Concepts: A Practioners Guide

Jeff Garland

Intro

"Where any answer is possible, all answers are meaningless." - Isaac Asimov

talk goals: climb up the concept ladder

- what's a concept
- using concepts in code
- reading & writing concepts: requires expressions & clauses
- designing with concepts

tools

- presentation involves lots of examples
- all modern compilers/libraries have excellent support
- godbolt links in the materials

Concept Basics

why do we want concepts?

- want to be able to write good generic libraries
- that are fast
- with reasonable error messages
- that our fellow programmers can understand and maintain
- **better interfaces**
    - more descriptive
    - better dependency management

what's a c++ concept?

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`

```
1  namespace io {
2    // Type T has print( std::ostream& ) const member function
3    template<class T>
4    concept printable = requires(std::ostream& os, T v)
5    {
6      v.print( os ) ; //<--an expression that if compiles yields true
7    };
8  }
9  class my_type
10 {
11   std::string s = "foo\n";
12 public:
13   void print( std::ostream& os ) const
14   {
15     os << "s: " << s;
16   }
17 };
18 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?

concepts 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'

concepts 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 type name 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)

concept usage: basic examples

```
1  template<class T>
2  concept printable = requires(std::ostream& os, T v)
3  {
4    v.print( os ) ; //<--an expression that if compiles yields true
5  }
6
7  void f(printable auto s) {...};
8
9  main {
10     printable auto s;
11 }
```

concept usage: error!

```
<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: initialize variable

```
1 main {
2
3     printable auto s = init_some_thing();
4 }
```

concept usage: function parameter or return value

```
1  template<printable T>
2  printable auto
3  print( const T& s )
4   {
5   //...
6   return s;
7  };
8
9  printable auto
10 print2( const printable auto& s )
11 {
12  //...
13  return s;
14 }
```

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

overload resolution - constrain function parameter

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

auto for function parameter, unconstrained

```
1  // auto parameter -- this is a template function!
2  // template<typename T_
3  // void print_ln( T p )
4  void print_ln( auto p )
5  {
6   std::cout << p << "\n";
7  }
8
9  class my_type {};
10
11 int main()
12 {
13     print_ln( "foo" );
14     print_ln( 100 );
15
16     my_type m;
17     print_ln ( m ); //compile error
18 }
```

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

auto for function parameter, unconstrained

```
1  // auto parameter -- this is a template function!
2  // template<typename T_
3  // void print_ln( T p )
4  void print_ln( auto p )
5  {
6   std::cout << p << "\n";
7  }
8
9  class my_type {};
10
11 int main()
12 {
13     print_ln( "foo" );
14     print_ln( 100 );
15
16     my_type m;
17     print_ln ( m ); //compile error
18 }
```

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

concrete overload `print_ln` for `my_type`

```
1  void print_ln( auto p )
2  {
3   std::cout << p << "\n";
4  }
5
6  //selected ahead of print_ln (auto) because better match
7  void print_ln( my_type p )
8  {
9   p.print( std::cout ); std::cout  << "\n";
10 }
11
12 int main()
13 {
14     print_ln( "foo" );
15     print_ln( 100 );
16
17     //good
18     my_type m;
19     print_ln ( m );
20 }
```

- https://godbolt.org/z/85cEdqMEc

concrete overload `print_ln` for `my_type`

```
1  void print_ln( auto p )
2  {
3   std::cout << p << "\n";
4  }
5
6  //selected ahead of print_ln (auto) because better match
7  void print_ln( my_type p )
8  {
9   p.print( std::cout ); std::cout  << "\n";
10 }
11
12 int main()
13 {
14     print_ln( "foo" );
15     print_ln( 100 );
16
17     //good
18     my_type m;
19     print_ln ( m );
20 }
```

- https://godbolt.org/z/85cEdqMEc

concepts `printable` and `output_streamble`

```
1  // Type T has print ( std::ostream& ) member function
2  template<typename T>
3  concept printable = requires(std::ostream& os, T v)
4  {
5    v.print( os ) ; //<--an expression that if compiles yields true
6  };
7
8  template<class T>
9  concept output_streamable = requires (std::ostream& os, T v)
10 {
11    os << v;
12 };
```

