An introduction to C++ Concepts

Thomas Peters

Dec 13, 2017

Bonjour

Summary

1) Background, statement of problem.
2) Overview of Concepts.
3) How we deal without Concepts today.

Brief summary of templates

- Mechanism for generic programming in C++
- Instead of:

```
int add(int x, int y) { return x + y; }
double add(double x, double y) { return x + y; }
```

Can write “a single function”

```
template <class T>
T add(T x, T y) { return x + y; }

add(1, 2);
add(3.0, 4.0);
add<float>(1, 2.0);
add(“c++”, “mtl”); // compile error
```

Similar to dynamic languages

```
template <class T>
T add(T x, T y) { return x + y; }
```

Similar to Python:

```
def add(x, y):
    return x + y
```

Similar to dynamic languages

```
template <class T>
T add(T x, T y) { return x + y; }
```

Actually more similar to:

```
def meta(T):
    def add_T(x, y):
        return x + y
    return add_T
```

Templates: compile time programming on types

- Objects returned by meta functions are (mostly) functions and classes.
- Specialization and overloading introduce branching, recursion
- Class constants allow computation of (integral) numeric values

Template metaprogramming is Turing complete

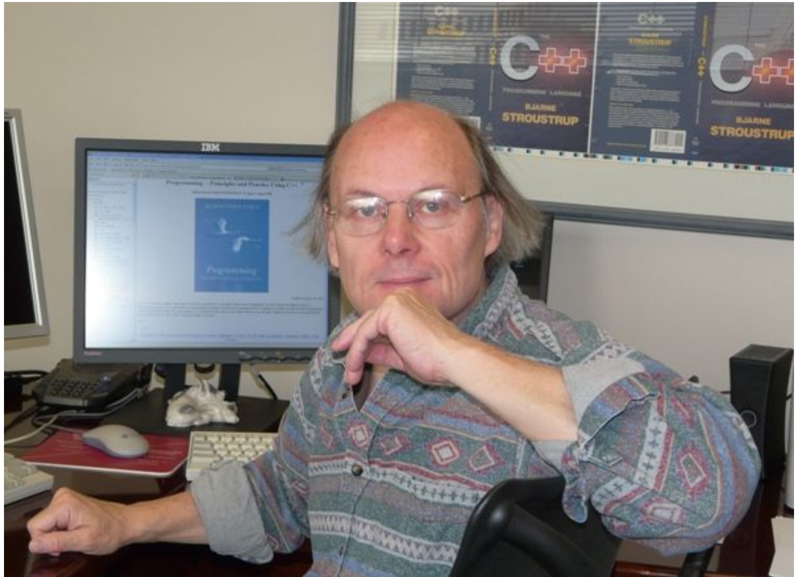
Design goals of templates (1987-8)

Bjarne wanted:

- Full generality/expressiveness
- Zero overhead compared to hand coding
- Well-specified interfaces

Instead got:

- Turing completeness
- better than hand coded performance
- *Lousy interfaces*



A solution to the interfaces problem is called *Concepts.*

Originally proposed by Alexander Stepanov

(father of the STL)

Lousy interfaces.

```
std::vector<int> vec { 1, 2, 3, 4 };

std::sort(vec.begin(), vec.end());    // OK

std::list<int> a_list { 1, 2, 3, 4 };

std::sort(a_list.begin(), a_list.end());
```

What could go wrong? (hop over to godbolt.org)

Compiler barfs

- `std::sort` is an unconstrained template
- Templates use compile time duck typing
- Errors are very much in the weeds, and don't offer *helpful* clues

Implicit interfaces

```
template<class T>

void sort(T& c) {
    // code for sorting (depending on various properties of T,
    // such as having [ ] and a value type with <
}
```

Concepts provide explicit requirements:

```
template<class T>

    requires Sortable<T>

void sort(T& c) {
    // ...
}
```

// or:

```
void sort(Sortable& c) {
    // ...
}
```

```
//or
template <Sortable T>
void sort(T t) { … }
```

Concepts make code more *readable*

Better error messages

```
sort(lst);
```

```
error: cannot call function 'void sort(T&) [with T = std::list<int>]'
note:   concept 'Sortable()' was not satisfied
```

Status of Concepts

- Voted down for C++11 (different version)
- Voted down for C++17 (“concepts-lite”)
- Candidate for C++20 (merged into working draft)
- GCC 6 already ships with concepts (Andrew Sutton)
  - Only the language features
  - No standard library
  - Need `-fconcepts` flag
  - Try on godbolt.org!

