Back To Basics
Concepts

NICOLAI JOSUTTIS

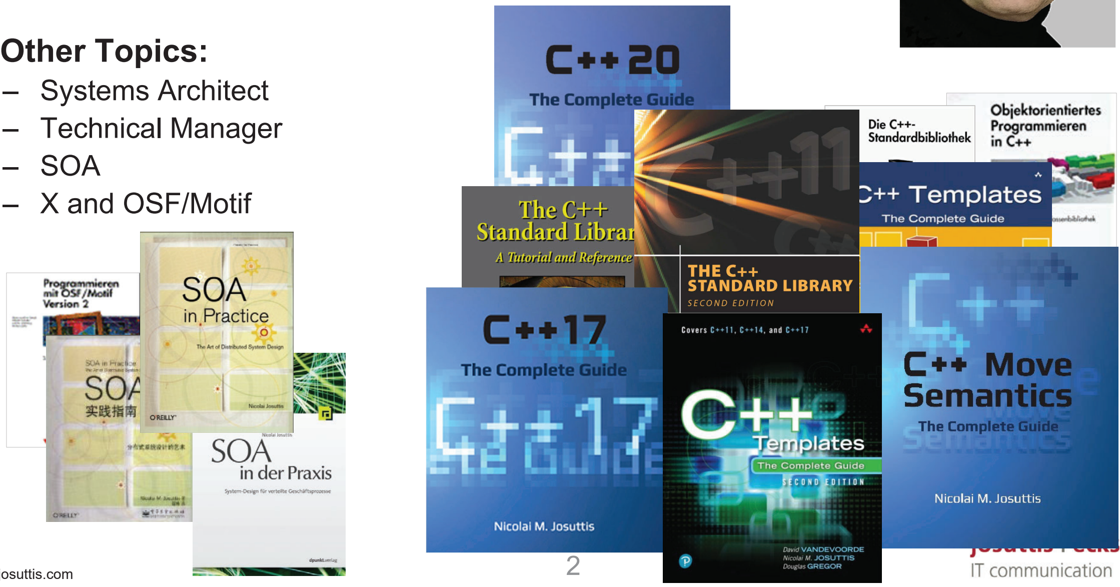
Cppcon
The C++ Conference

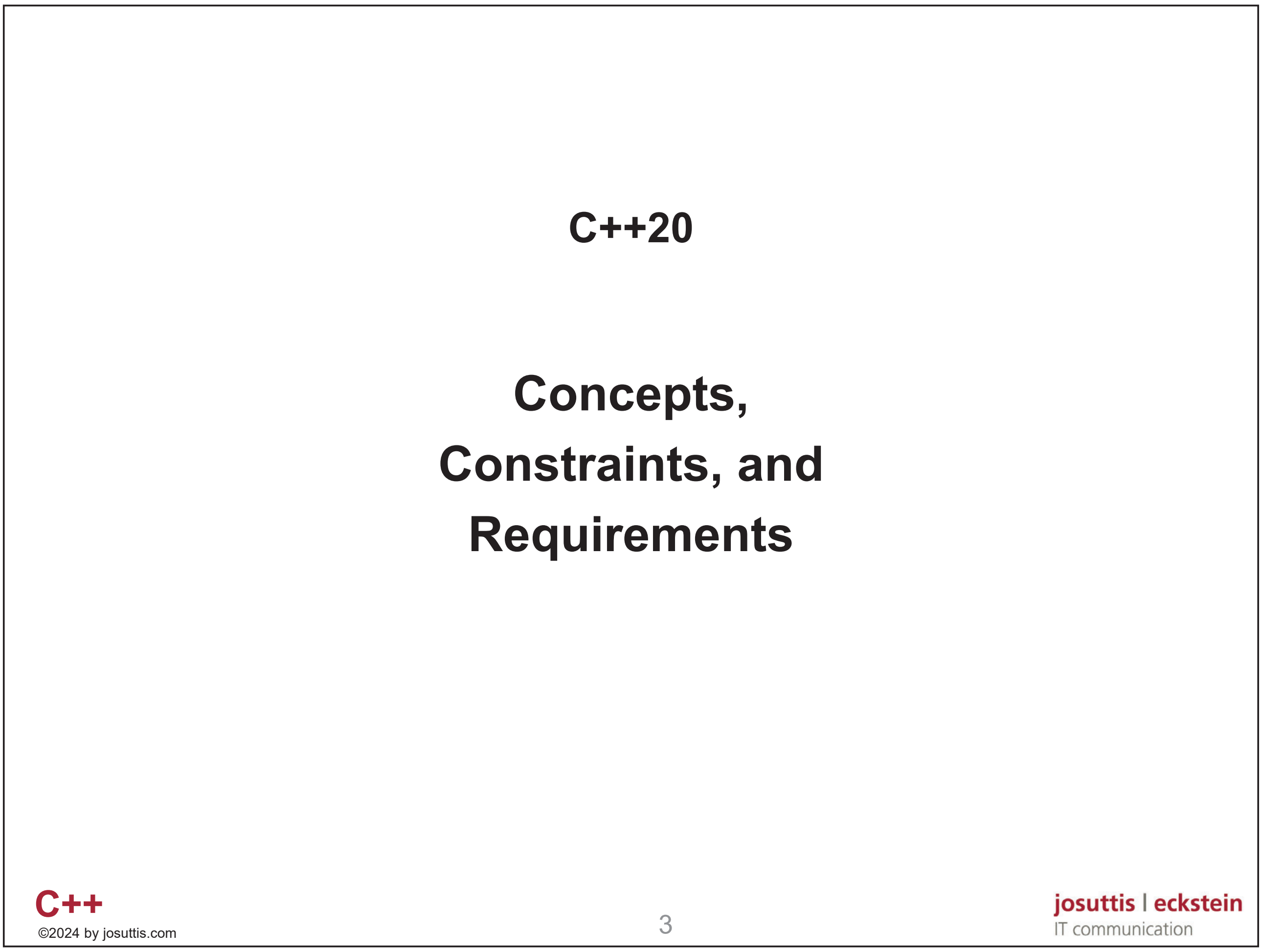
2024
September 15 - 20

September 18, 2024
14:00 - 15:00 MDT

Nicolai M. Josuttis

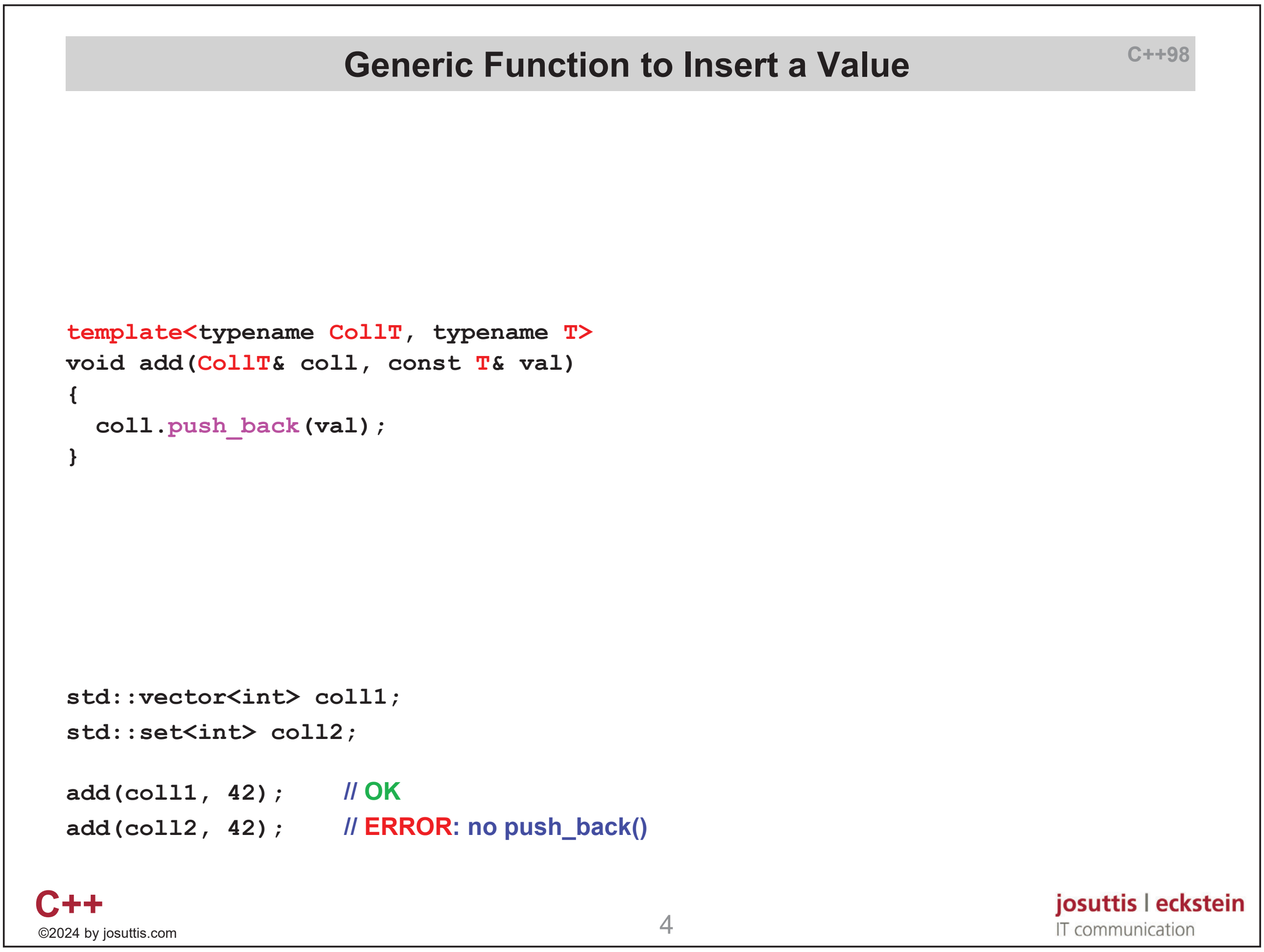
- **Independent consultant**
  - Continuously learning since 1962
- **C++:**
  - since 1990
  - ISO Standard Committee since 1997
- **Other Topics:**
  - Systems Architect
  - Technical Manager
  - SOA
  - X and OSF/Motif





C++20

Concepts, Constraints, and Requirements



Generic Function to Insert a Value

C++98

```
template<typename CollT, typename T>
void add(CollT& coll, const T& val)
{
  coll.push_back(val);
}
```

```
std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);      // OK
add(coll2, 42);      // ERROR: no push_back()
```

Overloading Function Templates

C++98

```
template<typename CollT, typename T>
void add(CollT& coll, const T& val)
{
  coll.push_back(val);
}

template<typename CollT, typename T>
void add(CollT& coll, const T& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);      // ERROR: two definitions of add()
add(coll2, 42);      // ERROR: two definitions of add()
```

Overload resolution cares only for declarations (ignoring return types)

Constraints with Concepts

C++20

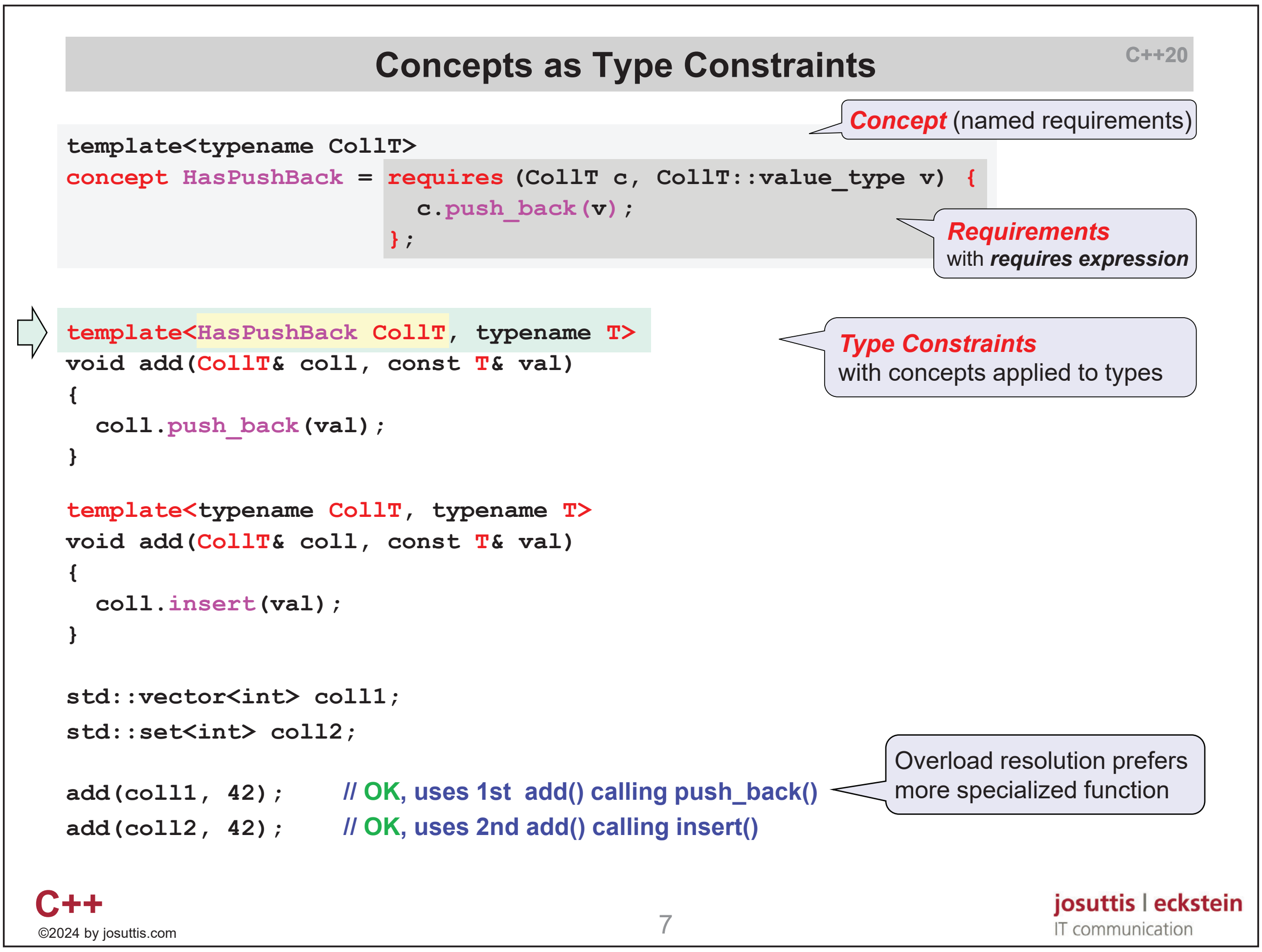
```
template<typename CollT>
concept HasPushBack = requires (CollT c, CollT::value_type v) {
                        c.push_back(v);
                      };

template<typename CollT, typename T>
requires HasPushBack<CollT>
void add(CollT& coll, const T& val)
{
  coll.push_back(val);
}

template<typename CollT, typename T>
void add(CollT& coll, const T& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);      // OK, uses 1st  add() calling push_back()
add(coll2, 42);      // OK, uses 2nd add() calling insert()
```

Concept (named requirements)

Requirements with ***requires expression***

Constraints formulated by a ***requires clause***

Overload resolution prefers more specialized function



Concepts as Type Constraints

C++20

Concept (named requirements)

```
template<typename CollT>
concept HasPushBack = requires (CollT c, CollT::value_type v) {
                        c.push_back(v);
                      };
```

