

# The Concepts of concepts

---

Sandor DARGO

Meeting C++

12th November 2021

# Who Am I?

Sándor DARGÓ

Principal Engineer in Amadeus

Enthusiastic blogger <https://www.sandordargo.com>

(A former) Passionate traveller

Curious home baker

Happy father of two



# Agenda

Why do we need something like `concepts`?

What are `concepts` and `constraints`?

How to use `concepts` with functions?

How to use `concepts` with classes?

How to write `concepts`?

Real-life examples

Why do we need something like concepts?

---

# Overloads don't scale

Tedious to write all

Verbose

Difficult to maintain

```
double add(double a, double b) {  
    return a+b;  
}  
  
int add(int a, int b) {  
    return a+b;  
}  
  
int main() {  
    add(42, 66);  
    add(4.2, 6.6);  
}
```

# Templates allow just anything

No constraints

Potentially unexpected  
behaviour

Is that really what you want?

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

int main() {
    add(42, 66);
    add(42.42L, 66.6L);
    add('a', 'b');
}
```

# Forbid template specializations

Works

But doesn't scale!

```
#include <string>

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

template<>
std::string add(std::string,
                std::string) = delete;

int main() {
    // ERROR
    add(std::string{"a"}, std::string{"b"});
}
```

# What about type traits?

Nice(r) error messages!

Less overhead

Everything is at one place

Not intuitively easily  
reusable

Hidden requirements

```
template <typename T>
T add(T a, T b) {
    static_assert(std::is_integral_v<T> ||
        std::is_floating_point_v<T>,
        "Call add only with numbers!");
    return a+b;
}

int main() {
    add(std::string{"a"},
        std::string{"b"});
}
```



# concepts to the rescue

Readable

Reusable

Scalable

Safer

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

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

```
int main() {
    add(1, 2);
}
```

What are concepts and constraints?

---

# What are concepts?

One of the new major features of C++20

## **Concepts**

Coroutines

Modules

Ranges

# Extension for templates

Compile-time booleans to validate  
template arguments

```
template <typename T>  
bool concept Any = true;
```

```
template <typename T>  
concept any = true;
```

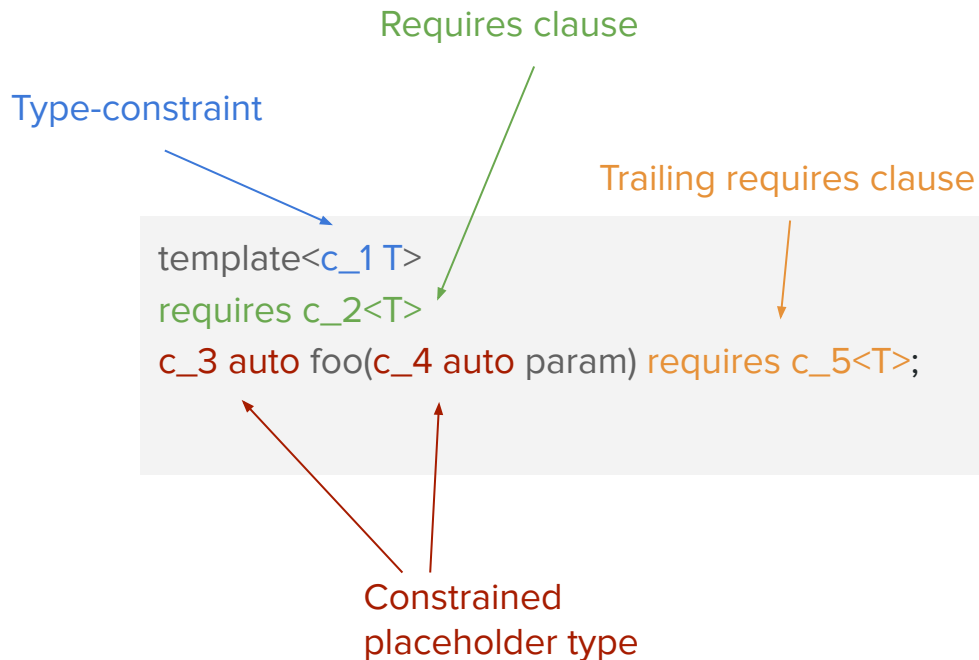
# Concepts as constraints

Template arguments can be constrained

A concept can be used as a constraint

A concept is a set of requirements

# Concepts appearing at different positions



# 4 ways to use concepts with functions

---

# Disclaimer: **concept incomplete!**

We are going use the  
concept `number`

**It's incomplete!**

Bear with me

