsky

FUNCTION TEMPLATES

- This is not allowed:

```
6   template <typename T>
7   void print(T t) {fmt::print("T: {}\n", t);};
8
9   template <typename T>
10  void print<T*>(T* p) {fmt::print("T*: {}\n", *p);}
```

```
<source>:10:6: error: function template partial specialization is not allowed
void print<T*>(T* p) {fmt::print("T: {}\n", *t);};
```

FUNCTION TEMPLATES

- But we can overload

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

template <typename T>
void print(T* p) {fmt::print("T*: {}\n", *p);};


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

FUNCTION TEMPLATES

- But we can overload

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

template <typename T>
void print(T* p) {fmt::print("T*: {}\n", *p);};


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

T*: 1
T: 1

Overloading and specializations are tricky.

FUNCTION TEMPLATES

```cpp
template <typename T>
void print(T) {fmt::print("Generic");};
template <typename T>
void print(T*) {fmt::print("Overload");};
template<>
void print(double*) { fmt::print("Specialization");};

int main(){
    double d = 1.5;
    print(&d);
};
```

Alex Dathskovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathskovsky

FUNCTION TEMPLATES

```cpp
template <typename T>
void print(T) {fmt::print("Generic");};
template <typename T>
void print(T*) {fmt::print("Overload");};
template<>
void print(double*) { fmt::print("Specialization");};

int main(){
    double d = 1.5;
    print(&d);
};
```

Specialization

Alex Dathskovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathskovsky

FUNCTION TEMPLATES

```cpp
template <typename T>
void print(T) {fmt::print("Generic");};
template<>
void print(double*) { fmt::print("Specialization");};
template <typename T>
void print(T*) {fmt::print("Overload");};

int main(){
    double d = 1.5;
    print(&d);
};
```

Alex Dathshkovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathshkovsky

FUNCTION TEMPLATES

```
template <typename T>
void print(T) {fmt::print("Generic");};
template<>
void print(double*) { fmt::print("Specialization");};
template <typename T>
void print(T*) {fmt::print("Overload");};

int main(){
    double d = 1.5;
    print(&d);
};
```

Overload

Overload resolution considers only base templates.

CLASS TEMPLATES

- Support partial and complete specialization.

```cpp
template <typename T>
struct more {};
template <typename T>
struct more<T*> {};
template<>
struct more<int>{};


int main() {
    auto m1 = more<double>();
    auto m2 = more<double*>();
    auto m3 = more<int>();
}
```

TRAIT LIBRARY

Alex Dathskovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathskovsky

TRAITS

- C++11 introduced the standard type trait library

  Example of useful traits:

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

Alex Dathskovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathskovsky

CONSTRAINTS WITH TRAITS

Alex Dathskovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathskovsky

TRAITS

- Will this always work?

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

Alex Dathskovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathskovsky

TRAITS

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

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

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

Alex Dathskovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathskovsky

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):

```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<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

```
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

```
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);
}
```

Alex Dathshkovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathskovsky

TRAITS

- We can simplify things with Tag Dispatch

  std::is_pointer<T>::type is std::true_type or std::false_type;

```cpp
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);
};
```

TRAITS

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

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

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

Alex Dathskovsky | calebxyz@gmail.com | 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

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

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

Alex Dathshkovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathskovsky

CONTAINER DETECTION

Alex Dathskovsky | calebxyz@gmail.com | 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

Alex Dathskovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathskovsky

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

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

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

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

Alex Dathskovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathskovsky

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 | calebxyz@gmail.com | 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);
        }
    }
}
```

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

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;
```

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}});
};
```

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}});
};
```

18
123

Alex Dathskovsky | calebxyz@gmail.com | 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}});
}
```

Alex Dathskovsky | calebxyz@gmail.com | 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}});
}
```