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

concepts `printable` and `output_streamble`

```
1  // Type T has print ( std::ostream& ) member function
2  template<typename T>
3  concept printable = requires(std::ostream& os, T v)
4  {
5    v.print( os ) ; //<--an expression that if compiles yields true
6  };
7
8  template<class T>
9  concept output_streamable = requires (std::ostream& os, T v)
10 {
11    os << v;
12 };
```

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

a type satisfying `printable`

```
1  class my_type
2  {
3    int i = 1;
4    std::string s = "foo\n";
5
6  public: //<-- concept not satisfied if print isn't public
7
8    void print( std::ostream& os) const
9    {
10     os << "i: " << i << " s: " << s;
11   }
12 };
13 static_assert( printable<my_type> );
14
15 class my_type2 {};
```

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

a type satisfying `printable`

```
1  class my_type
2  {
3    int i = 1;
4    std::string s = "foo\n";
5
6  public: //<-- concept not satisfied if print isn't public
7
8    void print( std::ostream& os) const
9    {
10     os << "i: " << i << " s: " << s;
11   }
12 };
13 static_assert( printable<my_type> );
14
15 class my_type2 {};
```

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

constrained overload for `print_ln`

```
1  void print_ln( auto p )
2  {
3   std::cout << p << "\n";
4  }
5
6  //auto parameter -- this is a template function!
7  void print_ln( printable auto p ) //<-- constrained resolution
8  {
9    p.print( std::cout );
10   std::cout << "\n";
11 }
12
13 int main()
14 {
15     print_ln( "foo" );
16     print_ln( 100 );
17
18     my_type m;
19     print_ln ( m );
20 }
```

- https://godbolt.org/z/36cdsGzzo

constrained overload for `print_ln`

```
 1 void print_ln( auto p )
 2 {
 3  std::cout << p << "\n";
 4 }
 5
 6 //auto parameter -- this is a template function!
 7 void print_ln( printable auto p ) //<-- constrained resolution
 8 {
 9   p.print( std::cout );
10   std::cout << "\n";
11 }
12
13 int main()
14 {
15     print_ln( "foo" );
16     print_ln( 100 );
17
18     my_type m;
19     print_ln ( m );
20 }
```

- https://godbolt.org/z/36cdsGzzo

overloaded functions

```
1  // example of overload resolution
2  void print_ln( output_streamable auto p)
3  {
4    std::cout << p << "\n";
5  }
6
7  void print_ln( printable auto p)
8  {
9    p.print(std::cout);
10   std::cout << "\n";
11 }
12
13 class my_type2 {};
14
15 int main()
16 {
17     print_ln( "foo" );
18     my_type m;
19     print_ln ( m );
20     //compile error of course
```

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

overloaded functions

```
1  // example of overload resolution
2  void print_ln( output_streamable auto p)
3  {
4    std::cout << p << "\n";
5  }
6
7  void print_ln( printable auto p)
8  {
9    p.print(std::cout);
10   std::cout << "\n";
11 }
12
13 class my_type2 {};
14
15 int main()
16 {
17     print_ln( "foo" );
18     my_type m;
19     print_ln ( m );
20     //compile error of course
```

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

overloaded functions

```
1  // example of overload resolution
2  void print_ln( output_streamable auto p)
3  {
4    std::cout << p << "\n";
5  }
6
7  void print_ln( printable auto p)
8  {
9    p.print(std::cout);
10   std::cout << "\n";
11 }
12
13 class my_type2 {};
14
15 int main()
16 {
17     print_ln( "foo" );
18     my_type m;
19     print_ln ( m );
20     //compile error of course
```

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

pointers and concepts

- useful for things like factory functions

```
1  //ouput:
2  //s: foo
3  //s: foo
4
5  // based on
6  //https://godbolt.org/z/d7bGhn
7
8  int main()
9  {
10    const printable auto* m = new my_type();
11    m->print(std::cout);
12    const std::unique_ptr<printable auto> upm = std::make_unique<my_type>();
13    upm->print(std::cout);
14 }
```

pointers and concepts

- useful for things like factory functions

```
1  //ouput:
2  //s: foo
3  //s: foo
4
5  // based on
6  //https://godbolt.org/z/d7bGhn
7
8  int main()
9  {
10    const printable auto* m = new my_type();
11    m->print(std::cout);
12    const std::unique_ptr<printable auto> upm = std::make_unique<my_type>();
13    upm->print(std::cout);
14 }
```