```
#include <concepts>

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



# Using the `requires` clause

`requires` following the  
template parameter list

After `requires` write your  
concept(s) to be satisfied

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

# Concepts can be combined in different ways

In fact not only concepts

Type traits

Any boolean expression

Compile-time values

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

# Function calls as usual with better error messages

```
add(2, 3);
```

```
add(2, 3.14);
```

```
no matching function for call to 'add(int,  
double)'
```

```
17 | std::cout << add(2, 3.14);
```

```
candidate: 'template<class T> requires  
number<T> auto add(T, T)'
```

```
9 | auto add(T a, T b) {
```

```
template argument deduction/substitution  
failed:
```

```
deduced conflicting types for parameter 'T'  
('int' and 'double')
```

```
17 | std::cout << add(2, 3.14);
```

# Trailing requires clause

requires comes after cv  
qualifiers

Apart from that, same as  
“normal” requires clause

Supports

- combinations

- multiple parameter types

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

# Constrained template parameter

No more `requires` clause

`typename` is replaced by the  
concept

Supports multiple parameters

Doesn't support a  
combination of concepts

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

# Abbreviated function templates

No `requires`, no template parameter list

Use ***concept\_name*** `auto` in the parameter list

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

*T.12: Prefer concept names over auto for local variables*

Parameter types can be implicitly different

No support for requirements expressions

# How to choose among the 4 ways?

Do you have a complex requirement?

Choose requires or trailing requires!

Only a simple requirement?

*T.13: Take abbreviated function templates!*

Simple requirements? Want to keep it short and want to control to binded types of the parameters?

Go with the constrained template parameter!

# Are they all the same after all?

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

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

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



# There is an expected difference

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

# How to use concepts with classes

---

# Fewer syntactical choices

Abbreviated function templates wouldn't make sense

Trailing `requires` clause only in certain circumstances

# The `requires` clause

`requires` following the  
template parameter list

After `requires` write your  
concept(s) to be satisfied

You can use complex  
constraints

```
template <typename T>
requires number<T>
class WrappedNumber {
    public:
        WrappedNumber(T num) :
            m_num(num) {}
    private:
        T m_num;
};
```

# Constrained template parameters

Instead of `typename`  
keyword use simply a  
concept

No complex constraints

```
template <number T>
class WrappedNumber {
public:
    WrappedNumber(T num) :
        m_num(num) {}
private:
    T m_num;
};
```

# Overload by constraints

Class level templates  
with concepts on  
functions

Provide different  
“overloads” for different  
parameter types

```
template <typename T>
class MyNumber {
public:

    T divide(const T& divisor)
        requires std::integral<T> {
        // ...
    }

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

# Functions are added based on constraints

```
template <typename T>
concept car = requires (T car) {
    car.startEngine();
    car.openDoor();
};

template <typename T>
concept is_convertible = car<T> && requires (T car) {
    car.openRoof();
};

template<car C>
class VacationDriver {
public:
    void cruise() { /* .. */ }
    void openRoof() requires is_convertible<C> { /* .. */}
private:
    C m_car;
};
```

```
class SUV {
public:
    void openDoor() {}
    void startEngine() {}
};

int main(){
    VacationDriver<SUV> vd;

    // error
    // invalid reference to function 'openRoof':
    // constraints not satisfied
    vd.openRoof()
}
```

# How to write concepts?

---



# How to write a simple concept?

List all template parameters

After the keyword `concept` comes the name

Then the requirements

```
template<typename T>  
concept any = true;
```

# What kind of requirements can we express?

Expectations on the public interface

Syntactic requirements

Also semantic requirements

*T.20: Avoid “concepts” without meaningful semantics*

```
template <typename C>
concept car = requires (C c) {
    c.openDoor();
    c.closeDoor();
    c.startEngine();
    c.stopEngine();
    c.accelerate();
    c.brake();
};

class Tank {
    // ...
};

// syntactically valid but
// a car is still not a tank...
static_assert(car<Tank>);
```

# Combine already defined concepts

*T.11: Whenever possible use standard concepts*

User defined ones are also OK

Use them in any logical combination\*

```
#include <concepts>

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

What does combining concepts mean?