Requirements
with *requires expression*

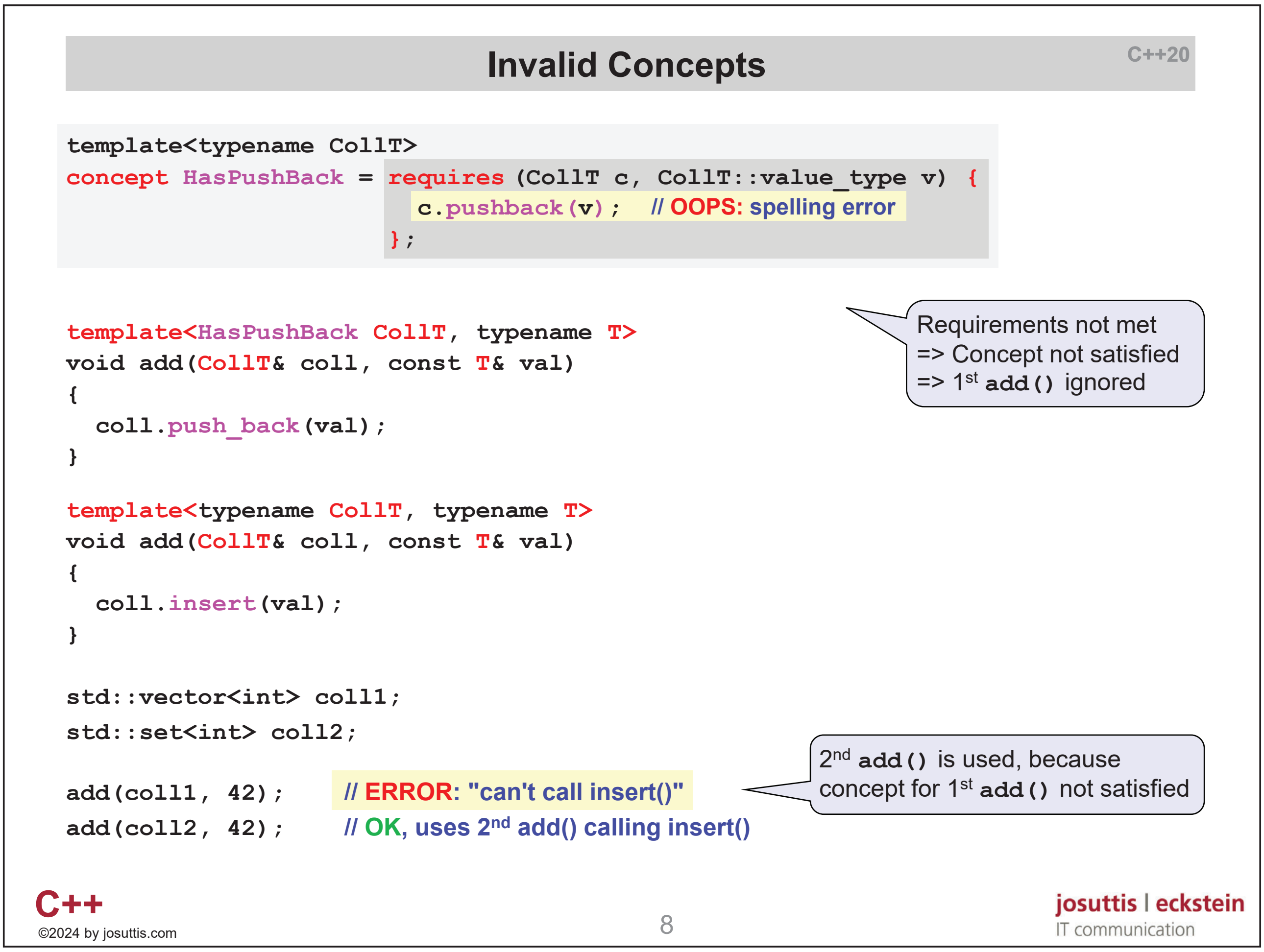
```
template<HasPushBack CollT, typename T>
void add(CollT& coll, const T& val)
{
  coll.push_back(val);
}

template<typename CollT, typename T>
void add(CollT& coll, const T& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);      // OK, uses 1st  add() calling push_back()
add(coll2, 42);      // OK, uses 2nd add() calling insert()
```

Type Constraints
with concepts applied to types

Overload resolution prefers
more specialized function

C++

josuttis | eckstein
IT communication



Invalid Concepts

C++20

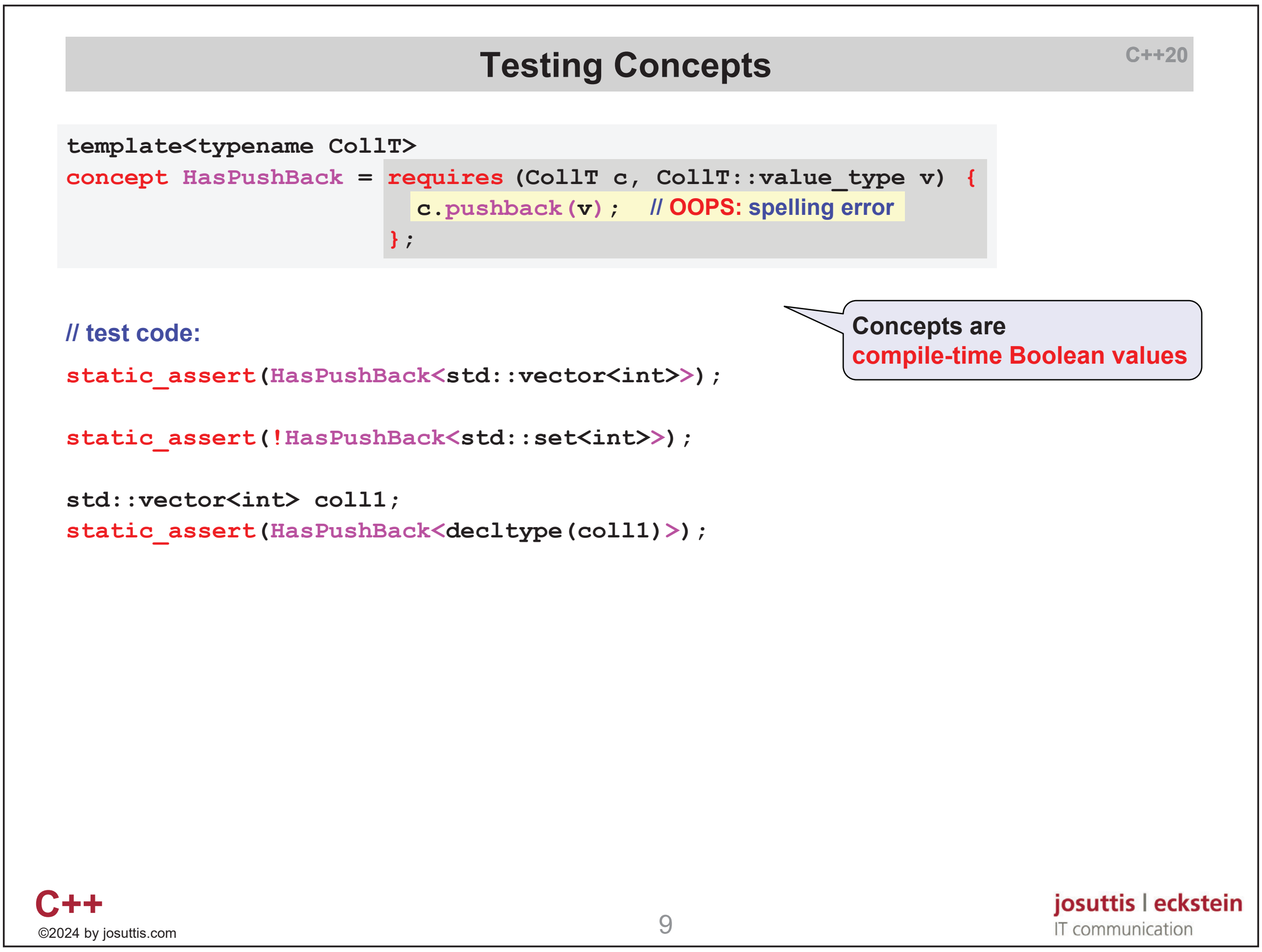
```
template<typename CollT>
concept HasPushBack = requires (CollT c, CollT::value_type v) {
                        c.pushback(v);   // OOPS: spelling error
                      };

template<HasPushBack CollT, typename T>
void add(CollT& coll, const T& val)
{
  coll.push_back(val);
}

template<typename CollT, typename T>
void add(CollT& coll, const T& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);      // ERROR: "can't call insert()"
add(coll2, 42);      // OK, uses 2nd add() calling insert()
```

Requirements not met
=> Concept not satisfied
=> 1st `add()` ignored

2nd `add()` is used, because
concept for 1st `add()` not satisfied

C++

josuttis | eckstein
IT communication



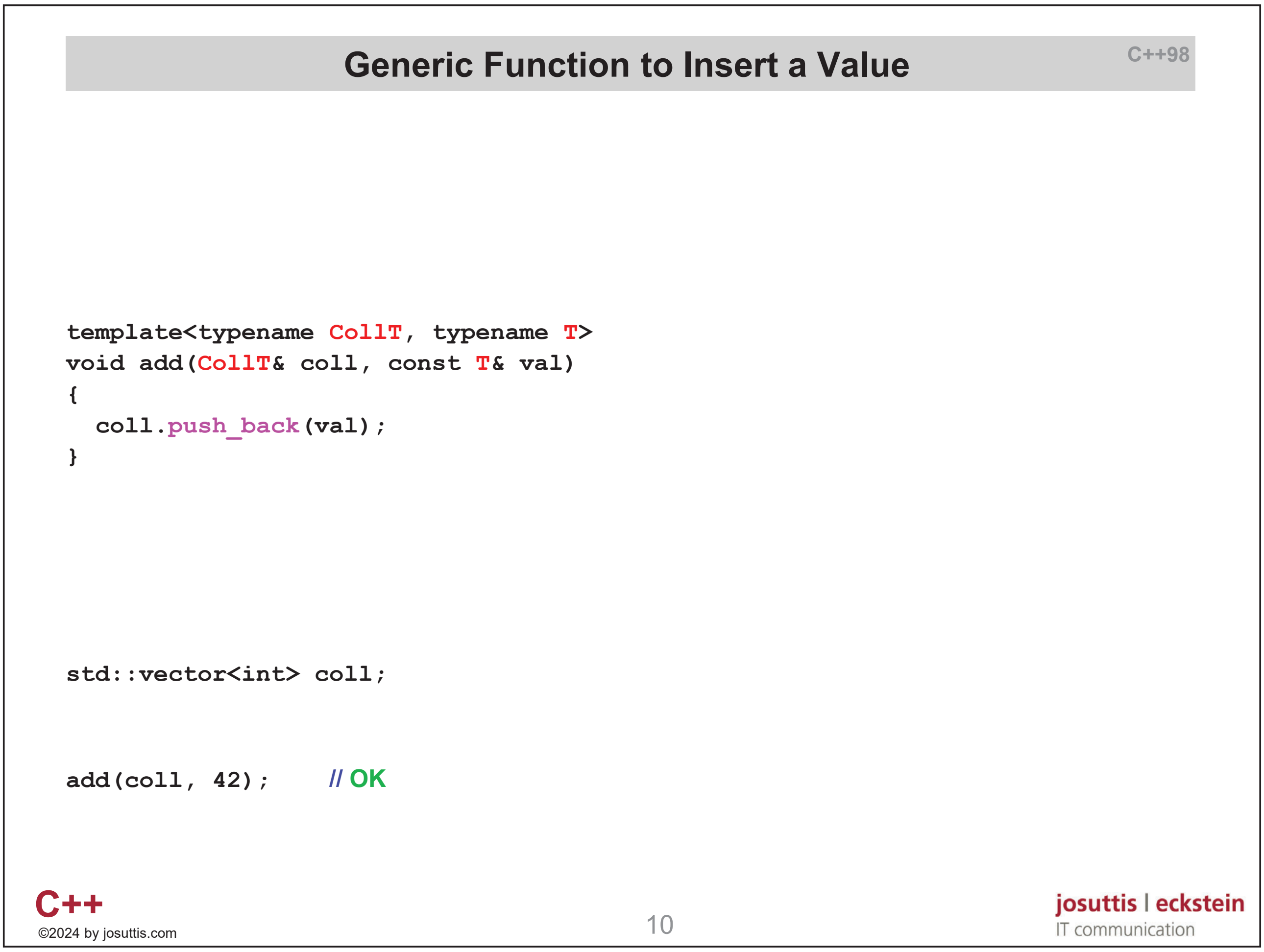
Testing Concepts

C++20

```
template<typename CollT>
concept HasPushBack = requires (CollT c, CollT::value_type v) {
                        c.pushback(v);   // OOPS: spelling error
                      };
```

Concepts are
compile-time Boolean values

```
// test code:
static_assert(HasPushBack<std::vector<int>>);

static_assert(!HasPushBack<std::set<int>>);

std::vector<int> coll1;
static_assert(HasPushBack<decltype(coll1)>);
```



Generic Function to Insert a Value

C++98

```
template<typename CollT, typename T>
void add(CollT& coll, const T& val)
{
  coll.push_back(val);
}
```

```
std::vector<int> coll;
```

```
add(coll, 42);      // OK
```

auto as Function Parameters

C++20

```
void add(auto& coll, const auto& val)
{
  coll.push_back(val);
}
```

"***Abbreviated function template***"

- Generic code
- Equivalent to:

```
template<typename T1, typename T2>
void add(T1& coll, const T2& val) {
  coll.push_back(val);
}
```

- Definition usually in header files
- No **`inline`** necessary

```
std::vector<int> coll;

add(coll, 42);     // OK
```

C++

josuttis | eckstein
IT communication

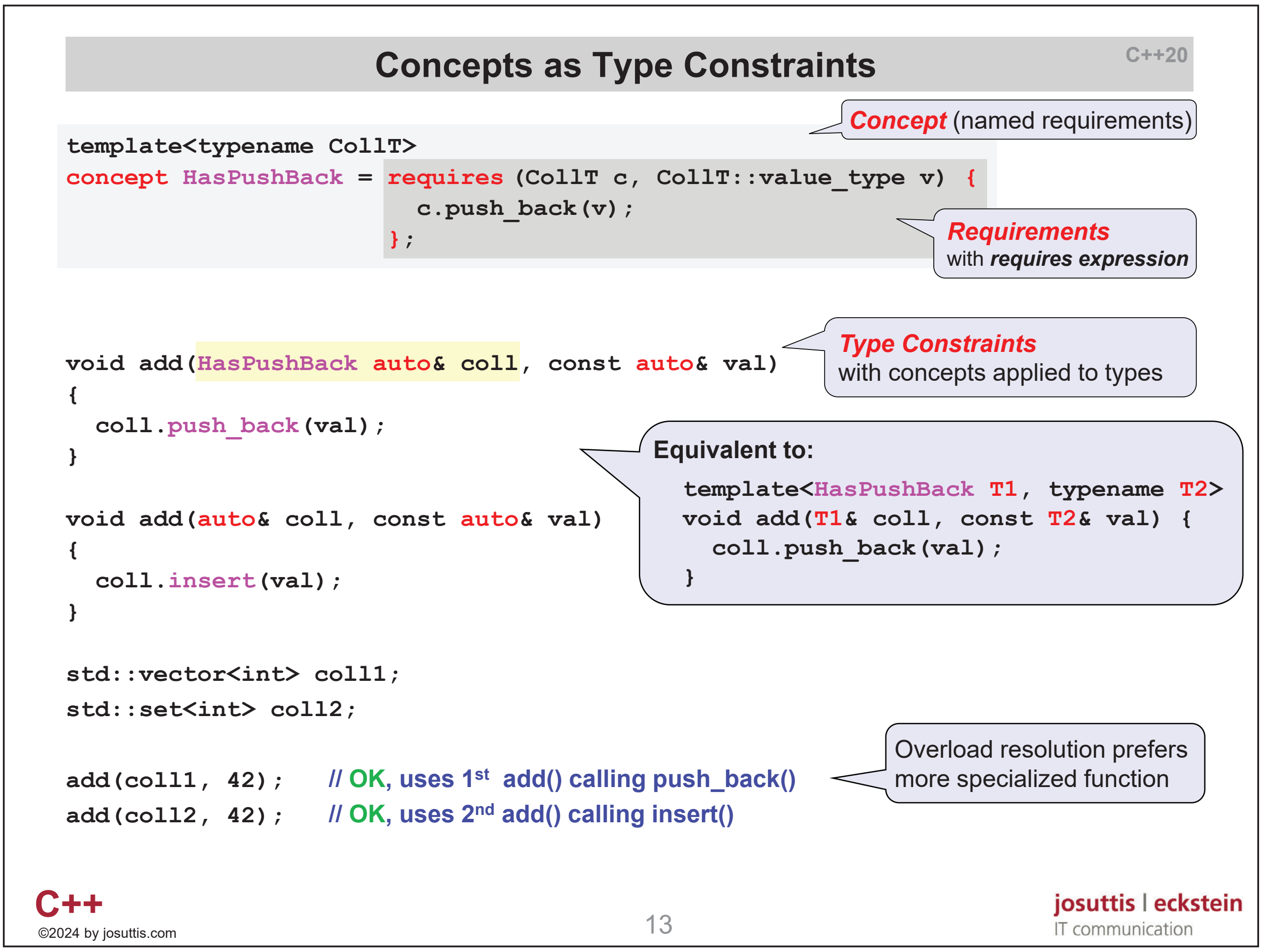
auto as Function Parameters

C++20

```
void add(auto& coll, const auto& val)
{
  coll.push_back(val);
}

void add(auto& coll, const auto& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);    // ERROR: two definitions of add()
add(coll2, 42);    // ERROR: two definitions of add()
```

C++

josuttis | eckstein
IT communication



Concepts as Type Constraints

C++20

```
template<typename CollT>
concept HasPushBack = requires (CollT c, CollT::value_type v) {
                        c.push_back(v);
                      };
```

Concept (named requirements)

