Jagiellonian University
Faculty of Mathematics and Computer Science
Theoretical Computer Science

Bachelor's Thesis

C++ Concepts - complete overview

Jakub Cisło
Album number:*1115689*

Supervisor:
dr hab. Marcin Kozik

Kraków, 2017

Contents

1 What are the concepts? 7

2 The very beginning (1994) 8
2.1 Constraints by inheritance . . . 8
2.2 Constraints by usage . . . 9

3 Texas Proposal (2003) 10
3.1 The base-class approach . . . 10
3.2 The function-match approach . . . 10
3.3 The usage-pattern approach . . . 10
3.3.1 Additional features . . . 11
3.3.2 Implicit modeling . . . 12
3.4 The pseudo-signature approach . . . 12

4 Concepts' design aims (2003) 12

5 Indiana Proposal (2005) 13
5.1 Syntax . . . 13
5.2 Additional features . . . 14
5.3 Explicit modeling . . . 15
5.3.1 Input and Forward Iterators issue . . . 16

6 Revised Texas Proposal (2005) 17
6.1 Negative assertions . . . 18

7 ConceptGCC and Revised Indiana Proposal (2005) 18

8 Compromise (2006) 19
8.1 Concept definition . . . 19
8.2 Constraining functions and classes . . . 19
8.3 Explicit and implicit modeling . . . 20
8.4 Axioms . . . 20

9 Simplification (2009) 21
9.1 Explicit refinement . . . 21
9.2 Voting . . . 21

10 Concepts Lite (2013) 22
10.1 Syntax . . . 22
10.2 Caller-site checking . . . 22
10.3 Additional features . . . 23
10.3.1 Concept-based overloading . . . 23
10.4 Logical operations in concepts . . . 24
10.4.1 Various syntax of constraining template arguments . . . 24
10.5 Usage-patterns again . . . 25
10.6 Implicit modeling again . . . 25

11 Concepts TS (2015) 26

12 C++17 (2016) **26**

13 Conclusions **26**
13.1 Project development 26
13.2 Future of the concepts 27

Introduction

An idea of constraining templates' arguments goes back to 1990s. Since that time the concepts were discussed and developed but they are still absent in the current standard. In this paper I would like to introduce the concepts, take a tour through the history and try to explain why they have not been fully implemented yet, despite their quite easy definition and valuable application.

Section 1 presents a concepts' main idea and a reason why they are desired. The history of the concepts starts in section 2 followed by the Texas Proposal (the first study concerning the concepts) (section 3). Main goals of the concepts are described in section 4. They are followed by the Indiana Proposal, section 5 (an alternative to the Texas Proposal). Two next sections 6 and 7 describe revisions of the Texas Proposal and the Indiana Proposal. After the revisions, the results of the meeting at Adobe Systems are presented in section 8. A suggested simplification of the design is the subject of section 9. Section 10 explains Concepts Lite. Section 11 is devoted to formalization of the Concepts Lite in Technical Specification. Section 12 concerns presence of the Concepts Lite in C++17 standard. Section 13 summarizes the history of the development and contains considerations about future of the concepts.

Enjoy your reading!

1 What are the concepts?

Templates are one of the most valuable facilities in C++. They provide a flexible approach to generic programming and do not imply any runtime overhead. The templates are used very often and became a crucial part of the language. However, they suffer from a number of problems. Let us think about generic function for minimum.

```
template <typename T>
T mymin(T a, T b) {
    return (a<b?a:b);
}
```

The function seems to be correct, unfortunately, there is a problem with this code. There is a silent assumption that type is comparable. If the function is called on a complex number:

```
#include <complex>

int main() {
    std::complex<float> a {1,2};
    std::complex<float> b {3,4};
    std::complex<float> c = mymin(a, b);
}
```

the following error message will be generated.[1]

```
In instantiation of 'T mymin(T, T) [with T = std::complex<float>]':
error: no match for 'operator<'
  (operand types are 'std::complex<float>' and 'std::complex<float>')
  return (a<b?a:b);
             ~^~
In file included
  from /usr/include/c++/6.3.1/bits/ios_base.h:46:0,
  from /usr/include/c++/6.3.1/ios:42,
  from /usr/include/c++/6.3.1/istream:38,
  from /usr/include/c++/6.3.1/sstream:38,
  from /usr/include/c++/6.3.1/complex:45,
  from code.cpp:6:
/usr/include/c++/6.3.1/system_error:274:3: note: candidate:
  bool std::operator<(const std::error_condition&, const std::error_condition&)
  operator<(const error_condition& __lhs,
  ^~~~~~~~

[120 lines more...]
```

The error is caused by an instantiation of a template with a type which does not provide the appropriate operator. The message is long and complicated because the compiler does not know the template requirements. The concepts are supposed, among other things, to fix the problem of intricate error messages.

```
template <typename T>
concept bool Comparable = requires(T a, T b) {
    {a<b} -> bool;
};

```

[1] This output was generated by GCC 7.1 without concepts support.

```
template <Comparable T>
T mymin(T a, T b) {
    return (a<b?a:b);
}
```

The `Comparable` concept requires the `<` operator which returns boolean-convertible value.[2] The function template `mymin()` expects a `Comparable` type as its template argument (notice `Comparable` instead of `typename`). As a result, the error message is more much easier to understand.[3]

```
In function 'int main()':
error: cannot call function 'T mymin(T, T) [with T = std::complex<float>]'
  std::complex<float> c = mymin(a, b);
                                ^

note: constraints not satisfied
  T mymin(T a, T b) {
    ^~~~~
note: within 'template<class T> concept const bool Comparable<T>
  [with T = std::complex<float>]'
  concept bool Comparable = requires(T a, T b) {
               ^~~~~~~~~~
note: with 'const std::complex<float> a'
note: with 'const std::complex<float> b'
note: the required expression '(a < b)' would be ill-formed
```

A concept is a set of constraints on the arguments of templates which are enforced by the compiler. This leads to shorter and more precise error messages, but additional functionality can be accomplished with this mechanism as well (to be described later).

