Concept Based Testing

Dietmar Kühl
dkuhl@bloomberg.net

Bloomberg

Overview

- The vision
- C++20 concepts
- Detection idiom
- Examples of generic tests
- Short demo

Vision

- Duplicating code is bad, including test code
- Recurring operations are repeatedly tested the same way
  - e.g., comparisons, copying, iteration, etc.
- Concepts provide an abstraction for related operation
- Idea: tests based on concepts for common operations
  - only specific behavior needs custom tests

Example

- Test behavior of equality providing specimen

```
struct X { int v; };
bool operator== (X x0, X x1) { return x0.v == x1.v; }
bool operator!= (X x0, X x1) { return x0.v == x1.v; }

int main() {
   return test_equality_comparable({ { X{1}, X{1}, X{1} }, { X{4} }, { X{9} } })
        ? EXIT_SUCCESS: EXIT_FAILURE;
}
```

Example

- Test behavior of equality providing specimen

```
struct X { int v; };
bool operator== (X x0, X x1) { return x0.v == x1.v; }
bool operator!= (X x0, X x1) { return x0.v == x1.v; }

int main() {
   return test_equality_comparable({ { X{1}, X{1}, X{1} }, { X{4} }, { X{9} } })
        ? EXIT_SUCCESS: EXIT_FAILURE;
}
```

Vision

- Spend time once on comprehensive semantic tests
  - More likely to actually cover all cases
  - Interface could direct users to more thorough coverage
- Basic for standard concept tests may become available off the shelf
- Assumption: common generic vocabulary, i.e., concepts

Concepts

- Abstraction for entities manipulated in generic code
- Support present in the C++2x draft
- Define a generic interface and its semantics
- Formal representation of the required syntax
- Semantics (*axioms*) remain informal

Concept Definition

- Boolean expression on types verifying requirements

```
template <typename T>
concept EqualityComparable
    = requires(T a, T b) {
        { a == b } -> bool;
        { a != b } -> bool;
    };
// !(a == b) <=> a != b
// a == a                                (reflexive)
// a == b <=> b == a                     (symmetric)
// a == b and b == c => a == c           (transitive)
```

Concept Definition

- Boolean expression on types verifying requirements

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

Concept Use

- Boolean expression on types verifying requirements

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

template <typename T>
requires EqualityComparable<T>
void f(T value);
```

Concept Use

- Boolean expression on types verifying requirements

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

template <typename T>
requires EqualityComparable<T> // requires clause
void f(T value);
```

Concept Use

- Boolean expression on types verifying requirements

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

template <EqualityComparable T>

void f(T value);
```

Concept Use

- Boolean expression on types verifying requirements

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

```
void f(EqualityComparable value);
```

Concept Use

- Boolean expression on types verifying requirements

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

~~void f(EqualityComparable value);~~ // so far not in C++20

Concept Definition

- Boolean expression on types verifying requirements

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

Requires Expression

- Boolean, non-evaluated test for available syntax
- Simple requirement: `a + b;`
- Compound requirement: `{ a + b } -> T;`
- Type requirement: `typename T::inner;`
- Nested requirement: `requires sizeof(a) == 4u;`

Simple Requirement

- Tests if the expression is valid for the declared arguments

```
requires(S a, T b) { a == b; }

requires(S const& a, T const& b) { a == b; }
```

Type Requirement

- Tests whether a type exists

```
requires {
    typename T::value_type;
    typename ValueType<T>;
}
```

Compound Requirement

- Tests an expression and its result type

```
requires(S a, T b) {
    { a == b };                          // same as a == b;
    { a == b } -> bool;                  // result [implicitly] convertible
    { a == b } noexcept -> bool;         // detectably noexcept
    { a == b } -> Same<bool>;            // concept check result
}
```

Nested Requirement

- Test properties based on non-evaluated arguments

