Using Concepts - C++ Design in a Concept World

Jeff Garland

Created: 2021-05-03 Mon 20:42

Intro

> *"All things change except barbers, the ways of barbers, and the surroundings of barbers. These never change." -Mark Twain*

about me

talk goals

- climb up the concept ladder
  - what's a concept
  - using concepts in code
  - reading concepts: requires expressions & clauses
  - writing concepts: it's hard…
  - designing with concepts

talk outline - part 1

- concept basics
  - overview of concepts
  - concepts vs types
- using a concepts in code
  - overloading, variables, pointers
  - std library concepts
- reading concepts
  - requires expressions

talk outline - part 2

- writing concepts
  - concept details 102
  - writing `sleep_for` with concepts
  - good concepts, bad concepts
- designing with concepts
  - what is design?
  - review of some 'design principles'
  - concepts and dependencies
  - impact on multi-paradigm design in c++
  - concept serialization

tools

- gcc 10.2 on linux, predominantly
- other platforms also coming along
  - clang 10, msvc partial
  - std:lib concepts partial

Concept Basics

why do we want concepts?

- want to be able to write good generic libraries
- that are fast
- with reasonable error messages
- the our fellow programmers can understand and maintain

why do we want concepts II?

- unconstrained template programming has issues
- hundreds of lines of error messages
- solutions involve meta programming
- direct langauge support means
  - improved tooling
  - better compile times
- **better interfaces**
  - more descriptive
  - better dependency management

long history

- Stepanov 'algebraic structures' in 1981
- 1994 STL specification
- 2000 libraries – boost uses concepts (concept checking
- 2011 Palo Alto meeting –> concepts lite
- 2017 concepts technical specification
- c++20 ranges library specified by concepts
- c++20 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0898r3.p
- `PascalCase` versus `snake_case`
- `concept bool` versus `concept`

what's a concept?

- boolean predicate on types and values
- type requirement examples
  - required methods
  - required semantics*
  - required subtypes or base types
- c++ realization includes
  - new keywords `concept` and `requires`
  - `<concept_name> auto` for describing a set of types
  - new rules for function overloading

boolean predicate composition

- support complex compile time logic composition
- conjuction and disjunction (and/or) logic
- used to classify types
  - in or out of a set
  - that share syntax/semantic
  - although semantics is the desire only syntax is checked

types versus concepts

- type
  - describes a set of operation that can perform
  - relationships with other types
    - example: base class
    - example: declared dependent type
  - describes a memory layout
  - for built in types this can be implicit
- concept
  - describes how a type can be used
  - operations it can perform
  - relationships with other types

simple one parameter concept `printable`

```
namespace io {

  // Type T has print ( std::ostream& ) const member function
  template<class T>
  concept printable = requires //...more later
}

class my_type
{
  std::string s = "foo\n";
public:
  void print( std::ostream& os ) const
  {
    os << "s: " << s;
  }
};
static_assert( io::printable<my_type> ); //good
```

notable concept properties

- concepts evaluated completely at compile time
  - no runtime footprint – linker never sees
  - compatible with high performance code
- concepts can be scoped in namespaces
- bridge between 'pure auto' and a specific type
- difficult to change without recompiling all code
- core use case is constraining templates

Using Concepts in Code

> *"All models are wrong, some are useful." - George Box (1976)*

What can we do, not do with a concept?

can do

- constrain an overload set
- initialize a variable with `<concept_name> auto`
- conditional compilation with constexpr if
- can use a pointer or `unique_ptr` of concept
- partially specialize a template with concept
- make template code into 'regular code'

cannot do

- cannot inherit from concept
- cannot constrain a concrete type using requires
- cannot 'allocate' via new
- cannot apply requires to virtual function

where can we write `<concept_name> auto`?