Number: 2

Container: 1 2 3

Alex Dathskovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathskovsky

VARIADIC TEMPLATES

Alex Dathshkovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathshkovsky

VARIADIC TEMPLATES : C++17 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");
}
```

Alex Dathskovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathskovsky

VOID_T

Alex Dathskovsky | calebxyz@gmail.com | 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

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

Alex Dathshkovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathshkovsky

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 ☺

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. beginIter has ++
    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

```
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 TNotEqualAble = decltype(std::declval<BI>() != std::declval<EI>());
```

CONCEPTS: IS_CONTAINER

- Let's create a better is_container

```
template <typename BI>
using TIncable = decltype(++std::declval<BI&>());

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

Alex Dathskovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathskovsky

CONCEPTS: IS_CONTAINER

- Let's create a better is_container

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

Alex Dathskovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathskovsky

CONCEPTS: IS_CONTAINER

- Let's create a better is_container

```
template <typename C>
struct is_container<C, std::void_t<
TBegin<C>,
TEnd<C>,
TIncable<TBegin<C>>,
TINeq<TBegin<C>, TEnd<C>>,
TDerefable<TBegin<C>>>>:
std::integral_constant<bool,
std::is_convertible_v<TINeq<TBegin<C>, TEnd<C>>, bool>
&& !std::is_void_v<TDerefable<TBegin<C>>>
&& std::is_destructible_v<TBegin<C>>
&& std::is_copy_constructible_v<TBegin<C>>
&& std::is_destructible_v<TEnd<C>>
&& std::is_copy_constructible_v<TEnd<C>>> {};
```

CONCEPTS: IS_CONTAINER

- Usage examples:

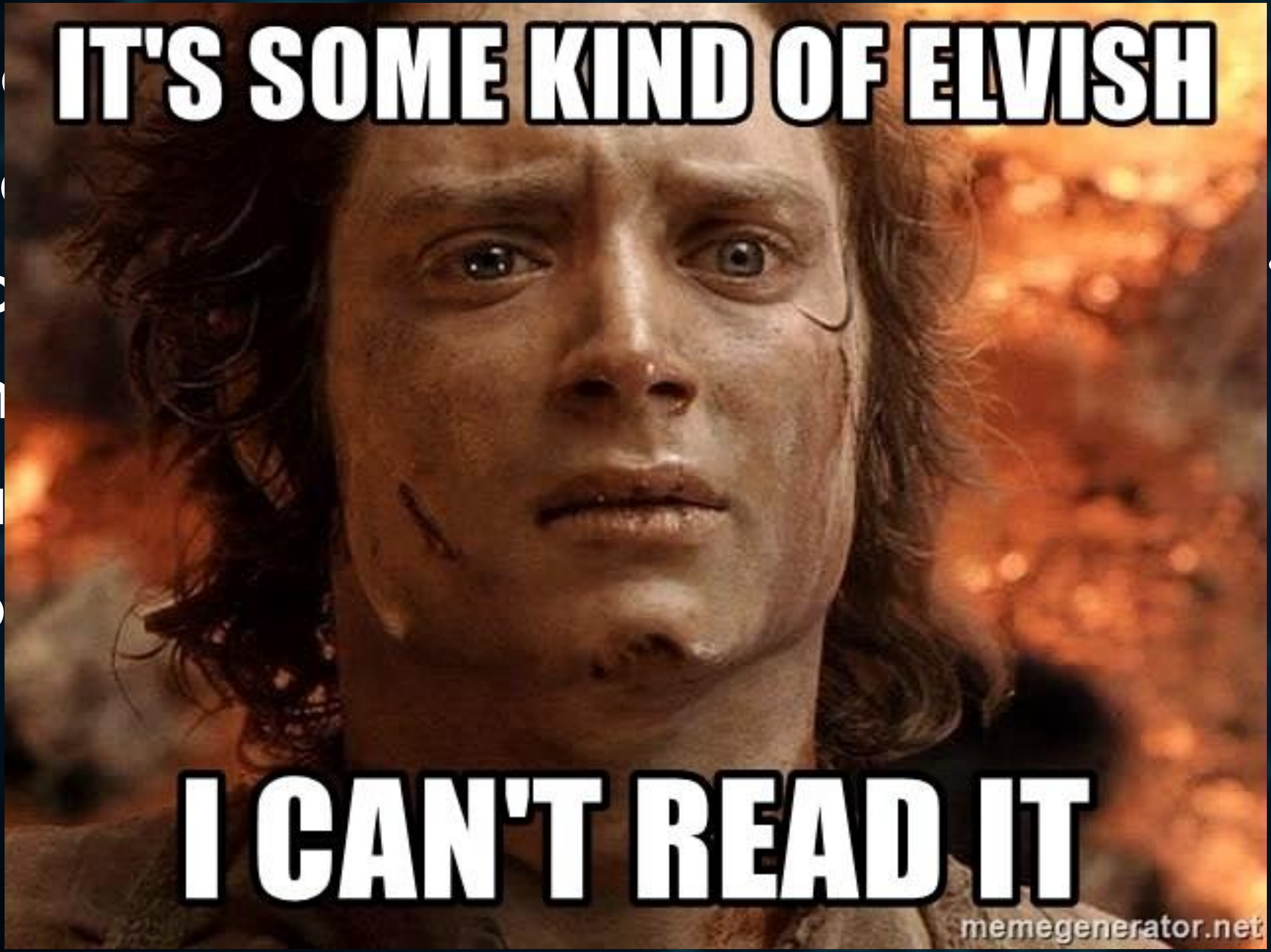
```
template <typename C>
constexpr bool isContainer(const C& c){
    return is_container<C>::value;
}

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();
}
```

