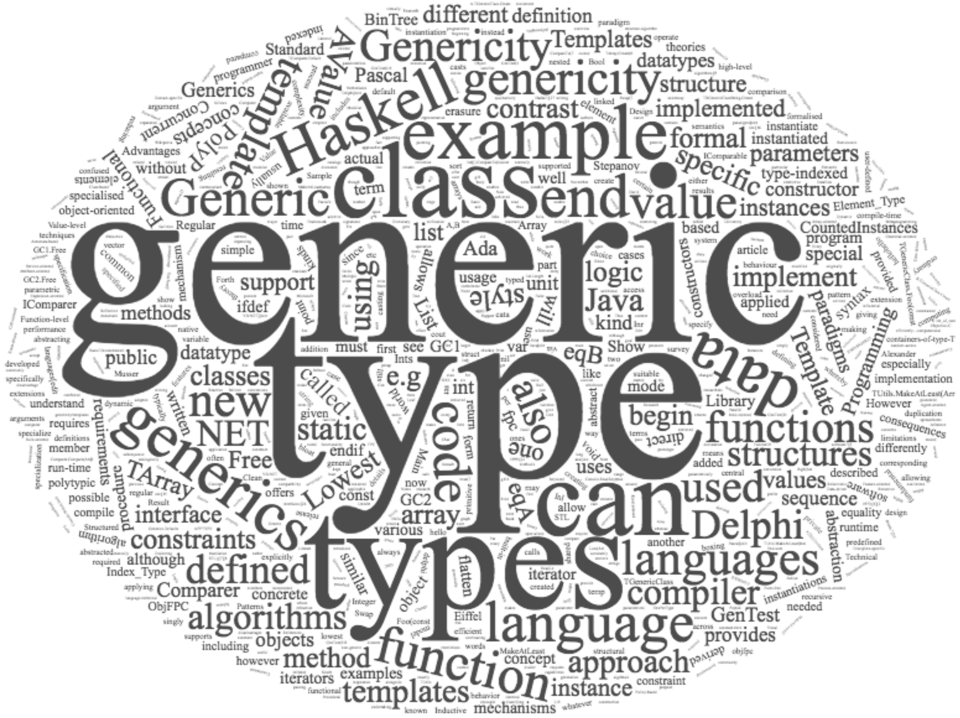
generic programming – c++



contents

generic programming in c++

- ***generic function***
  - performs the same operation on different data types
- how to ***implement*** a generic function in C++
  - ***overloading***
  - ***void pointers***
  - **templates**
- example: *swap the value of two variables*

generic function - overloading

overloading: set of methods all having

- *the same name*
- *different arguments list (signature)*

```
void my_swap (int &f, int &s ) {
     int tmp = f; f=s; s=tmp;
}
void my_swap (string &f, string &s ) {
     string tmp = f; f=s; s=tmp;
}
int main() {
   string a, b; a = "hello"; b = "world";
   cout << "before a = " << a << " b = " << b << endl;
   my_swap (a,b);
   cout << "after  a = " << a << " b = " << b << endl;
   int x, y; x = 33; y = 44;
   cout << "before x = " << x << " y = " << y << endl;
   my_swap(x,y);
   cout << "after x = " << x << " by = " << y << endl;

   double d1, d2; d1 = 3.3; d2 = 4.4;
   cout << "before d1 = " << d1 << " d1 = " << d2 << endl;
   // my_swap(d1,d2);      // compile time error
                           // no know conversion from double to &int ...
   cout << "after d1 = " << d1 << " d2 = " << d2 << endl;
   return 0;
}
```

generic function – void pointers

- we can write a function that takes a ***void pointer as an argument***, and then **use** that method with ***any pointer***
- this method is more ***general*** and can be used in more places
- we ***need cast*** from void pointer to a specific pointer

gneric function – void pointers

```
void my_swap (void* &f, void* &s ) {
     void* tmp = f;
     f=s;
     s=tmp;
 }

int main() {
   void* a; void* b;
   a = new std::string("hello"); b = new std::string("world");
   cout <<  *((string*) a) <<  *((string*) b) << endl;
   my_swap (a,b);
   cout <<  *((string*) a) <<  *((string*) b) << endl;

   void* x; void* y;
   x = new int(33); y = new int(44);
   cout <<  *((int*) x) <<  *((int*) y) << endl;
   my_swap(x,y);
   cout <<  *((int*) x) <<  *((int*) y) << endl;

   cout << "a = " << *((int*) a) << endl;
        // no compile time error, no runtime error
        // output a = 1919907594    :(
   return 0;
}
```