```
requires(T a) {
    requires 4u < sizeof(a);     // OK: a is only used in a non-evaluated context
    requires 4u < a;             // error: a's value would need to be known
}
```

Concepts Won’t be Available Soon

- Concept support is expected to come with C++20
- Informal definition of concepts is already in the standard
- For the purpose of generic tests basic functionality is sufficient
  - Detect if types or operations with appropriate signatures exist
  - Overloading based on concepts isn’t necessary
  - More involved implementation could be acceptable

Detect Operations

- Existence of nested types or specializations are easy to detect
- Operations can be detected without concept support
  - Making use of concepts without C++20 is viable
- Idea: determine if an expression has a type and fail instantiation otherwise

Detect Operations

requires(S a, T b) { a == b; }

Detect Operations

```
requires(                S a,                T b ) {


                a                    == b                    ;

}
```

Detect Operations

```
requires(                S a,                T b ) {

                std::declval<S>() == std::declval<T>() ;

}
```

Detect Operations

```
requires(                   S a,                T b ) {


    decltype(std::declval<S>() == std::declval<T>());

}
```

Detect Operations

```
requires(                   S a,                T b ) {

using Equality

   = decltype(std::declval<S>() == std::declval<T>());

}
```

Detect Operations

```
template <typename S  , typename T                                  >

using Equality

   = decltype(std::declval<S>() == std::declval<T>());
```

Detect Operations

```
template <typename S  , typename T    = S                    >

using Equality

   = decltype(std::declval<S>() == std::declval<T>());
```

Detect Operations

```
template <typename S  , typename T    = S, typename...>

using Equality

   = decltype(std::declval<S>() == std::declval<T>());
```

Detection Idiom

- Detect available syntax without requires support
  - Idea: detect if the type of an expression can be produced
  - Less convenient approach than actual concepts

```
template <typename S, typename T = S, typename...>
using Equality
    = decltype(std::declval<T>() == std::declval<T>());

if constexpr (is_detected_v<Equality, T>) { ... }
```

Implementing detect

```
template <typename T, template <typename...> class Op,
                      typename, typename... Args>
struct detect
    : std::false_type { using type = T; };
```

Implementing detect

```
template <typename T, template <typename...> class Op,
                      typename, typename... Args>
struct detect
    : std::false_type { using type = T; };
```

Implementing detect

```
template <typename T, template <typename...> class Op,
                      typename, typename... Args>
struct detect
    : std::false_type { using type = T; };
```

Implementing detect

```
template <typename T, template <typename...> class Op,
                  typename, typename... Args>
struct detect
    : std::false_type { using type = T; };
```

Implementing detect

```
template <typename T, template <typename...> class Op,
                      typename, typename... Args>
struct detect
    : std::false_type { using type = T; };

template <typename T, template <typename...> class Op,
                       typename... Args>
struct detect<T, Op, std::void_t<Op<Args...>>, Args...>
    : std::true_type { using type = Op<Args…>; };
```

void_t

```
template <typename…> using void_t = void;
```

is_detected

```
template <template <typename...> class Op, typename... A>
using is_detected = detect<nonesuch, Op, void, A...>;

template <template <typename...> class Op, typename... A>
inline constexpr bool is_detected_v
      = is_detected<Op, A…>::value;

template <template <typename...> class Op, typename... A>
inline constexpr bool detected_t
      = is_detected<Op, A…>::type;
```

Simulating Requires Clauses

- Use detection idiom to determine available declarations
- Yields compile-time checks
- Can be used with C++11 (assuming an implementation of std::void_t)

Simple Requirement

- With concept support:

```
requires(S a, T b) { a == b; }
```

- Simulation:

```
template <typename S, typename T = S, typename...>
using Equality
    = decltype(std::declval<S>() == std::declval<T>());

is_detected_v<Equality, S, T>
```

Type Requirement

- With concept support:

  requires { typename T::value_type; }

- Simulation:

  template <typename T, typename...>
  using ValueType = typename T::value_type;

  is_detected_v<ValueType, T>

Compound Requirement

- With concept support:

```
requires(S a, T b) { { a == b } -> bool; }
```

- Simulation:

```
template <typename S, typename T = S, typename...>
using Equality
    = decltype(std::declval<S>() == std::declval<T>());

std::is_convertible_v<detected_t<Equality, S, T>, bool>
```

Compound Requirement

- With concept support:

```
template <class S, class T> concept Same = std::is_same_v<S, T>;
requires(S a, T b) { { a == b } -> Same<bool>; }
```

- Simulation:

```
template <typename S, typename T>
inline constexpr bool Same = std::is_same_v<S, T>;

Same<detected_t<Equality, S, T>, bool>
```

Nested Requirement

- Becomes a bit more involved
- It can be done even though it may not be nice
- Nested requirements are likely relatively rare

Constraining Classes

- With concepts:

```
template <EqualityComparable T>
class example;
```

- Without concepts

```
template <typename T,
          typename = std::void_t<Equal<T>>>
class example;
```

Constraining Functions

- With concepts

```
template <EqualityComparable T>
void f(T value);
```

- Without concepts

```
template <typename T,
          typename = void_t<Equal<T>>>
void f(T value);
```

Concept Testing

- Concepts consist of syntax and semantics
  - Using requires expressions or is_detected can verify syntax is available
  - Compilers can’t verify the semantics at all
- Generic behavior applies to all models of a concept
- Semantic testings remains relevant with concepts support

Concept Testing

- Testing should work with partial implementations:
  - Inform users about missing operations
  - Exercise existing functionality
- Using is_detected allows to verify concept operations exist
- Using if constexpr (...) to report missing operations on one branch
- On the other branch to semantic tests can be run

Incomplete Models

- Classes may only partially implement concepts
  - Start implementing basic functionality
  - Find out from errors what is missing
- Running a test program can potentially suggest code
  - For example, when common strategies are used

EqualityComparable: Definition

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


// !(a == b) <=> a != b
// a == a                                (reflexive)
// a == b <=> b == a                     (symmetric)
// a == b and b == c => a == c           (transitive)
```

EqualityComparable: Operations

```
template <typename T>
using EqualityOperation
    = decltype(std::declval<T>() == std::declval<T>());

template <typename T>
using InequalityOperation
    = decltype(std::declval<T>() != std::declval<T>());
```

EqualityComparable: Existence Tests

```
template <typename T>
bool testEqualityComparable(std::initializer_list<std::initializer_list<T>> es) {
    if (!is_detected_v<EqualityOperation, T const&>) {
        std::cerr << “requires(T const& a, T const& b){ a == b; } not satisfied\n”;




        return false;
    }
    return true;
}
```

EqualityComparable: Existence Tests

```
template <typename T>
bool testEqualityComparable(std::initializer_list<std::initializer_list<T>> es) {
    if (!is_detected_v<EqualityOperation, T const&>) {
        std::cerr << “requires(T const& a, T const& b){ a == b; } not satisfied\n”;
        if (is_detected_v<InequalityOperation, T const&>)
            std::cerr << “potential implementation\n”
                      << “bool operator== (T const& a, T const& b) {\n”
                      << “    return !(a != b);\n}\n”;
        return false;
    }
    return true;
}
```

EqualityComparable: Existence Tests

```
template <typename T>
bool testEqualityComparable(std::initializer_list<std::initializer_list<T>> es) {
    if (!is_detected_v<EqualityOperation, T const&>) {
       return false;
    }
    else {
        T const& x = *(*es.begin()).begin();
        if (!(x == x))
            return (std::cerr << “a == a is false (not reflexive)\n”), false;
   }
   return true;
}
```

EqualityComparable: Existence Tests

```
template <typename T>
bool testEqualityComparable(std::initializer_list<std::initializer_list<T>> es) {
    if constexpr (!is_detected_v<EqualityOperation, T const&>) {
       return false;
    }
    else {
        T const& x = *(*es.begin()).begin();
        if (!(x == x))
            return (std::cerr << “a == a is false (not reflexive)\n”), false;
   }
   return true;
}
```

EqualityComparable: Semantic Tests

```
template <typename T>
bool testEqualityComparable(std::initializer_list<std::initializer_list<T>> es) {
    if constexpr (!is_detected_v<EqualityOperation, T const&>) {
       return false;
    }
    else {
        T const& x = *(*es.begin()).begin();
        if (!(x == x))
            return (std::cerr << “a == a is false (not reflexive)\n”), false;
   }
   return true;
}
```