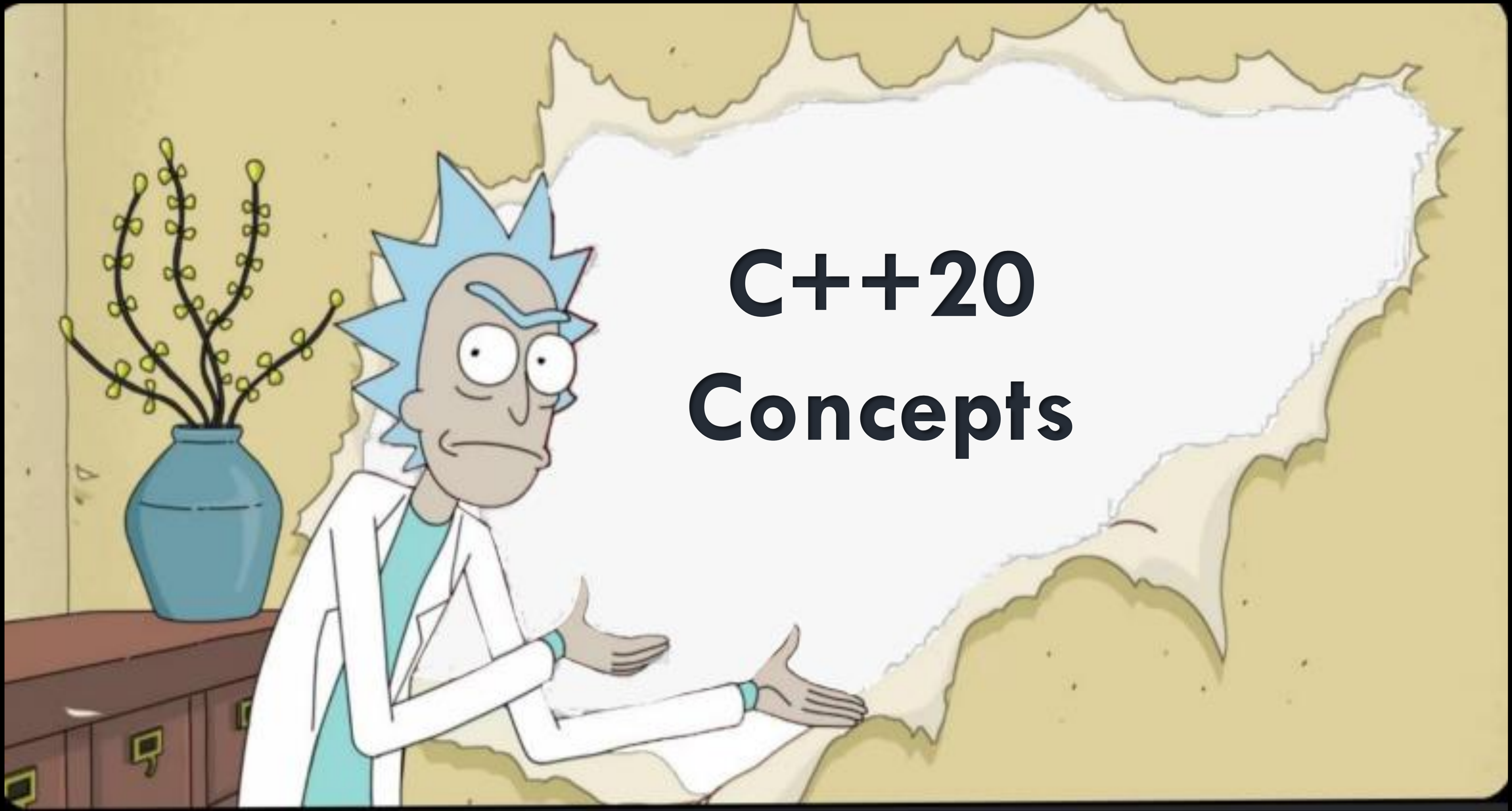
CONCEPTS: IS_CONTAINER

- 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

CONCEPTS: IS_CONTAINER

- Probl
  - Its h
  - Erro   ing when a con
  - ena   many peop





Alex Dathskovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathskovsky

CONCEPTS: C++20 CONCEPT LIBRARY

- With C++20 It's easier to create and use a Concepts

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

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

CONCEPTS: C++20 CONCEPT LIBRARY

- With C++20 It's easier to create and use a Concepts

```
foo(int(), long());
```

CONCEPTS: C++20 CONCEPT LIBRARY

- With C++20 It's easier to create and use a Concepts

```
foo(int(), std::vector<int>::iterator());
```

```
bool foo(BI, EI){
^
<source>:10:24: note: because
'Nequable<__gnu_cxx::__normal_iterator<int *, std::vector<int> >, int>' evaluated to false
```

```
template <typename BI, Nequable<BI> EI>
^
<source>:7:10: note: because 'bi != ei' would be invalid:
invalid operands to binary expression
```

CONCEPTS: C++20 CONCEPT LIBRARY

- We can write the same constraint in many ways

```
template <typename BI, typename EI>
    requires Neqable<BI, EI>
constexpr bool foo_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 foo_3(BI bi, EI ei) requires Neqable<BI, EI>{
    return true;
}
```

CONCEPTS: C++20 CONCEPT LIBRARY

- We can write the same constraint in many ways

```cpp
constexpr bool foo_4(auto bi, Neqable<decltype(bi)> auto ei) {
    return true;
}
```

CONCEPTS: IS_CONTAINER

Let's implement all other concepts that are needed

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

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

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

Alex Dathskovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathskovsky

CONCEPTS: IS_CONTAINER

Let’s implement all other concepts that are needed

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

template <typename C>
concept BeginDerefable  = requires(C c) {
     *std::begin(c);
};

template <typename C>
concept BeginDerefToVoid = requires(C c) {
     { *std::begin(c) } -> std::same_as<void>;
};
```

Alex Dathskovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathskovsky

CONCEPTS: IS_CONTAINER

Let’s implement all other concepts that are needed

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

Alex Dathskovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathskovsky

CONCEPTS: IS_CONTAINER

- Now let's implement all other concepts that are needed