- where a typename might otherwise appear
  - variable declaration
  - function parameter
  - function return type
  - class template member, if template argument
- but not
  - class member
  - base class
- template parameter and aliases (no auto)

using concepts - basic examples

```
// Type T has print ( std::ostream& ) member function
template<typename T>
concept printable = //...

template<typename T>
concept associative_container = //...
```

concept usage examples I

```
  //given concept definitions printable and associative_container

  void f(printable auto s) {...};

  template<associative_container T> //no auto here
  class MyType
  {
    T map_;
  }

main {

    printable auto s;

    associative_container auto myMap;
}
```

concept usage examples II

```
<source>: In function 'int main()':
<source>:16:4: error: declaration of
  'auto [requires ::printable<<placeholder>, >] s' has no initializer
   16 |    printable auto s;
      |
```

concept usage examples III

```
  //given concept definitions printable and associative_container

  void f(printable auto s) {...};

  template<associative_container T> //no auto here
  class MyType
  {
    T map_;
  }

main {

    printable auto s = init_some_thing();

    associative_container auto myMap = MyType<std::map<int, int>>{ {1,1
}
```

concept use - function parameter or return value

```
//https://godbolt.org/z/r3dY3dsvd

template<printable T>
printable auto
print( const T& s )
{
 //...
 return s;
};

printable auto
print2( const printable auto& s )
{
 //...
 return s;
}
```

overload resolution - constrain function parameter

- write function `print_ln`
- overload function based on concept

auto for function parameter, unconstrained

```
//https://godbolt.org/z/GKq8ns

#include <iostream>

//auto parameter -- this is a template function!
// template<typename T_
// void print_ln( T p )
void print_ln( auto p )
{
 std::cout << p << "\n";
}

class my_type {};

int main()
{
    print_ln( "foo" );
    print_ln( 100 );

    //compile error of course
    //my_type m;
    //print_ln ( m );
}
```

concrete overload `print_ln` for `my_type`

```
// https://godbolt.org/z/85cEdqMEc
  #include <iostream>

  void print_ln( auto p )
  {
   std::cout << p << "\n";
  }

  //selected ahead of print_ln (auto) because better match
  void print_ln( my_type p )
  {
   p.print( std::cout ); std::cout  << "\n";
  }

  int main()
  {
      print_ln( "foo" );
      print_ln( 100 );

      //good
      my_type m;
      print_ln ( m );
  }
```

concepts `printable` and `output_streamble`

```
//https://godbolt.org/z/dYdhW7
#include <iostream>
#include <memory>

// Type T has print ( std::ostream& ) member function
template<typename T>
concept printable = requires(std::ostream& os, T v)
{
  v.print( os ) ; //<--an expression that if compiles yields true
};

template<class T>
concept output_streamable = requires (std::ostream& os, T v)
{
   os << v;
};
```

a type satisfying `printable`

```
//https://godbolt.org/z/dYdhW7

class my_type
{
  int i = 1;
  std::string s = "foo\n";

public: //<-- still a concept error if print isn't public

  void print( std::ostream& os) const
  {
    os << "i: " << i << " s: " << s;
  }
};
static_assert( printable<my_type> );

class my_type2 {};
```

constrained overload for `print_ln`

```
// https://godbolt.org/z/36cdsGzzo
#include <iostream>

void print_ln( auto p )
{
 std::cout << p << "\n";
}

//auto parameter -- this is a template function!
void print_ln( printable auto p ) //<-- constrained resolution
{
  p.print( std::cout );
  std::cout << "\n";
}

int main()
{
    print_ln( "foo" );
    print_ln( 100 );

    my_type m;
    print_ln ( m );
}
```

overloaded functions

```
//https://godbolt.org/z/dYdhW7

// example of overload resolution
void print_ln( output_streamable auto p)
{
  std::cout << p << "\n";
}

void print_ln( printable auto p)
{
  p.print(std::cout);
  std::cout << "\n";
}

class my_type2 {};

int main()
{
    print_ln( "foo" );
    my_type m;
    print_ln ( m );
    //compile error of course
    //my_type2 m2;
```

```
    //print_ln ( m2 );
}
```

pointers and concepts

- useful for things like factory functions

```
//ouput:
//s: foo
//s: foo

// based on
//https://godbolt.org/z/d7bGhn

int main()
{
   const printable auto* m = new my_type();
   m->print(std::cout);
   const std::unique_ptr<printable auto> upm = std::make_unique<my_type
   upm->print(std::cout);
}
```

pointer to a concept - compile error

```
//     #1 with x86-64 gcc 10.2

// <source>: In function 'int main()':
// <source>:29:37: error: deduced initializer does not satisfy
//                                                 placeholder constraint
//    29 |     printable auto* m = new whatever{};
//       |                                    ^
// <source>:29:37: note: constraints not satisfied
// <source>:6:9:   required for the satisfaction of 'printable<whatever
// <source>:6:21:   in requirements with 'std::ostream& os',
//                                                'T v' [with T = whateve
// <source>:8:10: note: the required expression 'v.print(os)' is invali
//     8 |   v.print( os ) ;
//       |

class whatever {}; //no print

int main()
{
   printable auto* m = new whatever();
}
```

