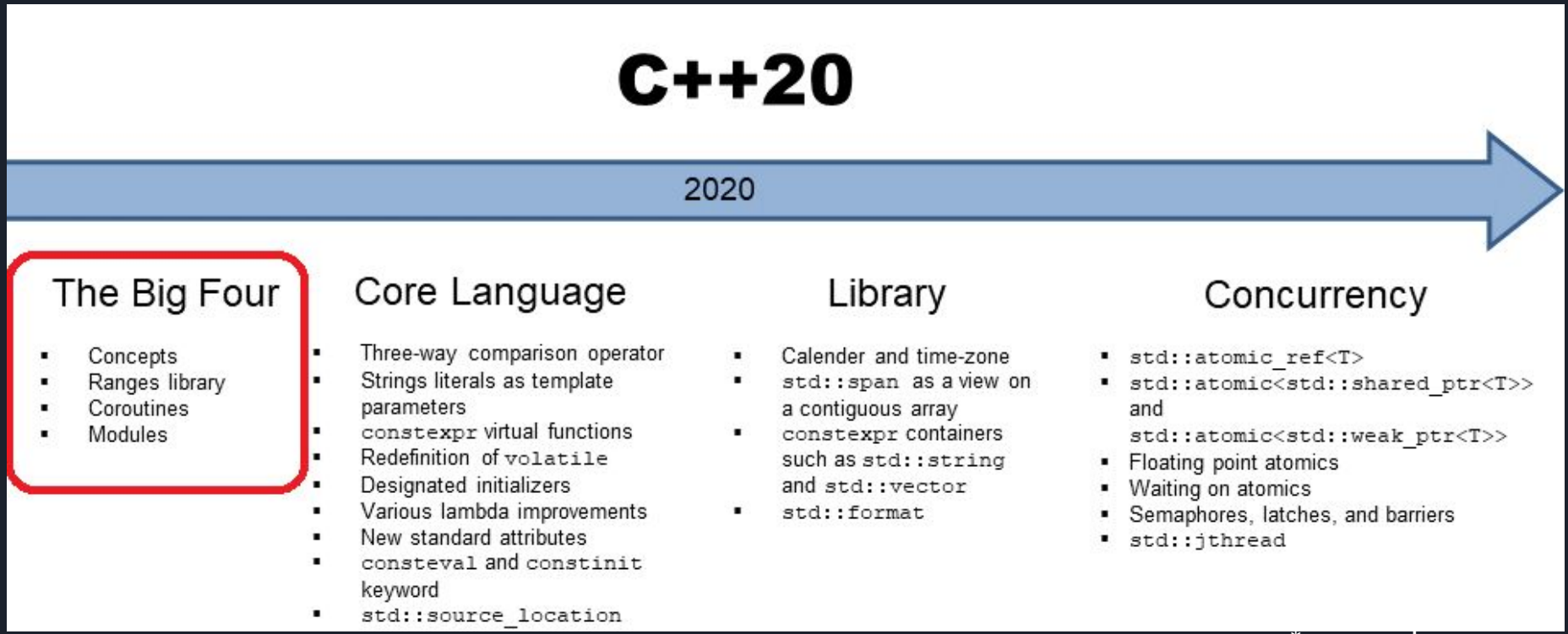
Easing into concepts

Mr.K

Big 4 C++20 features



*src: moderncpp.com

Table of contents

Overview of Concepts

Problems with templates

```
template <typename T>
T add(const T &a, const T &b)
{
    return a + b;
}
```