2 The very beginning (1994)

In 1994 Bjarne Stroustrup in his book [Str94] introduced an idea of constraining templates. He proposed two approaches.

2.1 Constraints by inheritance

In this approach, requirements on the template parameters were expressed by inheritance.

```
template <typename T>
struct Comparable {
    virtual bool operator<(T) = 0;
};

template <typename T : Comparable>
T mymin(T a, T b) {
    return (a<b?a:b);
}

struct X : public Comparable<X> {
    bool operator<(X) override { return true; }
```

[2] From a design perspective, a comparable type should implement more operators (e.g., `>`, `<=`, `==`). They are skipped to keep examples short and simple. Similar simplifications will by applied in the whole document.

[3] This output was generated by GCC 7.1 with experimental concepts support (used `-fconcepts` switch).

```
};

int main() {
    X x = mymin(X(), X());
}
```

The approach reused the inheritance mechanism and there was only one new syntactic element to be introduced. This solution would be easy to implement, moreover a similar solutions are currently used in programming languages like C# and Java. The approach had three drawbacks. First, it mixed different levels of programming: concepts and abstract classes - they were undistinguishable. Second, built-in types were not in hierarchy of classes and needed a workaround. Last, the requirements on functions were not flexible enough: the functions required exactly the same signature. E.g, if a concept required a function with an argument passed by constant reference but a class implemented that function with the argument passed by value then the concept would not be satisfied.

2.2 Constraints by usage

The second approach was based on usage cases. It was possible to write a function assuring all requirements on the template parameter:

```
template <typename T>
class MyClass {
    void comparable_constraints(T a, T b) {
        bool r = a<b;
    }
};

struct X {
    bool operator<(X) { return true; }
};

int main() {
    MyClass<X> myClass;
}
```

If the `comparable_constraints()` function (*constraining function*) compiled correctly after instantiation then all the expressions in its body were supposed to be valid. Therefore, it was enough to write simple expressions with all the required operations in order to ensure that `T` satisfied the concept. In the example, the concept needs the `<` operator so the expression `bool r = a<b` is written.

In contrast to the constraints by inheritance, the constraints by usage were applicable only to class templates' arguments. Moreover, they required only one change in a compiler: compiling constraining functions before other functions. In case of failure, the process of compilation should be stopped and the appropriate message ought to be reported.

In the old versions of compilers this approach was almost working because they compiled all the functions inside a class template. The current versions of the compilers compile only the code which is necessary so the unused constraining functions are not compiled and the concepts are not checked.

3 Texas Proposal (2003)

The idea of constraining template continued evolving. In 2003 Bjarne Stroustrup published another paper [N1510] which presented four approaches.

3.1 The base-class approach

The base-class approach was reminiscent of inheritance-based solution (see 2.1). Stroustrup noticed that such a solution had to impose runtime overhead as the function calls had to be realized by virtual function tables (i.e. by a true inheritance). On the other hand, the syntax was very simple and easy to understand.

3.2 The function-match approach

The function-match approach was a refinement of the base-class approach. A concept required functions with the same signatures as stated but the calls would not be realized by virtual functions. The inheritance was omitted.

```
match Comparable {
    bool operator<(Comparable);
};

template <typename T match Comparable>
T mymin(T a, T b) {
    return (a<b?a:b);
}
```

However, there was still the same problem: very strict requirements. A free-standing function and a inside-class function, arguments passed by value or by (constant) reference - all of them could express almost the same but only one of them satisfied the concept.

3.3 The usage-pattern approach

The usage-pattern approach, later called the Texas Proposal, was very similar to the constraints by usage (see 2.2). The author suggested the following syntax.

```
concept Comparable {
    constraints(Comparable a, Comparable b) {
        bool r = a<b;
    }
};

template <Comparable T>
T mymin(T a, T b) {
    return (a<b?a:b);
}
```

The constraining clause consisted of the `constraints` keyword followed by parameter list and the body. The body contained expressions: requirements on the constrained type.

The `constraints` clause was similar to a function definition. However, it did not need any constructor for its arguments itself. Its parameter list introduced new variables of given types,

in contrast to a parameter list of a normal function which required copy/move constructors of the arguments' types.

3.3.1 Additional features

The Texas Proposal included some additional features.