Requirements with *requires expression*

```
void add(HasPushBack auto& coll, const auto& val)
{
  coll.push_back(val);
}

void add(auto& coll, const auto& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);    // OK, uses 1st  add() calling push_back()
add(coll2, 42);    // OK, uses 2nd add() calling insert()
```

Type Constraints with concepts applied to types

Equivalent to:

```
template<HasPushBack T1, typename T2>
void add(T1& coll, const T2& val) {
  coll.push_back(val);
}
```

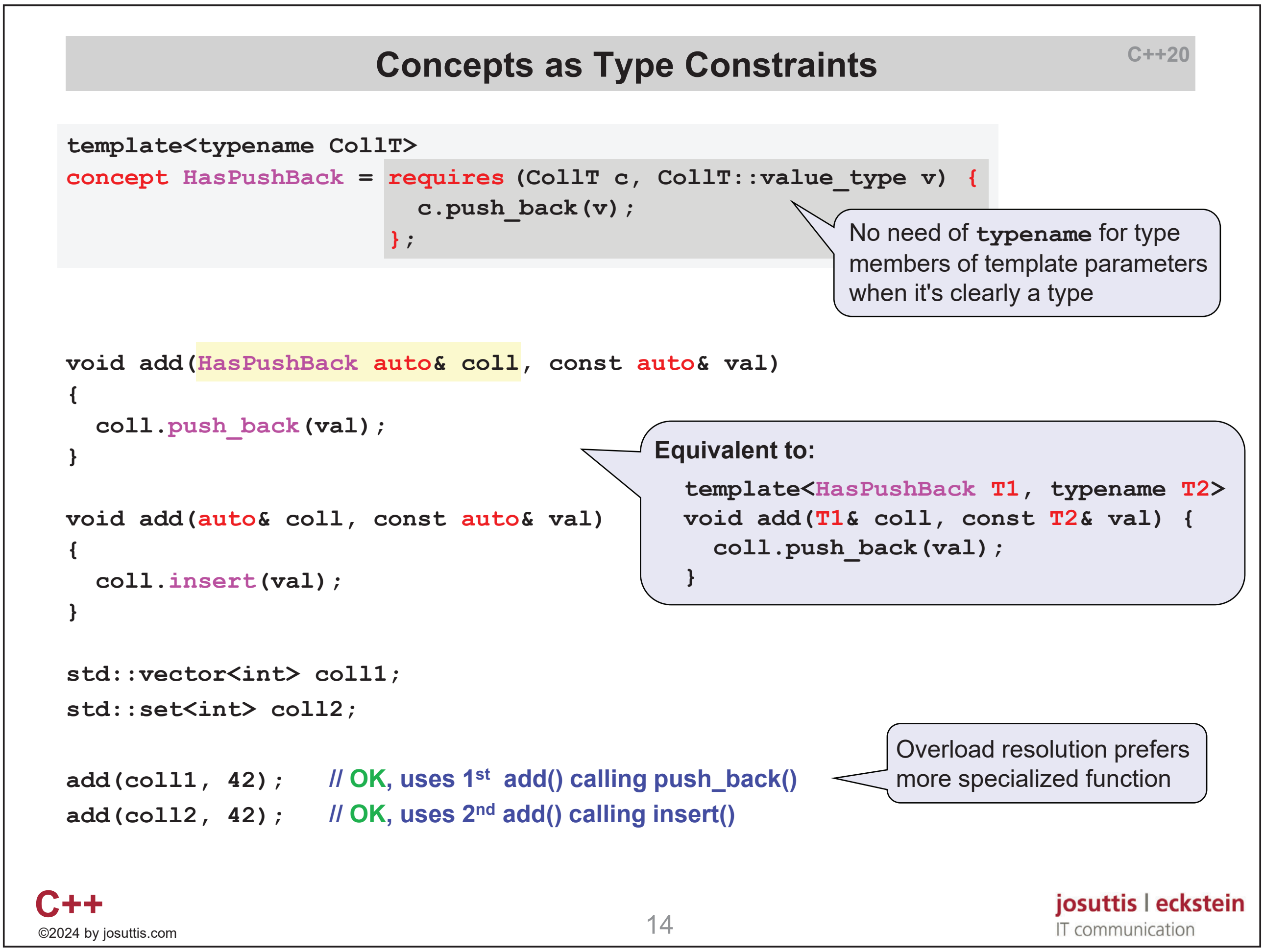
Overload resolution prefers more specialized function

C++

©2024 by josuttis.com

13

josuttis | eckstein IT communication



Concepts as Type Constraints

C++20

```
template<typename CollT>
concept HasPushBack = requires (CollT c, CollT::value_type v) {
                        c.push_back(v);
                      };
```

No need of `typename` for type members of template parameters when it's clearly a type

```
void add(HasPushBack auto& coll, const auto& val)
{
  coll.push_back(val);
}

void add(auto& coll, const auto& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);    // OK, uses 1st  add() calling push_back()
add(coll2, 42);    // OK, uses 2nd add() calling insert()
```

Equivalent to:

```
template<HasPushBack T1, typename T2>
void add(T1& coll, const T2& val) {
  coll.push_back(val);
}
```

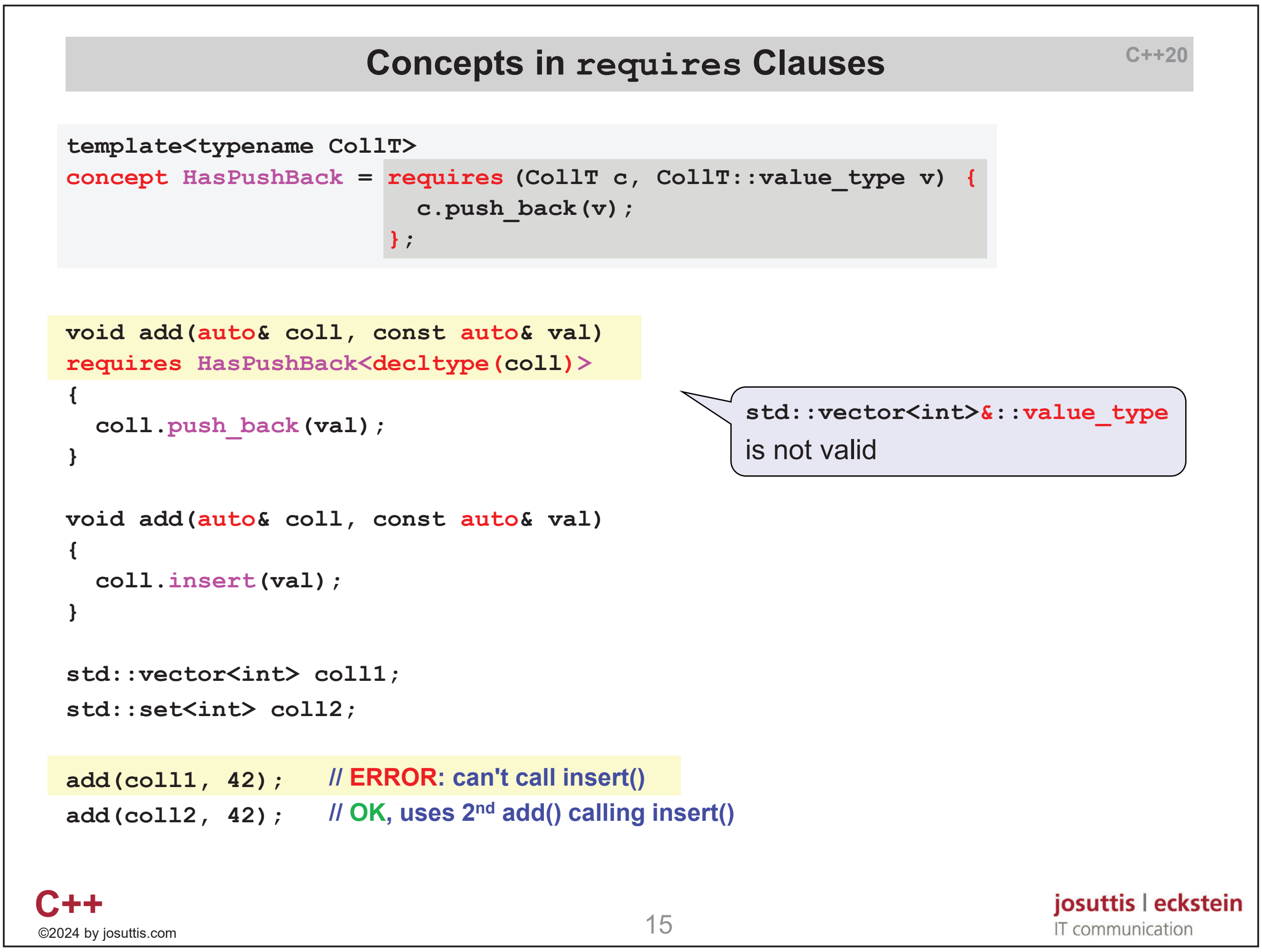
Overload resolution prefers more specialized function

C++

©2024 by josuttis.com

14

josuttis | eckstein IT communication



Concepts in requires Clauses

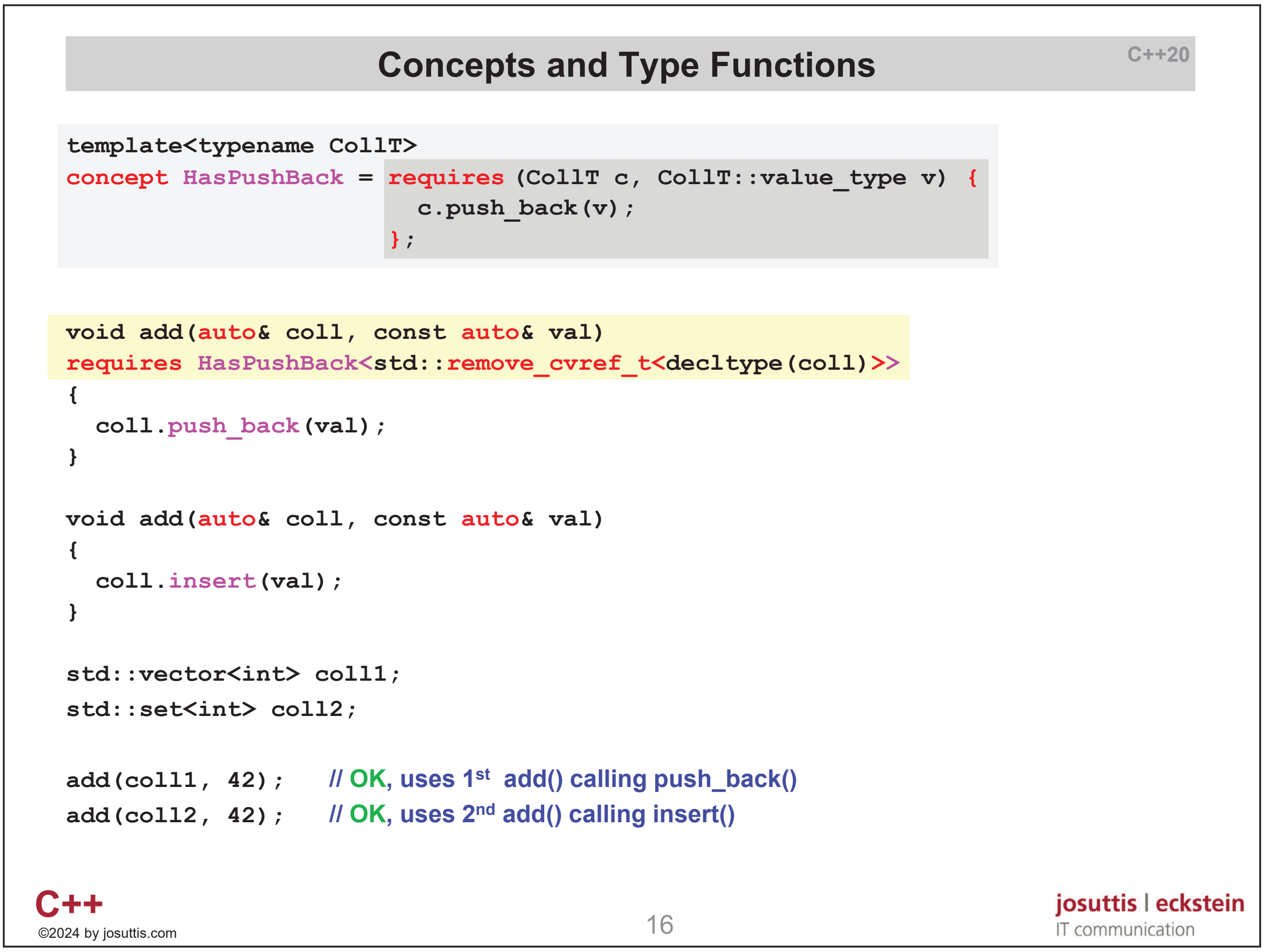
C++20

```
template<typename CollT>
concept HasPushBack = requires (CollT c, CollT::value_type v) {
                        c.push_back(v);
                      };

void add(auto& coll, const auto& val)
requires HasPushBack<decltype(coll)>
{
  coll.push_back(val);
}

void add(auto& coll, const auto& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);     // ERROR: can't call insert()
add(coll2, 42);     // OK, uses 2nd add() calling insert()
```

`std::vector<int>&::value_type` is not valid

C++

josuttis | eckstein
IT communication



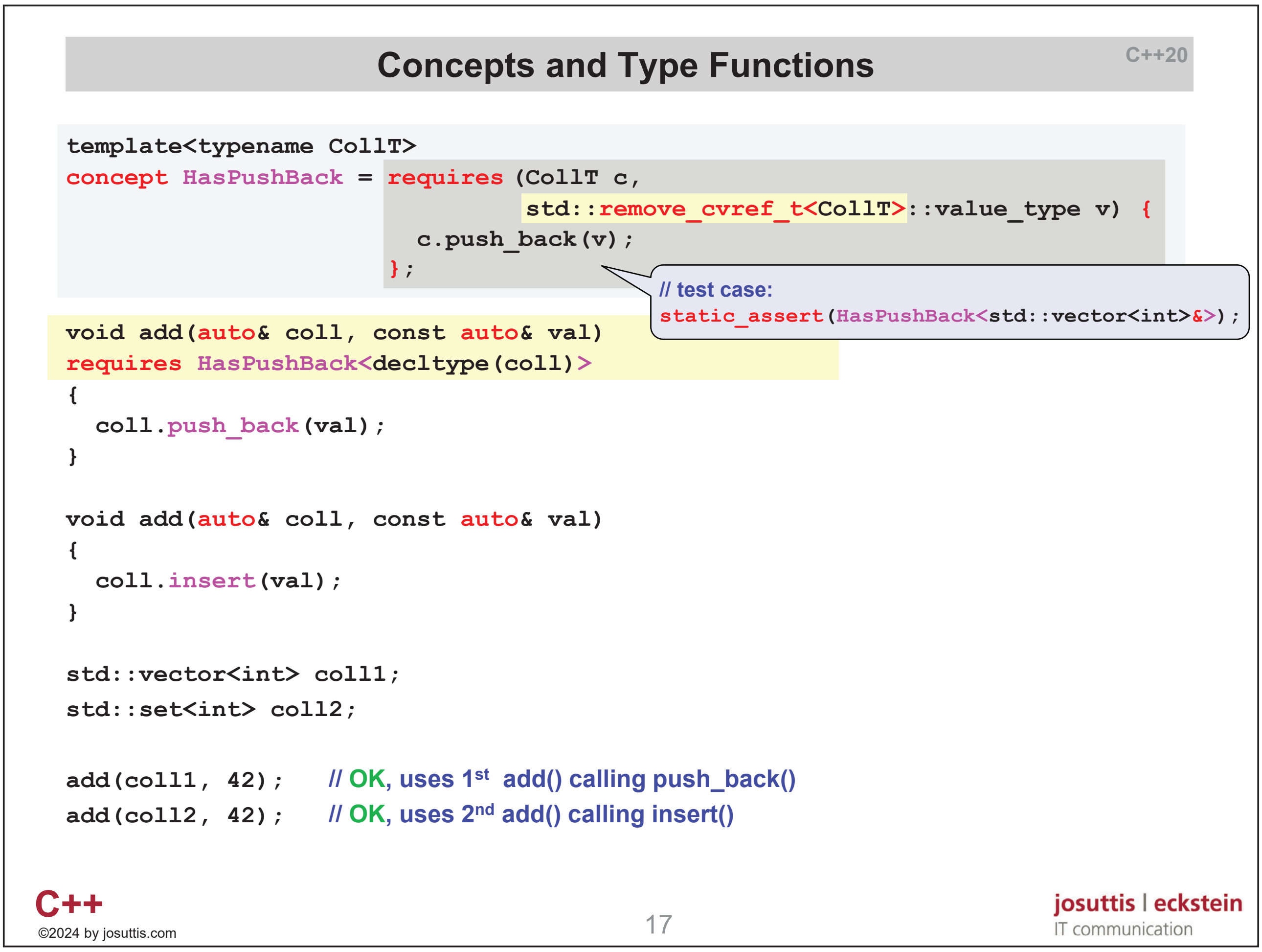
Concepts and Type Functions

C++20

```
template<typename CollT>
concept HasPushBack = requires (CollT c, CollT::value_type v) {
                        c.push_back(v);
                      };

void add(auto& coll, const auto& val)
requires HasPushBack<std::remove_cvref_t<decltype(coll)>>
{
  coll.push_back(val);
}

void add(auto& coll, const auto& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);     // OK, uses 1st  add() calling push_back()
add(coll2, 42);     // OK, uses 2nd add() calling insert()
```