---

# Conjunctions (&&) and disjunctions (||) are OK

You are free to combine

concepts

bool literals

bool expressions

type traits (::value, \_v)

requires expressions

Beware of the !

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

template <typename T>
concept number =
    std::integral<T> ||
    (std::is_floating_point_v<T> ==
     true);
```

# What does the ! say?

For boolean expressions,  
subexpressions

Are well-formed

Compile

Return `false`

For concepts, a subexpression

Might return `false`

Can be ill-formed

The rest can be still satisfied

# The opposite of `true` is not `false`

It doesn't have to be compilable

It might return `false`

Expecting `false` is possible

With a cast to `bool`

Or with a more explicit way

```
template <typename T, typename U>
    requires std::unsigned_integral<
        typename T::Blah> ||
        std::unsigned_integral<
            typename U::Blah>
void foo(T bar, U baz) { /*...*/ }

class MyType {
public:
    using Blah = unsigned int;
    // ...
};

foo(MyType{}, 42);
```

# There is always another way

```
template <typename T, typename
U>

requires
  (bool(std::unsigned_integral<
        typename T::Blah> ||
        std::unsigned_integral<
        typename U::Blah>))

void foo(T bar, U baz) {
  /*...*/ }
}
```

```
template <typename T, typename
U>

requires (
  requires {typename T::Blah;}
  && requires {
    typename U::Blah;}) &&
(std::unsigned_integral<
  typename T::Blah> ||
std::unsigned_integral<
  typename U::Blah>)

void foo(T bar, U baz) {
  /*...*/ }
}
```



How to find the most constrained constraint?

---

# A few words on overload resolution

The best candidate is the most constrained one

A more constrained concept is based on another one

Multiple overloads with the same priority => ambiguity

# The constrained one will be automatically chosen

The right path will be chosen based on the template parameter type's characteristics

*T.25: Avoid complementary constraints*

```
template <typename Key>
class Ignition {
public:
    void start(Key key)
        requires (!Smart<Key>) {
        // ...
    }

    void start(Key key)
        requires Smart<Key> {
        // ...
    }
};
```

# The most constrained will be automatically chosen

The most appropriate overload will be chosen at compile-time

All this happens via concepts subsumption

```
template <typename Key>
class Ignition {
public:
    void start(Key key) {}

    void start(Key key)
        requires Smart<Key> {}

    void start(Key key)
        requires Smart<Key> &&
                Personal<Key> {}
};
```

# Negations bring ambiguity - use named concepts

```
template <typename Key>
class Ignition {

public:

    void start(Key key) {}

    void start(Key key)
        requires (!Smart<Key>) {}

    void start(Key key)
        requires (!Smart<Key>) &&
            Personal<Key> {}

};
```



```
template <typename Key>
class Ignition {

public:

    void start(Key key) {}

    void start(Key key)
        requires NotSmart<Key> {}

    void start(Key key)
        requires NotSmart<Key> &&
            Personal<Key> {}

};
```



So how to write concepts?

---

# Simple requirements on the interface

Use *wishful writing*!

Write down the operation you expect to be compiled

List all the variables used in the requirements after the `requires` keyword

```
template <typename T>
concept test_concept =
requires (T a, T b,
           int exponent) {
    a + b;
    t.square();
    t.power(exponent);
};
```

# Requirements on return types (compound requirements)

Constraint the return types with:

```
std::convertible_to<From, To>
```

```
std::same_as<T, U>
```

Don't forget the braces!

No bare type for future generalizations

```
template <typename T>
concept has_square = requires
(T t) {
    {t.square()} ->
        std::convertible_to<int>;
};
```



# Type requirements

A certain nested type exists

A class template specialization names a type

An alias template specialization names a type

# Require nested types

`std::vector` has inner member  
type `value_type`

`int` doesn't have any member type  
=>

*error: deduced initializer  
does not satisfy  
placeholder constraints  
... the required type  
'typename T::value\_type'  
is invalid*

```
template<typename T>
concept type_requirement =
requires {
    typename T::value_type;
};

int main() {
    type_requirement auto myVec =
        std::vector<int>{1, 2, 3};
    type_requirement auto
        myInt{3};
}
```

# Require nested template specializations

Make sure that a type can be used as a template parameter for another type

Error: *the required type 'Other<T>' is invalid ... constraints not satisfied ...*  
*'template<class T> requires !(same\_as<T, int>) struct Other [with T = int]'*...