EqualityComparable: Semantic Tests

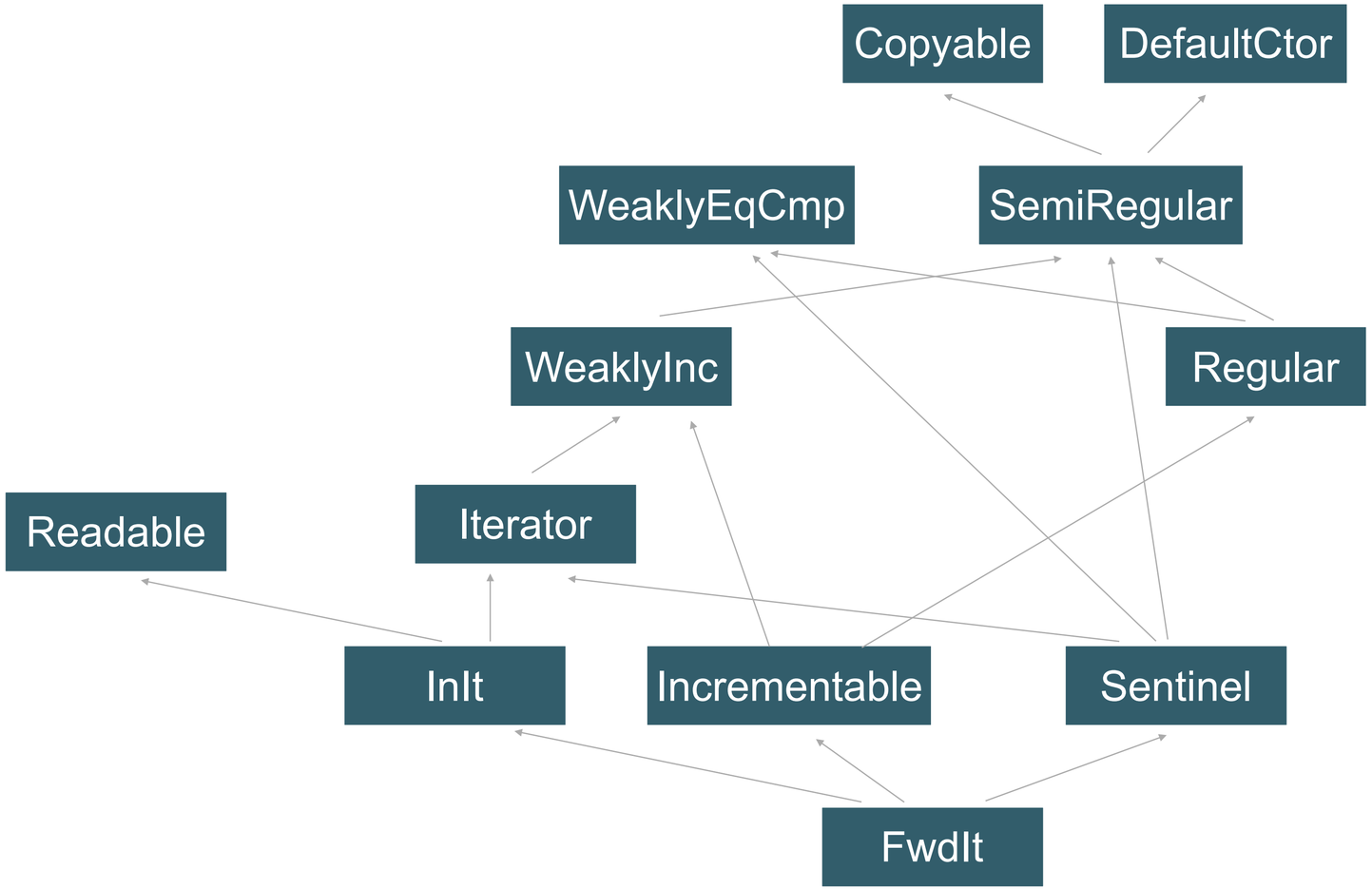
```
template <typename T>
bool testEqualityComparable(std::initializer_list<std::initializer_list<T>> es) {
    if constexpr (is_detected_v<EqualityOperation, T const&>) {
        …
    }
    if constexpr (is_detected_v<InequalityOperation, T const&>) {
        …
    }


    return true;
}
```