generic function - templates

```
template <class T>
void my_swap(T& f, T& s) {
     T tmp = f;
     f = s;
     s = tmp;
}

int main()
{
   int a = 3; int b = 4;
   cout << "before a = " << a << " b = " << b << endl;
   my_swap<int> (a,b);
   cout << "after  a = " << a << " b = " << b << endl;

   string s1 = "hello";
   string s2 = "world";
   cout << "before s1 = " << s1 << " s2 = " << s2 << endl;
   my_swap<string> (s1,s2);
   cout << "after  s1 = " << s1 << " s2 = " << s2 << endl;

   return 0;
}
```

we add a type parameter to the function

template

- templates allows ***functions*** and ***classes*** to operate with ***generic types***
- with templates a function or a class can work on many different data types without being rewritten for each one
- the C++ Standard Library provides many useful functions within a framework of connected templates
- kinds of templates:
  - ***function*** templates
  - ***class*** templates
  - variable templates (C++14)

function template

- a function template defines a family of functions

```
template <class identifier>
function_declaration;
template <typename identifier>
function_declaration;
```

template: array central element

T must be a type

primitive type

class

```
template <typename T>
T centralElement(T data[], int cont)
{
    return data[cont/2];
}
```

```
int i[] = {10,20,30,40,50};
int ci = centralElement(i,5);

string s[] = {"alpha","beta","gamma"};
string cs = centralElement(s,3);

float f[] = {2.2,3.3,4.4};
float cf = centralElement<float>(f,3);
```

type parameters are **inferred** from the values in a function invocation

or **explicitly** passed as type parameter

argument deduction

```
template <typename T>
T min (T a, T b) {
   return a < b ? a : b;
}
int main() {
   std::cout << min(3,4);     // OK (output 3) 'int', 'int' inferred
   std::cout << min(3.3,4);   // compile time error
   // template argument deduction/substitution failed:
   // deduced conflicting types for parameter ‘T’ (‘double’ and ‘int’)
   std::cout << min(3.3,(double)(4)); // OK (output 3.3) 'double', 'double' inferred
   std::cout << min(3.3,static_cast<double>(4));
                     // OK (output 3.3) 'double', 'double' inferred
   std::cout << min<double>(3.3,4); // OK (output 3.3) 'double' explicitly passed
}
```

multiple type parameters

```
template <typename T1, typename T2>
T1 min (T1 a, T2 b) {
   return a < b ? a : b;
}

int main() {
   std::cout << min(3,4) << std::endl;     // output 3 : 'int', 'int' -> 'int'
   std::cout << min(3.3,4) << std::endl;  // output 3.3 'double', 'int' -> 'double'
   std::cout << min(4, 3.3) << std::endl; // output 3 'int', 'double' -> 'int'
}
```

return type parameter

```
template <typename T1, typename T2, typename RT>
RT min (T1 a, T2 b) {
    return static_cast<RT>(a < b ? a : b);
}
int main() {
   std::cout << min<int,int,int>(3,4);
   // output 3 : 'int', 'int' -> 'int'
   std::cout << min<double,int,double>(3.3,4);
   // output 3.3 'double', 'int' -> 'double'
   std::cout << min<int,double,double>(4, 3.3);
   // output 3.3 'int', 'double' -> 'double'
}
```

under the hood

C++ function template

- in c++, templates are a ***pure compile-time feature***
- template is a ***factory*** that can be used to ***produce functions***
- c++ provide ***substitutions of types*** during compile time
  - *in c# substitutions are performed at runtime*
- each ***set*** of different template ***parameters*** may cause the generation at compile time of a ***different internal function definition***
- the resulting program is ***bigger in size*** due to the boilerplate code created during compilation

boilerplate code

- *sections of code included in many places with little or no alteration*

more on function templates

- ***a generic function*** is also called ***template function***
    - when the compiler creates a specific version of a generic function, it is said to have created a ***generated function***
    - the act of generation is said ***instantiation***
    - a generated function is a specific ***instance*** of a template function
- no code is generated from a source file that contains only template definitions
- in order for any code to appear, a template must be instantiated → the template arguments must be determined so that the compiler can generate an actual function

