Evolving a Nice Trick

PATRICE ROY

EVOLVING A NICE TRICK

Patrice Roy

Patrice.Roy@USherbrooke.ca; Patrice.Roy@clg.qc.ca

CeFTI, Université de Sherbrooke; Collège Lionel-Groulx

Who am I?

- Father of five, ages 26 to 8
- Feeds and cleans up after a varying number of animals
  - Look for *Paws of Britannia* with your favorite search engine
- Used to write military flight simulator code, among other things
  - CAE Electronics Ltd, IREQ
- Full-time teacher since1998
  - Collège Lionel-Groulx, Université de Sherbrooke
  - Works a lot with game programmers
- Incidentally, WG21 and WG23 member (although I've been really busy recently)
  - Involved in SG14, among other study groups
  - Occasional WG21 secretary
- And so on…

WE HAVE A PROBLEM

The Situation

- Writing code that performs « sensible » comparisons for values of a given type is not really difficult

The Situation

- Writing code that performs « sensible » comparisons for values of a given type is not really difficult

```
bool are_equal(int a, int b) {
    return a == b;
}
```

The Situation

- Writing code that performs « sensible » comparisons for values of a given type is not really difficult

```
bool are_equal(int a, int b) {
    return a == b;
}
bool close_enough(float a, float b) {
    return abs(a - b) <= 0.000001f;
}
```

The Situation

- Writing code that performs « sensible » comparisons for values of a given type is not really difficult

```
bool are_equal(int a,
    return a == b;
}
bool close_enough(float a, float b) {
    return abs(a - b) <= 0.000001f;
}
```

I know this is not rigorous enough in practice, but this is not the issue we're adressing today (for the time being, it will suffice)

The Situation

- Writing *generic* code that performs « sensible » comparisons for numerical values is more complicated

The Situation

- Writing *generic* code that performs « sensible » comparisons for numerical values is more complicated
  - First, we want homogeneous naming to simplify the expression of client code

```
bool close_enough(int a, int b) {
    return a == b;
}
bool close_enough(float a, float b) {
    return abs(a - b) <= 0.000001f;
}
```

The Situation

- Writing *generic* code that performs « sensible » comparisons for numerical values is more complicated
  - First, we want homogeneous naming to simplify the expression of client code
  - Then, if we want to cover a variety of types, it's not just a matter of turning the functions into templates

```
template <class T>
bool close_enough(T a, T b) {
    return a == b;
}
template <class T>
bool close_enough(T a, T b) {
    return abs(a - b) <= 0.000001f;
}
```

The Situation

- Writing *generic* code that performs « sensible » comparisons for numerical values is more complicated
  - First, we want homogeneous naming to simplify the expression of client code
  - Then, if we want to cover a variety of types, it's not just a matter of turning the functions into templates

```
template <class T>
bool close_enough(T a, T b) {
    return a == b;
}
template <class T>
bool close_enough(T a, T b) {
    return abs(a - b) <= 0.000001f;
}
```

A call such as close_enough(3,3) is now ambiguous

The Situation

- The problem statement is simple: we want a generic close_enough() function that performs « sensible » comparisons for values of a given arithmetic type
  - For integrals, comparing the values with == will suffice
  - For floating point numbers, we will try to see if the difference between the values is within our error tolerance threshold

The Situation

- Writing close_enough() is not in itself hard
  - It is also a problem for which other popular languages often offer solutions that are… unsatisfactory
    - Either do case-by-case overloads for a large number of types
    - Otherwise, rely on interfaces (e.g.: IEquatable<T> and similar) and incur indirection costs for each comparison

Without C++…

```
// C# (example), non-generic
class Integral
{
   public int Value{ get; init; }
   // ...
   public override bool Equals(object other) =>
      other is Integral &&
            Value == ((Integral) other).Value;
// ...
}
```

Without C++…

```
// C# (example), non-generic
class Integral
{
    public int Value{ get; init; }
    // ...
    public override bool Equals(object other) =>
        other is Integral &&
              Value == ((Integral) other).Value;
// ...
}
```

We could add operator==, but that would mean adding operator!= and, if we want to silence warnings, adding GetHashCode

Without C++…

```
// C# (example), non-generic
class Integral
{
   public int Value{ get; init; }
   // ...
   public override bool Equals(object other) =>
      other is Integral &&
            Value == ((Integral) other).Value;
// ...
}
```

Notice that we have an implicit test against null, a downcast and an indirection here… Whew!