use in `if constexpr`

- use concepts without `concept` keyword
- compile time if – `if constexpr`

if constexpr and printable

```
//https://godbolt.org/z/nsojqK5e1
template<class T>
std::ostream&
print_ln( std::ostream& os, const T& v )
{
  if constexpr ( requires{ printable<T>; } )
//  if constexpr ( printable<T> )    <-- shorter version
  {
    v.print(os);
  }
  else { //no print function
    os << v;
  }
  os << "\n";
  return os;
}
int main()
{
   my_type m;
   print_ln( std::cout, m );
   int i = 100;
   print_ln( std::cout, i );
}
```

`if constexpr` requires expression

```
//https://godbolt.org/z/nsojqK5e1
template<class T>
std::ostream&
print_ln( std::ostream& os, const T& v )
{
  if constexpr ( requires{ v.print( os ); } )
  {
    v.print(os);
  }
  else { //no print function
    os << v;
  }
  os << "\n";
  return os;
}
int main()
{
   my_type m;
   print_ln( std::cout, m );
   int i = 100;
   print_ln( std::cout, i );
}
```

non-template member function of template class

```
https://godbolt.org/z/e6zWPqYzh
template<class T>
class wrapper
{
  T val_;
  public:
   wrapper(T val) : val_(val) {}
   T operator*() requires is_pointer_v<T>
   { return val_; }
};
int main()
{
  int i = 1;
  wrapper<int*> wi{&i};
  cout << *wi << endl;

  //no match for operator*
  //wrapper<int> wi2{i};
  //cout << *wi2 << endl;
}
```

constraining existing templates

- can we build an `std::vector<printable>`?
- without rewriting `vector`
- yes!

template alias with concept shorthand

```cpp
#include <iostream>
#include <vector>
#include <string>

//template alias using concepts
//All elements must satisfy concept printable
template<printable T> using
vec_of_printable = std::vector<T>;

int main()
{
  vec_of_printable<my_type> vp{ {}, {} };
  for ( const auto& e : vp )
  {
    e.print(std::cout);
  }
}
```

template alias error message

```
//template alias using concepts
//All elements must satisfy concept printable
template<printable T> using
vec_of_printable = std::vector<T>;

vec_of_printable<int> vp;

//compile error
// template_alias.cpp:16:21: error: template constraint
//                                               failure for ‘template<cl
//     requires  printable<T> using vec_of_printable = std::vector<T>’
//    16 | vec_of_printable<int> vp; //compile error
//       |                     ^
// template_alias.cpp:16:21: note: constraints not satisfied
// template_alias.cpp:6:9:   required for the satisfaction of
//                                                ‘printable<T>’ [with T
// template_alias.cpp:6:21:   in requirements with ‘std::ostream& os’,
//                                                [with T = int]
// template_alias.cpp:8:10: note: the required expression ‘v.print(os)’
//     8 |   v.print( os ) ;
//       |   ~~~~~~~^~~~~~
```

using standard library concepts

- in headers <concepts>, <type_traits>, <iterator> or <ranges>
- groups of std concepts
  - core language concepts
  - comparison concepts
  - object concepts
  - callable concepts
  - ranges concepts

an aside about <type_traits>

- the standard library has all sorts of trait types
- traits are compile time values like `is_arithmetic`
- traits can be used like/in concepts
- some traits require **compiler magic**
- many could be replaced with concepts

std concepts - numerics

concept	description
`floating_point<T>`	float, double, long double
`integral`	char, int, unsigned int, bool
`signed_integral`	char, int
`unsigned_integral`	char, unsigned

std concepts - comparison

concept	description
`equality_comparable<T>`	
`equality_comparable_with<T,U>`	operator== is an equivalence

std concepts - comparison II

concept	**description**
`totally_ordered<T>`	
`totally_ordered_with<T,U>`	`==`,`!=`,`<`,`>`,`<=`,`>=` are a total ordering

std concepts - object relations

concept	**description**
`same_as<T,U>`	types are same
`derived_from<T,U>`	T is subclass of U
`convertible_to<T,U>`	T converts to U
`assignable_from<T,U>`	T can assign from U