Andrew Sutton, architect of concepts ->

Concepts today

Consider `std::find` (http://en.cppreference.com/w/cpp/algorithm/find)

```
template <class InputIt, class T>

InputIt find(InputIt first, InputIt last, const T& value);
```

Type requirements

- `InputIt` must meet the requirements of InputIterator.

Example Concept: InputIterator

http://en.cppreference.com/w/cpp/concept/InputIterator

- Gives a list of requirements that must be satisfied by an InputIterator.
- *Not enforced by the compiler*

Similar definitions exist in Boost too.

Eventually may see:

```
template <class R, class T>
    requires Range<R> && EqualityComparableWith<Value_type<R>, T>
Iterator_of<R> find(R& range, const T& value);


// Using template aliases
template <class X> using Value_type = X::value_type;
template <class X> using Iterator_of = X::iterator;
```

- Concepts not only introduce requirements on single types, but on *relationships between types*
    - Eg, `EqualityComparableWith<Value_type<R>, T>`
- See the Ranges TS for a language feature built with Concepts (http://en.cppreference.com/w/cpp/experimental/ranges)

Defining concepts

template < *template-parameter-list* **>**

concept [bool] *concept-name* **=** *constraint-expression***;**

Examples:

```
// variable concept
template <class T, class U>
concept bool Derived = std::is_base_of<U, T>::value;

// function concept (must be invoked)
template <class T>
concept bool EqualityComparable() {
    return requires(T a, T b) { {a == b} -> Boolean;
                                {a != b} -> Boolean; };
}
```

Constraint expression

Constraints

1) conjunctions
2) disjunctions
3) predicate constraints
4) expression constraints (only in a requires-expression)
5) type constraints (only in a requires-expression)
6) implicit conversion constraints (only in a requires-expression)
7) argument deduction constraints (only in a requires-expression)
8) exception constraints (only in a requires-expression)
9) parametrized constraints (only in a requires-expression)

More examples: Range

```
template <class T>

concept bool Range = requires(T t) {

      typename Value_type<T>; // must have a value type typename Iterator_of;

      typename Iterator_type<T>; // must have an iterator type

      { begin(t) } -> Iterator_of<T>; // must have begin() and end() which return iterators

      { end(t) } -> Iterator_of<T>;

      requires Input_iterator<Iterator_of<T>>;

      requires Same_type<Value_type<T>, Value_type<Iterator_of<T>>>;

};
```

More examples: Sortable

```
template <class T>
concept bool Sortable =
    Range<T>
    && Random_access_iterator<Iterator_of<T>>
    && Less_than_comparable<Value_type<T>>
    ;
```

Template overloading

```
template<class T1, class T2> void f(T1, T2);
template<class T> void f(T);

void f(int, int);    // takes precedence over template functions

void f(int, int, int);
```

It's hard to provide overloads based on particular properties of T.

Concepts: overloading

```
void advance(Forward_iterator p, int n) { while(n--) ++p; }

void advance(Random_access_iterator p, int n) { p+=n; }

void use(vector& vs, list& ls) {

    auto pvs = find(vs, "foo");

    advance(pvs, 2); // use fast advance

    auto pls = find(ls, "foo");

    advance(pls, 2); // use slow advance

}
```

Concepts: constrained type deduction

```
// the type of x1 is deduced to whatever f returns
auto x1 = f(y);

// Return type of f must satisfy Sortable concept to compile
Sortable x2 = f(y);
```