Without C++…

```
// C# (example), generic
class Integral : IEquatable<Integral>
{
    public int Value{ get; init; }
    // ...
    public bool Equals(Integral other) =>
        other != null && Value == other.Value;
    public override bool Equals(object other) =>
        other is Integral n && Equals(n);
}
```

Without C++…

```
// C# (example), generic
class Integral : IEquatable<Integral>
{
    public int Value{ get; init; }
    // ...
    public bool Equals(Integral other) =>
        other != null && Value == other.Value;
    public override bool Equals(object other) =>
        other is Integral n && Equals(n);
}
```

This will be slightly more efficient when comparing two references to Integral objects, but we still have indirections and a test for null

Without C++…

```
// C# (example), generic
// ...
static bool CloseEnough<T>(T a, T b)
   where T : IEquatable<T> =>
      a.Equals(b);
```

Without C++…

```
// C# (example), generic
// ...
static bool CloseEnough<T>(T a, T b)
    where T : IEquatable<T> =>
        a.Equals(b);
```

This can work through and indirection and assuming T implements an interface intrusively… and assuming a != null

The Situation

- Writing close_enough() is not in itself hard
  - This is C++, so we want a non-intrusive solution, and we do not want to make client code pay overhead costs for each comparison

The Situation

- Writing close_enough() is not in itself hard
  - It is, however, a problem for which solutions have benefitted from the evolution of the C++ language
  - **In this talk, we will « travel through time », starting in 1998 and going forward, to see how C++ gradually makes our life better and better**

C++98/03

C++98/03

- A technique that works well even with C++98 is ***tag dispatching***
  - It still works well today, of course
  - C++ is nothing if not backward-compatible!

C++98/03

- The idea:
  - Write functions with different signatures for each algorithm
    - These functions are not meant to be called directly
  - Use instances of empty classe (*tag types*) to make these signatures distinct from one another

C++98/03

- The idea:
  - Write functions with different signatures for each algorithm
    - These functions are not meant to be called directly
  - Use instances of empty classe (*tag types*) to make these signatures distinct from one another
  - Write another function that considers the types involved, and *dispatches* the work to one of the specialized algorithms
    - This one is meant to be used directly by client code
    - Optimizers are very, very good at inlining this level of the call chain

C++98/03: Tag Dispatching

```
// Defining tag types
class exact {}; // int, char, bool, etc.
class floating{}; // float, double, long double
```

Technique : *Tag Dispatching*

```
#include <cmath>
class exact {};
class floating {};
// specialized algorithms
template <class T>
bool close_enough(T a, T b, exact) {
   return a == b;
}
template <class T>
bool close_enough(T a, T b, floating) {
   return std::abs(a - b) <= static_cast<T>(0.000001);
}
```

Technique : *Tag Dispatching*

```
#include <cmath>
class exact {};
class floating {};
// specialized algorithms
template <class T>
bool close_enough(T a, T b, exact) {
   return a == b;
}
template <class T>
bool close_enough(T a, T b, floating) {
   return std::abs(a - b) <= static_cast<T>(0.000001);
}
```

Tag arguments are deliberately unnamed to avoid warnings on unused variables

Technique : *Tag Dispatching*

```
#include <cmath>
class exact {};
class floating {};
// specialized algorithms
template <class T>
bool close_enough(T a, T b, exact) {
   return a == b;
}
template <class T>
bool close_enough(T a, T b, floating) {
   return std::abs(a - b) <= static_cast<T>(0.000001);
}
```

We moved from a float literal to a type T literal to avoid mixed type comparisons

Technique : *Tag Dispatching*

- In C++98/03, unless one could use third party libraries for this, one had to define one's own traits manually
  - In this specific case, one could also write close_enough() for every single floating point type, but it's a bit of a pain