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 constraints
//    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 = whatever]
// <source>:8:10: note: the required expression 'v.print(os)' is invalid
//     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

```
1  template<class T>
2  std::ostream&
3  print_ln( std::ostream& os, const T& v )
4   {
5    //if constexpr ( requires{ printable<T>; } ) more verbose
6    if constexpr ( printable<T> )   //<-- shorter version
7    {
8      v.print(os);
9    }
10   else { //no print function
11     os << v;
12   }
13   os << "\n";
14   return os;
15 }
16 int main()
17 {
18    my_type m;
19    print_ln( std::cout, m );
20    int i = 100;
```

- https://godbolt.org/z/nsojqK5e1

if constexpr and printable

```cpp
template<class T>
std::ostream&
print_ln( std::ostream& os, const T& v )
 {
  //if constexpr ( requires{ printable<T>; } ) more verbose
  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;
```

- https://godbolt.org/z/nsojqK5e1

if constexpr and printable

```cpp
template<class T>
std::ostream&
print_ln( std::ostream& os, const T& v )
 {
  //if constexpr ( requires{ printable<T>; } ) more verbose
  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 );
```

- https://godbolt.org/z/nsojqK5e1

`if constexpr` requires expression

```
1  template<class T>
2  std::ostream&
3  print_ln( std::ostream& os, const T& v )
4   {
5    if constexpr ( requires{ v.print( os ); } )
6    {
7      v.print(os);
8    }
9    else { //no print function
10     os << v;
11   }
12   os << "\n";
13   return os;
14 }
15 int main()
16 {
17    my_type m;
18    print_ln( std::cout, m );
19    int i = 100;
20    print_ln( std::cout, i );
```

- https://godbolt.org/z/nsojqK5e1

`if constexpr` requires expression

```cpp
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 );
}
```

- https://godbolt.org/z/nsojqK5e1

non-template member function of template class

```
1  https://godbolt.org/z/e6zWPqYzh
2  template<class T>
3  class wrapper
4  {
5    T val_;
6    public:
7     wrapper(T val) : val_(val) {}
8     T operator*() requires is_pointer_v<T> // <-- type trait
9     { return val_; }
10 };
11 int main()
12 {
13   int i = 1;
14   wrapper<int*> wi{&i};
15   cout << *wi << endl;
16
17   //no match for operator*
18   //wrapper<int> wi2{i};
19   //cout << *wi2 << endl;
20 }
```

non-template member function of template class

```
1  https://godbolt.org/z/e6zWPqYzh
2  template<class T>
3  class wrapper
4  {
5    T val_;
6    public:
7     wrapper(T val) : val_(val) {}
8     T operator*() requires is_pointer_v<T> // <-- type trait
9     { return val_; }
10 };
11 int main()
12 {
13   int i = 1;
14   wrapper<int*> wi{&i};
15   cout << *wi << endl;
16
17   //no match for operator*
18   //wrapper<int> wi2{i};
19   //cout << *wi2 << endl;
20 }
```

non-template member function of template class

```
1  https://godbolt.org/z/e6zWPqYzh
2  template<class T>
3  class wrapper
4  {
5    T val_;
6    public:
7     wrapper(T val) : val_(val) {}
8     T operator*() requires is_pointer_v<T> // <-- type trait
9     { return val_; }
10 };
11 int main()
12 {
13   int i = 1;
14   wrapper<int*> wi{&i};
15   cout << *wi << endl;
16
17   //no match for operator*
18   //wrapper<int> wi2{i};
19   //cout << *wi2 << endl;
20 }
```

constraining existing templates

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

template alias with concept shorthand

```
1  //template alias using concepts
2  //All elements must satisfy concept printable
3  template<printable T> using
4  vec_of_printable = std::vector<T>;
5
6  int main()
7  {
8    vec_of_printable<my_type> vp{ {}, {} };
9    for ( const auto& e : vp )
10   {
11     e.print(std::cout);
12   }
13 }
```

template alias with concept shorthand

```
1  //template alias using concepts
2  //All elements must satisfy concept printable
3  template<printable T> using
4  vec_of_printable = std::vector<T>;
5
6  int main()
7  {
8    vec_of_printable<my_type> vp{ {}, {} };
9    for ( const auto& e : vp )
10   {
11     e.print(std::cout);
12   }
13 }
```