```
template <typename C>
concept Container =
    Beginable<C> && Endable<C> &&
    BeginIncrementable<C> && BeginDerefable<C> &&
    NeqableBeginAndEnd<C> &&
    !BeginDerefToVoid<C> &&
    BeginAndEndCopyConstructibleAndDestructible<C>;
```

Alex Dathskovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathskovsky

CONCEPTS: IS_CONTAINER

Usage Examples:

```
static_assert(Container<std::vector<int>>, "Must Be a container");
```

CONCEPTS: IS_CONTAINER

Usage Examples:

```
    static_assert(Container<std::vector<int>>, "Must Be a container");
```

```
constexpr bool isFirstElemTheSame(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, 1, 1};
    return isFirstElemTheSame(v, a);
}
```

CONCEPTS: IS_CONTAINER

Error Example:

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

int main(){
    std::vector v{1, 2, 3};
    std::tuple t{1, "hello"sv};
    return isFirstElemTheSame(v, t);
}
```

Alex Dathskovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathskovsky

CONCEPTS: IS_CONTAINER

Error Example:

```
<source>:149:12: error: no matching function for call to 'isFirstElemTheSame'
    return isFirstElemTheSame(v, t);
           ^~~~~~~~~~~~~~~~~~
<source>:142:16: note: candidate template ignored: constraints not satisfied [with c1:auto = std::vector<int>, c2:auto = std::tuple<int, std::basic_string_view<char>>]
constexpr bool isFirstElemTheSame(Container auto c1, Container auto c2){
               ^
<source>:142:54: note: because 'std::tuple<int, std::basic_string_view<char>>' does not satisfy 'Container'
constexpr bool isFirstElemTheSame(Container auto c1, Container auto c2){
                                                     ^
<source>:105:5: note: because 'std::tuple<int, std::basic_string_view<char>>' does not satisfy 'Beginable'
    Beginable<C> && Endable<C> &&
    ^
<source>:71:5: note: because 'std::begin(c)' would be invalid: no matching function for call to 'begin'
    std::begin(c);
    ^
```

Alex Dathskovsky | calebxyz@gmail.com | 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

CONCLUSION

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

```cpp
template <typename C>
struct is_container<C, std::void_t<
TBegin<C>,
TEnd<C>,
TIncable<TBegin<C>>,
TINeq<TBegin<C>, TEnd<C>>,
TDerefable<TBegin<C>>>>:
std::integral_constant<bool,
std::is_convertible_v<TINeq<TBegin<C>, TEnd<C>>, bool>
&& !std::is_void_v<TDerefable<TBegin<C>>>
&& std::is_destructible_v<TBegin<C>>
&& std::is_copy_constructible_v<TBegin<C>>
&& std::is_destructible_v<TEnd<C>>
&& std::is_copy_constructible_v<TEnd<C>>> {};
```

