ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

C++20 Concepts

TEMPLATES

Unconstrained

Constrained

```
class MyClass { /* just a dummy class */ };

int main()
{
  std::vector<int&> v;

  MyClass one, other;
  auto biggest = std::max(one, other);

  std::set<MyClass> objects;
  objects.insert(MyClass{});

  std::list numbers{ 4, 1, 3, 2 };
  std::sort(begin(numbers), end(numbers));
}
```

TEMPLATES

Unconstrained

Constrained

```cpp
class MyClass { /* just a dummy class */ };

int main()
{
  std::vector<int&> v;

  MyClass one, other;
  auto biggest = std::max(one, other);

  std::set<MyClass> objects;
  objects.insert(MyClass{});

  std::list numbers{ 4, 1, 3, 2 };
  std::sort(begin(numbers), end(numbers));
}
```

```
  required from '_IIter std::find(_IIter, _IIter, const _Tp&) [with _IIter = _
  required from here
error: no match for 'operator==' in '__first.__gnu_cxx::__normal_iterator<_Ite
note: candidates are:
note: template<class _T1, class _T2> bool std::operator==(const std::pair<_T1,
note:   template argument deduction/substitution failed:
note:   'std::vector<int>' is not derived from 'const std::pair<_T1, _T2>'
note: template<class _Iterator> bool std::operator==(const std::reverse_iterat
note:   template argument deduction/substitution failed:
note:   'std::vector<int>' is not derived from 'const std::reverse_iterator<_I
note: template<class _IteratorL, class _IteratorR> bool std::operator==(const
note:   template argument deduction/substitution failed:
note:   'std::vector<int>' is not derived from 'const std::reverse_iterator<_I
note: template<class _T1, class _T2> bool std::operator==(const std::allocato
note:   template argument deduction/substitution failed:
note:   'std::vector<int>' is not derived from 'const std::allocator<_T1>'
note: template<class _Tp> bool std::operator==(const std::allocator<_Tp1>&, co
note:   template argument deduction/substitution failed:
note:   'std::vector<int>' is not derived from 'const std::allocator<_Tp1>'
note: template<class _Tp, class _Alloc> bool std::operator==(const std::vecto
note:   template argument deduction/substitution failed:
note:   mismatched types 'const std::vector<_Tp, _Alloc>' and 'const int'
```

TEMPLATES

Unconstrained

```
class MyClass { /* just a dummy class */ };

int main()
{
  std::vector<int&> v;

  MyClass one, other;
  auto biggest = std::max(one, other);

  std::set<MyClass> objects;
  objects.insert(MyClass{});

  std::list numbers{ 4, 1, 3, 2 };
  std::sort(begin(numbers), end(numbers));
}
```

```
  required from '_IIter std::find(_IIter, _IIter, const _Tp&) [with _IIter = _
  required from here
error: no match for 'operator==' in '__first.__gnu_cxx::__normal_iterator<_Ite
note: candidates are:
note: template<class _T1, class _T2> bool std::operator==(const std::pair<_T1,
note:   template argument deduction/substitution failed:
note:   'std::vector<int>' is not derived from 'const std::pair<_T1, _T2>'
note: template<class _Iterator> bool std::operator==(const std::reverse_iterat
note:   template argument deduction/substitution failed:
note:   'std::vector<int>' is not derived from 'const std::reverse_iterator<_I
note: template<class _IteratorL, class _IteratorR> bool std::operator==(const
note:   template argument deduction/substitution failed:
note:   'std::vector<int>' is not derived from 'const std::reverse_iterator<_I
note: template<class _T1, class _T2> bool std::operator==(const std::allocator
note:   template argument deduction/substitution failed:
note:   'std::vector<int>' is not derived from 'const std::allocator<_T1>'
note: template<class _Tp> bool std::operator==(const std::allocator<_Tp1>&, co
note:   template argument deduction/substitution failed:
note:   'std::vector<int>' is not derived from 'const std::allocator<_Tp1>'
note: template<class _Tp, class _Alloc> bool std::operator==(const std::vecto
note:   template argument deduction/substitution failed:
note:   mismatched types 'const std::vector<_Tp, _Alloc>' and 'const int'
```

Constrained

- The signature specifies the template argument constraints.
- Template arguments type checking.
- A template is only instantiated if the template arguments satisfy all constraints.
- Error messages closer to the root cause of the problem.

SFINAE DISADVANTAGES

1. Hard to implement.
2. Template errors.
3. Readability.
4. Nested templates usually won't work in enable_if statements.

CONCEPTS - *A NAMED SET OF REQUIREMENTS*

Syntactic

```
i + n;
i += n;      Compiled
i[n];
...
```

Semantic

```
*(i + n)  ~  i[n]
i += -n   ~  i -= n
i += n    ~  ++i (n times)
...
```

Complexity

```
i += n
i + n          O(1) complexity
i -= n
...
```

i - random-access iterator;
n - integral value;

CONCEPT DEFINITION

Concept definition is a template for a named set of constraints.

```
template <parameter list>
concept name = constraints;
```

- Concepts are never instantiated by the compiler.
- The compiler evaluates at compile time.
- Parameter list can contain non-type parameters.