template alias error message

```
1  //template alias using concepts
2  //All elements must satisfy concept printable
3  template<printable T> using
4  vec_of_printable = std::vector<T>;
5
6  vec_of_printable<int> vp;
7
8  //compile error
9  // template_alias.cpp:16:21: error: template constraint
10 //                                                failure for ‘template<class T>
11 //     requires  printable<T> using vec_of_printable = std::vector<T>’
12 //    16 | vec_of_printable<int> vp; //compile error
13 //       |                     ^
14 // template_alias.cpp:16:21: note: constraints not satisfied
15 // template_alias.cpp:6:9:   required for the satisfaction of
16 //                                                 ‘printable<T>’ [with T = int]
17 // template_alias.cpp:6:21:   in requirements with ‘std::ostream& os’, ‘const T& v’
18 //                                                 [with T = int]
19 // template_alias.cpp:8:10: note: the required expression ‘v.print(os)’ is invalid
20 //     8 |   v.print( os ) ;
21 //       |   ~~~~~~~^~~~~
```

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

```
1  //trivial with ~std::regular~ concept and ~static_assert~
2  #include <string>
3  #include <concepts>
4
5  class my_type
6  {
7    std::string s = "foo\n";
8  public:
9    void print( std::ostream& os ) const
10   {
11     os << "s: " << s;
12   }
13 };
14
15 static_assert( std::regular<my_type> );
```

enforcing regular error

```
1  //   008a_regular.cpp:20:21: error: static assertion failed
2  //    20 | static_assert( std::regular<my_type> );
3  //       |                ~~~~~^~~~~~~~~~~~~~~~~
4  // 008a_regular.cpp:20:21: note: constraints not satisfied
5  // In file included from /usr/include/c++/10/compare:39,
6  //                  ...
7  //                  from /usr/include/c++/10/iostream:39,
8  //                  from 008a_regular.cpp:2:
9  // /usr/include/c++/10/concepts:280:15:
10 //           required for the satisfaction of '__weakly_eq_cmp_with<_Tp, _Tp>'
11 //           [with _Tp = my_type]
12 // /usr/include/c++/10/concepts:290:13:
13 //           required for the satisfaction of 'equality_comparable<_Tp>'
14 //           [with _Tp = my_type]
```

enforcing regular error

```
1  //   008a_regular.cpp:20:21: error: static assertion failed
2  //    20 | static_assert( std::regular<my_type> );
3  //       |                ~~~~~^~~~~~~~~~~~~~~~~
4  // 008a_regular.cpp:20:21: note: constraints not satisfied
5  // In file included from /usr/include/c++/10/compare:39,
6  //                   ...
7  //                  from /usr/include/c++/10/iostream:39,
8  //                  from 008a_regular.cpp:2:
9  // /usr/include/c++/10/concepts:280:15:
10 //           required for the satisfaction of ‘__weakly_eq_cmp_with<_Tp, _Tp>’
11 //           [with _Tp = my_type]
12 // /usr/include/c++/10/concepts:290:13:
13 //           required for the satisfaction of ‘equality_comparable<_Tp>’
14 //           [with _Tp = my_type]
```

enforcing regular fix

```
1  class my_type
2  {
3    std::string s = "foo\n";
4  public:
5    void print( std::ostream& os ) const
6    {
7      os << "s: " << s;
8    }
9    //added this line
10   bool operator==( const my_type& ) const = default;
11 };
12
13 static_assert( std::regular<my_type> );
```

using range concepts

```
1 void print_ints( const std::ranges:range auto& R )
2 {
3   for ( auto i :  R )
4   {
5      std::cout << i << endl;
6   }
7 }
```

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

using range concepts II

```
1  //this function works on all the types below
2  void print_ints( const std::ranges:range auto& R ) {...}
3
4  int main()
5  {
6    vector<int> vi = { 1, 2, 3, 4, 5 };
7    print_ints( vi );
8
9    array<int, 5> ai = { 1, 2, 3, 4, 5 };
10   print_ints( ai );
11
12   span<int>     si2 ( ai );
13   print_ints( si2 );
14
15   int cai[] = { 1, 2, 3, 4, 5 };
16   span<int>     si3( cai );
17   print_ints( si3 );
18
19   ranges::iota_view iv{1, 6};
20   print_ints( iv );
```