Concept parametrization: In order to express a concept for more compound types, the parametrized concepts were suggested. For example, `std::vector` and `std::list` could be treated as collections of elements of the same type for each type. The concept for this purpose might be created as follows.

```
template <typename Item>
concept Collection {
    constraints(Collection<Item> c, Item item) {
        c.push_back(item);
        c.pop_back();
        //...
    }
};
```

Concept inheritance: A derived concept could be created with the inheritance syntax. Then the derived concept had the requirements from the base concept as well as additional ones.

```
concept EqualityComparable {
    constraints(EqualityComparable a, EqualityComparable b) {
        bool r = a==b;
    }
};

concept Comparable : EqualityComparable {
    constraints(Comparable a, Comparable b) {
        bool r = a<b;
    }
};
```

Concept composition: To avoid unnecessary definitions of concepts, the composition syntax was proposed.

```
template <(C1 && C2) T> class A1 {};
template <(C1 || C2) T> class A2 {};
template <(C1 && !C2) T> class A3 {};
```

Concept-based overloading: For a few templates with the same name, the proper one was chosen based on the satisfied concepts on the call site. Due to concept-based overloading, optimizations and special cases without runtime if-conditions were possible.

```
template<RandomAccessIterator Iter>
void advance(Iter iter, int n) { /* O(1) algorithm */ }

template<Iterator Iter>
void advance(Iter iter, int n) { /* O(n) algorithm */ }
```

```
6
7  int main() {
8      std::list<int> l {1,2,3,4};
9      std::vector<int> v {1,2,3,4};
10     auto lit = l.begin();
11     auto vit = v.begin();
12     advance(lit, 2); // uses advance() with Iterator
13     advance(vit, 2); // uses advance() with RandomAccessIterator
14 }
```

3.3.2 Implicit modeling

Implicit modeling was a vital aspect of the Texas Proposal (in contrast to further proposal's explicit modeling): every class which had functions required by a concept just satisfied that concept. No special declarations were necessary, classes were automatically matched to concepts.

3.4 The pseudo-signature approach

At the end of the document, Stroustrup briefly presented the pseudo-signature approach. It expressed the same set of requirements as the usage-pattern approach but differed in syntax. An example of pseudo-signatures is shown in the listing below.

```
1  concept Comparable {
2      <(Comparable, Comparable) -> bool
3  };
4
5  concept Swappable {
6      swap(Swappable, Swappable) -> void
7  };
```

A pseudo-signature concept was satisfied by functions having various signatures. For instance, a class with a `<` operator taking its arguments by value and a class with the operator taking its arguments by reference would satisfy the `Comparable` concept. The pseudo-signature approach was not carefully analyzed in Stroustrup's document.

4 Concepts' design aims (2003)

Another document [N1522] was published the same day as [N1510]. The authors, Stroustrup and Dos Reis, attempted to approach the concept problem from a different angle. They considered the aims of the concepts' design.

Flexibility: In object oriented system, an exact agreement between creator of a function and user of that function is necessary - they have to choose common interface and specify the operations. Introducing the concepts was supposed to amend the templates' duck typing by a formalization. However, that formalization should not be as strict as in OOP model.

Modular type checking: The definition of a template should not rely on actual substituted types but use the concept declaration instead. Similarly the usage of the template ought to depend only on the concept.

More precise error messages: Before the concepts incorrect instantiation's error messages were complicated and unclear if the problem was deeply nested. The concepts were supposed to fix that. Improper use of a constrained template should be identified by the template argument failing the concept, on the other hand, the definition of the template was supposed to be checked: the template was allowed to use only the operations guaranteed by the concept.

Concept-based overloading: The concepts had to allow overloading. Among matching templates one could be chosen based on the concepts satisfied by a type argument.

No run-time overhead: Performance of the constrained templates had to be comparable to unconstrained ones.

Simplicity in implementation: It was difficult to implement the first templates in compilers. The concepts' implementation were expected to not be as problematic as the templates' one.

Backward compatibility: The concepts' design had to introduce some new keywords and constructions. However, it was supposed to generate a small amount of backward conflicts.

Separate compilation: The unconstrained templates generated code after substituting actual types so it was impossible to compile template code without providing the types. The concepts gave hope that it would be possible to compile constrained templates independently of their usage.

Expressing the requirements: The Standard Template Library's documentation contained the requirements on the arguments of the templates. The concepts were expected to be a formalized way of expressing those requirements (both syntactic and semantic).

These were the main goals for the concepts. It was predictable that achieving all of them simultaneously was impossible. A conflict between the separate compilation and the concept-based overloading, which appeared during the studies, confirmed that.

5 Indiana Proposal (2005)

In 2005 other research group consisting of Jeremy Siek, Douglas Gregor, Ronald Garcia1, Jeremiah Willcock, Jaakko Järvi and Andrew Lumsdaine published [N1758]. They proposed an extension to the pseudo-signature approach by Stroustrup and called it the Indiana Proposal.

5.1 Syntax

An example of the syntax is presented below.

```
1  template <typeid T>
2  concept Comparable {
3      bool operator<(T, T);
4  };
5
6  template <typeid T>
7      where {Comparable<T>}
8  T mymin(T a, T b) {
9      return (a<b?a:b);
10 }
```

The approach was similar to the function-match but less strict.

"In a simple signatures approach, a type `T` would have to have functions that match those signatures exactly. A pseudo-signature approach, on the other hand, treats these declarations more loosely. For instance, the declaration of operator < requires the existence of a < operator, either built in, as a free function, or as a member function, that can be passed two values of type `T` and returns a value convertible to `bool`." ([N1758])

"The pseudo-signature may be satisfied by a function in the enclosing scope of the model declaration. Function lookup is performed as for a function call expression whose function and arguments are given by the pseudo-signature. A forwarding function whose signature exactly matches the pseudo-signature is generated by the C++ implementation, and the body of this function consists of a function call to the result of the function lookup. The return type of the found function shall be convertible to the return type of the pseudo-signature, otherwise a diagnostic is required." ([N1758])

It is worth noting that a template parameter was declared by the *typeid* keyword. The authors suggested such reuse of this keyword to express the difference between constrained and unconstrained parameters.