Technique : *Tag Dispatching*

```
// « manual », C++98-style solution: define an is_floating<T>
// trait. I will deliberately use the ::value convention we
// settled on over time
template <class> struct is_floating {
  enum { value = false };
};
template <> struct is_floating<float> { enum { value = true }; };
template <> struct is_floating<double> { enum { value = true }; };
template <> struct is_floating<long double> {
  enum { value = true };
};
```

Technique : *Tag Dispatching*

```
// « manual », C++98-style solution, more clever
// (thanks to the Boost people!). Yes, I know there's
// more to true_type and false_type ;)
struct false_type { enum { value = false }; };
struct true_type { enum { value = true }; };
template <class> struct is_floating : false_type { };
template <> struct is_floating<float> : true_type { };
template <> struct is_floating<double> : true_type { };
template <> struct is_floating<long double> : true_type { };
```

Technique : *Tag Dispatching*

```
// « manual », C++98-style solution
// ...tag types, specialized algorithms...
// ...floating point type detection trait...
// compile-time if-like construct to pick a type from a pair
// of options based on a compile-time condition. The use of
// type is deliberate, based on what we collectively settled on
template <bool, class T, class F>
  struct static_if_else;
template <class T, class F>
  struct static_if_else<true, T, F> { typedef T type; };
template <class T, class F>
  struct static_if_else<false, T, F> { typedef F type; };
```

Technique : *Tag Dispatching*

```
// « manual », C++98-style solution
// ...tag types, specialized algorithms...
// ...floating point type detection trait...
// ... compile-time if-like construct ...
// function to be used by client code
template <class T>
  bool close_enough(T a, T b) {
    return close_enough(a, b, typename static_if_else<
      is_floating<T>::value, floating, exact
    >::type{});
  }
```

Technique : *Tag Dispatching*

- What's there not to love? And it works!
  - When I used this in actual projects, reviews were flattering!

Technique : *Tag Dispatching*

```
#include <cmath>

class exact {};

class floating {};

template <class T> bool close_enough(T a, T b, exact) { return a == b; }

template <class T> bool close_enough(T a, T b, floating) { return std::abs(a - b) <= static_cast<T>(0.000001); }

struct false_type { enum { value = false }; };

struct true_type { enum { value = true }; };

template <class> struct is_floating : false_type { };

template <> struct is_floating<float> : true_type { };

template <> struct is_floating<double> : true_type { };

template <> struct is_floating<long double> : true_type { };

template <bool, class T, class F> struct static_if_else;

template <class T, class F> struct static_if_else<true, T, F> { typedef T type; };

template <class T, class F> struct static_if_else<false, T, F> { typedef F type; };

template <class T>

  bool close_enough(T a, T b) { return close_enough(a, b, typename static_if_else<is_floating<T>::value, floating, exact>::type{});   }
```

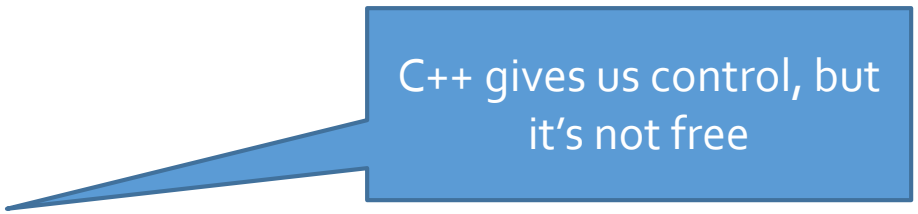
Technique : *Tag Dispatching*

```
#include <cmath>

class exact {};

class floating {};

template <class T> bool close_enough(T a, T b, exact) { return a == b; }

template <class T> bool close_enough(T a, T b, floating) { return std::abs(a - b) <= static_cast<T>(0.000001); }

struct false_type { enum { value = false }; };

struct true_type { enum { value = true }; };

template <class> struct is_floating : false_type { };

template <> struct is_floating<float> : true_type { };

template <> struct is_floating<double> : true_type { };

template <> struct is_floating<long double> : true_type { };

template <bool, class T, class F> struct static_if_else;

template <class T, class F> struct static_if_else<true, T, F> { typedef T type; };

template <class T, class F> struct static_if_else<false, T, F> { typedef F type; };

template <class T>

  bool close_enough(T a, T b) { return close_enough(a, b, typename static_if_else<is_floating<T>::value, floating, exact>::type{});   }
```