- Can pass anything
- Can produce unexpected results
- Or long cryptic error messages
- Increase in compile times

```
Could not execute the program
Compiler returned: 1
Compiler stderr
In file included from /opt/compiler-explorer/gcc-11.1.0/include/c++/11.1.0/algorithm:62,
                 from <source>:2:
/opt/compiler-explorer/gcc-11.1.0/include/c++/11.1.0/bits/stl_algo.h: In instantiation of 'void std::__sort(_RandomAccessIterator, _RandomAccessIterator, _Compare) [with
_RandomAccessIterator = std::_List_iterator<int>; _Compare = __gnu_cxx::__ops::_Iter_less_iter]':
/opt/compiler-explorer/gcc-11.1.0/include/c++/11.1.0/bits/stl_algo.h:4842:18:   required from 'void std::sort(_RAIter, _RAIter) [with _RAIter = std::_List_iterator<int>]'
<source>:7:11:   required from here
/opt/compiler-explorer/gcc-11.1.0/include/c++/11.1.0/bits/stl_algo.h:1955:50: error: no match for 'operator-' (operand types are 'std::_List_iterator<int>' and
'std::_List_iterator<int>')
 1955 |                               std::__lg(__last - __first) * 2,
      |                                         ~~~~~~~~~^~~~~~~~
In file included from /opt/compiler-explorer/gcc-11.1.0/include/c++/11.1.0/bits/stl_algobase.h:67,
                 from /opt/compiler-explorer/gcc-11.1.0/include/c++/11.1.0/list:60,
                 from <source>:1:
/opt/compiler-explorer/gcc-11.1.0/include/c++/11.1.0/bits/stl_iterator.h:557:5: note: candidate: 'template<class _IteratorL, class _IteratorR> constexpr decltype ((__y.base()
- __x.base())) std::operator-(const std::reverse_iterator<_Iterator>&, const std::reverse_iterator<_IteratorR>&)'
  557 |     operator-(const reverse_iterator<_IteratorL>& __x,
      |     ^~~~~~~~
/opt/compiler-explorer/gcc-11.1.0/include/c++/11.1.0/bits/stl_iterator.h:557:5: note:   template argument deduction/substitution failed:
In file included from /opt/compiler-explorer/gcc-11.1.0/include/c++/11.1.0/algorithm:62,
                 from <source>:2:
/opt/compiler-explorer/gcc-11.1.0/include/c++/11.1.0/bits/stl_algo.h:1955:50: note:   'std::_List_iterator<int>' is not derived from 'const std::reverse_iterator<_Iterator>'
 1955 |                               std::__lg(__last - __first) * 2,
      |                                         ~~~~~~~~~^~~~~~~~
In file included from /opt/compiler-explorer/gcc-11.1.0/include/c++/11.1.0/bits/stl_algobase.h:67,
                 from /opt/compiler-explorer/gcc-11.1.0/include/c++/11.1.0/list:60,
                 from <source>:1:
/opt/compiler-explorer/gcc-11.1.0/include/c++/11.1.0/bits/stl_iterator.h:1639:5: note: candidate: 'template<class _IteratorL, class _IteratorR> constexpr decltype
((__x.base() - __y.base())) std::operator-(const std::move_iterator<_IteratorL>&, const std::move_iterator<_IteratorR>&)'
 1639 |     operator-(const move_iterator<_IteratorL>& __x,
      |     ^~~~~~~~
/opt/compiler-explorer/gcc-11.1.0/include/c++/11.1.0/bits/stl_iterator.h:1639:5: note:   template argument deduction/substitution failed:
In file included from /opt/compiler-explorer/gcc-11.1.0/include/c++/11.1.0/algorithm:62,
                 from <source>:2:
/opt/compiler-explorer/gcc-11.1.0/include/c++/11.1.0/bits/stl_algo.h:1955:50: note:   'std::_List_iterator<int>' is not derived from 'const std::move_iterator<_IteratorL>'
 1955 |                               std::__lg(__last - __first) * 2,
      |                                         ~~~~~~~~~^~~~~~~~
```

```cpp
#include <list>
#include <algorithm>

int main()
{
    std::list<int> id{36, 21, 57, 19, 94 };
    std::sort(id.begin(), id.end());
    return 0;
}
```

Why does this happen??