```
#include <concepts>

template <typename T>
requires (!std::same_as<T, int>)
struct Other {};

template<typename T>
concept type_requirement = requires
{ typename Other<T>; };

int main() {
    type_requirement auto c = 'c';
    type_requirement auto i = 4;
}
```

# Requirements on alias template specializations

To make sure that an alias template specialization names a type

```
template<typename T> using  
Reference = T&;
```

```
template<typename T>  
concept type_requirement =  
requires {  
    typename Reference<T>;  
};
```

# Nesting is often overcomplication

Use nested requirements on local variables without declaring named concepts

Nested requirement to check what clonable returns

But...

```
#include <concepts>

struct Droid {
    Droid clone() { return {}; }
};
struct DroidV2 {
    Droid clone() { return {}; }
};

template <typename C>
concept cloneable = requires (C cloneable) {
    cloneable.clone();
    requires std::same_as<C, decltype(cloneable.clone())>;
};

int main() {
    cloneable auto c = Droid{};
    // nested requirement 'same_as<C, decltype(cloneable.clone())>'
    // is not satisfied [with C = DroidV2]
    // cloneable auto c2 = DroidV2{};
}
```

## But there is often a simpler option

```
template<typename C>
concept clonable = requires (C clonable) {
    { clonable.clone() } -> std::same_as<C>;
};
```

# Nest to simulate boolean expressions

```
template <typename T, typename
U>

requires
  (bool (std::unsigned_integral<
        typename T::Blah> ||
        std::unsigned_integral<
        typename U::Blah>))

void foo(T bar, U baz) {
  /*...*/ }

```

```
template <typename T, typename
U>
requires (
  requires {typename T::Blah;}
  && requires {
    typename U::Blah;}) &&
  (std::unsigned_integral<
    typename T::Blah> ||
    std::unsigned_integral<
    typename U::Blah>)
void foo(T bar, U baz) {
  /*...*/ }

```

# Real-life examples

---



# Numbers, nothing else, please

Some integral  
types are not  
numbers:

bool

char et al.

```
#include <concepts>
#include <iostream>

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

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

int main() {
    std::cout << add(1, 2) << '\n';
    std::cout << add(1, 2.14) << '\n';
    std::cout << add('1', '2') << '\n';
    std::cout << add(true, false) << '\n';
}
```

# Let's forbid unwanted types

No bools

No chars

```
#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>;

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

int main() {
  // constraints not satisfied, [with auto:11 = bool;
  // auto:12 = bool]': the expression '!(same_as<T, bool>)'
  // [with T = bool]' evaluated to 'false'
  add(true, false); // ERROR
}
```

# Turn poorly documented utility functions...

Usually taking any type

No static assertions

Bad template parameter names (T, U, etc.)

No documentation

```
template <typename
           BusinessObjectT>

void encodeSomeStuff(
    BusinessObjectT iBusinessObject)
{
    // ...
}
```

## ... into self-documenting code

No more naked Ts

No more unconstrained  
typenamees

*T.10: Specify concepts for  
all template arguments*

```
template <typename
BOWithEncodeableStuff_t>
concept bo_with_encodeable_stuff =
requires (BOWithEncodeableStuff_t bo)
{
    bo.interfaceA();
    bo.interfaceB();
    { bo.interfaceC() } ->
        std::same_as<int>;
};

void encode(bo_with_encodeable_stuff
auto iBusinessObject) { /*...*/ }
```

# Even if some concepts are not for reuse

Concept can be “inlined”

Cannot use a parameter  
in an unnamed context

Hence the nested  
`requires` clause

```
template <typename
BOWithEncodeableStuff_t>
requires requires
(BOWithEncodeableStuff_t bo) {

    bo.interfaceA();
    bo.interfaceB();
    { bo.interfaceC() } ->
        std::same_as<int>;
}

void
encodeSomeStuff(BOWithEncodeableStuff_t
iBusinessObject) { /*...*/ }
```

# Conclusion

---

# Key takeaways

Concepts help validate template arguments at compile-time

Concepts provide a reusable and scalable way to constrain templates

The standard library gives dozens of generic concepts

There are plenty of ways to define our concepts

# Call to action

Start using concepts as soon as you switch to C++20

Use them for your applications

No more naked `T`s and `typename`s



# The Concepts of concepts

---

Sandor DARGO

Meeting C++

12th November 2021