https://godbolt.org/z/EM7j5fhnM

Technique : *Tag Dispatching*

```
#include <cmath>

class exact {};

class floating {};

template <class T> bool close_enough(T a, T b, exact) { return a == b; }

template <class T> bool close_enough(T a, T b, floating) { return std::abs(a - b) <= static_cast<T>(0.000001); }

struct false_type { enum { value = false }; };

struct true_type { enum { value = true }; };

template <class> struct is_floating : false_type { };

template <> struct is_floating<float> : true_type { };

template <> struct is_floating<double> : true_type { };

template <> struct is_floating<long double> : true_type { };

template <bool, class T, class F> struct static_if_else;

template <class T, class F> struct static_if_else<true, T, F> { typedef T type; };

template <class T, class F> struct static_if_else<false, T, F> { typedef F type; };

template <class T>

  bool close_enough(T a, T b) { return close_enough(a, b, typename static_if_else<is_floating<T>::value, floating, exact>::type{});   }
```



Technique : *Tag Dispatching*

```
#include <cmath>

class exact {};

class floating {};

template <class T> bool close_enough(T a, T b, exact) { return a == b; }

template <class T> bool close_enough(T a, T b, floating) { return std::abs(a - b) <= static_cast<T>(0.000001); }

struct false_type { enum { value = false }; };

struct true_type { enum { value = true }; };

template <class> struct is_floating : false_type { };

template <> struct is_floating<float> : true_type { };

template <> struct is_floating<double> : true_type { };

template <> struct is_floating<long double> : true_type { };

template <bool, class T, class F> struct static_if_else;

template <class T, class F> struct static_if_else<true, T, F> { typedef T type; };

template <class T, class F> struct static_if_else<false, T, F> { typedef F type; };

template <class T>

  bool close_enough(T a, T b) { return close_enough(a, b, typename static_if_else<is_floating<T>::value, floating, exact>::type{});   }
```

The one function user code cares for is the one over there, at the bottom…

C++11

C++11

- Inspired by community practice and years of experience, C++11 introduced standard type traits
  - The <type_traits> header
- Such things as « compile-time type selection », « true_type / false_type », « is T a floating point type » were all standardized

C++11

- Inspired by community practice and years of experience, C++11 introduced standard type traits
  - The <type_traits> header
- Such things as « compile-time type selection » and many type traits
  - std::conditional<bool, class T, class F>
  - std::true_type, std::false_type
    - Derived from std::integral_constant<bool, true> and std::integral_constant<bool, false> respectively
  - std::is_floating_point<class T>
- … were all standardized

C++11, technique : *Tag Dispatching*

```
#include <cmath>
#include <type_traits>
class exact {};
class floating {};
template <class T> bool close_enough(T a, T b, exact) {
  return a == b;
}
template <class T> bool close_enough(T a, T b, floating) {
  return std::abs(a - b) <= static_cast<T>(0.000001);
}
template <class T>
  bool close_enough(T a, T b) {
     return close_enough(a, b, typename std::conditional<std::is_floating_point<T>::value, floating, exact>::type{});
  }
```

C++11, technique : *Tag Dispatching*

```
#include <cmath>
#include <type_traits>
class exact {};
class floating {};
template <class T> bool close_enough(T a, T b, exact) {
  return a == b;
}
template <class T> bool close_enough(T a, T b, floating) {
  return std::abs(a - b) <= static_cast<T>(0.000001);
}
template <class T>
  bool close_enough(T a, T b) {
     return close_enough(a, b, typename std::conditional<std::is_floating_point<T>::value, floating, exact>::type{});
  }
```