5.2 Additional features

Some additional features were introduced as well:

Concept inheritance (called concept refinement): Inheritance of concepts was possible. A derived concept inherited requirements from the base concept.

```
template <typeid T>
concept EqualityComparable {
    bool operator==(T, T);
};

template <typeid T>
concept Comparable : EqualityComparable<T> {
    bool operator<(T, T);
};
```

Concept-based overloading: In this proposal, concept-based overloading was allowed too. While considering the overloads, only templates with concepts satisfied by the type parameter were taken into account. Among them, a template with the most specific (in terms of the inheritance) concepts was chosen.[4]

```
template<typeid Iter> where {RandomAccessIterator<Iter>}
void advance(Iter iter, int n) { /* O(1) algorithm */ }

template<typeid Iter> where {Iterator<Iter>}
void advance(Iter iter, int n) { /* O(n) algorithm */ }

int main() {
    std::list<int> l {1,2,3,4};
    std::vector<int> v {1,2,3,4};
    auto lit = l.begin();
```

[4]In the example, it was assumed that modeling clauses were part of the STL and they were included from the library. See 5.3 for more information about the modeling syntax.

```
11      auto vit = v.begin();
12      advance(lit, 2); // uses advance() with Iterator
13      advance(vit, 2); // uses advance() with RandomAccessIterator
14  }
```

Associated types: Concepts could require type definitions inside classes. For example, iterators were supposed to have type of the value defined. An example below illustrates this.

```
1  template <typeid T>
2  concept Iterator {
3      typename value_type;
4      //...
5  };
```

Function definition inside a concept: A concept could define other functions based on provided ones. For instance, the `EqualityComparable` concept could define the `!=` operator based on the `==` operator as follows.

```
1  template <typeid T>
2  concept EqualityComparable {
3      bool operator==(T, T);
4      bool operator!=(T a, T b) { return !(a==b); }
5  };
```

If a class modeling the `EqualityComparable` concept contained the `!=` operator then its implementation was taken, otherwise the concept's one was used. This was supposed to decrease the amount of code written to satisfied.

5.3 Explicit modeling

In contrast to the Texas Proposal, the Indiana Proposal assumed explicit modeling. Every type (even built-in) needed explicit statement in order to assure that it satisfied a concept. The explicit modeling was declared using `model` keyword.

```
1  struct X {
2      bool operator<(X) { return true; }
3  };
4
5  model Comparable<X> { };
```

When a compiler encountered the `model` statement, it checked if the type had all the operations required by the concept. If the check was successful then the type could be used wherever the concept constrained the argument. Otherwise, a precise error message was generated.

A more compound example was modeling with a function definition. Even if a class did not have a proper function, it could be added using `model` syntax in order to satisfy a concept. Then, the function was treated as it was declared inside a class.

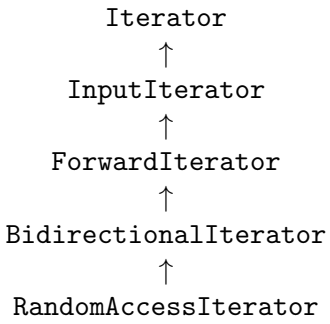
```
1  struct Y {
2      int v;
3  }
4
```

```
model Comparable<Y> {
    bool operator<(Y y) { return v < y.v; }
};
```

The reason for explicit modeling was a mismatch issue described in the next section.

5.3.1 Input and Forward Iterators issue

Iterators from hierarchy of concepts are presented in the diagram. Each concept is implemented as a refinement of its predecessor.



Concepts `InputIterator` and `ForwardIterator` offer the same set of operations (from syntactical point of view)[5] but `InputIterator` allows to view the sequence only once. Therefore, in implicit modeling every type satisfies either both concepts or none of them. Let us consider a template function overloaded for the both concepts. The overload for `ForwardIterator` will always be chosen over the overload for `InputIterator`, because the `ForwardIterator` concept is more specific. That may lead to runtime errors. It was Texas Proposal's issue.

```
concept InputIterator { /* ... */ };
concept ForwardIterator : InputIterator { /* ... */ };

template <InputIterator Iter>
void f(Iter iter) { /* ... */ }

template <ForwardIterator Iter>
void f(Iter iter) { /* ... */ }

template <InputIterator Iter>
void g(Iter iter) {
    //...
    f(iter);
    //...
}

class MyForwardIterator { /* ... */ }
class MyInputIterator { /* ... */ }

```

[5] A copy constructor could be considered as a difference between those iterators - `InputIterator` should not have been copied (moving was more suitable operation). However, because of historical reasons, `InputIterator` have to provide the copy constructor.

```
int main() {
    MyForwardIterator forwardIterator;
    MyInputIterator inputIterator;
    g(forwardIterator); // uses f() with ForwardIterator
    g(inputIterator); // uses f() with ForwardIterator too!
}
```