std concepts - object construction

concept	**description**
`default_initializable<T>`	default construction provided
`constructible_from<T,...>`	T can construct from variable pack
`move_constructible<T>`	support move
`copy_constructible<T>`	support move and copy

std concepts - regular semi-regular

- See Sean Parent talks on why regular is so useful
- tldr - type cooresponds to usual expectations (aka like int)

concept	description
`semiregular<T>`	copy move destruct default contruct
`regular<T>`	semiregular and equality comparible

enforcing regularity

```
//trivial with ~std::regular~ concept and ~static_assert~
#include <string>
#include <concepts>

class my_type
{
  std::string s = "foo\n";
public:
  void print( std::ostream& os ) const
  {
    os << "s: " << s;
  }
};

static_assert( std::regular<my_type> );
```

enforcing regular error

```
//   008a_regular.cpp:20:21: error: static assertion failed
//    20 | static_assert( std::regular<my_type> );
//       |                ~~~~~^~~~~~~~~~~~~~~~
// 008a_regular.cpp:20:21: note: constraints not satisfied
// In file included from /usr/include/c++/10/compare:39,
//                   ...
//                  from /usr/include/c++/10/iostream:39,
//                  from 008a_regular.cpp:2:
// /usr/include/c++/10/concepts:280:15:
//           required for the satisfaction of '__weakly_eq_cmp_with<_Tp
//           [with _Tp = my_type]
// /usr/include/c++/10/concepts:290:13:
//           required for the satisfaction of 'equality_comparable<_Tp>
//           [with _Tp = my_type]
```

enforcing regular fix

```
#include <string>
#include <concepts>


class my_type
{
  std::string s = "foo\n";
public:
  void print( std::ostream& os ) const
  {
    os << "s: " << s;
  }
  //added this line
  bool operator==( const my_type& ) const = default;
};

static_assert( std::regular<my_type> );
```

using range concepts

```
//https://godbolt.org/z/53qzE35M5
#include <string>
#include <span>
#include <ranges>
#include <array>
#include <iostream>
using namespace std;

void print_ints( const std::ranges:range auto& R )
{
  for ( auto i :  R )
  {
     cout << i << endl;
  }
}
```

using range concepts II

```
//https://godbolt.org/z/53qzE35M5

//this function works on all the types below
void print_ints( const std::ranges:range auto& R ) {...}

int main()
{
  vector<int> vi = { 1, 2, 3, 4, 5 };
  print_ints( vi );

  array<int, 5> ai = { 1, 2, 3, 4, 5 };
  print_ints( ai );

  span<int>      si2 ( ai );
  print_ints( si2 );

  int cai[] = { 1, 2, 3, 4, 5 };
  span<int>      si3( cai );
  print_ints( si3 );

  ranges::iota_view iv{1, 6};
  print_ints( iv );
```

```
}
```

Reading Concepts

> *"Read the source, Luke" - apologies to Yoda*

`requires` expression and `requires` clause

- clause -> a boolean expression
  - used after template and method declarations
  - clauses can contain an expression
- expression -> syntax for describing type constraints

`requires` expression basics

- parameters and expression are optional
- all requirements evaluate to true or false

```
requires { requirement-sequence }
requires ( ..parameters..) { requirement-sequence }

//simplest possible boolean example no parameters
template<typename T>
concept always = true;
```

`requires` expression more realistic example

```
// Type T has print ( std::ostream& ) member function
template<typename T>
concept printable = requires(std::ostream& os, T v)
{
  //this is a conjuntion AND --> all must be true
  v.print( os ) ;  //member function
  format ( v ) ;   //free function
  std::movable<T>;
  typename T::format; //declare a type called format
};

template<class T>
concept output_streamable = requires (std::ostream& os, T v)
{
   //constraint on return from operator<<
   { os << v } -> std::same_as<std::ostream&>;
};
```

constraint composition

- atomic constraints
- conjunction constraints (and)
- disjunction constraints (or)

constraint composition example

```
//disjunction
template<typename T>
concept printable_or_streamable =
     requires printable<T> || output_streamable<T>;

//same as above -- 'or' instead of ||
template<typename T>
concept printable_or_streamable =
     requires printable<T> or output_streamable<T>;

template<typename T>
concept fully_outputable =
     requires printable<T> and output_streamable<T>;
```