It's a net gain in terms of effort and source code size

C++11, technique : *Tag Dispatching*

```
#include <cmath>
#include <type_traits>
class exact {};
class floating {};
template <class T> bool close_enough(T a, T b, exact) {
  return a == b;
}
template <class T> bool close_enough(T a, T b, floating) {
  return std::abs(a - b) <= static_cast<T>(0.000001);
}
```

```
template <class T>
  bool close_enough(T a, T b) {
     return close_enough(a, b, typename std::conditional<std::is_floating_point<T>::value, floating, exact>::type{});
  }
```

It is however conceptually equivalent : we wrote three functions, one per algorithm and a (no-cost on any reasonable C++ compiler) dispatcher function

C++11, technique : *Tag Dispatching*

```
#include <cmath>
#include <type_traits>
class exact {};
class floating {};
template <class T> bool close_enough(T a, T b, exact) {
  return a == b;
}
template <class T> bool close_enough(T a, T b, floating) {
  return std::abs(a - b) <= static_cast<T>(0.000001);
}
```

https://godbolt.org/z/jb5sa95ra

```
template <class T>
  bool close_enough(T a, T b) {
     return close_enough(a, b, typename std::conditional<std::is_floating_point<T>::value, floating, exact>::type{});
  }
```

C++11: adding constexpr

- C++11 had this new thing called **constexpr**
- This allowed us to resolve at compile time simple (!) functions when
  - (a) the arguments were known at compile-time
  - (b) the context demanded it
    - That last part's important, as it makes constexpr functions debuggable

C++11: adding constexpr

- This constexpr feature appears to be a good fit for close_enough()

C++11: adding constexpr

```
#include <cmath>
#include <type_traits>
class exact {};
class floating {};
template <class T> constexpr bool close_enough(T a, T b, exact) {
  return a == b;
}
template <class T> bool close_enough(T a, T b, floating) {
  return std::abs(a - b) <= static_cast<T>(0.000001);
}
template <class T>
  bool close_enough(T a, T b) {
     return close_enough(a, b, typename std::conditional<is_floating_point<T>::value, floating, exact>::type{});
  }
```

Comparing to integers (say) at compile-time seems reasonable when said integers are also known at compile-time

C++11: adding constexpr

```
#include <cmath>
#include <type_traits>
class exact {};
class floating {};
template <class T> constexpr bool close_enough(T a, T b, exact) { return a == b; }
template <class T> bool close_enough(T a, T b, floating) {
  return std::abs(a - b) <= static_cast<T>(0.000001);
}
template <class T>
  bool close_enough(T a, T b) {
     return close_enough(a, b, typename std::conditional<
       std::is_floating_point<T>::value, floating, exact
     >::type{});
  }
```

Comparing two floating point numbers also seems like a good fit, except that std::abs() is in practice a C function, and these don't know constexpr

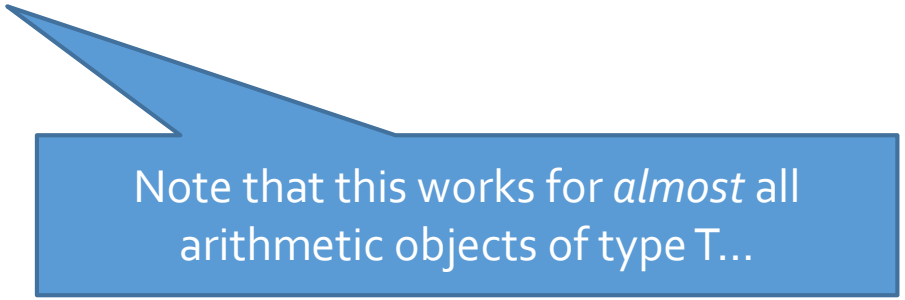
C++11: adding constexpr

```
#include <type_traits>
class exact {};
class floating {};
template <class T> constexpr bool close_enough(T a, T b, exact) { return a == b; }
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T> bool close_enough(T a, T b, floating) {
  return absolute(a - b) <= static_cast<T>(0.000001);
}
template <class T>
  bool close_enough(T a, T b) {
     return close_enough(a, b, typename std::conditional<
        std::is_floating_point<T>::value, floating, exact
     >::type{});
  }
```

… but we can write our own, can't we?