Just named requirements

```
#define RandomAccessIterator typename
```

```
template <RandomAccessIterator RandomIt>
void sort(RandomIt first, RandomIt last)
{
    //.......
}
```

```
#define TotallyOrdered typename
#define Pointer typename
#define Integral typename
#define InputIterator typename
#define OutputIterator typename
#define ForwardIterator typename
#define BidirectionalIterator typename
#define RandomAccessIterator typename
#define Incrementable typename
#define StrictWeakOrdering typename
#define EquivalenceRelation typename
```

What are concepts?

Idea of Concepts

```
template <typename T>
requires Addable<T>
void add(const T& a, const T& b)
{
    return a + b;
}
```

The idea is to restrict our the template to the desired type using the constraints.

The concept check will be made during the compile time

Concepts in the standard library

integral<T>

```cpp
#include <concepts>

template <typename T>
requires std::integral<T>
auto add(const T a, const T b)
{
    return a + b;
}

int main () {
    add(2356,6788);
    add(short(34),short(457));
    //add(35.5,57.3);
    return 0;
}
```

Concept integral<T> is satisfied if T is an integral type :

- short,
- int,
- long,
- long long
- bool,
- char,
- char8_t, char16_t, char32_t,
- wchar_t,

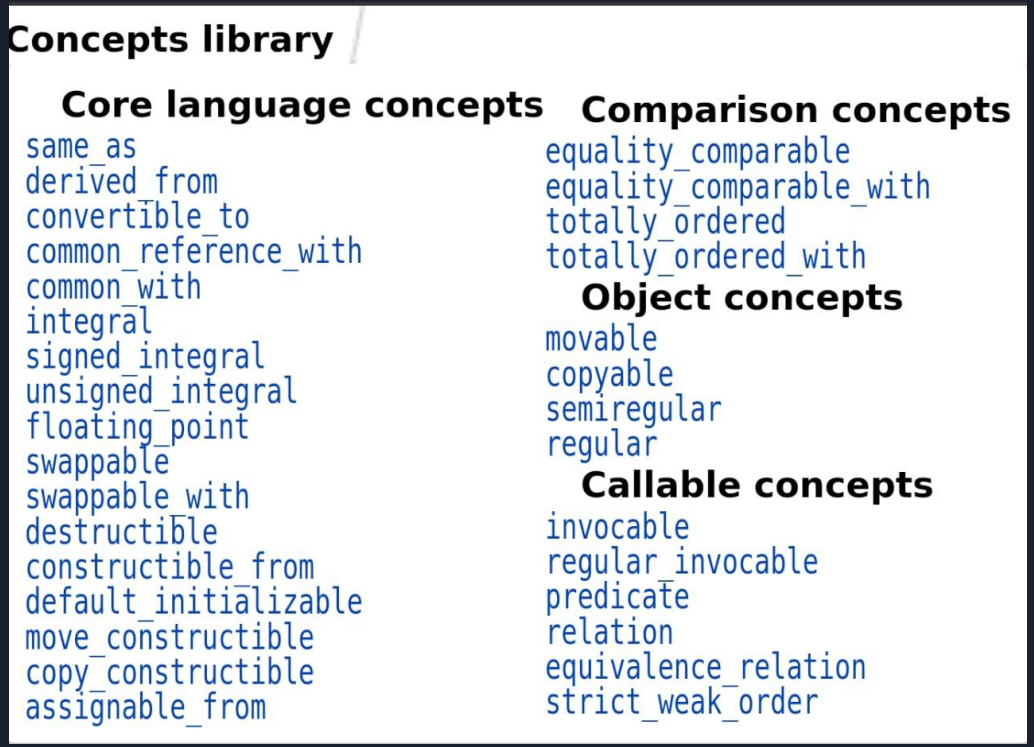
floating_point<T>

```cpp
#include <concepts>
template <typename T>
requires std::floating_point<T>
auto add(const T a, const T b)
{
    return a + b;
}
int main () {
    add(35.5,57.3);
    //add(2356,6788);    //error
    return 0;
}
```

The concept floating_point<T> is satisfied if and only if T is a floating-point type :

- float,
- double
- long double,

Concepts in standard library



Src: cppreference.com

How to write our own Concepts?