more constraint composition

```
// Type T has print ( std::ostream& ) member function
template<typename T>
concept printable = requires(std::ostream& os, T v)
{
  v.print( os ) ;
  std::moveable<T>;
  typename T::format; //declare a type called format
};

//same as above
template<typename T>
concept printable =
  std::moveable<T> and
  requires(std::ostream& os, T v)
  {
    v.print( os ) ;
    typename T::format;
   };
```

std example: `derived_from`

```
template< class Derived, class Base >
concept derived_from =
  std::is_base_of_v<Base, Derived> and
  std::is_convertible_v<const volatile Derived*, const volatile Base>
```

`is_arithmetic`

- it's really a trait that says type has plus, minus, etc
- unfortunately `char` and `bool` are in the group

example concept `number`

```
#include <concepts>
template<typename T, typename U>
concept not_same_as = not std::is_same_v<T, U>;

static_assert( not_same_as<int, double> );

template<typename T>
concept number =
    not_same_as<bool, T> and
    not_same_as<char, T> and
    std::is_arithmetic_v<T>;

static_assert( number<int> );
static_assert( number<double> );
static_assert( !number<bool> );
```

ranges and concepts

```
std::vector<int> vi{ 0, 1, 2, 3, 4, 5, 6 };

auto is_even = [](int i) { return 0 == i % 2; };

for (int i : ranges::filter_view( vi, is_even ))
{
  std::cout << i << " "; //0 2 4 6
}
```

std::ranges::filter_view

```
template<input_range V, indirect_unary_predicate<iterator_t<V>> Pred>
        requires view<V> && is_object_v<Pred>
        class filter_view : public view_interface<filter_view<V, Pred>>

  // public view_interface<filter_view<V, Pred>> <--- ????
```

std::ranges::view_interface

```
template<class D>
requires is_class_v<D> && same_as<D, remove_cv_t<D>>
class view_interface;
```

std::view_interface

```
template<class D>
requires is_class_v<D> && same_as<D, remove_cv_t<D>>
class view_interface
{
    constexpr const D& derived () const noexcept
    {
       return static_cast<const D&>(*this);
    }

    //concept based specialization of operator[]
    //only applies if subclass is random_access_range
    template<random_access_range R = const D>
    constexpr decltype(auto) operator[](range_difference_t<R> n) const
    {
       return ranges::begin(derived())[n];
    }
};
```

Writing Concepts

> *"Everything should be made as simple as possible, but not simpler" -Albert Einstein*

how constraints are evaluated

- normalize -> substitute all concepts bodies into – now list of atomic constraints with and/ors
- subsumption -> defines a partial ordering of constraints
- A constraint P subsumes constraint Q
  - if it can be proven that P implies Q
  - up to the identity of atomic constraints in P and Q
- concepts subsume, arbitrary expressions do not
- general principle is 'more constrained' is better match

good concepts and bad concepts

- single operation concepts `->` questionable
  - Addable `->` likely a poor concept, try Number
- good concepts express more than `just` what an algorithm needs
- typically based on analysis of the domain
- operations come in groups
  - numbers: plus, minus, multiply, etc
  - containers: insert, erase, iteration

evolving concepts

- unlikely to get it right on first attempt
- built up in iterations
- lots of recompiling of code as concepts change

concepts for `sleep_for`

durations and time_points

```
//this thread of execution sleeps for duration d
void sleep_for( time_duration d );

//this thread of execution sleeps until t
void sleep_until( time_point t );
```

using `sleep_for`

The following is an example of using this api:

```
//https://godbolt.org/z/3vreqf
std::this_thread::sleep_for(std::chrono::millisecond(2000));
std::this_thread::sleep_for(std::chrono::seconds(2));
```

the real `sleep_for` signature

```
template< class Rep, class Period >
void sleep_for( const std::chrono::duration<Rep, Period>& sleep_duratio

template< class Clock, class Duration >
void sleep_until( const std::chrono::time_point<Clock,Duration>& sleep_
```

breaking the grip of types

- how do we make this code work?

```
std::this_thread::sleep_for(boost::posix_time::millisecond(2000));
std::this_thread::sleep_for(boost::posix_time::seconds(2));
```

