Programming with C++ Constraints:

Background, Utility, and Gotchas

WALTER E. BROWN, PH.D.

< webrown.cpp @ gmail.com >

A little about me

- B.A. (math's); M.S., Ph.D. (computer science).
- Professional programmer for over 50 years, programming in C++ since 1982.
- Experienced in industry, academia, consulting, and research:
  - Founded a Computer Science Dept.; served as Professor and Dept. Head; taught and mentored at all levels.
  - Managed and mentored the programming staff for a reseller.
  - Lectured internationally as a software consultant and commercial trainer.
  - Retired from the Scientific Computing Division at Fermilab, specializing in C++ programming and in-house consulting.
- Not dead — still doing training & consulting. (Email me!)

 2

Emeritus participant in C++ standardization

- Written 125+ papers for WG21, proposing such now-standard C++ library features as gcd/lcm, cbegin/cend, and common_type, as well as the entirety of headers <random> and <ratio>.
- Influenced such core language features as *alias templates*, *contextual conversions*, and *variable templates*; working on *requires-expressions*, comparison operators, and more!
- Conceived and served as Project Editor for *Int'l Standard on Mathematical Special Functions in C++* (ISO/IEC 29124), now incorporated into C++17's <cmath>.
- Be forewarned: Based on my training and experience, I hold some rather strong opinions about computer software and programming methodology — these opinions are not shared by all programmers, but they should be! ☺

Coming soon to a compiler near you! ☺

- Compile-time constraints will soon be part of our routine C++ programming vocabulary:
  - Mainly due to requires, an expected new C++20 keyword …
  - That introduces *requires-clauses* and *requires-expressions*.
- While this syntax is new to C++, the ideas are not, so …
- This talk will explore:
  - What these ideas mean for us at a technical level, and …
  - How they affect our past, present, and future C++ code.
  - (Some material is adapted from my recent WG21 paper "enable_if vs. requires: A Case Study" [wg21.link/p0552].)

Contents

1. The Bigger Picture
2. Pre-Modern Constraints
3. Modern & Post-Modern Constraints
4. A Case Study
5. Lessons Learned
6. Addendum

5

std::disclaimer

- Code samples in this talk:
  - Freely use color and other highlighting techniques, …
  - Mostly ignore proper headers and namespaces, and …
  - May ignore (or subvert) a few other coding niceties.
- Why? In the interests of:
  - Clarity of exposition (my foremost concern), …
  - Coping with screen real estate (a secondary concern), and …
  - Emphasizing my main point(s).

6

1. The Bigger Picture

Instead of freaking out about these constraints, embrace them [and] use them to your advantage.

— Jason Fried

Nomenclature

- What's common to these terms?
  - Precondition
  - Narrow contract
  - Wide contract
  - Constraint
  - *requires-clause*
  - *requires-expression*
- All deal with requirements:
  - That a function or other component imposes, and …
  - That are expected/assumed to be met before use.
- How do they differ?
  - Some requirements affect a component's correct use.
  - Other requirements affect a component's very existence.

Preconditions and contracts

- A function's precondition is a predicate:
  - Expected/assumed true each time the function is called:
    - Typically at run time, but at compile time if constexpr applies.
    - A function that checks its precondition doesn't really have one.
  - If its precondition is false, a function's behavior is undefined; the program (by definition) has a bug (see [res.on.required]/1).
- Preconditions are part of a function's contract with clients:
  - No preconditions ⇒ wide contract.
  - Any preconditions ⇒ narrow contract.
- Tangential to constraints, so not further discussed today:
  - (But keep an eye out for [[expects:]], *etc.*, in a future C++.)

How are constraints different?

- A constraint is a compile-time predicate associated with some program component (*e.g.*, a function or a template):
  - The constraint must be satisfied (evaluated as true), or else!
  - Else the associated component WON'T/MUSTN'T BE COMPILED:
    - But that doesn't always imply that something is wrong!
    - Like preprocessor directives #if, #ifdef, *etc.*, in this regard.
- C++ has always had certain innate constraints:
  - Types impose constraints: *e.g.*, arg's must initialize parm's.
  - Template specialization imposes constraints on its arg's.
- The C++98 era limited us to such implicit techniques.

Newer technologies

- C++11 gave us tools for explicit constraint programming:
  - Via static_assert to force a diagnostic when constraint is false.
  - Via std::enable_if metafunction in a SFINAE context to prevent a template's instantiation (and thus to prevent its compilation).
  - Via *expression SFINAE*, additional contexts in which deduction failures do not result in hard errors.
- Most recently, WG21 added:
  - C++17: std::void_t to support the *detection idiom*.
  - C++17: *constexpr if* and *discarded statements*.
  - C++20 WD: the *Concepts [Lite] TS* requires keyword.

