Concepts: The Problem

```
class Account {
 public:
    Account() = default;
    Account(double bal): balance{bal} {}
 private:
    double balance{0.0};
};

template <typename T>
bool isSmaller(T t, T t2) {
    return t < t2;
}
```

account1.cpp

Concepts: The Concept `Smaller`

```
template <typename T>
concept Smaller = requires(T a, T b) {
    { a < b } -> std::convertible_to<bool>;
};


template <Smaller T>
bool isSmaller(T t, T t2) {
    return t < t2;
}
```

account2.cpp

Concepts: The Rescue

- Expresses requirements to the template parameters through the interface
- Generates better understandable error messages
- Can be used as placeholder for generic code
- Can be used for class templates, function templates, and member functions of class templates
- Supports function overloading and class template specialization

Concepts: The Concept `Smaller`

```
class Account {
 public:
    Account() = default;
    Account(double bal): balance{bal} {}
    bool operator < (const Account& oth) const {
        return balance < oth.balance;
    }
 private:
    double balance{0.0};
};
```

[account3.cpp]

Concepts: The Concept `Smaller`

```
class Account {
 public:
    Account() = default;
    Account(double bal): balance{bal} {}
    bool operator < (const Account& oth) const {
       return balance < oth.balance;
    }
 private:
    double balance{0.0};
};

template <typename T>
bool isGreater(T t, T t2) {
    return t > t2;
}
```

account4.cpp

Concepts: The Concept `Greater`

```
template <typename T>
concept Greater = requires(T a, T b) {
    { a > b } -> std::convertible_to<bool>;
};
class Account {
    ...
    bool operator > (const Account& oth) const {
       return balance > oth.balance;
    }
    ...
};
template <Greater T>
bool isGreater(T t, T t2) {
    return t > t2;
}
```

account5.cpp

Concepts

Concepts should model semantic categories but not syntactic constraints.

➡ The concepts `Smaller` and `Greater` model syntactic constraints.

- Haskells Typeclasses:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool

class Eq a => Ord a where
    compare :: a -> a -> Ordering
    (<) :: a -> a -> Bool
    (<=) :: a -> a -> Bool
    (>) :: a -> a -> Bool
    (>=) :: a -> a -> Bool
    max :: a -> a -> a
```

[account6.cpp]

- Concepts in C++

```
template<typename T>
concept Equal = requires(T a, T b) {
        { a == b } -> std::convertible_to<bool>;
        { a != b } -> std::convertible_to<bool>;
    };

template <typename T>
concept Ordering = Equal<T> && requires(T a, T b) {
        { a <= b } -> std::convertible_to<bool>;
        { a < b } -> std::convertible_to<bool>;
        { a > b } -> std::convertible_to<bool>;
        { a >= b } -> std::convertible_to<bool>;
    };
```

Concepts: Predefined Concepts

- Language related
  - same_as
  - derived_from
  - convertible_to
  - common_reference_with
  - common_with
  - assignable_from
  - swappable
- Arithmetic
  - integral
  - signed_integral
  - unsigned_integral
  - floating_point
- Compare
  - boolean
  - equality_comparable
  - totally_ordered
- Lifetime
  - destructible
  - constructible_from
  - default_constructible
  - move_constructible
  - copy_constructible
- Object
  - movable
  - copyable
  - semiregular
  - regular
- Callable
  - invocable
  - regular_invocable
  - predicate

Concepts: Application

- Requires clause

```
template<typename T>
requires Ordering<T>
T isSmaller(T a, T b);
```

- Trailing requires clause

```
template<typename T>
T isSmaller(T a, T b) requires Ordering<T>;
```

- Restricted template parameter

```
template<Ordering T>
T isSmaller(T a, T b);
```

- Abbreviated function templates syntax

```
Ordering auto isSmaller(Ordering auto a, Ordering auto b);
```

Concepts: Placeholders

Constrained placeholder (concepts) can be used when unconstrained placeholders (`auto`) are applicable.

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

std::integral auto getIntegral(int val){
    return val;
}
```

```
int main(){
    std::cout << std::boolalpha << '\n';

    std::vector<int> vec{1, 2, 3, 4, 5};
    for (std::integral auto i: vec) std::cout << i << " ";
    std::cout << '\n';

    std::integral auto b = true;
    std::cout << b << '\n';

    std::integral auto integ = getIntegral(10);
    std::cout << integ << '\n';

    auto integ1 = getIntegral(10);
    std::cout << integ1 << '\n';
}
```

placeholders.cpp

Concepts: Evolution or Revolution?

Evolution

- `auto` is a unconstrained placeholder
- Generic lambdas introduced a new syntax for defining templates

```
auto add = [](auto a, auto b) {
    return a + b;
}
```

Revolution

- Template requirements are checked by the compiler
- The declaration and definition of templates is significantly simplified.
- **Concepts define semantic categories and not syntactic constraints.**

Concepts

- [Modernes C++ Blog]
- [C++20: Get the Details]