reverse enginering the needs

```
//gcc 2021-01-17 version of sleep_for cleaned up for nanosleep only ver
//https://github.com/gcc-mirror/gcc/blob/master/libstdc++-v3/include/st
/// this_thread::sleep_for
template<typename _Rep, typename _Period>
  inline void
  sleep_for(const chrono::duration<_Rep, _Period>& __rtime)
  {
    if (__rtime <= __rtime.zero())
      return;
    auto __s = chrono::duration_cast<chrono::seconds>(__rtime);
    auto __ns = chrono::duration_cast<chrono::nanoseconds>(__rtime - __

    struct ::timespec __ts =
      {
        static_cast<std::time_t>(__s.count()),
        static_cast<long>(__ns.count())
      };
    while (::nanosleep(&__ts, &__ts) == -1 && errno == EINTR)
```

`time_duration` requirements

So the requirements as written are:

- a constant `zero` member function to return the zero value
- a comparison operator (less equal)
- ability to cast/retrieve the seconds and milliseconds of the duration
- a constant `count` function that casts to `long` and `std::time_t`

concept `time_duration`

```
//https://godbolt.org/z/1Tvefcasb
#include <concepts>
#include <chrono>

template <class D>
concept time_duration =
   std::totally_ordered<D> and
   requires(const D& v)
{
   v.count();
   v.zero();
};

static_assert( time_duration<std::chrono::seconds> );
```

constraining the representation type

```
#include <concepts>
#include <chrono>

template <class D>
concept time_duration =
  //D::rep is duration internal representation type
  not std::is_floating_point_v<typename D::rep> and
      std::totally_ordered<D> and
      requires(const D& v)
{
   v.count();
   v.zero();
};

static_assert( time_duration<std::chrono::seconds> );

int main()
{}
```

first draft of `sleep_for`

```
//https://godbolt.org/z/W5odTv

template <class T>
concept time_duration =
  std::totally_ordered<T> and
  requires(const T& v)
{
   v.count();
   v.zero();
};
static_assert( time_duration<std::chrono::seconds> );

void sleep_for( time_duration auto td )
{
  std::cout << "hello ";
  std::this_thread::sleep_for( td );
  std::cout << "there\n";
}

int main()
{
  sleep_for( std::chrono::seconds(2) );
}
```

now for boost `time_duration`

```
//https://godbolt.org/z/W53PGP
#include <boost/date_time.hpp>


void sleep_for( time_duration auto td)
{
  std::cout << "hello ";
  std::this_thread::sleep_for( td );
  std::cout << "there\n";
}

namespace bpt = boost::posix_time;

int main()
{
  sleep_for( bpt::seconds(2) );
}
```

compiler error with boost `time_duration`

```
27:30: error: use of function 'void sleep_for(auto:19) [with auto:19 =
   27 |    sleep_for( bpt::seconds(2) );
      |                                 ^
<source>:16:6: note: declared here
   16 | void sleep_for( time_duration auto td)
      |      ^~~~~~~~~
<source>:16:6: note: constraints not satisfied
<source>: In instantiation of 'void sleep_for(auto:19) [with auto:19 =
<source>:27:30:   required from here
<source>:8:9:   required for the satisfaction of 'time_duration<auto:19
<source>:8:52:   in requirements with 'const T& v' [with T = boost::pos
<source>:10:11: note: the required expression 'v.count()' is invalid
   10 |     v.count();
      |     ~~~~~~~^~
<source>:11:10: note: the required expression 'v.zero()' is invalid
   11 |     v.zero();
      |     ~~~~~~^~
cc1plus: note: set '-fconcepts-diagnostics-depth=' to at least 2 for mo
ASM generation compiler returned: 1
```

refactoring the `sleep_for` concept

duration concept refactored to `time_duration_access`

```
//https://godbolt.org/z/WW1dGM
template<class T>
concept time_duration_access = requires(const T& v)
{
   v.count();
   v.zero();
};

template <class T>
concept time_duration =
    std::totally_ordered<T> and
    time_duration_access<T>;

static_assert( time_duration<std::chrono::seconds> );

void sleep_for( time_duration auto td)
{
  std::cout << "hello ";
  std::this_thread::sleep_for( td );
  std::cout << "there\n";
}
```

```cpp
int main()
{
  sleep_for( std::chrono::seconds(2) );
}
```