2. Pre-Modern Constraints

There are no constraints …
except those we ourselves erect.

— Ronald Reagan

Statically asserting à la C++98

- Declare something that might, but needn't, be valid:
  - Let the validity depend on the truth of the constraint.
  - Guarantees a diagnostic when the constraint isn't satisfied.
  - Example: `int ensure[ 2 · int(`*constraint*`) − 1 ];`
- Or can apply tag dispatch for this purpose:
  - `template < bool b > struct ensure { };`
  - `int g_impl( T x, ensure<true> ) { return ⋯ ; }`
  - `int g( T x ) { return g_impl( x, ensure<`*constraint*`>( ) ); }`

Using constraints à la C++98

② O(log N) helper (a primary template with 2 partial spec's):

```
template< unsigned N, class T, bool = (N % 2 == 0) >
struct p {      // primary template handles only odd N (see below)
   T operator ( ) ( T x ) { return x · p<N-1, T>{ }( x ); } };
```

```
template< unsigned N, class T >
struct p< N, T, true > {                    // constrains N to be even
   T operator ( ) ( T x ) { return p<N/2, T>{ }( x · x ); } };
```

```
template< class T >
struct p< 0, T, true > {             // constrains N to be 0 (and even)
   T operator ( ) ( T ) { return T(1); } };
```

① Raise a value x (of type T) to the N[th] power:

```
template< unsigned N, class T >
T power( T x ) { return p<N, T>( )( x ); }   // dispatch to helper
```

Constraints à la C++11: the enable_if type trait metafunction

- If true, alias a given type; otherwise, provide no alias:
  - *// primary template assumes the* bool *value is* true*:*
    ```
    template< bool, class T = void >   // default is useful, not essential
    struct  enable_if  {  using type = T;  };
    ```
  - *// partial specialization recognizes a* false *value, so no alias:*
    ```
    template< class T >
    struct  enable_if<false, T>  {  };          // no member named type!
    ```
- Now consider a meta-call enable_if< false, … >::type :
  - Refers to a non-existent member named type.
  - Always an error, right?
  - No, only sometimes an error: SFINAE!

15

Mechanics of SFINAE 1: instantiation

- To instantiate a template's declaration, the compiler:

  ① Obtains (or figures out) each template argument:

  a) Take verbatim if explicitly supplied at template's point of use.

  b) Else deduce from function arguments, if any, at point of call.

  c) Else rely on (C++17) class template's deduction guide(s), if any.

  d) Else take from the declaration's default template arguments, if any.

  ② Replaces each occurrence of a template parameter by its corresponding template argument. (Substitution step.)
- This process has one of two possible outcomes.

16

Mechanics of SFINAE 2: outcomes

- An instantiation attempt may succeed or may fail:
  - ① If substitution produces a well-formed declaration, the instantiation is viable (succeeds), but …
  - ② If the resulting declaration is ill-formed, it is not viable (due to substitution failure) and is discarded w/ no diagnostic.
- In other words:

  SFINAE: Substitution Failure Is Not An Error!

- But what is it good for?

Constraints à la C++11: explicit overload set management

- At most one of the following overloads will be instantiated:
  - Why? At most one of the predicates can be true and …
  - The false predicates lead to ill-formed code that's discarded.
- 3 overloads to raise x (of type T) to the N^th^ power:

```
template< unsigned N, class T >
enable_if_t< N == 0u, T >
  power( T ) { return  T(1); }
```

```
template< unsigned N, class T >
enable_if_t< (N != 0u)  and  N % 2u == 0u, T >  // even, non-0 N
  power( T  x )  { return  power<N/2u, T>{ }( x · x ); }
```

```
template< unsigned N, class T >
enable_if_t< N % 2u != 0u, T >                  // odd N
  power( T  x )  { return  x · power<N-1u, T>{ }( x ); }
```

What's wrong with enable_if and SFINAE?

- Nothing at all. But …
- The idiom does have limitations:
  - It requires a template context.
  - It's more challenging to use when a constructor is involved.
- And the resulting code tends to be verbose/wordy:
  - It's often not very transparent to read/understand.
  - Newer techniques for expressing constraints can be both easier to write and easier to read.

19

3. Modern & Post-Modern Constraints

We are so Post-Modern that we don't realize how Post-Modern we are anymore.