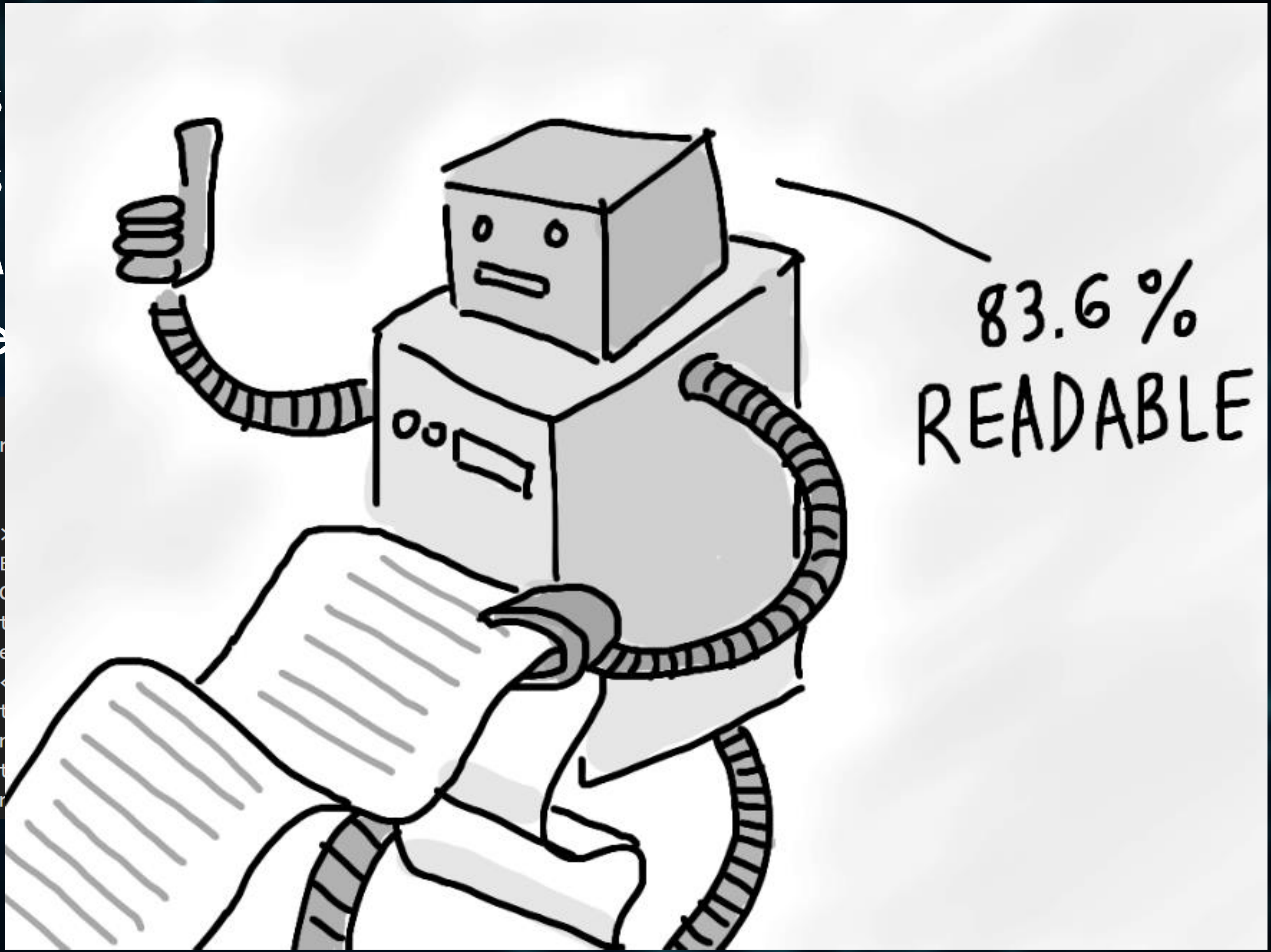
```cpp
template <typename C>
concept Container =
    Beginable<C> && Endable<C> &&
    BeginIncrementable<C> && BeginDerefable<C> &&
    NeqableBeginAndEnd<C> &&
    !BeginDerefToVoid<C> &&
    BeginAndEndCopyConstructibleAndDestructible<C>;
```

Alex Dathskovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathskovsky

CONCLUSION

- Concepts
- Concepts ... le
- Makes M
- Make the

```
template <typename
struct is_container
TBegin<C>,
TEnd<C>,
TIncable<TBegin<C>>
TINeq<TBegin<C>, TE
TDerefable<TBegin<C
std::integral_const
std::is_convertible
&& !std::is_void_v<
&& std::is_destruct
&& std::is_copy_con
&& std::is_destruct
&& std::is_copy_con
```



Alex Dathskovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathskovsky

QUESTIONS

?

Alex Dathskovsky | calebxyz@gmail.com | www.linkedin.com/in/alexdathskovsky

THANK YOU FOR LISTENING

Alex Dathshkovsky

+97254-7685001

calebxyz@gmail.com

www.linkedin.com/in/alexdathskovsky