duration concept with alternate duration access

```
//https://godbolt.org/z/7eqexh
#include <boost/date_time.hpp>
template<class T>
concept std_time_duration_access = requires(const T& v)
{
   v.count();
   v.zero();
};

template<class T>
concept boost_time_duration_access = requires(const T& v)
{
   v.total_milliseconds();
};

template <class T>
concept time_duration =
    std::totally_ordered<T> and
    (std_time_duration_access<T> or boost_time_duration_access<T>);

static_assert( time_duration<std::chrono::seconds> );
```

duration concept with alternate duration access

```
//https://godbolt.org/z/7eqexh

void sleep_for( time_duration auto td)
{
  std::cout << "hello ";
  if constexpr (std_time_duration_access<decltype(td)>) { //compile tim
    std::this_thread::sleep_for( td );
  }
  if constexpr (boost_time_duration_access<decltype(td)>) {
    auto d = std::chrono::milliseconds(td.total_milliseconds());
    std::this_thread::sleep_for( d );
  }
  std::cout << "there\n";
}

namespace bpt = boost::posix_time;

int main()
{
  sleep_for( std::chrono::seconds(2) );
  sleep_for( bpt::seconds(2) );
}
```

final thoughts on `sleep_for`

- demonstrates how difficult to get right
- requires lots of thought and experimentation
- refactoring 'to concepts'
    - promising way to evolve libraries
    - std::chrono could probably be opened up

Designing with Concepts

> *"If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization." - Gerald Weinberg (~1975)*

what is design?

- c++ supports 'multi-paradigm' design
  - structured
  - functional
  - generic
  - object oriented
- are these really that different?
- what's the real issue?

some ’design principles’

- DRY versus WET - don't repeat yourself OR write everything twice
- SOLID - see also keynote
  - single responsibility principle
  - open - closed principle
  - Liskov substitution principle
  - interface segregation principle
  - dependency inversion principle
- KISS - keep it simple stupid (aka Occam's razor)

divide and conquer (decomposition)

- need ways to break programs into manageable parts
- parts that can be tested
- parts that can be reasoned about
- separate concerns

divided programs are dependent

- part A depends on part B
- part B also depends on part A
- new design principle!
  - co-dependency is bad!
  - smelly big ball of mud!
  - consult and gain riches!

divided programs are dependent II

- divided monoliths need to be re-composed
- parts put together
- dependencies understood
- many of the 'principles' are about dependencies

concepts and dependencies

- move dependency to an abstraction from a type
- simple to test a type models a concept
- problems are just shifted
  - type may evolve to no longer model concept
    - working code now fails
  - concept may evolve so type no longer models
    - working code now fails

code readability and evolution

```
auto result = some_function(); //return type unknown, flexible

int result = some_function(); //return type obvious, brittle

time_duration auto result = some_function(); //flexible+clear
```

type, auto, or <concept_name> auto

- type expresses only one type can be returned
  - inflexible if return type evolves
- full auto means we really don't care
  - also means we don't know
  - still need recompile on change
- <concept_name> auto means a set of types
  - arguably a good sweet spot
  - allows for bounded evolution

Liskov Substitution - concepts vs inheritance

- principle that describes an inheritance rule
  - subtypes need have same pre/post-conditions as base type
  - behavior versus meaning
- concepts are similar without derivation
  - type that models a concept can be 'substituted'
  - concepts don't model function pre-conditions (aka contracts)
  - similar idea for generic programming

Parnas - information hiding (1972)

- ACM paper: On the Criteria To Be Used in Decomposing Systems into Modules
- Goals: independent development, composition, understandability

Examples:

- "A data structure, its internal linkings, accessing procedures and modifying procedure"
- "The sequence of instructions necessary to call a given routine and the routine itself are part of the same module."

information hiding and concepts

- concepts make needed interfaces concrete
- while hiding type specifics like storage
- while still allowing for type variations
- ranges example: algorithms over data structures
  - range required might be simply a `forward_range`
  - or maybe it's `contiguous_range`
  - provides separation of the algorithm from structure
  - but ensures performance

Coplien - multi-paradigm design (1999)