— Larry Wall

Constraints à la C++17: *constexpr if*

- An ordinary if, but evaluated at compile-time:
  - Branches not taken become discarded statements.
  - Each branch, even if discarded, must be parsable.
- Raise x to the Nth power — no dispatching, no overloading:
  - ```
    template< unsigned N, class T >
    T  power( T  x )  {
      if constexpr( N == 0u )   return  T(1);
      else if constexpr( N % 2u == 0u )
        return  power<N/2>( x · x );        // O(log N) multiplications
      else     // N is odd
        return  x  ·  power<N-1>( x );
    }
    ```
- While very useful, *constexpr if* doesn't fit all situations.

21

Constraints à la *Concepts [Lite]*

- *Concepts [Lite] TS* was published in late 2015:
  - Concepts parts weren't quite ready for C++17 integration.
  - Other parts specify syntax/semantics for a *requires-clause*.
- A *requires-clause* may be applied:
  - To any kind of template declaration, but …
  - Also to an ordinary (non-template) function declaration:
    - In which case it's termed a trailing *requires-clause*.

22
    ```

Technical details of requires

- A *requires-clause* induces constraints on its component:
  - These must be satisfied (true at compile time) …
  - To permit the component to be (instantiated and) compiled.
  - An unsatisfied constraint causes its associated component (an instantiation or a function) to be ignored à la SFINAE.
  - Mustn't use any component whose constraint is unsatisfied.
- Subsumption is a new consideration:
  - Applies when one constraint subsumes a second constraint.
  - Gives us the notion of more constrained.
  - Akin to templates' notion of *more specialized*.
- None of this machinery depends on any concept keyword.

 23

Constraints are useful even without templates

- *E.g.*, overloaded functions:
  - void xmit(char buffer[ ]) requires local == endian::big;
  - void xmit(char buffer[ ]) requires local == endian::little;
  - void xmit(char buffer[ ]) requires local == endian::native;
  - Each function implements, say, the transmission protocol appropriate to its supported environment.
- How can overloading work with identical argument lists?
  - The *requires-clause* is now part of the *signature*, so that …
  - Only declarations whose constraint is satisfied will go into a call's set of viable candidates, and …
  - As usual, the other declarations won't be further considered.

 24

A *requires-clause* has several forms

- Can take any compile-time predicate, *e.g.*, a type trait.
- Or can refer to a named set of requirements (a concept):
  - *E.g.*, using a concept C, can write … requires C<T> … .
  - The assoc. constraint is satisfied iff T meets C's specification.
- Or can use an equivalent shorter form:
  - Instead of template< class T > requires C<T> … ,
  - Can write template< C T > … (Stepanov/*Concepts TS* syntax).
- Or can use a *requires-expression* within a *requires-clause*:
  - *E.g.*, … requires requires { e_1 ; e_2 ; … ; } … .
  - The associated constraint is satisfied iff each expression e_i is well-formed.

25

4. A Case Study: Implementing std::swap

Every man lives by exchanging.

— Adam Smith

Overview of the experiment

- Implemented std::swap's specification twice:
  - Once with only enable_if technology, and then …
  - Separately with only requires technology.
  - Identical function bodies; different function introductions.
- Validated each, identically, to ensure conformance:
  - Used established test cases from well-respected public and private sources.
  - Unexpectedly, testing showed a somewhat subtle difference in the two implementations' behavior!

27

std::swap's specification

- Salient excerpts (from [utility.swap]):
  - template < class T >
    void swap(T&, T&) noexcept(⋯);
  - This function "shall not participate in overload resolution unless [T is a movable type]."
- The highlighted formulaic words specify a constraint:
  - Namely, they require SFINAE-like behavior.
  - In this case, conforming implementers must tell the compiler to instantiate std::swap<T> only when T is a movable type.

28

① Satisfying the spec via enable_if technology

- Declaration:
  - ```
    template< typename T >
    enable_if_t< is_movable_v<T>, void >
      swap( T&, T& )  noexcept( … );
    ```
  - The enable_if result is positioned as swap's return type.
- Two possibilities:
  - When the movability requirement holds, the metafunction call yields void (which is taken as swap's return type) and so the resulting function declaration is well-formed and viable.
  - But when that predicate does not hold, swap has no return type, thus forming an ill-formed (hence non-viable) declaration that's discarded due to SFINAE.