How to write our own Concepts??

- Concepts with functions
- Concepts with Classes
- Requires expression
- Combining Concepts
- Concepts subsumption rules

Concepts with functions

A.) defining concept using predefined concept

```cpp
#include <concepts>

template <typename T>
concept Number = std::integral<T> || std::floating_point<T>;

template <typename T>
requires Number<T>
auto add(T const a, T const b)
{
    return a + b;
}
```

- First, the concept is defined and then it is used
- Semicolon is necessary

```cpp
add(23, 45);
add(12.34 + 23.45);
//add(12, 12.34);
```

B.) Using require clause (ad-hoc constraint)

```
#include <concepts>


template<typename T>
requires std::integral<T> || std::floating_point<T>
auto add(T const a, T const b)
{
    return a + b;
}
```

- We define the constraint
- Need to re-define the concept before using at some other place
- Best for cases where you need a constraint at 1 place only

C.) Trailing require clause

```cpp
#include <concepts>

template <typename T>
concept Number = std::integral<T> || std::floating_point<T>;

template<typename T>
auto add(T const a, T const b) requires Number<T>
{
    return a + b;
}
```

- Requires clause after function definition
- apart from that same as requires clos clause
- supports combination

D.) Constraint template parameter

```cpp
#include <concepts>


template <typename T>
concept Number = std::integral<T> || std::floating_point<T>;


template <Number T>
auto add(const T a, const T b)
{
    return a + b;
}
```

- No requires clause
- The typename is replaced by Concept name
- Doesn't support combination of Concepts

E.) Abbreviated template

```
#include <concepts>


template <typename T>

concept Number = std::integral<T> || std::floating_point<T>;


auto add(const Number a, const Number auto b)

{

    return a + b;

}
```

- No requires
- No template parameter list
- Use Concept-Name auto in parameter list
- Doesn't support combination of Concepts

Are there any differences between these?

```
#ifdef INSIGHTS_USE_TEMPLATE
template<>
int addRequiresClause<int>(const int a, const int b)
{
  return a + b;
}
#endif
```

```
#ifdef INSIGHTS_USE_TEMPLATE
template<>
int addConstraintTemplate<int>(const int a, const int b)
{
  return a + b;
}
#endif
```

```
#ifdef INSIGHTS_USE_TEMPLATE
template<>
int addRequiresAdHoc<int>(const int a, const int b)
{
  return a + b;
}
#endif
```

```
#ifdef INSIGHTS_USE_TEMPLATE
template<>
int addTrailing<int>(const int a, const int b) requires Number<int>
{
  return a + b;
}
#endif
```

The abbreviated is different

```
#ifdef INSIGHTS_USE_TEMPLATE
template<>
int addAbbreviate<int, int>(const int a, const int b)
{
  return a + b;
}
#endif
```

Be careful with abbreviated template

```
auto addAbbreviate(const Number auto a,const Number auto b)
{
	return a + b;
}

int main()
{
	addAbbreviate(36, 34.79);
}
```

```
#ifdef INSIGHTS_USE_TEMPLATE
template<>
double addAbbreviate<int, double>(const int a, const double b)
{
  return static_cast<double>(a) + b;
}
#endif
```

How to choose between all the choices ??