generic function & overloading

- we can define the ***explicit overloading*** of a generic function
- the modified version overloaded, ***hide*** the generic function

```
template<class T>
T square(T b) {  return b * b; }

template<>
string square(string b) { return b + b; }

int main( ) {
    int i = 5; cout << "square "<< i << " = " << square(i) << endl;      //square 5 = 25
    double j = 5.5; cout << "square "<< j << " = " << square(j) << endl; //square 5.5 = 30.25
    string s = "hello";
    cout << "square "<< s << " = " << square(s) << endl;          //square hello = hellohello
    char c = 'h';
    cout << "square "<< c << " = " << square(c) << endl;          //square h = @
}
```

class template

template class can work with many different types of values

class template

- a class template provides a specification for ***generating classes*** based on parameters
- class templates are generally used to implement containers
- a class template is ***instantiated*** by passing ***a given set of types*** to it as template arguments

```
template < parameter-list > class-declaration
```

```
template <typename F, typename S>
class Pair
{
public:
    Pair(const F& f, const S& s);
    F get_first() const;
    S get_second() const;
private:
    F first;
    S second;
};
```

```
template <typename F, typename S>
Pair<F,S>::Pair(const F& f, const S& s)
{
    first = f;
    second = s;
};

template <typename F, typename S>
F Pair<F,S>::get_first() const
{
    return first;
};

template <typename F, typename S>
S Pair<F,S>::get_second() const
{
    return second;
};
```

class templates unlike function templates

- when we declare a ***variable*** of a ***template class*** you ***must specify*** the parameters ***type***
  - types are not inferred

```
Pair<int,double> p1(2,3.4);
int p1_first = p1.get_first();
double p1_second = p1.get_second();


Pair<string,int> p2("alpha",5);
string p2_first = p2.get_first();
int p2_second = p2.get_second();
```

templates & compiler

```
template<typename T>
class Foo
{
public:
      T& bar()
      {
           return subject;
      }
private:
      T subject;
};


Foo<int> fooInt;
Foo<double> fooDouble;
```

the compiler generates the code for the specific types given in the template class instantiation

left side and right side
will generate the same compiled code

```
class FooInt
{
public:
      int& bar()
      {
           return subject;
      }
private:
      int subject;
}

class FooDouble
{
public:
      double& bar()
      {
           return subject;
      }
private:
      double subject;
}


FooInt fooInt;
FooDouble fooDouble;
```

inheritance vs template

inheritance

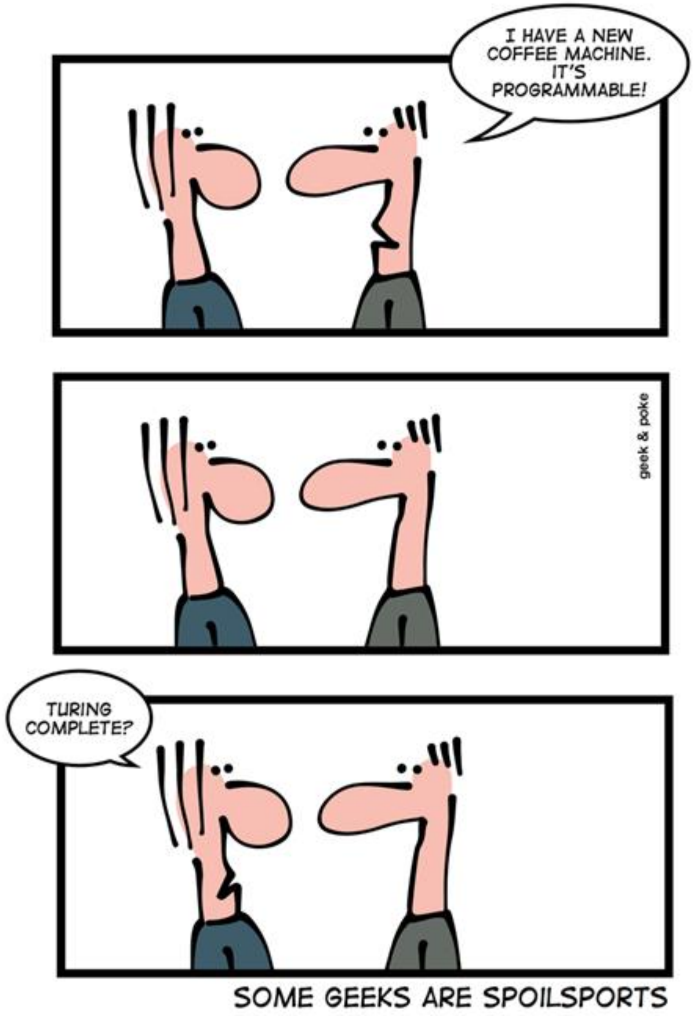
- ***run time*** polymorphism
- requires run time mechanism for binding
- late binding