The explicit modeling and the `model` syntax were solution to the mismatch problem.

```
template <typeid Iter>
concept InputIterator { /* ... */ };

template< typeid Iter>
concept ForwardIterator : InputIterator<Iter> { /* ... */ };

template <typeid Iter>
    where {InputIterator<Iter>}
void f(Iter iter) { /* ... */ }

template <typeid Iter>
    where {ForwardIterator<Iter>}
void f(Iter iter) { /* ... */ }

template <typeid Iter>
    where {InputIterator<Iter>}
void g(Iter iter) {
    //...
    f(iter);
    //...
}

class MyForwardIterator { /* ... */ }
class MyInputIterator { /* ... */ }

model ForwardItertor<MyForwardIterator> {}
model InputItertor<MyInputIterator> {}

int main() {
    MyForwardIterator forwardIterator;
    MyInputIterator inputIterator;
    g(forwardIterator); // uses f() with ForwardIterator
    g(inputIterator); // uses f() with InputIterator
}
```

6 Revised Texas Proposal (2005)

After publication of the Indiana Proposal, the Texas proposal was revised [N1782]. An example from [N1782] illustrates the changes.

```
concept ForwardIterator<class Iter> {
    Iter p;                          // uninitialized
    Iter q =p;                       // copy initialization
    p = q;                           // assignment
    Iter& q = ++p;                   // can pre-increment,
                                     // result usable as an Iter&

    const Iter& cq = p++;            // can post-increment,
                                     // result convertible to Iter
    bool b1 = (p==q);                // equality comparisons,
                                     // result convertible to bool
    bool b2 = (p!=q);
    Value_type Iter::value_type;     // Iter has a member type,
                                     // value_type which is
                                     // a Value_type
    Iter::value_type v = *p;         // *p is assignable to
                                     // Iter's value type
    *p = v;                          // Iter's value type is
                                     // assignable to *p
};
```

"Here, `Iter p;` introduces the name `p` of type `Iter` for us to use when expressing concept rules. It does not state that an `Iter` require a default constructor. That would be expressed as `Iter();` or `Iter p = Iter();`" ([N1782])

6.1 Negative assertions

In order to solve the mismatch problem (see 5.3.1), the Texas team suggested so called negative assertions. The syntax was as follows.

```
static_assert !ForwardIterator<MyIterator>;
```

This statement meant that `MyIterator` could not be matched to the `ForwardIterator` concept. Such statements were sufficient for problematic classes and concepts.

7 ConceptGCC and Revised Indiana Proposal (2005)

In August of 2005 two documents were published: [N1848] and [N1849]. The first one was a description of ConceptGCC. ConceptGCC was a branch of GCC compiler with support for the concepts based on Indiana Proposal from [N1849]. This prototype was significant for the Indiana Proposal because it proved that the proposal was feasible.

The revision of the Indiana Proposal removed the `typeid` keyword in context of the concepts. It was replaced back with the standard `typename`. The concept definition and usage looked as in the listing.

```
template <typename T>
concept Comparable {
    bool operator<(T, T);
};

```

```
template <typename T>
    where {Comparable<T>}
T mymin(T a, T b) {
    return (a<b?a:b);
}
```

8 Compromise (2006)

In 2006 the teams responsible for Texas and Indiana proposals met at Adobe Systems and a compromise was reached ([Sie10]) Soon after the meeting Indiana and Texas teams created a common design document [Gre+06] and published proposals [N2042; N2081] which were well received by the WG21 committee[6]. The main points from [N2042] are presented below.

8.1 Concept definition

Both usage-patterns and pseudo-signatures were equivalent in semantics; all the requirements could be translated from the first approach to the second and vice versa. Therefore, other aspects were considered.

Despite the fact that the usage-pattern approach was very similar to the concepts requirement in documentation, the pseudo-signature approach was chosen. The rationale behind that was the similarity to required functions from classes and consistency with the modeling syntax (in a `model` clause).

Another advantage was the ease of creating archetypes. An archetype was a minimal class which provided the required functions, operations and members. The archetypes were a part of the proposal intended for checking the constrained templates' definitions.

The code below illustrates the concept definition after the compromise:

```
concept Comparable<typename T> {
    bool operator<(T, T);
};
```

8.2 Constraining functions and classes

The teams allowed both methods of constraining the templates arguments.

```
template <Comparable T>
T mymin(T a, T b) {
    return (a<b?a:b);
}
```

```
template <typename T>
    where {Comparable<T>}
T mymin(T a, T b) {
    return (a<b?a:b);
}
```

[6] WG21 is a ISO C++ committee, a group of experts making decisions about C++ language.

The first was shorter and suitable when a class had simple constraints. The latter one could express more complicated requirements, even with multitype constraints. Additionally, the logical operators `&&`, `||` and `!` could be used inside the `where` clause.

8.3 Explicit and implicit modeling

The choice between explicit and implicit modeling proved to be a problematic issue. Explicit modeling was solving the mismatch issue (see 5.3.1), on the other hand, it increased complexity of simple programs - they would need a lot of modeling statements which would make it more obfuscated. The solution was to divide concepts into two categories: default concepts (explicitly modeled) and *auto-concepts* (implicitly modeled).

The `model` keyword was also changed into `concept_map` to decrease conflicts in existing codes.

Explicit modeling

```
concept Comparable<typename T> {
    bool operator<(T, T);
};

struct X {
    bool operator<(X) { return true; }
};

concept_map Comparable<X> { };
```

Implicit modeling

```
auto concept Comparable<typename T> {
    bool operator<(T, T);
};

struct X {
    bool operator<(X) { return true; }
};
```

8.4 Axioms

Axioms appeared in the design. It was a first attempt to express properties of a type, e.g., associativity of an operation or transitivity of a relation. They were not checked by the compiler but they might have influence on optimizations - the compiler could use them and substitute one expression by the other. A short example of an axiom is presented below.

```
concept Semigroup<typename Op, typename T> {
    T operator()(Op, T, T);
    axiom Associativity(Op op, T x, T y, T z) {
        op(x, op(y, z)) == op(op(x, y), z);
    }
```

```
};
```

Then any occurrence of `op(x, op(y, z))` on types satisfying the `Semigroup` concept could be replaced by a compiler with `op(op(x, y), z)`.