If you don't want to re-use the constraint/concept and have simple constraint

- Ad-hoc constraint (preferable)
- Trailing ad-hoc requires

If you have want to use the constraint again

- Requires clause
- Trailing requires clause
- Abbreviated template (be careful)
- Constraint template

Auto & checking types

```
Number auto add(Number auto a, Number auto b){
    return a + b;
}

int main(){
    int a{23}, b{36};
    if constexpr (Number<int>){
        Number auto result = add(a, b);
    }
}
```

- Use the concept with auto placeholder (constraint auto)
- Check whether a type satisfies a given constraint or not

Concepts with classes

Requires clause (ad-hoc) constraint

```
template <typename T>
requires std::integral<T> || std::floating_point<T>
class BoundedNumber {
   private:
          T  num_;
    public:
         BoundedNumber(T num) :  num_(num) {}
};
```

- Requires clause after the template parameter list
- After requires list all your constraint(s)
- You can use complex requirements

Requires clause

```
template <typename T>
requires Number<T>
class BoundedNumber {
  private:
      T  num_;
  public:
      BoundedNumber(T num) : num_(num){}
};
```

- Requires clause after the template head
- After requires list all concepts
- You can use complex requirements
- Good for Concepts you need at multiple places

Constraint template parameter

```
template <Number T>
class BoundedNumber {
  private:
      T  num_;
  public:
      BoundedNumber(T num) : num_(num) {}
};
```

- instead of typename, use the concept name
- You can use simple requirements only

Trailing return type constraint

```
template <typename T>
class BoundedNumber {
    public:
        T divide(const T& divisor)
        requires std::integral<T> {
            // …
        }
};
```

- Class level templates with overloads on functions
- Provides different overload for different parameters types

Trailing return type constraint

```
template <typename T>
class BoundedNumber {
  public:
    T divide(const T divisor)
    requires std::integral<T> {
        // …
    }

    T divide(const T divisor)
    requires std::floating_point<T> {
        // …
    }
};
```

- Class level templates with overloads on functions
- Provides different overload for different parameters types
- The first overload will be selected when the T is of integral type
- The 2nd overload will be selected of T is of floating point type

Requires expression

Requirement on Operations

```
template <typename T>
concept Addable = requires(const T &a, const T &b) {
                        a + b;
                    };

template <typename T>
requires Addable<T>
auto add(const T &a, const T &b)
{
    return a + b;
}
```

- You can define any operations, you needed for you type
- the semicolon after operation is compulsory
- You can list all the operations in the braces with semicolon at end of every requirement

Requirement on interface

```
template <typename T>
concept Square = requires (T &a) {
                        a.square();
                    };
```

```
template <typename T>
concept Power = requires(T& a,int exp){
                           a.power(exp);
                       };
```

- You can define any interface, you needed for you type
- the semicolon after operation is compulsory
- You can list all the operations in the braces with semicolon at end of every requirement

Nested requirements

```
template <typename T>
concept exponent = requires(T &a) {
                        a.square();
                        a.cube();
                    };
```

- You can define any interface, you needed for you type
- the semicolon after operation is compulsory
- You can list all the operations in the braces with semicolon at end of every requirement

Requirement on return types (compound requirements)

```
#include <concepts>


template <typename T>

concept Square = requires (T &t) {

    {t.square()} -> std::convertible_to<int>;

};
```

Constraint the return values using:

- std::convertible<T>
- std::same_as<T>

The braces are compulsory

Requirement on return types (compound requirements)

```
#include <concepts>


template <typename T>

concept Addable= requires(const T &a, const T &b){

            { a + b } noexcept -> std::same_as<T> ;

};
```

I'm this, are actually requiring 3 constraints:

- There should be a operator + for a and b
- The operator should be marked as noexcept
- The return type should be the same as of type a and b

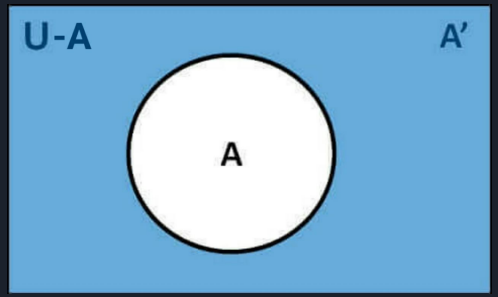
Concepts overloading

Concepts

```
template <typename T>
requires IsfreeAcc<T>
void billing(T & account) {
    //...
}

template <typename T>
void billing (T & account) {
    //...
}
```