template

- ***compile time*** polymorphism
- each set of different template parameters may cause the generation of a different internal function definition
- no run time cost

template → Turing complete

- templates are a compile time mechanism that is Turing-complete
  - *any computation expressible by a computer program can be computed, in some form, by a template metaprogram prior to runtime*
- *is this fact useful in practice?*



c++ templates with invalid parameters

compiler error messages are often misleading and obscure

```
class Point {
public:
   Point();
   Point(int, int);
   ~Point();

   void setX(int);
   int getX();
   void setY(int);
   int getY();
   void display();
private:
   int x;
   int y;
};
```

```
Point::Point () {
  x = 0;     y = 0;
}
Point::Point (int x, int y) {
  this->x = x;  this->y = y;
}
Point::~Point(){ }
void Point::setX (int x){
  this->x=x; }
int Point::getX() {
  return x; }
void Point::setY (int y){
  this->y=y; }
int Point::getY() {
  return y;
}
void Point::display() {
    cout<<"("<<x<<","<<y<<")"<<endl;
}
```

invalid type parameter

- the ***properties*** that a type parameter must satisfy are characterized only ***implicitly*** by the way instances of the type are used in the body of the template function

```
template <typename T>
T minValue(T v1, T v2)
{
    if (v1<v2)
        return v1;
    return v2;
}
```

```
int mi = minValue(3,6);             //(int int) OK
float mf1 = minValue(9.2,6.1); // (float float) OK
float mf2 = minValue(9.2,6);
// (float int) error:
template argument deduction/substitution failed

float mf3 = minValue<float>(9.2,6);
//explicit provide type parameter OK

Point p1(3.2,4.7);
Point p2(2.9,1.1);
Point p3 = minValue(p1,p2);
// error: no match for 'operator<`
(operand types are 'Point' and 'Point')
```

Compiler error messages are often misleading and obscure

problems

- the error message we get from minValue(p1,p2) is verbose and ***nowhere near as precise and helpful***
- to use minValue ***we need to provide its definition***, rather than just its declaration
- the ***requirements*** of minValue on its argument type are implicit ("***hidden***") in its function body
- the error message for minValue will appear only when the template is instantiated, and that may be long after the point of call
- proposed ***solution***:
- using a ***concept*** we can get to the root of the problem by properly specifying a ***template's requirements*** on its arguments

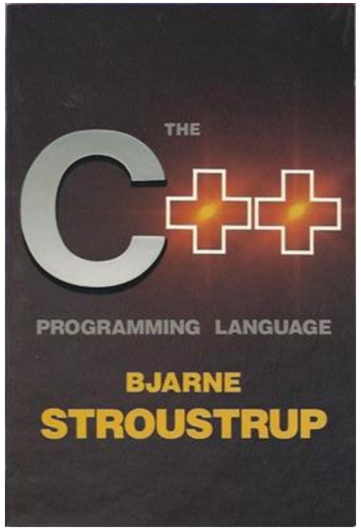
c++ concepts

concepts

Bjarne Stroustrup

The Future of Generic Programming and

how to design good concepts and use them well



once upon a time ...