Constraints are logical expressions that consist of conjunctions (&&) and/or disjunctions (||) of constant bool expressions.

CONCEPT EXPRESSION

```
template <typename T>
concept Small = sizeof(T) <= sizeof(int);

Small<char> or Small<double> - concept expressions
```

REQUIRES EXPRESSIONS

```
template <typename Iter>
concept RandomAccessIterator = BidirectionalIterator<Iter>
   && /* Additional syntactical requirements for random-access
         iterators... */;
```

REQUIRES EXPRESSIONS

```
template <typename Iter>
concept RandomAccessIterator = BidirectionalIterator<Iter>
   && /* Additional syntactical requirements for random-access
         iterators... */;
```

```
requires { requirements }
requires (parameter list) { requirements }
```

typed variables

expressions with declared variables

REQUIREMENT TYPES

1. Simple requirements.
2. Compound requirements.
3. Type requirements.
4. Nested requirements.

SIMPLE REQUIREMENTS

```
template <typename Iter>
concept RandomAccessIterator = BidirectionalIterator<Iter>
   && requires (Iter i, Iter j, int n)
      {
         /* int v; Error: not an expression statement */
         i + n; i - n; n + i; i += n; i -= n; i[n];
         i < j; i > j; i <= j; i >= j;
      }
```

Global variables or variables introduced in the parameter list

SIMPLE REQUIREMENTS

```
template <typename Iter>
concept RandomAccessIterator = BidirectionalIterator<Iter>
   && requires (Iter i, Iter j, int n)
      {
         /* int v; Error: not an expression statement */
         i + n; i - n; n + i; i += n; i -= n; i[n];
         i < j; i > j; i <= j; i >= j;
      }
```

Disadvantages?

Global variables or variables introduced in the parameter list

SIMPLE REQUIREMENTS

```
template <typename Iter>
concept RandomAccessIterator = BidirectionalIterator<Iter>
   && requires (const Iter i, const Iter j, Iter k, const int n)
      {
         i + n; i - n; n + i; i[n];
         k += n; k -= n;
         i < j; i > j; i <= j; i >= j;
      }
```

COMPOUND REQUIREMENTS

```
{ expr };
{ expr } noexcept;
{ expr } -> type-constraint;
{ expr } noexcept -> type-constraint;
```

COMPOUND REQUIREMENTS

```
{ expr };
{ expr } noexcept;
{ expr } -> type-constraint;
{ expr } noexcept -> type-constraint;
```

```
template <typename Iter>
concept NoExceptDestructible = requires (T& value)
     {
        { value.~T() } noexcept;
     }
```

COMPOUND REQUIREMENTS

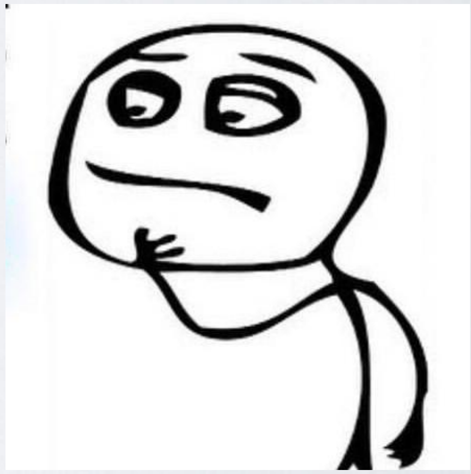
```
{ expr };
{ expr } noexcept;
{ expr } -> type-constraint;
{ expr } noexcept -> type-constraint;

template <typename Iter>
concept RandomAccessIterator = BidirectionalIterator<Iter>
  && requires (const Iter i, const Iter j, Iter k, const int n)
    {
      { i - n }    -> std::same_as<Iter>;
      { i + n }    -> std::same_as<Iter>;
      { k += n }   -> std::same_as<Iter&>;
      { i[n] }     -> std::same_as<decltype(*i)>;
      { i < j }    -> std::convertible_to<bool>;
      ...
    }
```

COMPOUND REQUIREMENTS

```
{ expr };
{ expr } noexcept;
{ expr } -> type-constraint;
{ expr } noexcept -> type-constraint;

template <typename Iter>
concept RandomAccessIterator = BidirectionalIterator<Iter>
  && requires (const Iter i, const Iter j, Iter k, const int n)
    {
      { i - n }     -> std::same_as<Iter>;
      { i + n }     -> std::same_as<Iter>;
      { k += n }    -> std::same_as<Iter&>;
      { i[n] }      -> std::same_as<decltype(*i)>;
      { i < j }     -> std::convertible_to<bool>;
      ...
    }
```