- If the requirement on the first function is satisfied, then the first billing function method is called
- If the requirement is not satisfied, Then the second billing method is called



Conjunctions (&&) and disjunction (||)

```cpp
#include <concepts>

template <typename T>
concept Number = std::integral<T> || std::floating_point;
```

You can combine

- Concepts
- Boolean literals
- Boolean expressions
- Type Traits
- Requires expression

You can also use !

Overloading

```
template <typename T>
requires FreeAcc<T>
void billing (T & account) {
    //......
}

template <typename T>
requires FreeAcc<T> && DiscountCode<T>
void billing (T & account) {
    //.....
}
```

- The most constraint overload will be selected by compiler at compile time
- This is done by the Concept Subsumption rules

Concepts overloading

```
template <typename T>
requires (not FreeAcc<T>)
void billing (T & account) {
    ....
}

template <typename T>
requires (not FreeAcc<T>) && DiscountCode<T>
void billing (T & account) {
    ....
}
```

Will result in ambiguity

Concepts overloading

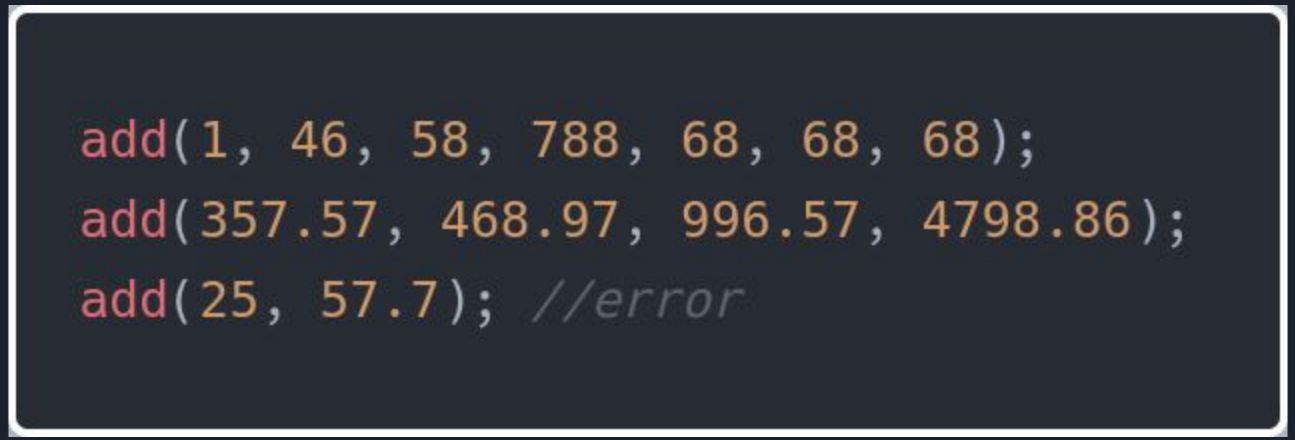
```
template <typename T>
concept NotfreeAccount = not isfreeAcc<T>;

template <typename T>
requires notFreeAcc<T>
void billing(T & account) {
     .....
}

template <typename T>
requires notFreeAcc<T> && DiscountCode<T>
void billing(T & account) {
     .....
}
```

//not ambiguous

Real life examples


add(1, 46, 58, 788, 68, 68, 68);
add(357.57, 468.97, 996.57, 4798.86);
add(25, 57.7); //error