C++

josuttis | eckstein
IT communication



Concepts and Type Functions

C++20

```
template<typename CollT>
concept HasPushBack = requires (CollT c,
                                 std::remove_cvref_t<CollT>::value_type v) {
                        c.push_back(v);
                      };
```

// test case:
`static_assert(HasPushBack<std::vector<int>&>);`

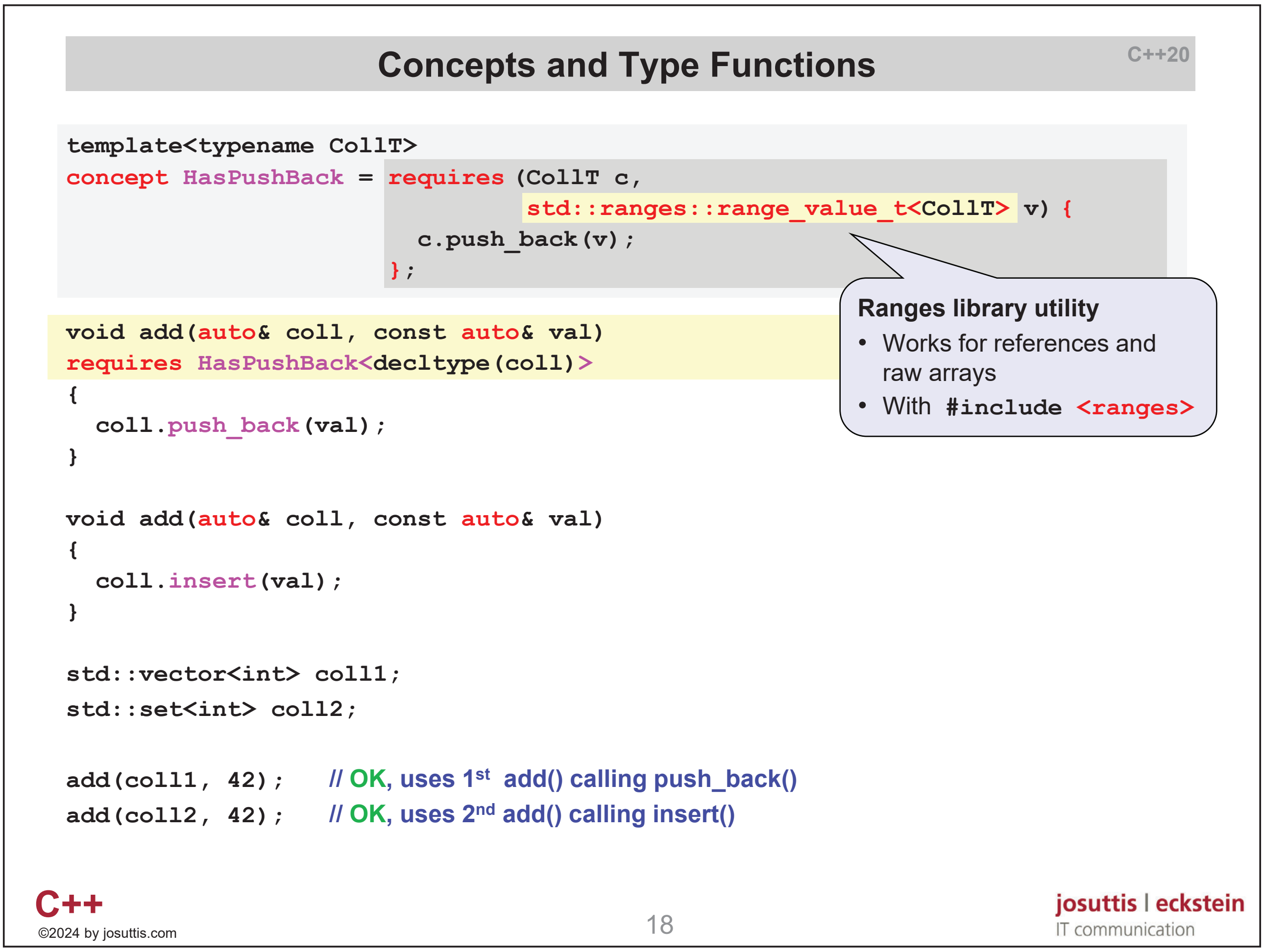
```
void add(auto& coll, const auto& val)
requires HasPushBack<decltype(coll)>
{
  coll.push_back(val);
}

void add(auto& coll, const auto& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);    // OK, uses 1st  add() calling push_back()
add(coll2, 42);    // OK, uses 2nd add() calling insert()
```

C++

©2024 by josuttis.com

17

josuttis | eckstein
IT communication



Concepts and Type Functions

C++20

```
template<typename CollT>
concept HasPushBack = requires (CollT c,
                                 std::ranges::range_value_t<CollT> v) {
                        c.push_back(v);
                      };
```

Ranges library utility

- Works for references and raw arrays
- With `#include <ranges>`

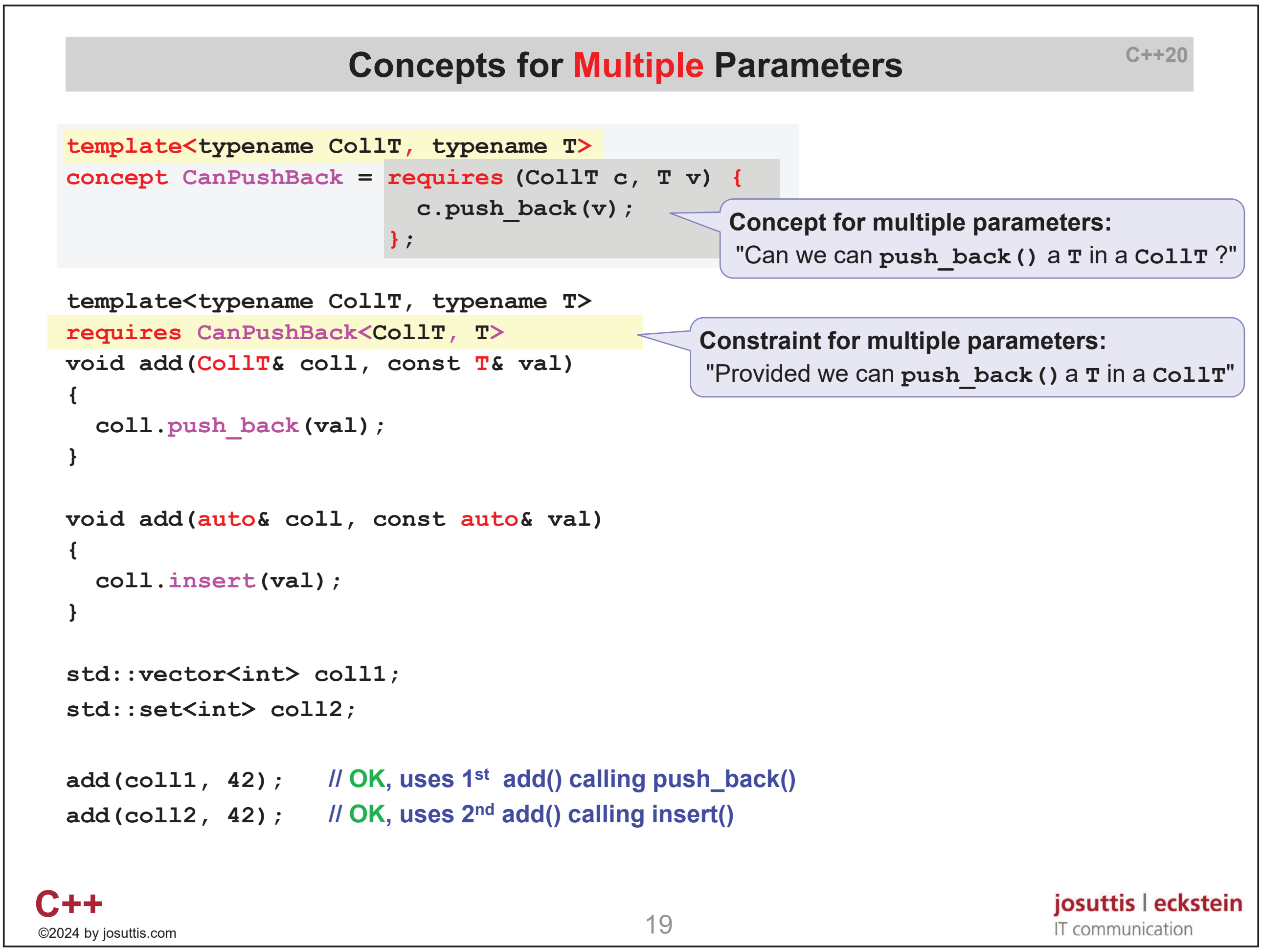
```
void add(auto& coll, const auto& val)
requires HasPushBack<decltype(coll)>
{
  coll.push_back(val);
}

void add(auto& coll, const auto& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);    // OK, uses 1st  add() calling push_back()
add(coll2, 42);    // OK, uses 2nd add() calling insert()
```

C++

©2024 by josuttis.com

18

josuttis | eckstein
IT communication



Concepts for Multiple Parameters

C++20

```
template<typename CollT, typename T>
concept CanPushBack = requires (CollT c, T v) {
                        c.push_back(v);
                      };
```

Concept for multiple parameters:
"Can we can `push_back()` a `T` in a `CollT` ?"

```
template<typename CollT, typename T>
requires CanPushBack<CollT, T>
void add(CollT& coll, const T& val)
{
  coll.push_back(val);
}
```

Constraint for multiple parameters:
"Provided we can `push_back()` a `T` in a `CollT`"

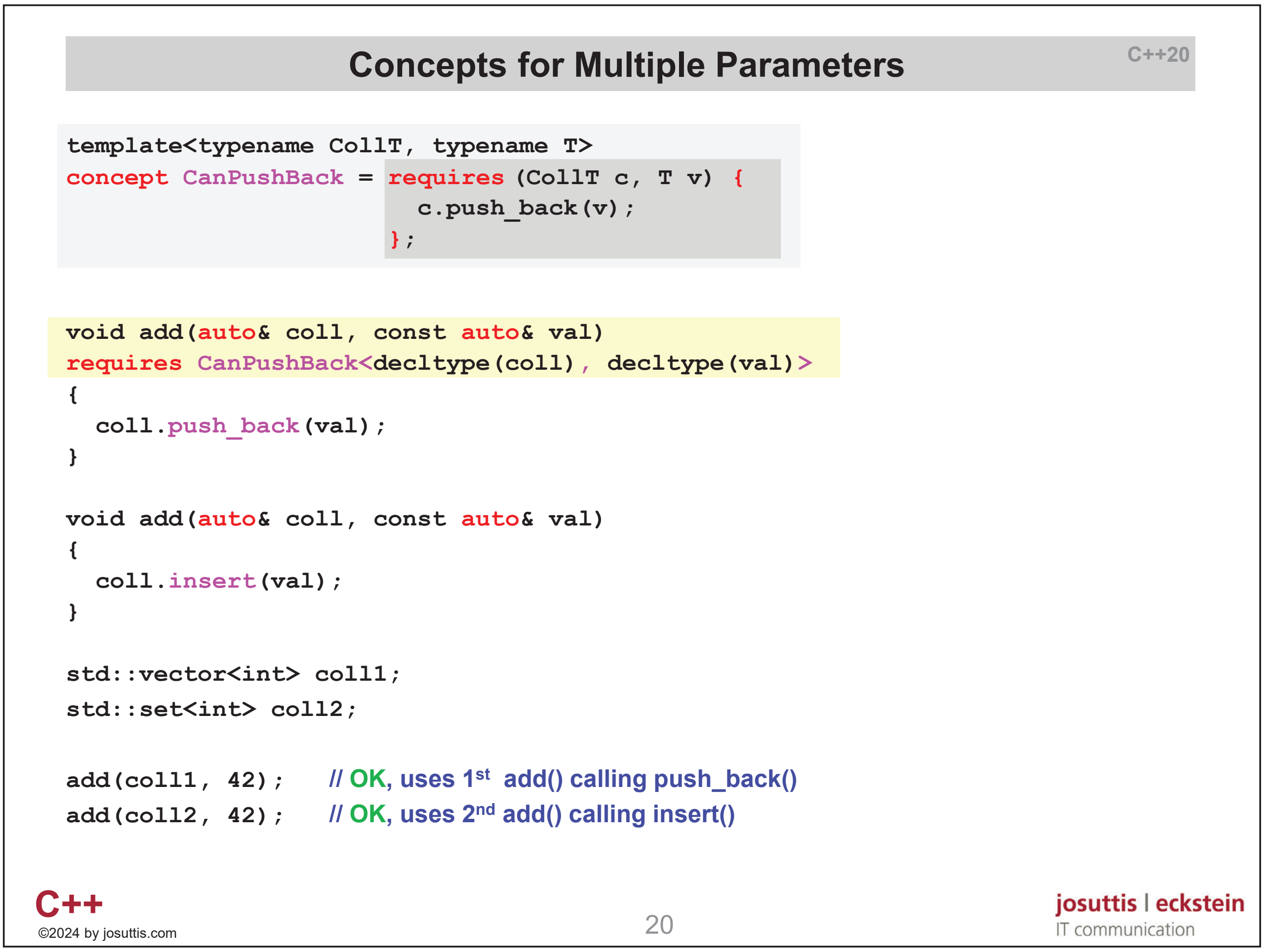
```
void add(auto& coll, const auto& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);    // OK, uses 1st  add() calling push_back()
add(coll2, 42);    // OK, uses 2nd add() calling insert()
```

C++

©2024 by josuttis.com

19

josuttis | eckstein
IT communication



Concepts for Multiple Parameters

C++20

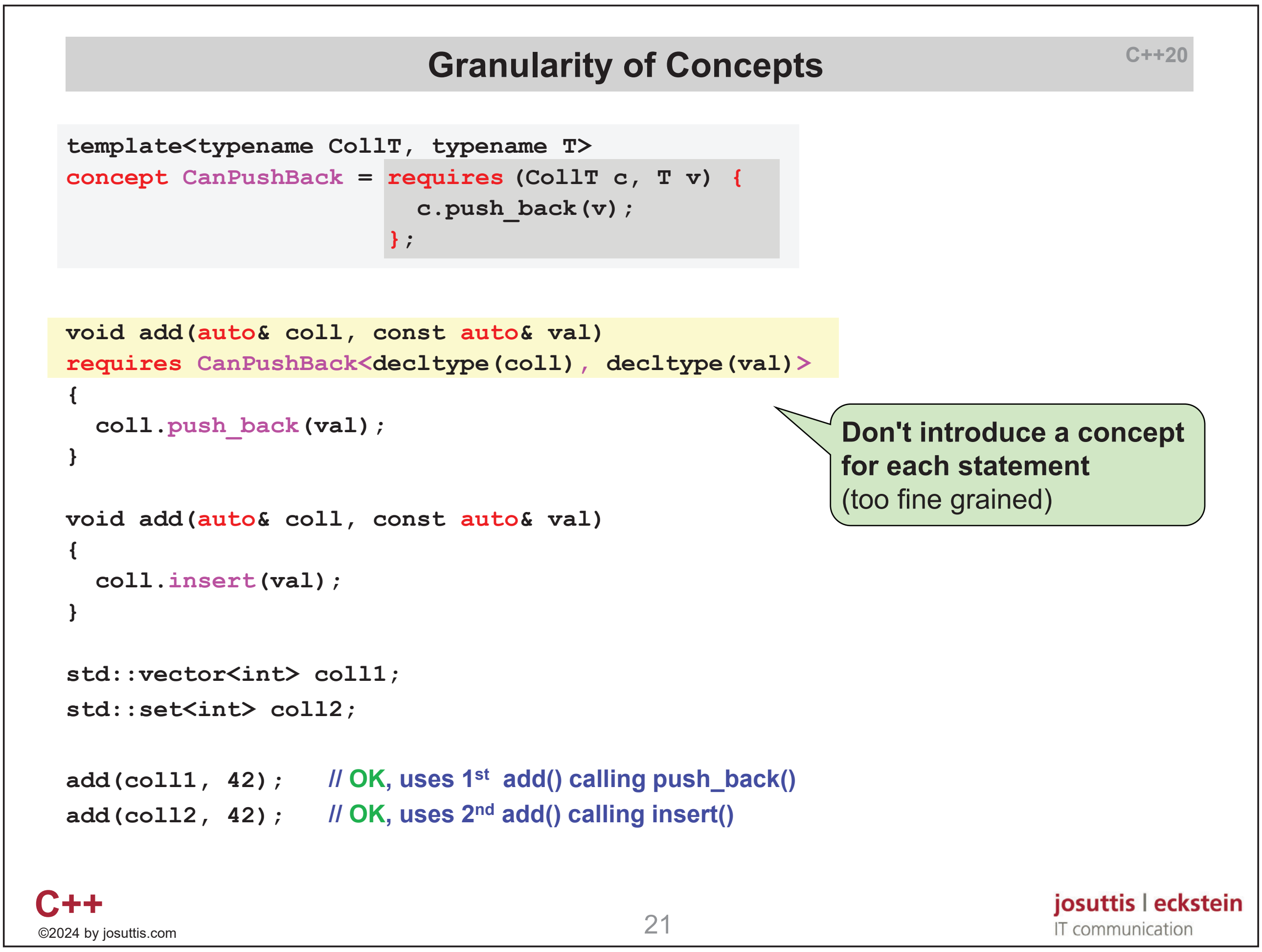
```
template<typename CollT, typename T>
concept CanPushBack = requires (CollT c, T v) {
                        c.push_back(v);
                      };


void add(auto& coll, const auto& val)
requires CanPushBack<decltype(coll), decltype(val)>
{
  coll.push_back(val);
}

void add(auto& coll, const auto& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);    // OK, uses 1st  add() calling push_back()
add(coll2, 42);    // OK, uses 2nd add() calling insert()
```