> *"most designs in fact have a nontrivial component that is not object-oriented" - James O. Coplien*

commonality and variability analysis

- "good systems, commonality captures mechanism and variability captures policy" (p24)
- "dimensions of commonality include: structure, name and behavior, algorithm" (p42)
- commonalities identify "abstractions that will remain stable over time" (p60)

name and behavior commonality

- "use commonality of name to group items (such as functions)" (p48)
- a name often defines a common behavior
- behavior versus meaning
  - each overloaded or **specialized** function `should` have different behavior
  - meaning to the calling client should be the same

implementing commonality and variability

- commonality techniques
  - factor commonalities into a base class
  - factor policy into traits (policy based design)
  - value oriented programming - vocabulary types
  - factor commonalities to **concepts**
- variation tools
  - pre-processor (build)
  - inheritance (build or run)
  - templates (build)
  - overloading (build)
  - now **concepts** (build)

multi-paradigm design and concepts

- concepts express a set of types
- 'if a type' `models` a concept then it is part of the set
- concepts can express type variations and relations
- negative variability
  - can be expressed with `requires` on type methods
  - a class member can be removed if type doesn't support
- positive variability
  - a type the models a concept can add behaviors/methods as desired

concept serialization

- how many familiar with serialization pattern
- old design pattern - originally OO only
- by boost.serialization design is mostly template

serialization 101- key abstractions

- Archive - a type that encapsulates data formatting for device
  - input and output archives
  - concrete examples: xmlInput, xmlOutput, jsonInput, rdbOutput, protoBufOutput
- Serializeable - types that have encapsulated data
- double dispatch (visitor) between archive and serializable
- type factory for input

properties of the design

- type data is nicely encapsulated
  - method to serialize is in the type
- archive format is nicely encapsulated
  - it only knows about fundamental types
- separation of concerns

serializable

```
class myType {
   int foo;
   string bar;
   vector<int> baz;

   //one method for both input/output
   template<class Archive>
   serialize(Archive& archive)
   {
      ar( "foo", foo );
      ar( "bar", bar );
      ar( "baz", baz );
   }

};
```

archive types

```
class OutputArchive
{
   put(string name, const string s);
   put(string name, int i);
   put(string name, double d);
   ...
};
```

problems

- no base type, so depend on docs for archive form
- compile errors are ugly
- need external extensions for collection types

conceptification

```
template<class T, class A>
concept serializable = requires(T val, A archive)
{
   val.archive<A>();
};

template<class A>
concept Archive = //...

template<class A>
concept OutputArchive = Archive<A> and requires(A archive)
{
   put(string, int);
   put(string, double);
   put(string, string)
   ...
};
```

advantages?

- looks quite doable
- fixes the docs/compile issues
  - static assert your archive type
- allows factoring of other subtle policies
  - archive ordering
  - devices like files or database also become policies
- template aliases can help with collections

Conclusions & Resources

is concepts everything we want?

- NO!
- modern generic designs depend on
  - customization point objects
  - tagged invoke
  - these express desired variabilties
  - none are as clear and obvious as a virtual method
- P2279R0 We need a language mechanism for customization points, Barry Revzin
  - compares Rust traits to current techniques
  - some of what we need was in c++0x concepts

final thoughts

- concepts are a powerful new tool in c++20
- adoption will take time
- best practices will take time
- hopefully motivated to explore the space

final thoughts

- concepts are a powerful new tool in c++20
- adoption will take time
- best practices will take time
- hopefully motivated to explore the space

> *"I can taste mythical fountains. False hope, perhaps. But the truth never got in my way." – Tool, Invincible*

papers, blogs

- Concept Design for STL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3351.pdf
- std library concepts http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0898r3.p
- https://www.cppfiddler.com/2019/06/09/concept-based-interfaces/
- https://en.cppreference.com/w/cpp/language/constrai
- Working Draft, C++extensions for Concepts

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4553.pdf http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0734r0.pdf

- on return type conventions

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1452r2.htm

- Parnas information hiding 1972: http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.132.7232

videos

- Stroustrup 2018 keynote cppcon
  https://www.youtube.com/watch?v=HddFGPTAmtU
- Sutton 2018 talk reference
  https://www.youtube.com/watch?v=ZeU6OPaGxwM
- CppCon 2018: Arthur O'Dwyer "Concepts As She Is Spoke"

  https://www.youtube.com/watch?v=CXn02MPkn8Y

- Jason Turner short video on if constexpr and requires https://www.youtube.com/watch?v=sy32kAtsIKg

- Sar Raaz C++20 Concepts: A Day in the Life

CppCon 2019 https://www.youtube.com/watch?v=qawSiMIXtE4