Template Pack (C++17)

```cpp
#include <type_traits>

template <typename Value, typename... RestValues>
constexpr bool are_same_v = std::conjunction_v< std::is_same<Value, RestValues>...>;

template <typename Value, typename... RestValues>
struct first_arg {
    using type = Value;
};

template <typename... Values>
using first_arg_t = typename first_arg<Values...>::type;
```

```cpp
template <typename... Args>
std::enable_if_t<are_same_v<Args...>, first_arg_t<Args...>>
add(Args &&... args) noexcept
{
    return (args + ...);
}
```

Template Pack (C++20)

```cpp
#include <type_traits>
#include <concepts>

template <typename Value, typename... RestValues>
constexpr bool are_same_v = std::conjunction_v<std::is_same<Value, RestValues>...>;

template <typename Args...>
requires are_same_v<Args...>
auto add(Args &&... args) noexcept {
    (args + ...);
}
```

improving Template Pack (C++20)

- Add requires more than 1 parameter add(46); //error
- Type used in Args must have operator +
- The + operator must be noexcept
- The return value of the + should be same as type used in Args

```
template <typename... Args>
requires are_same_v<Args...>
        && (sizeof...(Args) > 1)
        && requires(Args... args)
        {
            (args + ...);
            { (args + ...) } noexcept;
            {(args + ...)}-> std::same_as< first_arg_t<Args...>>;
        }
auto add(Args &&... args) noexcept
{
    return (args + ...);
}
```

```
requires(Args... args)
{
    { (args + ...) } noexcept -> std::same_as< first_arg_t<Args...>>;
}
```

COM Like Wrapper

```
struct defaultDestructible {};      // default destructible typea


struct userDefinedDestructible{
    //....
    public:
    ~userDefinedDestructible() {    //not default destructible
        release ();
    }
    void release () {               //release method to release the resources
        //.....
    }
};

static_assert( std::is_trivially_destructible_v< ComWrapper< defaultDestructible >>);
static_assert( not std::is_trivially_destructible_v< ComWrapper< userDefinedDestructible>>);
```

COM Like Wrapper(C++17, incomplete)

```cpp
template <typename T, typename = void>
struct has_release : std::false_type {};

template <typename T>
struct has_release<T,decltype(std::declval<T>().release())> :std::true_type{};

template <typename T>
class ComWrapper {
  private:
    T data_;

  public:
    ~ComWrapper() {
        if constexpr (has_release<T>::value) {
            data_.release();
        }
    }
};
```

C++20 COM Wrapper (Dependant Destructor)

```
template <typename T>
concept hasRelease = requires(const T t) {
    t.release();
};

template <typename T>
class ComWrapper {
    T data_;

  public:
    ~ComWrapper() requires hasRelease<T> {
        data_.release();
    }

    ~ComWrapper() = default;
};
```

Conclusion

Better type checking

```
#include <concepts>

template <typename T>
concept Number= (std::integral<T> || std:: floating_point<T>)
                            && !std::same_as<T, bool>
                            && !std::same_as<T, char>
                            && !std::same_as<T, unsigned char>
                            && !std::same_as<T, char8_t>
                            && !std::same_as<T, char16_t>
                            && !std::same_as<T, char32_t>
                            && !std::same_as<T, wchar_t>;

template <typename T>
requires Number<T>
class BoundedNumber {
    T num;
};
```