C++

©2024 by josuttis.com

20

josuttis | eckstein
IT communication

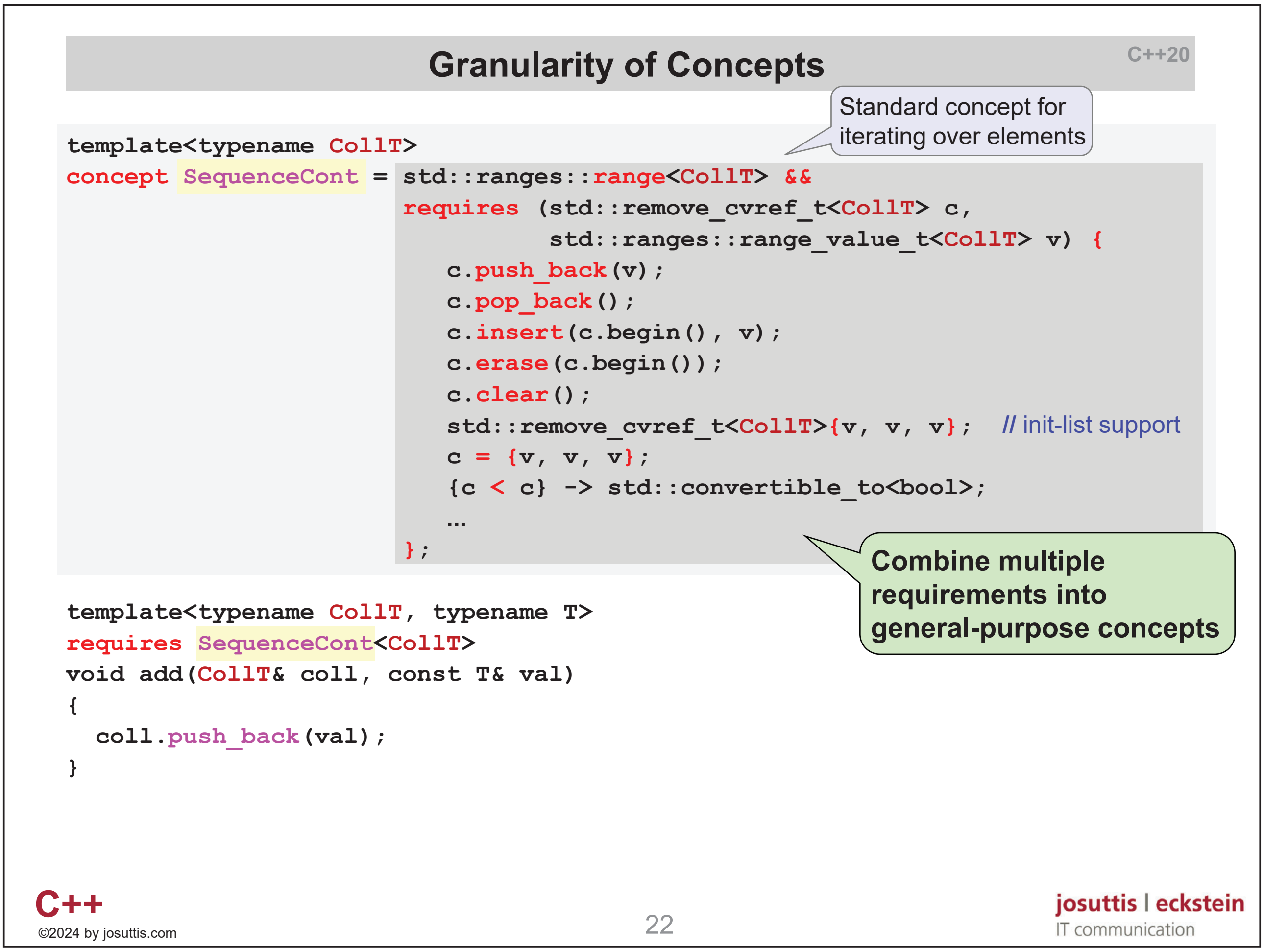


Granularity of Concepts

C++20

```
template<typename CollT, typename T>
concept CanPushBack = requires (CollT c, T v) {
                        c.push_back(v);
                      };


void add(auto& coll, const auto& val)
requires CanPushBack<decltype(coll), decltype(val)>
{
  coll.push_back(val);
}

void add(auto& coll, const auto& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);    // OK, uses 1st add() calling push_back()
add(coll2, 42);    // OK, uses 2nd add() calling insert()
```

Don't introduce a concept for each statement
(too fine grained)



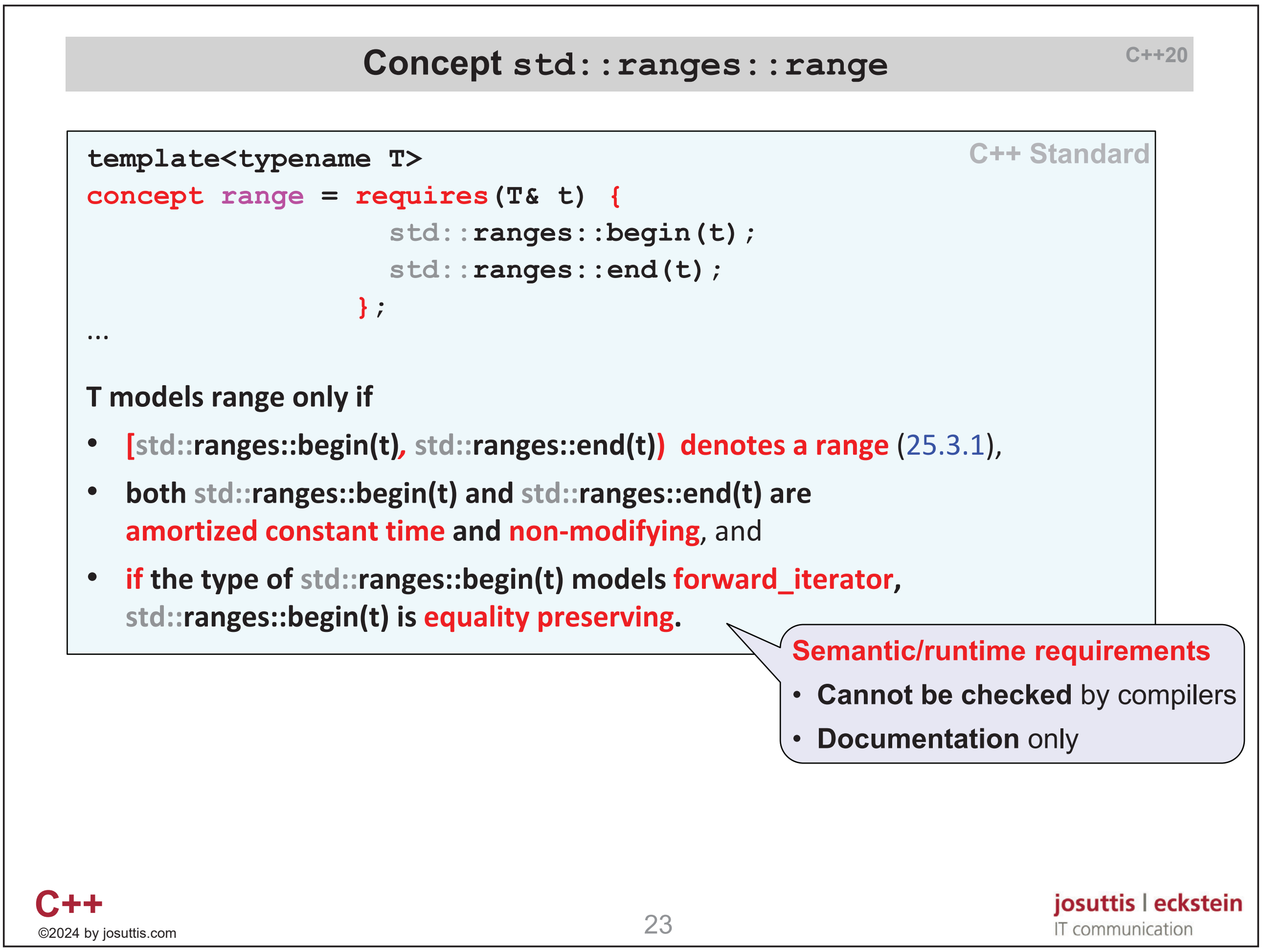
Granularity of Concepts

C++20

Standard concept for iterating over elements

```
template<typename CollT>
concept SequenceCont = std::ranges::range<CollT> &&
                       requires (std::remove_cvref_t<CollT> c,
                                 std::ranges::range_value_t<CollT> v) {
                         c.push_back(v);
                         c.pop_back();
                         c.insert(c.begin(), v);
                         c.erase(c.begin());
                         c.clear();
                         std::remove_cvref_t<CollT>{v, v, v};   // init-list support
                         c = {v, v, v};
                         {c < c} -> std::convertible_to<bool>;
                         …
                       };

template<typename CollT, typename T>
requires SequenceCont<CollT>
void add(CollT& coll, const T& val)
{
  coll.push_back(val);
}
```

Combine multiple requirements into general-purpose concepts



Concept `std::ranges::range`

C++20

C++ Standard

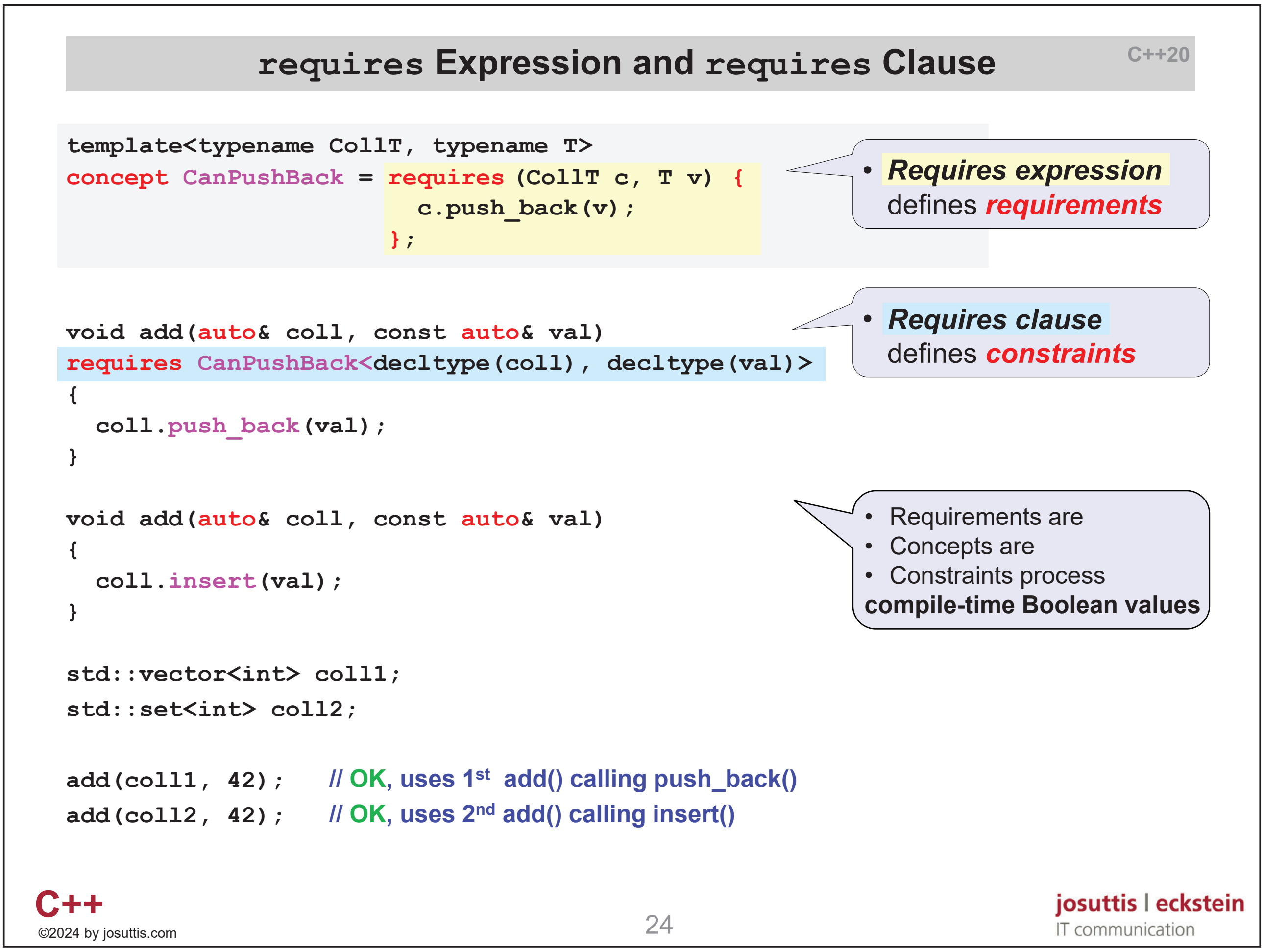
```
template<typename T>
concept range = requires(T& t) {
                  std::ranges::begin(t);
                  std::ranges::end(t);
                };
...
```

T models range only if

- [std::ranges::begin(t), std::ranges::end(t)) denotes a range (25.3.1),
- both std::ranges::begin(t) and std::ranges::end(t) are amortized constant time and non-modifying, and
- if the type of std::ranges::begin(t) models forward_iterator, std::ranges::begin(t) is equality preserving.