 29

② Satisfying the spec via requires technology

- Declaration:
  - ```
    template< typename T >
      requires  is_movable_v<T>
    void
      swap( T&, T& )  noexcept( … );
    ```
  - Has an ordinary return type (namely void), because …
  - Now the requires clause induces the specified constraint.
- Again, two possibilities:
  - When the moveability constraint is satisfied, the (well-formed) declaration will be a viable candidate.
  - But when the constraint is not satisfied, the declaration is considered non-viable and is silently discarded à la SFINAE.

 30

Then came validation

- Consisted of several public and private test program suites.
- All went well, until …
- When tested against is_swappable.pass.cpp (from libc++):
  - ✔ The enable_if-based implementation passed, but …
  - ✘ The requires-based implementation failed!
- So what went wrong?

31

The failing test probes the is_swappable trait

- Relevant excerpt (names shortened):
  - ```
    namespace ns {
      struct A { };
      template< class T > void swap( T&, T& ) { }
    }
    ```
- Consider `ns::A a, b;  using std::swap;  swap( a, b );` :
  - Ordinary (unqualified) name lookup finds std::swap, and …
  - Argument-dependent lookup (ADL) finds ns::swap.
  - Which is better to call? Classic partial ordering says neither!
- `static_assert( ! std::is_swappable_v< ns::A > );`
  - The test passes if A is not swappable — which it isn't, here, due to the expected ambiguity re which function to call.
  - This works fine for enable_if, but not for requires!

32
    ```

So what's different about a *requires-clause*?

- It comes with a new rule (a tie-breaker of sorts):
  - When partial ordering considers two viable candidates, …
  - The more constrained (by a *requires-clause*) candidate wins.
- In the second implementation:
  - std::swap was constrained by a *requires-clause*, so …
  - The new rule applies (even though ns::swap was unconstrained).
- Result? The test code is no longer ambiguous:
  - std::swap is now the more constrained overload, and wins.
  - And the test's assertion, expecting ambiguity, now fails!

 33

A caveat

- This talk uses the requires syntax and semantics as formulated in the *Concepts [Lite] TS*:
  - And as implemented by gcc -fconcepts for several years now.
- Much of that wording has been recently absorbed into the C++20 *Working Draft*, but:
  - That wording has already been considerably tweaked, and …
  - WG21 discussions will continue, and so …
  - Further technical adjustments seem likely.
- Therefore, what we actually get in C++20 may be slightly (or not-so-slightly 😊) different from today's exposition.
- But the thrust of today's talk will hold up, namely the …

 34

5. Lessons Learned

Ultimately there is no such thing as failure. There are lessons learned in different ways.

— Twyla Tharp

About language constraints

- C++ constraints via *requires-clauses* are more than just a language solution for the ugliness of enable_if.
- If present, a constraint comes into play:
  - Whenever instantiating a template, or even when just using a template name.
  - As part of overload resolution, when compared during partial ordering.
- *E.g.*, given `void g( ) requires false { }` *// not overloaded*
  - `g( );` *// no; can't call* g
  - `void (*p)( ) = g;` *// no; can't take* g*'s address*
  - `decltype(g)* p = nullptr;` *// no;* decltype(g) *is an invalid type*

 36

The big picture

- Core language constraints (*requires-clauses*, *etc.*) become an integral part of a declaration.
- Traditional declarations, not constrained in this way, just aren't quite the same!
- With *requires-clauses* come an additional set of rules for overload resolution.
- We programmers must be cognizant of those new rules in our coding.
  - Most of these details seem not yet widely well-understood.
  - Sample misleading advice: "If an implementation turns [enable_if] constraints into concepts, you won't be able to tell the difference, programmatically." — NAME WITHHELD

37

Consequences

- "You potentially change overload resolutions every time you add constrained overloads [to] a set containing [only] unconstrained templates with equivalent types."
  — Andrew Sutton
- "enable_if cannot be directly replaced with requires without a semantic difference."
  — David Sankel
- Vendors are already free to meet today's standard library specifications via this new technology:
  - Doing so can (greatly!) simplify some implementations, …
  - But, as shown, this (a) is detectable, (b) can affect user code in edge cases, and (c) affects the ABI (due to name mangling).

38

6. Addendum

He who never made a mistake,
never made a discovery.

— SAMUEL SMILES

Test your understanding

- How do these declarations differ?
  a) `void f ( );` *// unconstrained*
  b) `void f ( ) requires true;`
- What about these?
  c) `void g( ) = delete;`
  d) `void g( ) requires false { }`
- And these?
  e) `template< ⋯ > requires true and false`
     `void h( );`
  f) `template< ⋯ > requires true`
     `void h( ) requires false;`

When you train/consult/discuss

- I respectfully suggest that only experienced programmers (journeymen and masters) should learn these details in the order I've presented them, namely bottom-up.
- Novices and apprentices should likely best approach this material from first principles, namely top-down.
- As you pass on this information, please keep in mind your various listeners' level of experience.

 41

Programming with C++ Constraints:

Background, Utility, and Gotchas

FIN

WALTER E. BROWN, PH.D.

< webrown.cpp @ gmail.com >