Accepts number only

- No bool
- No chars

Turn normal code into self-documenting code

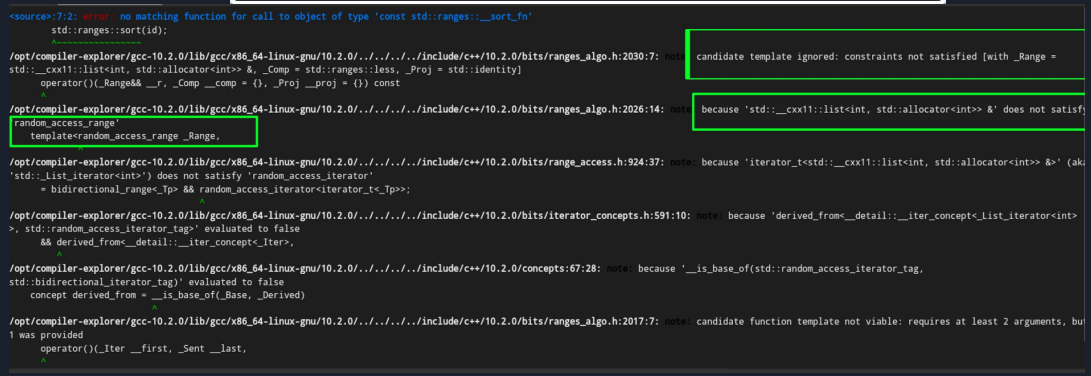
```
template <typename BussinessObject>
void process(BusinessObject & ob)
{
    //......
}
```

```
template <typename BussinessObject>
requires std::regular<BussinessObject>
        && std::derived_from<BussinessObject, BaseClass>
        && std::movable<BussinessObject>
        && has_method<BussinessObject>
void process(BusinessObject & ob) {
        //......
}
```

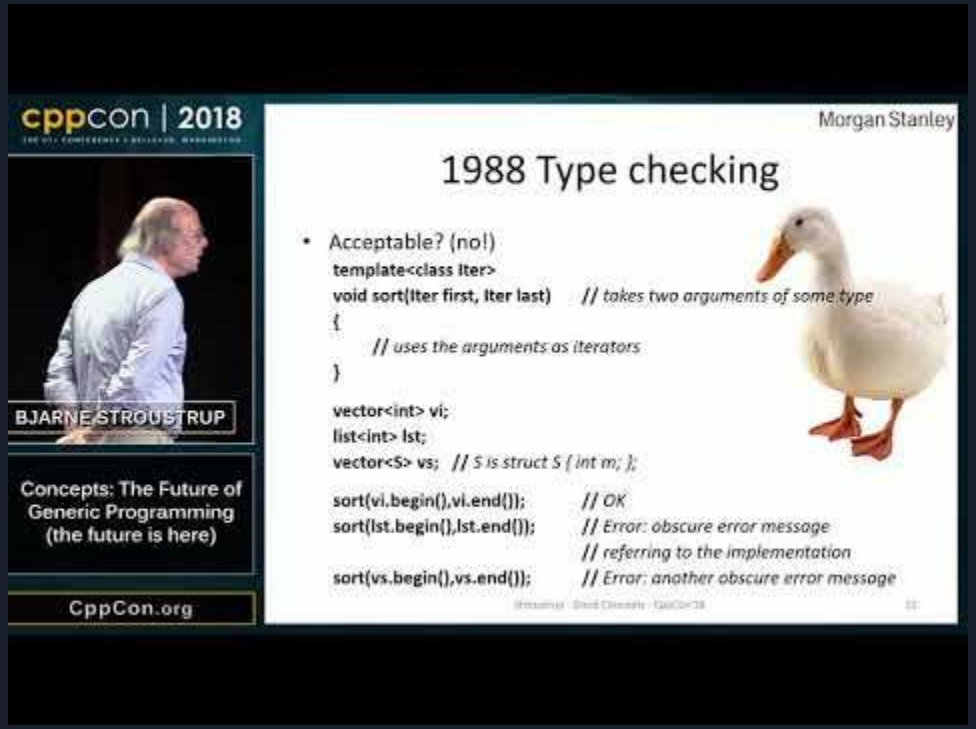
Ranges library (error example)

```cpp
#include <list>
#include <algorithm>
#include <ranges>

int main() {
    std::list<int> id{35, 57, 56, 24, 19};
    std::ranges::sort(id);
    return 0;
}
```



Reduced Compile Times




Why not to
use Concepts??
CHANGE MY MIND
imgflip.com


There is not
a single reason
not to use concepts
CHANGE MY MIND
imgflip.com