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

Reading & Writing 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 expressions
- expression -> syntax for describing type constraints

`requires` expression basics

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

```
1 requires { requirement-sequence }
2 requires ( ..parameters..) { requirement-sequence }
3
4 //simplest possible boolean example no parameters
5 template<typename T>
6 concept always = true;
```

`requires` expression more realistic example

```
1  // Type T has print ( std::ostream& ) member function
2  template<typename T>
3  concept printable = requires(std::ostream& os, T v)
4  {
5    //this is a conjuntion AND --> all must be true
6    v.print( os ) ;  //member function
7    format ( v ) ;   //free function
8    std::movable<T>;
9    typename T::format; //declare a type called format
10 };
11
12 template<class T>
13 concept output_streamable = requires (std::ostream& os, T v)
14 {
15    //constraint on return from operator<<
16    { os << v } -> std::same_as<std::ostream&>;
17 };
```

constraint composition

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

constraint composition example

```
1  //disjunction
2  template<typename T>
3  concept printable_or_streamable =
4       requires printable<T> || output_streamable<T>;
5
6  //same as above -- 'or' instead of ||
7  template<typename T>
8  concept printable_or_streamable =
9       requires printable<T> or output_streamable<T>;
10
11 template<typename T>
12 concept fully_outputable =
13      requires printable<T> and output_streamable<T>;
```

more constraint composition

```
1  template<typename T>
2  concept printable = requires(std::ostream& os, T v)
3  {
4    v.print( os ) ;
5    std::moveable<T>;
6    typename T::format; //declare a type called format
7  };
8
9  //same as above
10 template<typename T>
11 concept printable =
12   std::moveable<T> and
13   requires(std::ostream& os, T v)
14   {
15     v.print( os ) ;
16     typename T::format;
17    };
```

std example: `derived_from`

```
1 template< class Derived, class Base >
2 concept derived_from =
3   std::is_base_of_v<Base, Derived> and
4   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

```
 1 #include <concepts>
 2 template<typename T, typename U>
 3 concept not_same_as = not std::is_same_v<T, U>;
 4
 5 static_assert( not_same_as<int, double> );
 6
 7 template<typename T>
 8 concept number =
 9     not_same_as<bool, T> and
10     not_same_as<char, T> and
11     std::is_arithmetic_v<T>;
12
13 static_assert( number<int> );
14 static_assert( number<double> );
15 static_assert( !number<bool> );
```

ranges and concepts

```
1 std::vector<int> vi{ 0, 1, 2, 3, 4, 5, 6 };
2
3 auto is_even = [](int i) { return 0 == i % 2; };
4
5 for (int i : ranges::filter_view( vi, is_even ))
6 {
7   std::cout << i << " "; //0 2 4 6
8 }
```

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];
    }
};
```

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?

divide and conquer (decomposition)

- need ways to break programs into manageable parts
- parts that can be tested
- parts that can be reasoned about
- separates concerns
- when programs are divided - dependencies created

divided programs are dependent

- divided monoliths need to be re-composed
- parts put together
- dependencies understood
- many 'design 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 changed
    - 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

breaking the grip of type dependency

```
1  //this function works on all the types below
2  void print_ints( const std::ranges:range auto& R ) {...}
3
4  int main()
5  {
6    vector<int> vi = { 1, 2, 3, 4, 5 };
7    print_ints( vi );
8
9    array<int, 5> ai = { 1, 2, 3, 4, 5 };
10   print_ints( ai );
11
12   span<int>     si2 ( ai );
13   print_ints( si2 );
14
15   int cai[] = { 1, 2, 3, 4, 5 };
16   span<int>     si3( cai );
17   print_ints( si3 );
18
19   ranges::iota_view iv{1, 6};
20   print_ints( iv );
```

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

Final thoughts

- concepts are a powerful new tool in c++20
- practical and easy to use
- alter designs in important ways
- integrated into many aspects of the standard libray
- C++Now 2021 talk
    - Part 1: https://www.youtube.com/watch?v=Ffu9C1BZ4-c
    - Part 2: https://www.youtube.com/watch?v=IXbf5lxGtr0