Semantic/runtime requirements

- **Cannot be checked** by compilers
- **Documentation** only



`requires` Expression and `requires` Clause

C++20

```
template<typename CollT, typename T>
concept CanPushBack = requires (CollT c, T v) {
                        c.push_back(v);
                      };
```

- ***Requires expression*** defines ***requirements***

```
void add(auto& coll, const auto& val)
requires CanPushBack<decltype(coll), decltype(val)>
{
  coll.push_back(val);
}
```

- ***Requires clause*** defines ***constraints***

```
void add(auto& coll, const auto& val)
{
  coll.insert(val);
}
```

- Requirements are
- Concepts are
- Constraints process

compile-time Boolean values

```
std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);     // OK, uses 1st  add() calling push_back()
add(coll2, 42);     // OK, uses 2nd add() calling insert()
```

Combining requires Expression and requires Clause

C++20

```
void add(auto& coll, const auto& val)
requires requires { coll.push_back(val); }
{
  coll.push_back(val);
}

void add(auto& coll, const auto& val)
{
  coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);    // OK, uses 1st  add() calling push_back()
add(coll2, 42);    // OK, uses 2nd add() calling insert()
```

- ***Requires expression*** defines ***requirements***
- ***Requires clause*** defines ***constraints***

- Requirements are
- Concepts are
- Constraints process

compile-time Boolean values

requires and Compile-Time if

C++20

```
void add(auto& coll, const auto& val)
{
  if constexpr (requires { coll.push_back(val); }) {
    coll.push_back(val);
  }
  else {
    coll.insert(val);
  }
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);    // OK, calls push_back()
add(coll2, 42);    // OK, calls insert()
```

- Requirements are
- Concepts are
- Constraints process

compile-time Boolean values

Concepts and Error Messages

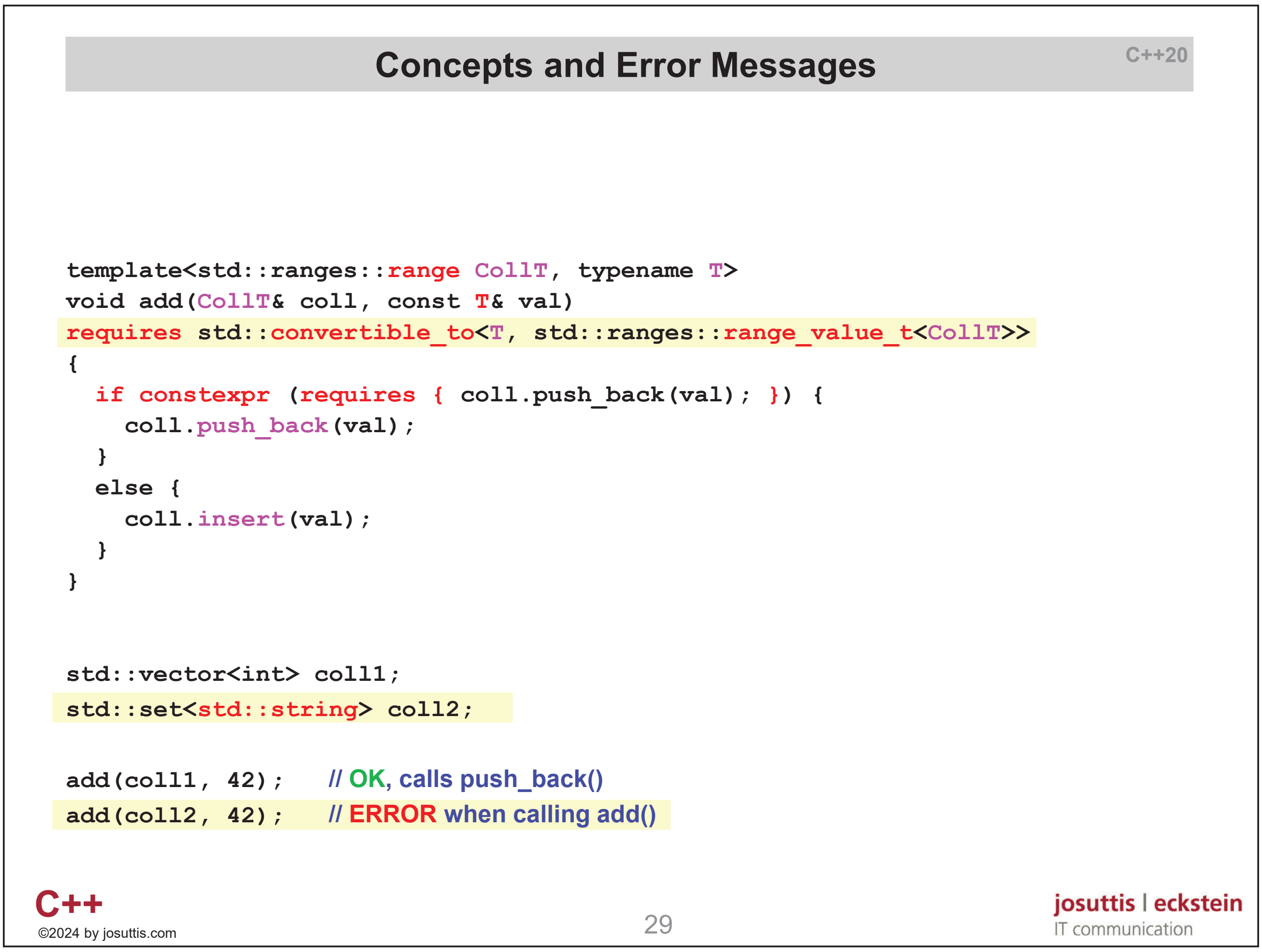
C++20

```
void add(auto& coll, const auto& val)
{
  if constexpr (requires { coll.push_back(val); }) {
    coll.push_back(val);
  }
  else {
    coll.insert(val);
  }
}


std::vector<int> coll1;
std::set<std::string> coll2;

add(coll1, 42);    // OK, calls push_back()
add(coll2, 42);    // ERROR
```

Possible Error Message:

```
prog.cpp:16:10: error: no matching member function for call to 'insert'
    coll.insert(val);
    ~~~~~^~~~~~
prog.cpp:30:1: note: in instantiation of function template specialization
'add<std::set<std::basic_string<char>>, int>' requested here
add(coll2, 42);
^
/include/c++/12.0.1/bits/stl_set.h:509:7: note: candidate function not viable: no
known conversion from 'const int' to 'const
std::set<std::basic_string<char>>::value_type' (aka 'const std::basic_string<char>')
for 1st argument
      insert(const value_type& __x)
      ^
/include/c++/12.0.1/bits/stl_set.h:518:7: note: candidate function not viable: no
known conversion from 'const int' to 'std::set<std::basic_string<char>>::value_type'
(aka 'std::basic_string<char>') for 1st argument
      insert(value_type&& __x)
      ^
…
/include/c++/12.0.1/bits/stl_set.h:603:7: note: candidate function not viable:
requires 2 arguments, but 1 was provided
      insert(const_iterator __hint, node_type&& __nh)
      ^
1 error generated.
```

```
void add(auto& col
{
  if constexpr (re
    coll.push_back
  }
  else {
    coll.insert(va
  }
}

std::vector<int> coll1;
std::set<std::string> coll2;

add(coll1, 42);    // OK, calls push_back()
add(coll2, 42);    // ERROR in the code of std::set<> when calling insert()
```



Concepts and Error Messages

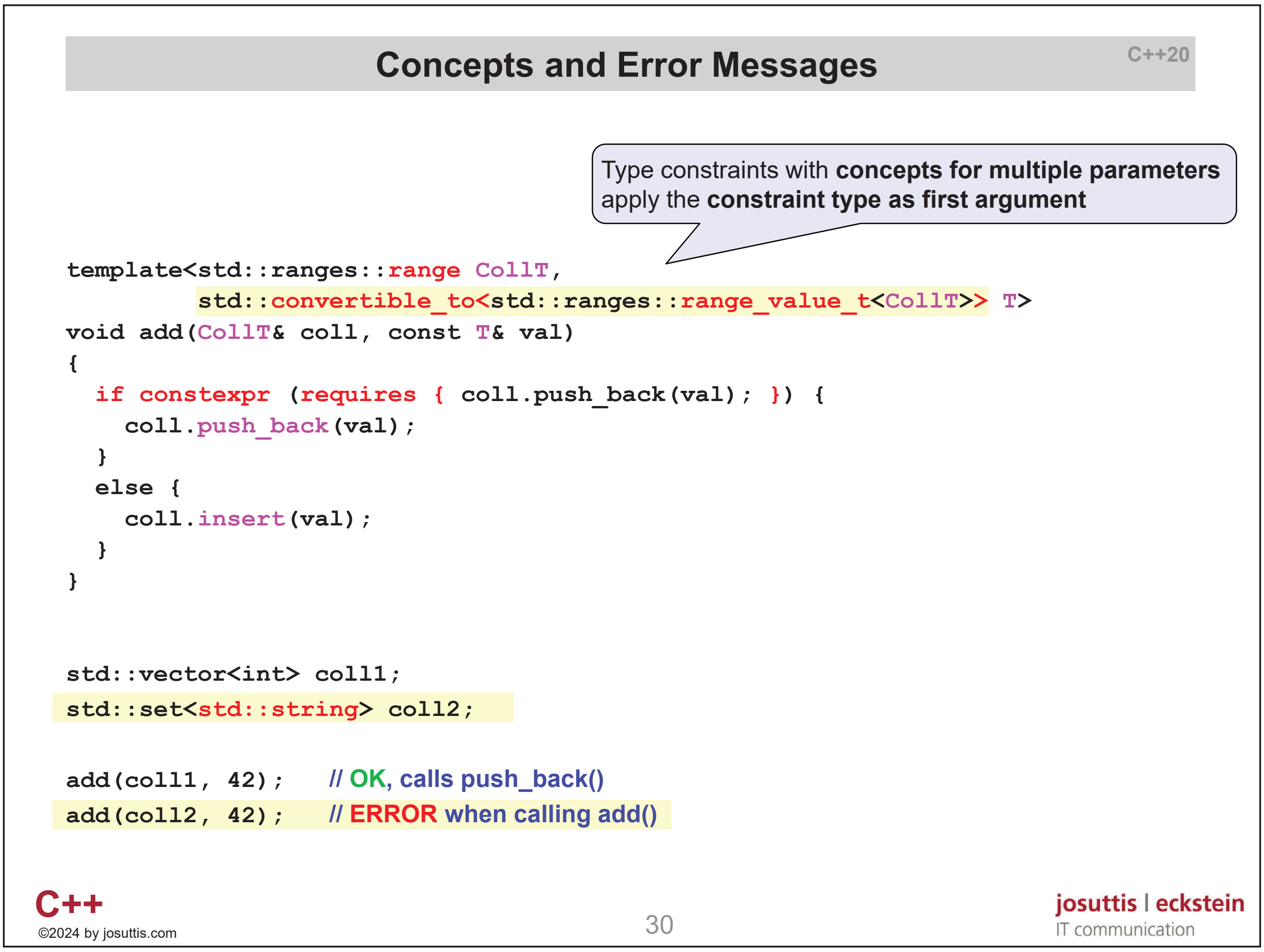
C++20

```
template<std::ranges::range CollT, typename T>
void add(CollT& coll, const T& val)
requires std::convertible_to<T, std::ranges::range_value_t<CollT>>
{
  if constexpr (requires { coll.push_back(val); }) {
    coll.push_back(val);
  }
  else {
    coll.insert(val);
  }
}


std::vector<int> coll1;
std::set<std::string> coll2;

add(coll1, 42);    // OK, calls push_back()
add(coll2, 42);    // ERROR when calling add()
```

C++

©2024 by josuttis.com

josuttis | eckstein
IT communication



Concepts and Error Messages

C++20

Type constraints with **concepts for multiple parameters** apply the **constraint type as first argument**

```
template<std::ranges::range CollT,
         std::convertible_to<std::ranges::range_value_t<CollT>> T>
void add(CollT& coll, const T& val)
{
  if constexpr (requires { coll.push_back(val); }) {
    coll.push_back(val);
  }
  else {
    coll.insert(val);
  }
}


std::vector<int> coll1;
std::set<std::string> coll2;

add(coll1, 42);    // OK, calls push_back()
add(coll2, 42);    // ERROR when calling add()
```

C++

©2024 by josuttis.com

josuttis | eckstein
IT communication

Concepts and Error Messages

C++20

Possible Error Message (clang):

```
prog.cpp:30:1: error: no matching function for call to 'add'
add(coll2, 42);
^~~
prog.cpp:9:6: note: candidate template ignored: constraints not satisfied [with CollT
= std::set<std::basic_string<char>>, T = int]
void add(CollT& coll, const T& val)
     ^
prog.cpp:10:15: note: because 'std::convertible_to<int,
std::ranges::range_value_t<set<basic_string<char> > > >' evaluated to false
requires std::convertible_to<T, std::ranges::range_value_t<CollT>>
              ^
/include/c++/12.0.1/concepts:72:30: note: because 'is_convertible_v<int,
std::basic_string<char> >' evaluated to false
    concept convertible_to = is_convertible_v<_From, _To>
                             ^
1 error generated.
```

```
template<std::rang
         std::conv
void add(CollT& co
{
  if constexpr (re
    coll.push_back
  }
  else {
    coll.insert(va
  }
}


std::vector<int> coll1;
std::set<std::string> coll2;

add(coll1, 42);    // OK, calls push_back()
add(coll2, 42);    // ERROR when calling add()
```

C++20

Concepts in Detail

Concepts Terminology

C++20

- **Requirements**
  - Expressions to specify a restriction with `requires{...}`
    - Operations that have to be valid
    - Types that have to be defined/returned
- **Concepts**
  - Names for one or more requirements
- **Constraints**
  - Restrictions for the availability/usability of generic code
  - Specified as
    - `requires` clauses of concepts or ad-hoc requirements
    - Type constraints (concepts applied to template parameters or `auto`)
- **No code is generated**
  - Code is evaluated only to decide *whether/what* to compile

Different Constraints Can Create Ambiguities

C++20

```
template<typename CollT>
concept HasSize = requires (CollT c) {
                    { c.size() } -> std::convertible_to<int>;
                  };
```

```
template<typename CollT>
concept HasIndexOp = requires (CollT c) { c[0]; };
```

```
template<typename CollT>
requires HasSize<CollT>                                // has to support size()
void foo(CollT& coll) {
  std::cout << "foo() for container with size()\n";
}

template<typename CollT>
requires HasIndexOp<CollT>                             // has to support []
void foo(CollT& coll) {
  std::cout << "foo() for container []\n";
}

std::list<int> lst{0, 8, 15};
std::vector<int> vec{0, 8, 15};

foo(lst);   // OK: calls first foo()
foo(vec);   // ERROR: ambiguous
```

Output:

```
foo() for container with size()
```

Constraints with Concepts Can Subsume

C++20

```
template<typename CollT>
concept HasSize = requires (CollT c) {
                    { c.size() } -> std::convertible_to<int>;
                  };
```

```
template<typename CollT>
concept HasIndexOp = requires (CollT c) { c[0]; };
```

```
template<typename CollT>
requires HasSize<CollT>                          // has to support size()
void foo(CollT& coll) {
  std::cout << "foo() for container with size()\n";
}

template<typename CollT>
requires HasSize<CollT> && HasIndexOp<CollT>     // has to support size() and []
void foo(CollT& coll) {
  std::cout << "foo() for container with size() and []\n";
}

std::list<int> lst{0, 8, 15};
std::vector<int> vec{0, 8, 15};
foo(lst);   // OK: calls first foo()
foo(vec);   // OK: calls second foo()
```

HasSize<> && HasIndexOp<> *subsumes* HasSize<>

Output:

```
foo() for container with size()
foo() for container with size() and []
```

C++

©2024 by josuttis.com

josuttis | eckstein
IT communication

Concept Subsumption

C++20

- **Only concept constraints are checked for subsumption**

```
template<typename T>
concept BigType = sizeof(T) > 8;

template<typename T>
concept BigClassType0 = sizeof(T) > 8 && std::is_class_v<T>;
```

Does not subsume concept BigType

```
void foo1(BigType auto) { … }
void foo1(BigClassType0 auto) { … }
std::string s;
foo1(s);            // ERROR: ambiguous
```

```
template<typename T>
concept BigClassType = BigType<T> && std::is_class_v<T>;
```

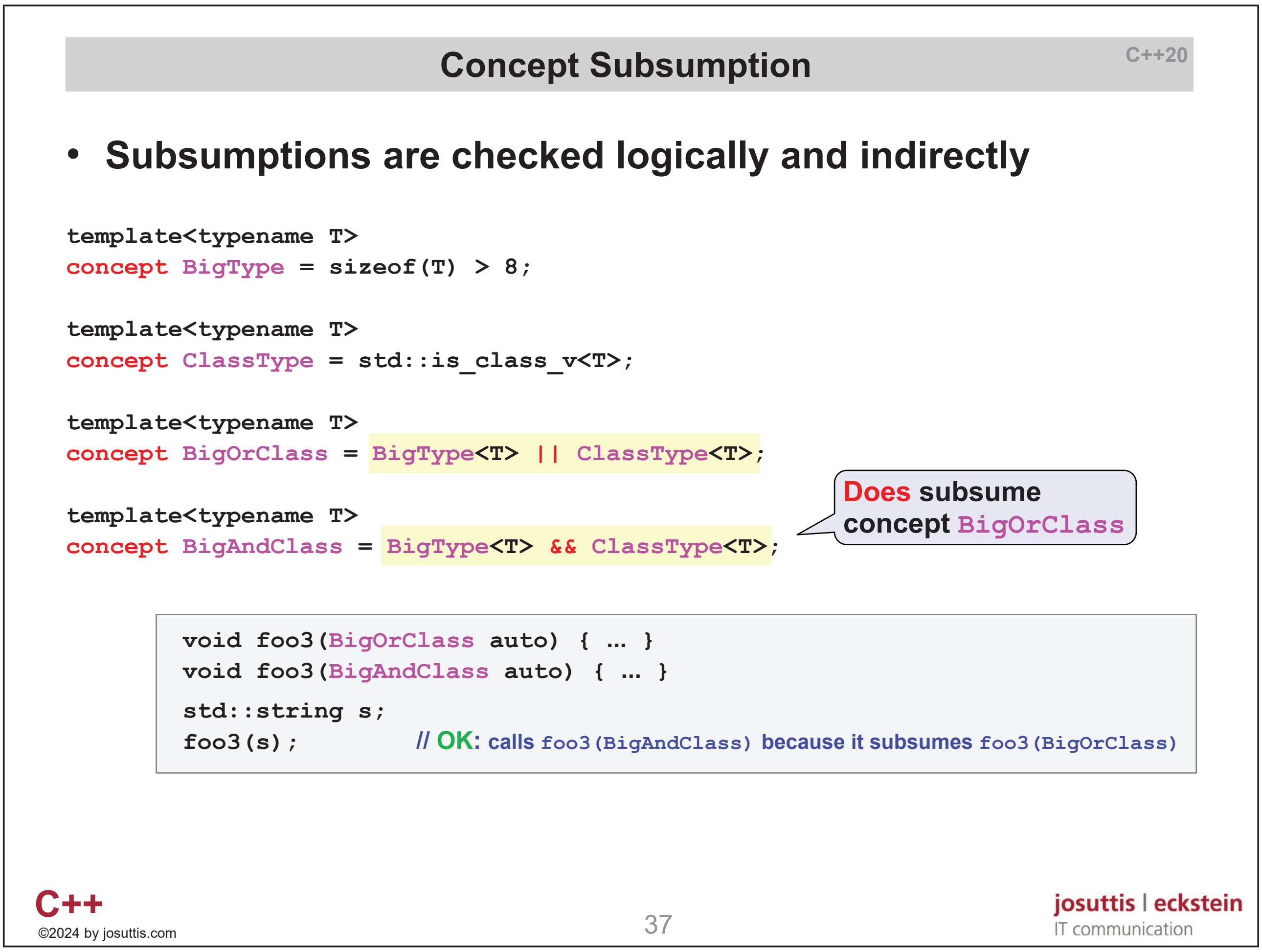
Does subsume concept BigType

```
void foo2(BigType auto) { … }
void foo2(BigClassType auto) { … }
std::string s;
foo2(s);            // OK: calls foo2(BigClassType)
```