9 Simplification (2009)

In 2009 Bjarne Stroustrup wrote a paper [N2906] where he summarized worries of the WG21 committee members. They were concerned that the concepts were to complicated to an average C++ programmer. Stroustrup urged to simplify the design.

9.1 Explicit refinement

Bjarne Stroustrup recommended to remove explicit concepts and to introduce *explicit refinement* into implicit modeling instead.

```
concept ForwardIterator<typename Iter>
    : explicit InputIterator<Iter> { /* ... */ };
```

The syntax above expressed that despite the fact the type `T` satisfied the more specific concept (`ForwardIterator`), `T` could not be generally treated as `ForwardIterator`. That allowed to create a hierarchy of auto-concepts and to turn off the automatic match for specific concepts in the hierarchy. The explicit refinement was another solution for mismatch problem (see 5.3.1).

In order to not lose the optimizations for types which were actually `ForwardIterator`, the following code was proposed.

```
concept_map ForwardIterator<MyForwardIterator> { };
```

Introducing the explicit refinement and decreased number of `concept_map`s as its result were expected to make the concepts more available for an average programmer.

9.2 Voting

At Frankfurt meeting in July of 2009, a month after issuing [N2906], the voting took place. Stroustrup gave 3 alternatives for the concepts:

1. issue the concepts with the current specification,
2. implement the fixes from [N2906] and issue the concepts in C++0x standard,
3. remove the concepts from C++0x.

The WG21 committee noticed drawbacks of the current design and the votes split between the second and the third option. However, the majority chose safer way: removing the concepts from the current standard. The committee members were anxious about the time plan. C++0x standard was planned in the first decade of the new millennium but it was already 2009 and fixing the concepts could delay the issue date more. The Stroustrup's fixes were not straightforward refinements too. Moreover, there were some worries about the efficiency of the ConceptGCC - this implementation was slow. Finally, the committee decided to postpone the concepts for the next standard. The members were disappointed but they preferred to deliver a high-quality solution ([Str09; Sie12]).

10 Concepts Lite (2013)

After an unsuccessful adoption of concepts into C++11 standard, radical steps were taken. Not only the simplification of the design was desired but the path of the development was changed. Being aware of the difficulty of introducing such a complex feature into the language, Stroustrup and the WG21 committee split the concept design and focused on the first part - *template constraints* also called *Concepts Lite* ([N3576; N3580; N3701]).

10.1 Syntax

```
template<typename T>
concept bool Comparable() {
    return requires (T a, T b) {
        {a < b} -> bool;
    };
}

template <Comparable T>
T mymin(T a, T b) {
    return (a<b?a:b);
}
```

A concept definition was changed. It was equivalent to constant expression function template definition. The `concept` keyword had the same meaning as `constexpr` but with additional compile-time checks, e.g., the function was supposed to return `bool` and to have no functional parameters.

A new clause appeared. The `requires` clause might introduce a new variables and it checked if the statements inside its body compiled and returned values which could be converted to the expected types. In the example above: if the comparison with operator `<` was implemented and if it returned a boolean-convertible value. If the check was successful then the `requires` clause returned `true`, otherwise `false`.

10.2 Caller-site checking

The approach was called “lite” becuase it no longer verified the template definition. For instance, a template function could require a type satisfying the concept `C` but inside the definition it might use a syntax not allowed by `C`. It was a step back in the design but made concepts much easier to implement and therefore, more likely to appear sooner in the standard.

```
template<typename T>
concept bool Comparable() {
    return requires (T a, T b) {
        {a < b} -> bool;
    };
}

template <Comparable T>
T mymin(T a, T b) {
    return (a<b?a:b);
```

```
}

template <Comparable T>
T mymax(T a, T b) {
    return (a>b?a:b); //not provided by Comparable
}

struct X {
    bool operator<(X) { return true; }
    bool operator>(X) { return false; }
};

struct Y {
    bool operator<(Y) { return true; }
};

struct Z {
    bool operator>(Z) { return false; }
};

int main() {
    X x1, x2;
    mymin(x1, x2);
    mymax(x1, x2);
    Y y1, y2;
    mymin(y1, y2);
    //mymax(y1, y2); //error from inside the function
    Z z1, z2;
    //mymin(z1, z2); //error: constraint not satisfied
    //mymax(z1, z2); //error: constraint not satisfied
}
```

10.3 Additional features

The Concepts Lite proposal included additional features.

10.3.1 Concept-based overloading

The concept-based overloading was present in the design.

```
template<RandomAccessIterator Iter>
void advance(Iter iter, int n) { /* O(1) algorithm */ }

template<Iterator Iter>
void advance(Iter iter, int n) { /* O(n) algorithm */ }

int main() {
    std::list<int> l {1,2,3,4};
    std::vector<int> v {1,2,3,4};
```

```
    auto lit = l.begin();
    auto vit = v.begin();
    advance(lit, 2); // uses advance() with Iterator
    advance(vit, 2); // uses advance() with RandomAccessIterator
}
```

10.4 Logical operations in concepts

A concept definition could consist of logical operations.

```
template<typename T>
concept bool C1() { /* ... */ }
template<typename T>
concept bool C2() { /* ... */ }

template <typename T>
concept bool C3() {
    return (C1<T>() && C2<T>()) || requires(T a) { /* ... */};
]
```

10.4.1 Various syntax of constraining template arguments

To ease and increase speed of writing constrained functions and classes, the following facilities were suggested.