COMPOUND REQUIREMENTS

```
{ expr };
{ expr } noexcept;
{ expr } -> type-constraint;
{ expr } noexcept -> type-constraint;

template <typename Iter>
concept RandomAccessIterator = BidirectionalIterator<Iter>
   && requires (const Iter i, const Iter j, Iter k, const int n)
     {
        { i - n }    -> std::same_as<Iter>;
        { i + n }    -> std::same_as<Iter>;
        { k += n }   -> std::same_as<Iter&>;
        { i[n] }     -> std::same_as<decltype(*i)>;
        { i < j }    -> std::convertible_to<bool>;
        ...
     }
```

```
{ expr } -> concept<Args...>; => concept<decltype(expr), Args...>
```

COMPOUND REQUIREMENTS

```
{ expr };
{ expr } noexcept;
{ expr } -> type-constraint;
{ expr } noexcept -> type-constraint;
```

```
template <typename Iter>
concept RandomAccessIterator = BidirectionalIterator<Iter>
  && requires (const Iter i, const Iter j, Iter k, const int n)
    {
      { i - n } -> Iter; /*Error: Iter is a type, not a type constraint*/
      ...
    }
```

TYPE & NESTED REQUIREMENTS

```
typename name;          // name is a valid type name
requires constraints;   // same as in
                        // 'template <params> concept = constraints;'
```

TYPE & NESTED REQUIREMENTS

```
typename name;          // name is a valid type name
requires constraints; // same as in
                        // 'template <params> concept = constraints;'

template <typename S>
concept String = requires (S& s, const S& cs)
{
   typename S::value_type;
   requires Character<typename S::value_type>; // ~ predicates
   { cs.length() } -> std::integral;
   ...
}
```

REQUIRES EXPR IN REQUIRES EXPR

```
template <typename S>
concept String = requires (S& s, const S& cs)
{
   typename S::value_type;
   requires requires (typename S::value_type x) { ++x; }
   ...
}
```

REQUIRES CLAUSE

```
//By requires clause
template <typename Container>
   requires Sortable<Container>
void sort(Container& container);

//By requires clause
template <typename Container>
void sort(Container& container) requires Sortable<Container>;

//By concept
template <Sortable Container>
void sort(Container& container);
```

bool expressions

with && / ||

REQUIRES CLAUSE

```
template <typename T>
  requires !is_trivial_v<T>
void function(T param);
```

Not compiled

REQUIRES CLAUSE

```
template <typename T>
  requires (!is_trivial_v<T>)
void function(T param);
```

Compiled

SPECIALIZATION

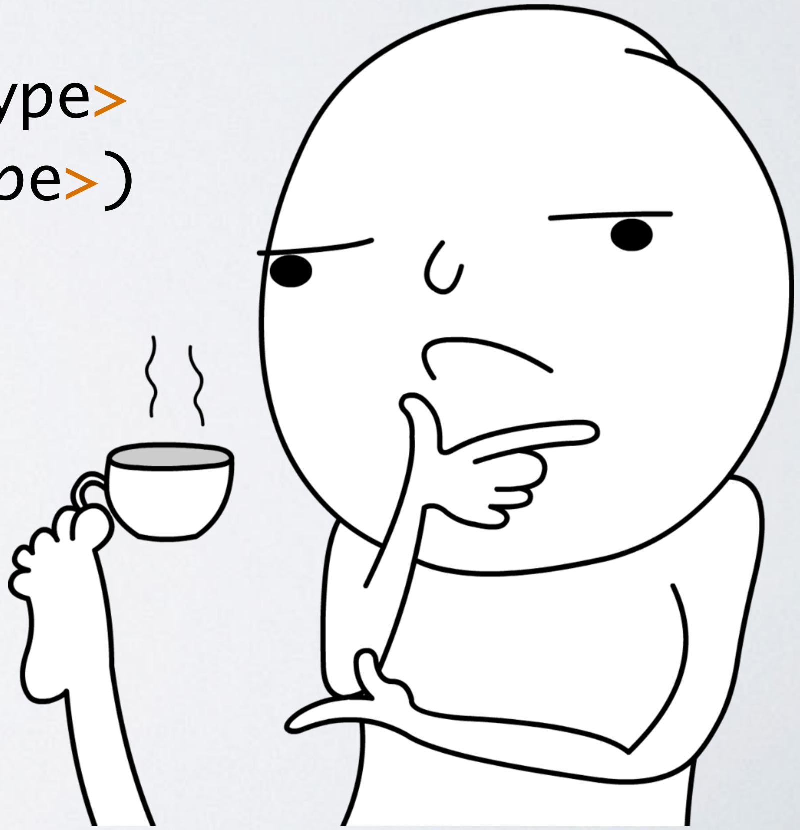
```
template <typename T>
  requires is_trivial_v<typename T::value_type>
void function(T param);

template <typename T>
void function(T param);
```

CONJUNCTION AND DISJUNCTION

```
template <typename T, typename U>
   requires is_trivial_v<typename T::value_type>
         ||  is_trivial_v<typename U::value_type>
void function(T param);

template <typename T, typename U>
   requires (is_trivial_v<typename T::value_type>
         ||  is_trivial_v<typename U::value_type>)
void function(T param);
```



КОНЕЦ ТРЕТЬЕЙ ЛЕКЦИИ