Dealing without concepts 1

1. Do nothing. Cry.
2. Use documentation (like the STL)
3. Use gcc 6 with `-fconcepts`

Dealing without concepts 2: plain ol' templates

```
template<class T1, class T2> struct Can_copy {
     static void constraints(T1 a, T2 b) { T2 c = a; b = a; }
     Can_copy() { constraints; }
};

struct X {};
struct Y {};

Can_copy<X, Y>();
```

```
// Gcc 6.1 output

<source>: In instantiation of 'static void Can_copy<T1, T2>::constraints(T1, T2) [with T1 = X; T2 = Y]':
7 : <source>:7:22:   required from 'Can_copy<T1, T2>::Can_copy() [with T1 = X; T2 = Y]'
22 : <source>:22:23:   required from here
6 : <source>:6:54: error: conversion from 'X' to non-scalar type 'Y' requested
     static void constraints(T1 a, T2 b) { T2 c = a; b = a; }
                                                  ^
6 : <source>:6:59: error: no match for 'operator=' (operand types are 'Y' and 'X')
     static void constraints(T1 a, T2 b) { T2 c = a; b = a; }
                                                       ~~^~~
```

Dealing without concepts 3: static_assert

```
template <class T>

void foo(T&t) {

    static_assert(std::is_integral<T>::value);

    // ....

}
```

- Errors still come out of body, just as without.
- Again, interface is specified in implementation (function body)
- Doesn’t allow overloading

Dealing without concepts 4: tagged-dispatch

```
namespace std {
  struct input_iterator_tag { };
  struct bidirectional_iterator_tag { };
  struct random_access_iterator_tag { };

  namespace detail {
    template <class InputIterator, class Distance>
    void advance_dispatch(InputIterator& i, Distance n,
                          input_iterator_tag) {
      while (n--) ++i;
    }

    template <class BidirectionalIterator, class Distance>
    void advance_dispatch(BidirectionalIterator& i, Distance n,
       bidirectional_iterator_tag) {
      if (n >= 0)
        while (n--) ++i;
      else
        while (n++) --i;
    }

    template <class RandomAccessIterator, class Distance>
      void advance_dispatch(RandomAccessIterator& i, Distance n,
         random_access_iterator_tag) {
        i += n;
      }
    }

    template <class InputIterator, class Distance>
    void advance(InputIterator& i, Distance n) {
      typename iterator_traits<InputIterator>::iterator_category
category;
      detail::advance_dispatch(i, n, category);
    }
  }

// A technique for selecting implementations based on compile time
properties.
```

Dealing without concepts 5: constexpr if

```
template <class T>
void foo(T t) {        // C++17 feature
    if constexpr (std::is_integral<T>::value) {
        // provide int implementation
    } else {
        // provide non-int implementation
    }
}
```

Dealing without concepts 6: SFINAE

```
template <class T, typename = std::enable_if_t<std::is_integral<T>::value>>  // C++14, almost C++11

void foo(T t) { }  // basically equivalent to requires Integral<T>
```

```
template <class T>

auto bar(T t) -> decltype(t.member_func(), void())

{}   // disabled unless has a member function named

     // `member_func` which accepts no arguments
```

Also see `void_t`, or any talk by Walter Brown. (C++17, yet trivial to implement)

Dealing without concepts 6: SFINAE

- SFINAE is almost as powerful as concepts. Can almost implement concepts!
  - https://stackoverflow.com/questions/26513095/void-t-can-implement-concepts
  - https://akrzemi1.wordpress.com/2016/03/21/concepts-without-concepts/
  - http://www.boost.org/doc/libs/1_60_0/libs/concept_check/concept_check.htm
- SFINAE can’t apply to things like constructors in a template class
- SFINAE is *ugly,* and hard to get right.

Why concepts didn't make C++17?

- Only one implementation (gcc)
- Not enough use cases seen (Ranges TS biggest example)
- No library of concepts
- <u>No checking of template definitions</u>
- *Sometimes, error messages are worse*
  - You can end up with a large set of discarded overloads, each with its own reason for rejection.
  - See http://honermann.net/blog/ for examples

Conclusion: Concepts simplify generic programming

- Improve readability
- Improve error messages (hopefully)
- Provide explicit interfaces for templates
- Easier overloading, specialization of templates
- Restricted type deduction (`auto` replacement)
- Unclear if will make it as a real language feature.

Summary:

A C++ concept is a compile-time predicate on zero or more template argument type argument or value arguments.

References

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

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0557r0.pdf

Contact: `thomas.d.peters@gmail.com`