Logical operations: Logical expressions on concepts could be a requirement.

```
template<typename T>
concept bool C1() { /* ... */ }
template<typename T>
concept bool C2() { /* ... */ }

template <typename T>
    requires C1<T>() && C2<T>()
void f(T);
```

Concepts as type introducers: A concept's name could introduce the types it constrained. This was useful especially with multitype concepts.

```
template<typename T, typename S>
concept bool Convertible() {
    return requires (T a) {
        {a} -> S;
    };
}

Convertible{InT, OutT}
OutT convert(InT input) {
    return input;
}
```

Terse notation: The following syntax introduces a concept and a template function. The function `sort` is a template function because its argument's type is a concept.

```
template<typename T>
concept bool Container() { /* ... */ }

void sort(Container& container);
```

Terse notation in lambdas functions: The same notation could be used in lambdas functions.

```
template<typename T> concept bool Number() { /* ... */ }

[](Number n) { return 2*n + 3; }
```

10.5 Usage-patterns again

The approach was changed and the usage-patterns appeared instead of the pseudo-signatures but it is not quite clear why. One of the advantages might be an ease of conversion the requirements from the documentation of Standard Template Library. The statements there had been written as usage examples so it was not difficult to convert text into that syntax. Moreover, usage patterns were more abstract than pseudo-signatures (they were focused on *what* was provided, not *how*) and they encouraged a programmer to write more general code ([N3351]).

10.6 Implicit modeling again

Implicit modeling appeared again. However, there were no `concept_maps`. Thus, the mismatch issue (see 5.3.1) had another solution. The current one was based on *iterator tags*.

The `std::iterator_traits` template could be specialized for any new iterator type. The iterators from the STL had their own specializations provided. The trait had `iterator_category` typedef. This typedef indicated the kind of the iterator. The base kinds were represented by the following tags (which were just empty classes):

- `std::input_iterator_tag`,
- `std::output_iterator_tag`,
- `std::forward_iterator_tag`,
- `std::bidirectional_iterator_tag`,
- `std::random_access_iterator_tag`.

Having such tags, the concepts could require the proper tag to be defined beside the other requirements. This solution was a workaround similar to the `concept_maps` but no new syntax was necessary - the unconstrained templates and their specializations were sufficient.

11 Concepts TS (2015)

The aim of C++14 was to complete C++11 features and to fix existing bugs. There was not enough time to introduce new facilities too. The committee decided to prepare a separate document - Technical Specification ([SW14]).

The Technical Specification documents were invented to speed up the process of introducing new features into the language. Each Technical Specification could have individual pace of development, have a separate group of contributors and finally it could be included into the standard.

The work on the Concepts TS started in 2013 and in 2015 the final draft [N4553] was published. A month after that the official ISO document [ISOTS] was issued. The concepts introduced into GCC version 6.1 were based on this document (compilation required the `-fconcepts` switch).

12 C++17 (2016)

It was expected that the Concepts Lite were included into C++17. Sutton in [P0248R0] and Voutilainen in *Why I want Concepts, and why I want them sooner rather than later* [P0225R0] argued for including them. On the other hand, there were votes against it, e.g. *Why I want Concepts, but why they should come later rather than sooner* [P0240R0]. The arguments of the opposite sides were as follows.

It was undeniable that constrained templates (even with restricted version without template definition checking) offered what the programmers expected: better error messages, documentation, notation and overloading. Some experiments were carried in various projects and the concepts passed them. Only a few issues reminded. Also presentations in academic environment gave positive feedback. Moreover, the programmers had been waiting for the concepts so long and the lack of the concepts could lead to development of various independent libraries for concepts-like techniques.

The arguments against included the lack of conceptualized standard library. It was difficult to write high-quality concepts without base concepts in the library. Even experts from team responsible for new STL had problems with creating reliable hierarchy of concepts. Moreover, Concepts Lite were to be the first part of the whole concepts idea and the second part could require significant changes in the design of the first one. In addition ([Hon16]) it was short time since the [ISOTS] was published, there was only one implementation of the concepts in GCC written by one of the authors of the Indiana Proposal and there were still some problems with the syntax.

Finally, at the meeting in Jacksonville in 2016, the committee decided to postpone the concepts introduction again. The concepts issue is still open.

13 Conclusions

In this paper I presented the history of the concepts, the decisions which were made and the rationale behind them. Based on these considerations, several conclusions can be drawn.

13.1 Project development

The main idea of the concepts was very clear and easy: an ability to constrain template arguments was desired. However, while the idea was being developed, various issues occurred.

Even after splitting into parts and focusing on the Concepts Lite, there were still problems. This shows that designers have to be always aware of possible difficulties in their projects.

Another aspect is the responsibility for the design. It is unwise to create or modify the design without considering the effects. On the other hand, it is very hard to foresee all the consequences of the decisions. The risk depend also on a project category. In a small project it is easier to accept the cost of a wrong decision. In a complex and one, introducing improper idea may lead to irreversible consequences. The concepts' designers did their best to avoid such catastrophic decisions.