C++

©2024 by josuttis.com

josuttis | eckstein
IT communication



Concept Subsumption

C++20

- **Subsumptions are checked logically and indirectly**

```
template<typename T>
concept BigType = sizeof(T) > 8;

template<typename T>
concept ClassType = std::is_class_v<T>;

template<typename T>
concept BigOrClass = BigType<T> || ClassType<T>;

template<typename T>
concept BigAndClass = BigType<T> && ClassType<T>;
```

Does subsume concept BigOrClass

```
void foo3(BigOrClass auto) { … }
void foo3(BigAndClass auto) { … }
std::string s;
foo3(s);          // OK: calls foo3(BigAndClass) because it subsumes foo3(BigOrClass)
```

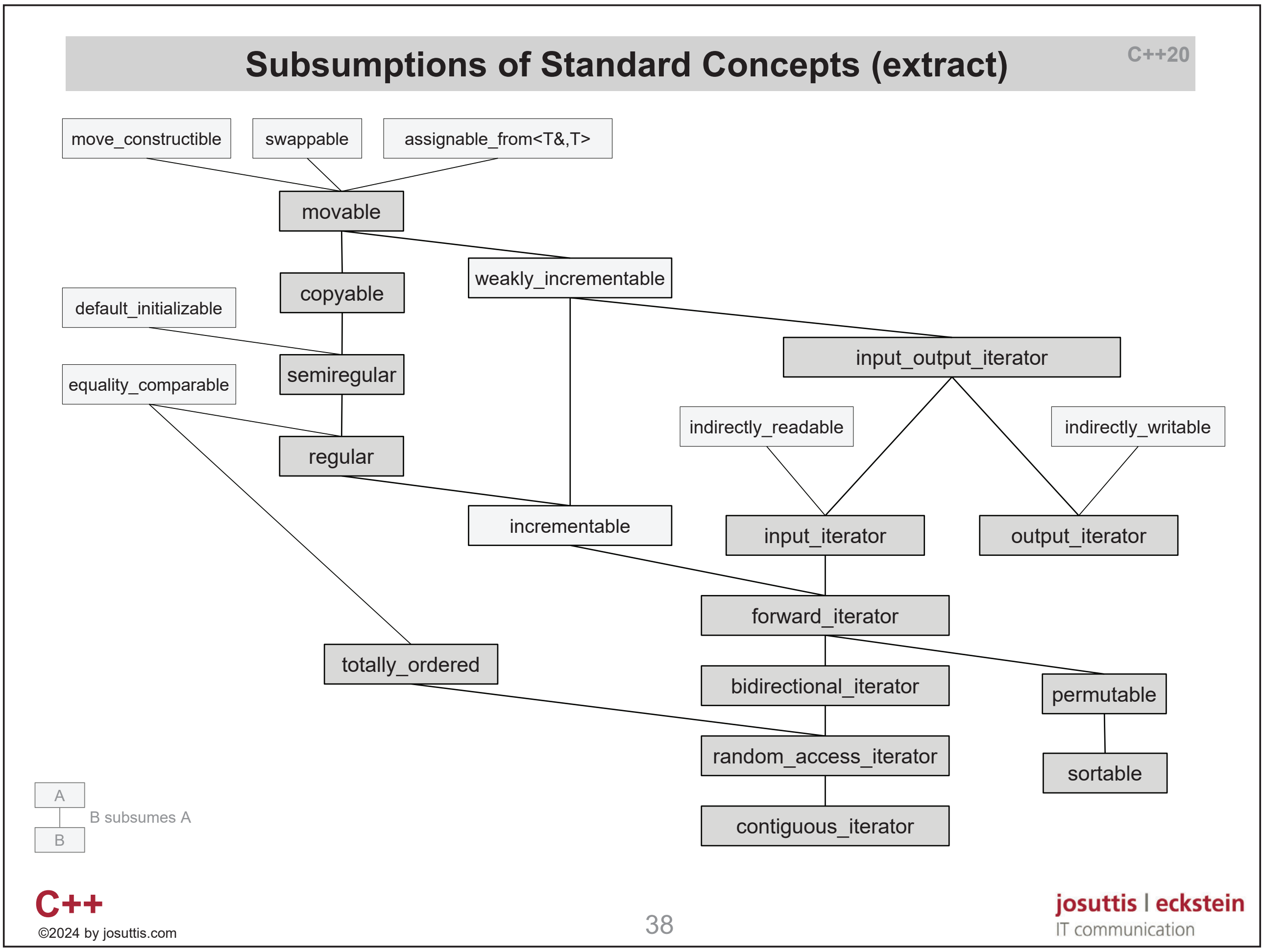
C++

©2024 by josuttis.com

josuttis | eckstein
IT communication

Subsumptions of Standard Concepts (extract)

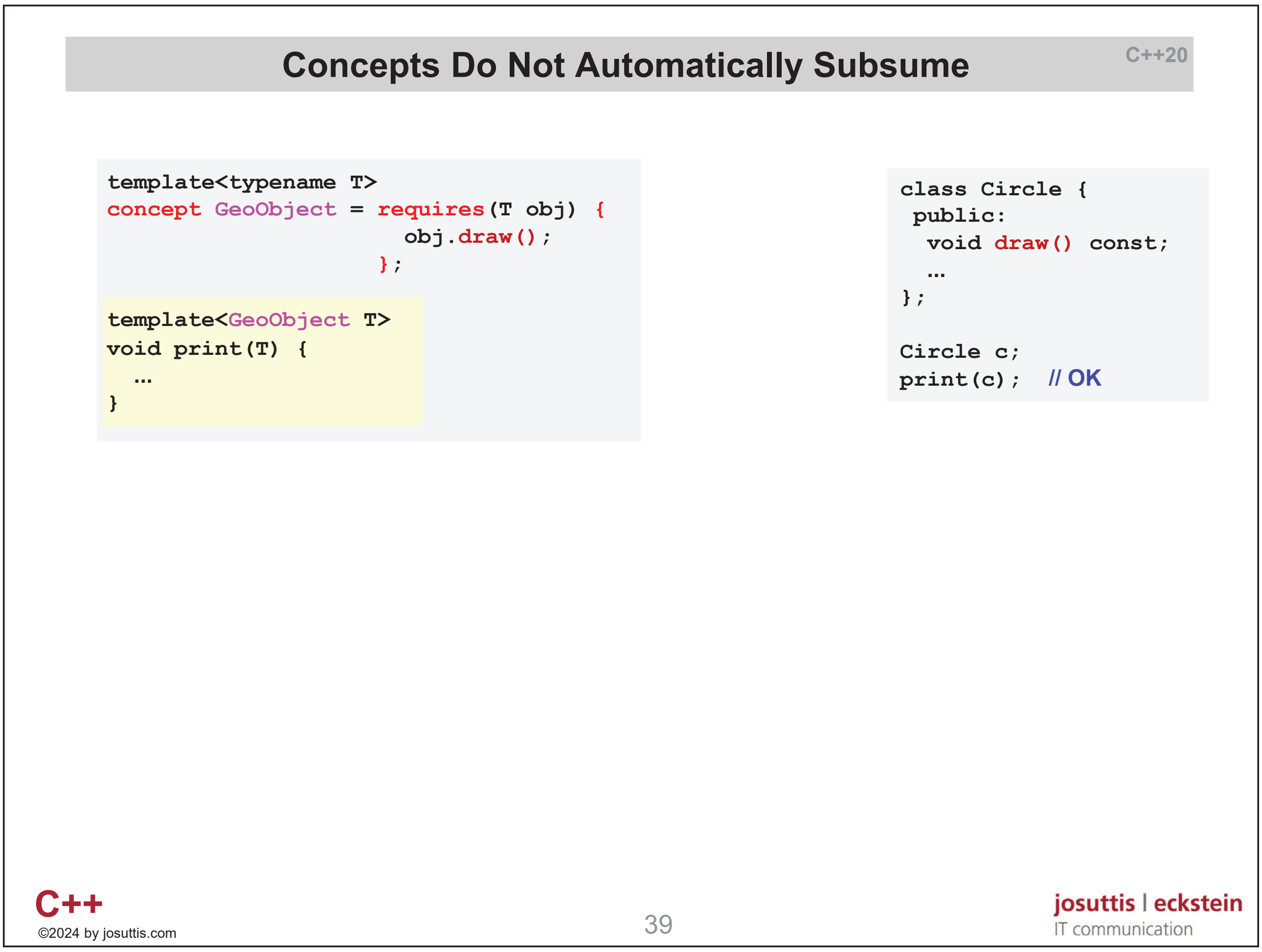
C++20



C++

©2024 by josuttis.com

josuttis | eckstein
IT communication



Concepts Do Not Automatically Subsume

C++20

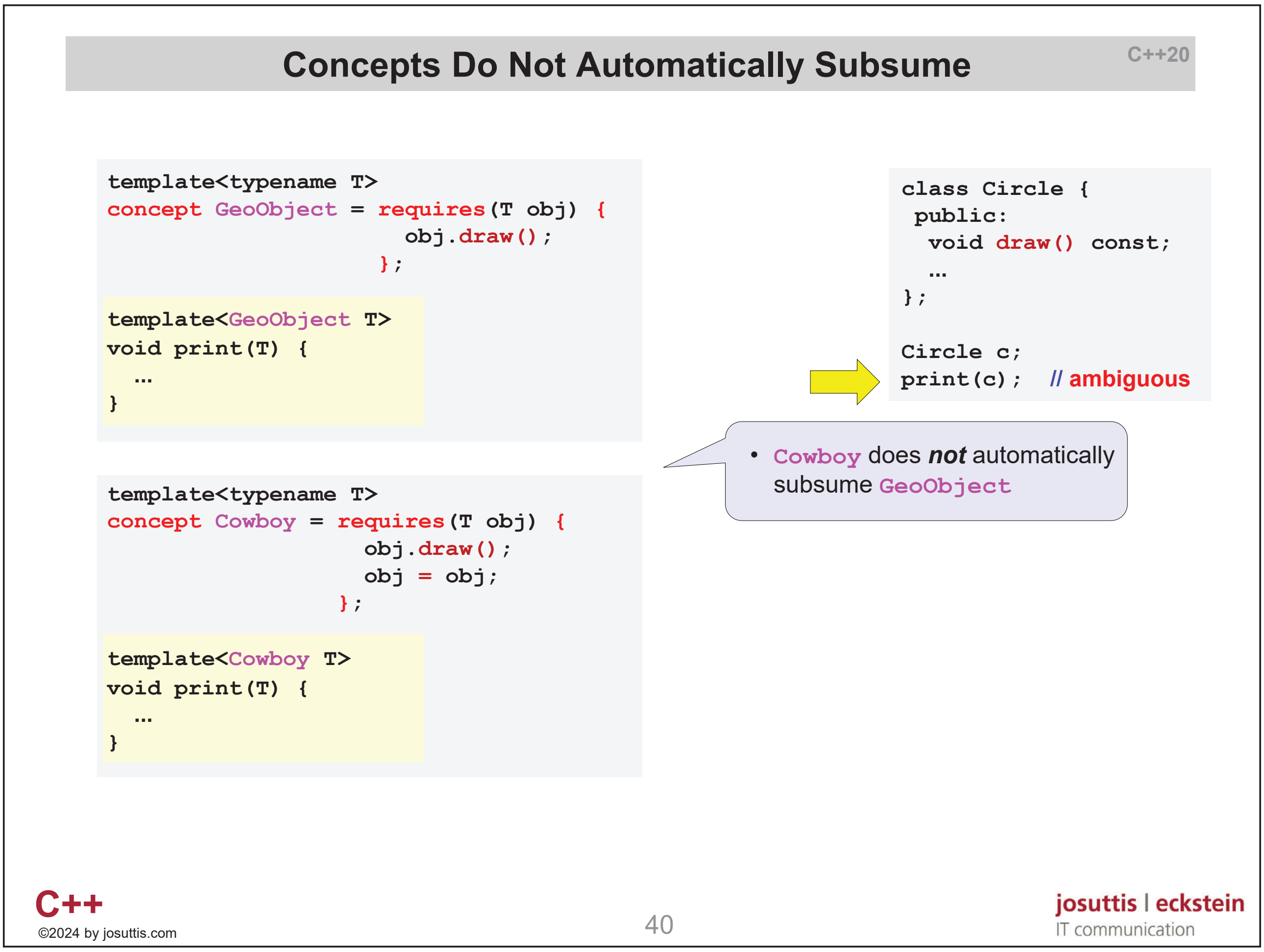
```
template<typename T>
concept GeoObject = requires(T obj) {
                      obj.draw();
                    };

template<GeoObject T>
void print(T) {
  …
}
```

```
class Circle {
 public:
  void draw() const;
  …
};

Circle c;
print(c);   // OK
```

C++

©2024 by josuttis.com

josuttis | eckstein
IT communication



Concepts Do Not Automatically Subsume

C++20

```
template<typename T>
concept GeoObject = requires(T obj) {
                      obj.draw();
                    };

template<GeoObject T>
void print(T) {
  …
}
```

```
template<typename T>
concept Cowboy = requires(T obj) {
                   obj.draw();
                   obj = obj;
                 };

template<Cowboy T>
void print(T) {
  …
}
```