- In about ***1987***, I (Bjarne ***Stroustrup***) tried to design ***templates*** with proper interfaces. I ***failed***. I wanted three ***properties*** for templates:
  - Full generality/expressiveness
  - Zero overhead compared to hand coding
  - Well-specified interfaces
- Then, nobody could figure out how to get all three, so we got
  - :) Turing completeness
  - :) Better than hand-coding performance
  - **:(Lousy interfaces (basically compile-time duck typing)**
- The lack of well-specified interfaces led to the ***spectacularly bad error messages*** we saw over the years. The other two properties made templates a run-away success.
- The solution to the interface specification problem was named “concepts” by Alex Stepanov

https://www.stroustrup.com/good_concepts.pdf

timeline – traditional code

```
double sqrt(double d); // C++84: accept any d that is a double
double d = 7;
double d2 = sqrt(d); // fine: d is a double
vector<string> vs = { "Good", "old", "templates" };
double d3 = sqrt(vs); // error: vs is not a double
```

- we have a function sqrt ***specified*** to require a double
- if we give it a double (as in sqrt(d)) all is well
- if we give it something that is not a double (as in sqrt(vs)) we promptly get ***a helpful error message***, such as “a vector<string> is not a double.”

timeline – 1990s style generic code

```
template<class T> void sort(T& c) // C++98: accept a c of any type T
{
  // code for sorting (depending on various properties of T,
  // such as having [] and a value type with <
}
vector<string> vs = { "Good", "old", "templates" };
sort(vs); // fine: vs happens to have all the syntactic properties required by sort
double d = 7;
sort(d); // error: d doesn't have a [] operator
```

- the **error message** we get from sort(d) is verbose and nowhere near as precise and helpful
- will ***appear*** only when the template is instantiated, and that may be long after the point of call
- to use sort, ***we need to provide its definition***, rather than just its declaration, this differs from ordinary code and changes the model of how we organize code
- the ***requirements*** of sort on its argument type are implicit ("***hidden***") in its function body

timeline – cpp20 with concepts

```
// Generic code using a concept (Sortable):
void sort(Sortable& c); // Concepts: accept any c that is Sortable
vector<string> vs = { "Hello", "new", "World" };
sort(vs); // fine: vs is a Sortable container
double d = 7;
sort(d); // error: d is not Sortable (double does not provide [], etc.)
```

- this code is analogous to the sqrt example
- the only real difference is that for ***double***, a language designer (Dennis ***Ritchie***) built it into the compiler as a ***specific type*** with its meaning specified in documentation
- for ***Sortable***, a ***user*** specified what it means in code
    - a type is Sortable if it has begin() and end() providing random access to a sequence with elements that can be compared using <
- we get an ***error message*** much as indicated in the comment
- the message is generated immediately ***at the point*** where the compiler sees the erroneous call (sort(d))

concepts as constraints

- ***templates*** may be associated with a ***constraint***
  - it specifies the requirements on template arguments
- constraints may also be used to ***limit automatic type deduction*** in variable declarations and function return types to only the types that satisfy specified requirements
- named ***sets of*** such ***requirements*** are called ***concepts***
- each ***concept is a predicate***, evaluated at ***compile time***, and becomes a part of the interface of a template where it is used as a constraint
- violations of constraints are detected at compile time, early in the template instantiation process, which leads to easy to follow error messages

e.g. EqualityComparable

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

```
template <Equality_comparable T>
bool twoEquals(T v1, T v2, T v3)
{
    if (v1==v2 || v1==v3 || v2==v3)
        return true;
    return false;
}
```

- `Equality_comparable` is proposed as a **standard-library concept**
- like many concepts it takes more than one argument: ***concepts describe not just types but relationships among types***
- a `require` expression is never actually executed → the compiler looks at the requirements and compiles only if all are `true`

compiler errors

```
cout<<twoEquals(2,3,2)<<endl;            //(int int int) OK
cout<<twoEquals(9.2,6.1,5.8)<<endl;      //(float float float) OK
cout<<twoEquals(2,3.1,2)<<endl;          //(int float int) ERROR
cout<<twoEquals<float>(9.2,6,6)<<endl;   //explicit provide type parameter OK
cout<<twoEquals("alpha","beta","beta")<<endl;   //(string string string) OK
Point p1(3,4); Point p2(5,2);   Point p3(3,4);
cout<<twoEquals(p1,p2,p3)<<endl;
// error: cannot call function ‘bool twoEquals(T, T, T) [with T = Point]’
// error no match for 'operator<' (operand types are 'Point' and 'Point')
// note:   constraints not satisfied
// bool twoEquals(T v1, T v2, T v3)
// ^~~~~~~~~~~~~
// note:within 'template<class T> concept bool Equality_Comparable() [with T = Point]'
// concept bool Equality_Comparable()
//               ^~~~~~~~~~~~~~~~~~~
// note:     with ‘Point a’
// note:     with ‘Point b’
// note: the required expression ‘(a == b)’ would be ill-formed
// note: the required expression ‘(a != b)’ would be ill-formed
```