Concept ForwardIterator

```
template <typename I>
concept ForwardIterator
    =   InputIterator<I>
    && DerivedFrom<iterator_category_t<I>,
                   forward_iterator_tag>
    && Incrementable<I>
    && Sentinel<I, I>
    ;
```

Concepts



Bloomberg

Concept BidirectionalIterator

```
template <typename I>
concept BidirectionalIterator
    =   ForwardIterator<I>
    && DerivedFrom<iterator_category_t<I>, bidirectional_iterator_tag>
    && requires(I i) { { ++i } -> Same<I&>; }
    && requires(I i) { { i++ } -> Same<I>; }
   ;
// axiom: &—a == &a
// axiom: bool(a == b) => (a— == b)
// axiom: bool(a == b), then —a and b— => bool(a == b)
// axiom: bool(a == b) => bool(—(++a) == b)
// axiom: bool(a == b) => bool(++(—a) == b)
```

Compound Concept Check

- requires(I i) { { ++i } -> Same<I&>;

```
if (!Same<detected_t<OpPreDecrement, I>, I>()) {
    return false;
}
```

- requires(I i) { { i++ } -> Same<I&&>;

```
if (!Same<detected_t<OpPostDecrement, I>, I>()) {
    return false;
}
```

Post-Decrement Test

- Pre-decrement defined ⇒ suggest post-deccrement

```
if (detected_v<PreDec, T> && !detected_v<PostDec, T>) {
    std::cout << “potential operator—(int) implementation:\n”
              << “T tmp(*this); —*this; return tmp;\n”;
}
```

- Alternatively it could also suggest use of a suitable CRTP class template

Decrement Test

```
if constexpr(is_detected_v<Equal, T>
                    && is_detected_v<CopyCtor, T>
                    && is_detected_v<PreDec, T>
                    && is_detected_v<PostDec, T>) {
    T t1(end), t2(end);
    if (!(t1 == begin)) {
        —t1; t2—;
        assert(t1 == t2);
    }

}
```

Decrement Test

```
if constexpr(is_detected_v<Equal, T>
                    && is_detected_v<CopyCtor, T>
                    && is_detected_v<PreDec, T>
                    && is_detected_v<PostDec, T>) {
    T t1(end), t2(end);
    if (!(t1 == begin)) {
        —t1; t2—;
        assert(t1 == t2);
    }

}
```

Decrement Test

```
if constexpr(is_detected_v<Equal, T>
                    && is_detected_v<CopyCtor, T>
                    && is_detected_v<PreDec, T>
                    && is_detected_v<PostDec, T>) {
    T t1(end), t2(end);
    for (int c(0); c != 1000 && !(t1 == begin); ++c) {
        —t1; t2—;
        assert(t1 == t2);
    }

}
```

Decrement Test

```
if constexpr(is_detected_v<Equal, T>
                    && is_detected_v<CopyCtor, T>
                    && is_detected_v<PreDec, T>
                    && is_detected_v<PostDec, T>) {
    T t1(end), t2(end);
    for (int c(0); c != 1000 && !(t1 == begin); ++c) {
        —t1; t2—;
        assert(t1 == t2);
    }
    assert(t1 == end);
}
```

Basic Decrement Test

- Semantic requirement: &it == &—it

```
template <typename It>
bool testDecIncrement(It&& begin, It&& end) {
    if constexpr (is_detected_v<PreDec, It>
                        && is_detected_v<Equality, It>) {
        auto tmp = &end;
        return begin != end && tmp == &—end;
    }
    else
        return false; // needed operations don’t exist
}
```

Demo

Bloomberg

Questions

Bloomberg

Thank you!

dkuhl@bloomberg.net
http://www.bloomberg.com/careers/technology

References

- Demo implementation: http://github.com/dietmarkuehl/kuhllib/test/cbt
- Concepts for the C++ Library: http://wg21.link/p0898
- Ranges TS (for iterator concepts): http://wg21.link/n4651