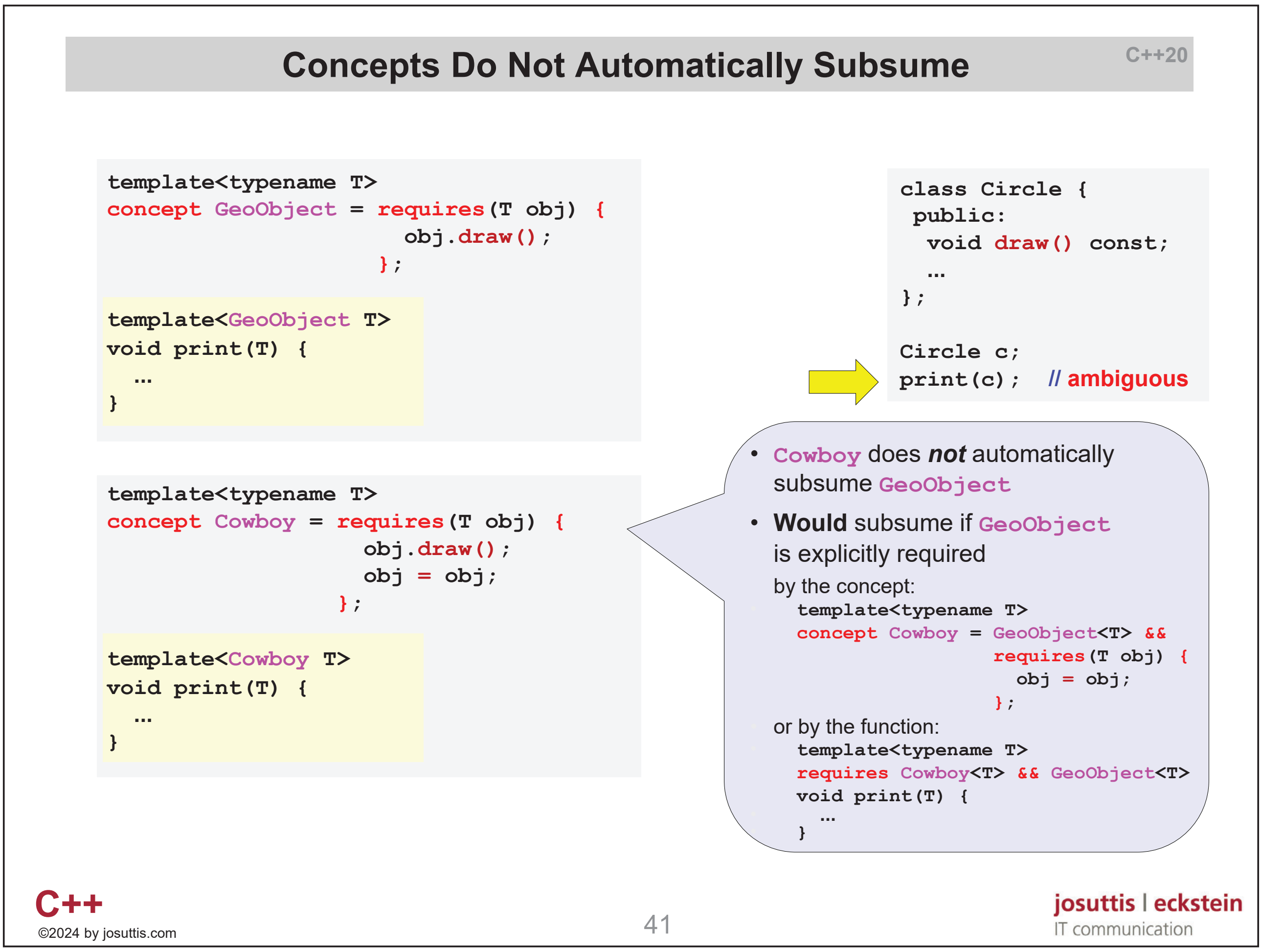
```
class Circle {
 public:
  void draw() const;
  …
};

Circle c;
print(c);   // ambiguous
```

- Cowboy does *not* automatically subsume GeoObject

C++

©2024 by josuttis.com

josuttis | eckstein
IT communication



Concepts Do Not Automatically Subsume

C++20

```
template<typename T>
concept GeoObject = requires(T obj) {
                      obj.draw();
                    };

template<GeoObject T>
void print(T) {
  …
}
```

```
template<typename T>
concept Cowboy = requires(T obj) {
                   obj.draw();
                   obj = obj;
                 };

template<Cowboy T>
void print(T) {
  …
}
```

```
class Circle {
 public:
  void draw() const;
  …
};

Circle c;
print(c);    // ambiguous
```

- Cowboy does *not* automatically subsume GeoObject
- **Would** subsume if GeoObject is explicitly required

by the concept:

```
template<typename T>
concept Cowboy = GeoObject<T> &&
                 requires(T obj) {
                   obj = obj;
                 };
```

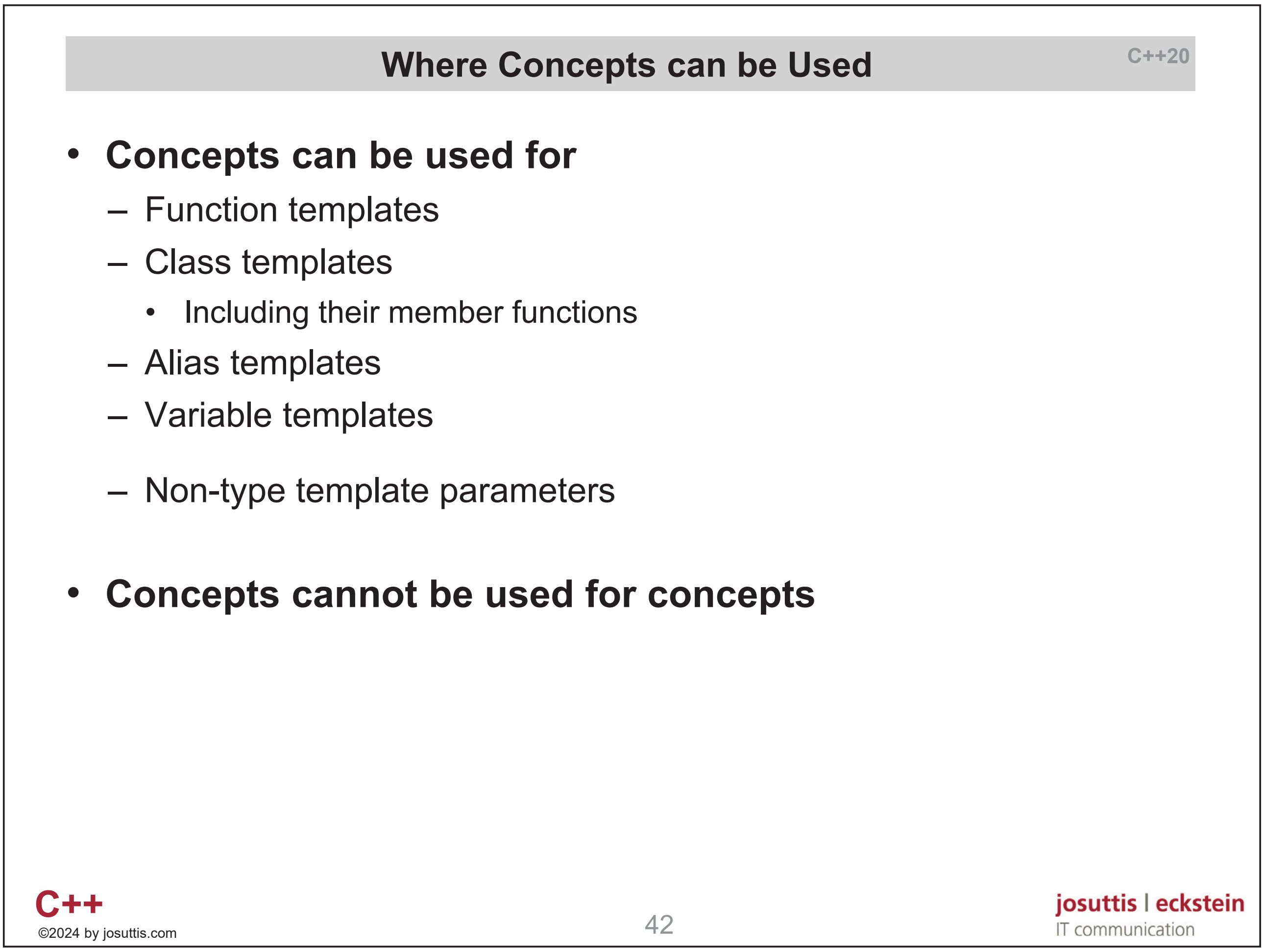
or by the function:

```
template<typename T>
requires Cowboy<T> && GeoObject<T>
void print(T) {
  …
}
```

C++

©2024 by josuttis.com

josuttis | eckstein
IT communication



Where Concepts can be Used

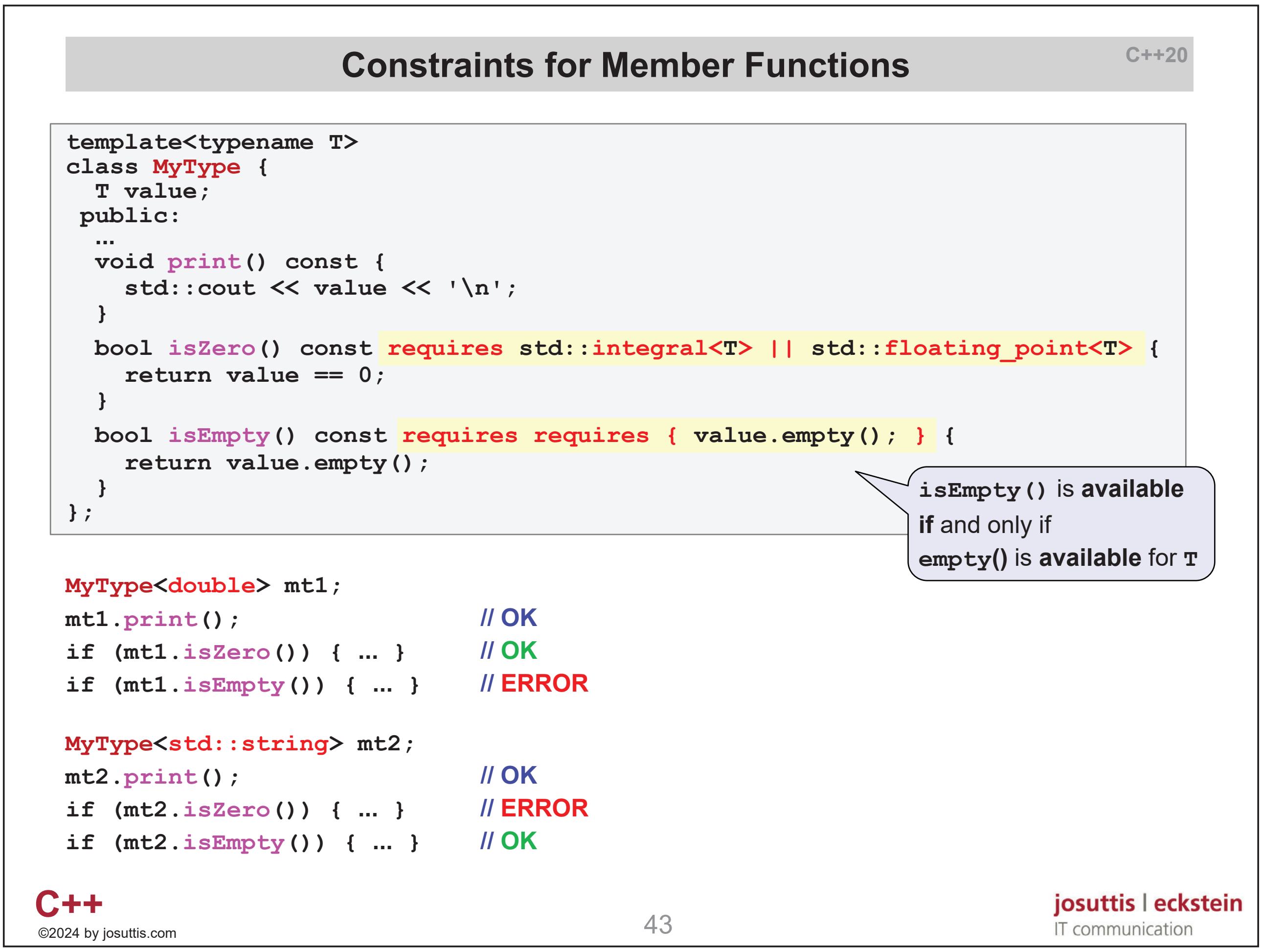
C++20

- **Concepts can be used for**
  - Function templates
  - Class templates
    - Including their member functions
  - Alias templates
  - Variable templates
  - Non-type template parameters
- **Concepts cannot be used for concepts**

C++

©2024 by josuttis.com

josuttis | eckstein
IT communication



Constraints for Member Functions

C++20

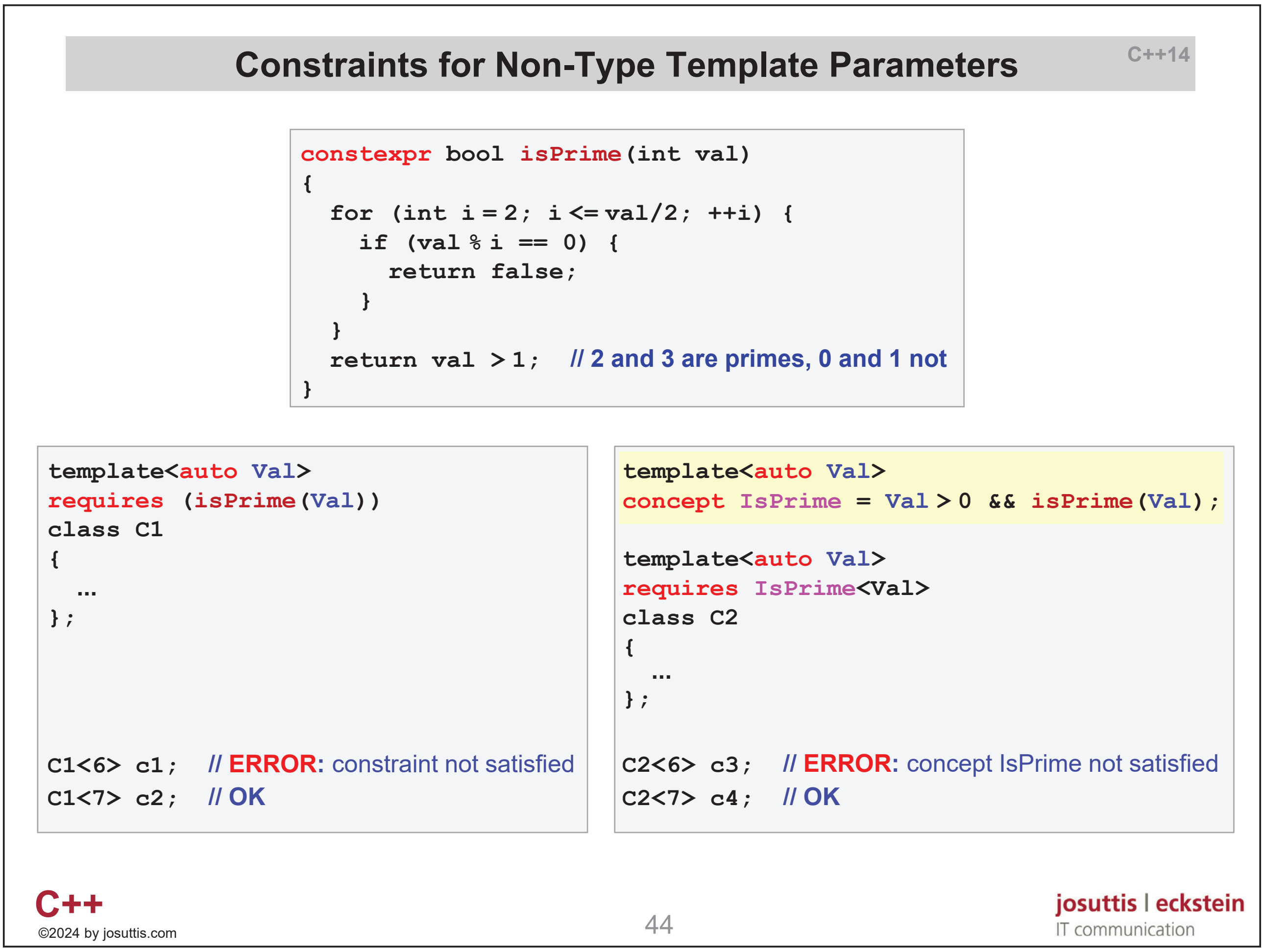
```
template<typename T>
class MyType {
   T value;
  public:
   …
   void print() const {
     std::cout << value << '\n';
   }
   bool isZero() const requires std::integral<T> || std::floating_point<T> {
     return value == 0;
   }
   bool isEmpty() const requires requires { value.empty(); } {
     return value.empty();
   }
};
```

`isEmpty()` is available if and only if **`empty()` is available** for `T`

```
MyType<double> mt1;
mt1.print();                   // OK
if (mt1.isZero()) { … }        // OK
if (mt1.isEmpty()) { … }       // ERROR

MyType<std::string> mt2;
mt2.print();                   // OK
if (mt2.isZero()) { … }        // ERROR
if (mt2.isEmpty()) { … }       // OK
```

C++

josuttis | eckstein
IT communication



Constraints for Non-Type Template Parameters

C++14

```
constexpr bool isPrime(int val)
{
  for (int i = 2; i <= val/2; ++i) {
    if (val % i == 0) {
      return false;
    }
  }
  return val > 1;   // 2 and 3 are primes, 0 and 1 not
}
```

```
template<auto Val>
requires (isPrime(Val))
class C1
{
  …
};

C1<6> c1;   // ERROR: constraint not satisfied
C1<7> c2;   // OK
```

```
template<auto Val>
concept IsPrime = Val > 0 && isPrime(Val);

template<auto Val>
requires IsPrime<Val>
class C2
{
  …
};

C2<6> c3;   // ERROR: concept IsPrime not satisfied
C2<7> c4;   // OK
```

C++

josuttis | eckstein
IT communication

Thank You!

There is way more to teach about concepts and C++20.
Enjoy my next C++20 class

Nicolai M. Josuttis

www.josuttis.com
nico@josuttis.com
@NicoJosuttis