example

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

template<typename T>
concept Addable = requires (T x) { x + x; }; // requires-expression

template<typename T> requires Addable<T> // requires-clause, not requires-expression
T mySum(T a, T b) { return a + b; }

int main() {
  std::cout << mySum(3,4) << std::endl;
  return 0;
}
```

https://en.cppreference.com/w/cpp/language/constraints

https://coliru.stacked-crooked.com/

concept summary

- concepts are ***named boolean predicates*** on template parameters, ***evaluated at compile time***
- a concept may be associated with a ***template***, it serves as a ***constraint*** (***limits the set of arguments that are accepted as template parameters***)
- concepts ***simplify compiler diagnostics*** for failed template instantiations
- if a programmer attempts to use a template argument that does not satisfy the requirements of the template, the compiler will generate an error

concepts as semantic categories

- the intent of concepts is to model ***semantic*** categories rather than syntactic restrictions
- the ability to specify a meaningful semantics is a defining characteristic of a true concept, as opposed to a syntactic constraint
- *“Concepts are meant to express semantic notions, such as ‘a number’, ‘a range’ of elements, and ‘totally ordered.’ Simple constraints, such as ‘has a + operator’ and ‘has a > operator’ cannot be meaningfully specified in isolation and should be used only as building blocks for meaningful concepts, rather than in user code.”*

Avoid "concepts" without meaningful semantics
(ISO C++ core guideline T.20)
Bjarne Stroustrup - Herb Sutter

e.g. Number

```
template<typename T>
concept Number = requires (T x) {
    x + x;
    x - x;
    x * x;
    x / x;
    -x;
    x += x;
    …
};

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

this is extremely unlikely
to be matched unintentionally

concepts library (c++20)

- the concepts library provides definitions of fundamental ***library concepts*** that can be used to perform ***compile-time validation of template arguments***
- most concepts in the standard library impose both ***syntactic*** and ***semantic*** requirements
  - a standard concept is ***satisfied*** if its ***syntactic*** requirements are met and its ***semantic*** requirements (if any) are also met
  - ***only the syntactic requirements can be checked by the compiler***

https://en.cppreference.com/w/cpp/concepts

concepts library (c++20)

- core language concepts
- comparison concepts
- object concepts
- callable concepts
- *additional concepts can be found in the iterators library, the the algorithms library, and the ranges library*

https://en.cppreference.com/w/cpp/concepts

```
template <class From, class To>
concept convertible_to =
  std::is_convertible_v<From, To> &&
  requires(std::add_rvalue_reference_t<From> (&f)()) {
    static_cast<To>(f());
  };
```

The concept convertible_to<From, To> specifies that an expression of the same type and value category as those of std::declval<From>() can be implicitly and explicitly converted to the type To, and the two forms of conversion are equivalent.

```
template <class T>
concept totally_ordered =
  std::equality_comparable<T> && __PartiallyOrderedWith<T, T>;
```

The concept totally_ordered<T> specifies that the comparison operators ==,!=,<,>,<=,>= on T yield results consistent with a strict total order on T.

```
template <class T>
concept copyable =
  std::copy_constructible<T> &&
  std::movable<T> &&
  std::assignable_from<T&, T&> &&
  std::assignable_from<T&, const T&> &&
  std::assignable_from<T&, const T>;
```

The concept copyable<T> specifies that T is an movable object type that can also copied (that is, it supports copy construction and copy assignment).

https://en.cppreference.com/w/cpp/concepts

references

- *Dehnert, James & Stepanov, Alexander. (1998). Fundamentals of Generic Programming. LNCS. 1766. 1-11. 10.1007/3-540-39953-4_1.*
- *B. Stroustrup: Concepts: The Future of Generic Programming (or "How to design good concepts and use them well)". January 2017.*
- *https://en.cppreference.com/w/cpp/concepts*