The variety of the points of view is also important. On each step of the development of the concepts there was at least two alternatives. This broadened the perspective and improved the proposals (e.g., Texas and Indiana Proposals created the compromised proposal). The lack of diversity was strong reason to postpone the concepts inclusion into standard (e.g., only GCC implemented the concepts, see 12). Thus, it is valuable to confront and discuss different approaches and to extensively test the solutions.

13.2 Future of the concepts

In my opinion Concepts Lite will be a part of the C++20 standard. A long time of planning and the number of examined scenarios imply high quality of the design.

A conflict between usage-patterns and pseudo-signatures (which was one of the objections) has been discussed very deeply by the WG21 committee. Advantages and disadvantages of both approaches are known very well.

The implementation has existed since April of 2016 and was tested in various environments. Microsoft and Clang team have also begun work on Concepts Lite in their compilers.

Additionally, the evolution of the Concepts Technical Specification has not been stopped and there are new voices that concepts are ready ([P0606R0]).

However, in my opinion, the full concepts will not be available soon. The issue of checking constrained template definition remains unresolved; integration with Concepts Lite can cause further problems and additional yet undiscovered issues can delay the final release of concepts. Nevertheless, I consider the concepts an important feature and hope they will be available rather sooner than later.

References

[Gre+06] Douglas Gregor et al. "Concepts: Linguistic Support for Generic Programming in C++". In: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA. 2006.

[Hon16] Tom Honermann. *Why Concepts didn't make C++17*. Tom Honermann's blog. 2016. URL: `http://honermann.net/blog/2016/03/06/why-concepts-didnt-make-cxx17/`.

[ISOTS] *Information technology – Programming languages – C++ Extensions for concepts*. Standard ISO/IEC TS 19217:2015. International Organization for Standardization, 2015.

[N1510] Bjarne Stroustrup. *Concept checking – A more abstract complement to type checking*. Tech. rep. N1510. WG21, 2003.

[N1522] Bjarne Stroustrup and Gabriel Dos Reis. *Concepts – Design choices for template argument checking*. Tech. rep. N1522. WG21, 2003.

[N1758] Jeremy Siek et al. *Concepts for C++0x*. Tech. rep. N1758. WG21, 2005.

[N1782] Bjarne Stroustrup and Douglas Gregor. *A concept design (Rev. 1)*. Tech. rep. N1782. WG21, 2005.

[N1848] Douglas Gregor and Jeremy Siek. *Implementing Concepts*. Tech. rep. N1848. WG21, 2005.

[N1849] Jeremy Siek et al. *Concepts for C++0x. Revision 1*. Tech. rep. N1849. WG21, 2005.

[N2042] Douglas Gregor and Bjarne Stroustrup. *Concepts*. Tech. rep. N2042. WG21, 2006.

[N2081] Douglas Gregor and Bjarne Stroustrup. *Concepts (Revision 1)*. Tech. rep. N2081. WG21, 2006.

[N2906] Bjarne Stroustrup. *Simplifying the use of concepts*. Tech. rep. N2906. WG21, 2009.

[N3351] Bjarne Stroustrup and Andrew Sutton. *A Concept Design for the STL*. Tech. rep. N3351. WG21, 2012.

[N3576] Herb Sutter. *SG8 Concepts Teleconference Minutes*. Tech. rep. N3576. WG21, 2013.

[N3580] Andrew Sutton, Bjarne Stroustrup, and Gabriel Dos Reis. *Concepts Lite: Constraining Templates with Predicates*. Tech. rep. N3580. WG21, 2013.

[N3701] Andrew Sutton, Bjarne Stroustrup, and Gabriel Dos Reis. *Concepts Lite*. Tech. rep. N3701. WG21, 2013.

[N4553] Andrew Sutton. *Working Draft, C++ extensions for Concepts*. Tech. rep. N4553. WG21, 2015.

[N4641] Andrew Sutton. *Working Draft, C++ extensions for Concepts*. Tech. rep. N4641. WG21, 2017.

[P0225R0] Ville Voutilainen. *Why I want Concepts, and why I want them sooner rather than later*. Tech. rep. P0225R0. WG21, 2016.

[P0240R0] Matt Calabrese. *Why I want Concepts, but why they should come later rather than sooner*. Tech. rep. P0240R0. WG21, 2016.

[P0248R0] Andrew Sutton. *Concepts in C++17*. Tech. rep. P0248R0. WG21, 2016.

[P0606R0] Gabriel Dos Reis. *Concepts Are Ready*. Tech. rep. P0606R0. WG21, 2017.

[Sie10] Jeremy Siek. *Concepts in C++*. Presentation from Spring School on Generic and Indexed Programming. 2010. URL: `http://ecee.colorado.edu/~siek/concepts-ssgip2010.pdf`.

[Sie12] Jeremy Siek. "The C++0x "Concepts" Effort". In: *Generic and Indexed Programming: International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures*. Springer Berlin Heidelberg, 2012.

[Str09] Bjarne Stroustrup. "The C++0x "Remove Concepts" Decision". In: *Dr. Dobb's Journal* (2009). URL: `http://www.drdobbs.com/cpp/the-c0x-remove-concepts-decision/218600111`.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[SW14] Bjarne Stroustrup and William Wong. "Bjarne Stroustrup Talks About C++14". In: *Electronic Design* (2014). URL: `http://www.electronicdesign.com/dev-tools/bjarne-stroustrup-talks-about-c14`.