C++11: adding constexpr

```
#include <type_traits>
class exact {};
class floating {};
template <class T> constexpr bool close_enough(T a, T b, exact) { return a == b; }
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T> bool close_enough(T a, T b, floating) {
  return absolute(a - b) <= static_cast<T>(0.000001);
}
template <class T>
  bool close_enough(T a, T b) {
     return close_enough(a, b, typename std::conditional<
        std::is_floating_point<T>::value, floating, exact
     >::type{});
  }
```



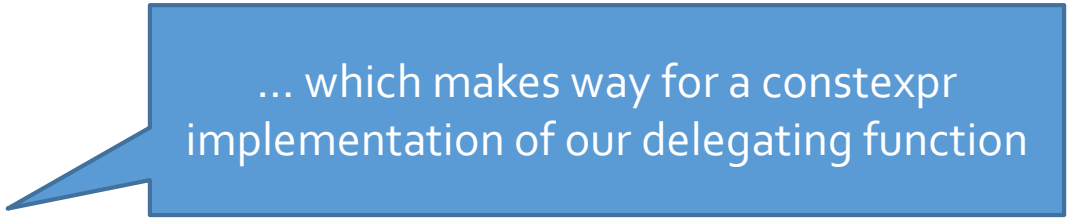
C++11: adding constexpr

```
#include <type_traits>
class exact {};
class floating {};
template <class T> constexpr bool close_enough(T a, T b, exact) { return a == b; }
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T> constexpr bool close_enough(T a, T b, floating) {
  return absolute(a - b) <= static_cast<T>(0.000001);
}
template <class T>
  bool close_enough(T a, T b) {
     return close_enough(a, b, typename std::conditional<
        std::is_floating_point<T>::value, floating, exact
     >::type{});
  }
```

With this small nudge, we have a constexpr implementation here…

C++11: adding constexpr

```
#include <type_traits>
class exact {};
class floating {};
template <class T> constexpr bool close_enough(T a, T b, exact) { return a == b; }
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T> constexpr bool close_enough(T a, T b, floating) {
  return absolute(a - b) <= static_cast<T>(0.000001);
}
template <class T>
  constexpr bool close_enough(T a, T b) {
     return close_enough(a, b, typename std::conditional<
        std::is_floating_point<T>::value, floating, exact
     >::type{});
  }
```



C++11: adding constexpr

```
#include <type_traits>
class exact {};
class floating {};
template <class T> constexpr bool close_enough(T a, T b, exact) { return a == b; }
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T> constexpr bool close_enough(T a, T b, floating) {
  return absolute(a - b) <= static_cast<T>(0.000001);
}
template <class T>
  constexpr bool close_enough(T a, T b) {
     return close_enough(a, b, typename std::conditional<
        std::is_floating_point<T>::value, floating, exact
     >::type{});
  }
```

https://godbolt.org/z/Mz48jq84T (and note that we can now test with static_assert, a C++11 feature)

C++11: adding constexpr

- Being able to add constexpr was a stupefying step ahead
  - We're still building a lot of modern C++ today on that feature, and on the way this feature transformed the ways in which we see code

C++11: adding enable_if

- There's still the matter of writing three functions when we only have two algorithms
  - It works fine, but some would say it lacks in elegance somewhat

C++11: adding enable_if

- There's still the matter of writing three functions when we only have two algorithms
  - It works fine, but some would say it lacks in elegance somewhat
- C++11 also standardized something called **enable_if**
  - It's … strange
  - A way (through SFINAE) of removing from the available set of options those functions or classes we do not want there

C++11: adding enable_if

```
#include <type_traits>
template <class T>
   constexpr
      typename std::enable_if<!std::is_floating_point<T>::value, bool>::type
         close_enough(T a, T b) { return a == b; }
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T>
   constexpr
      typename std::enable_if<std::is_floating_point<T>::value, bool>::type
         close_enough(T a, T b) {
            return absolute(a - b) <= static_cast<T>(0.000001);
         }
```

C++11: adding enable_if

To the compiler's « eyes », the first version of close_enough<T>() only exists if T *is not* a floating point type

```
#include <type_traits>
template <class T>
   constexpr
      typename std::enable_if<!std::is_floating_point<T>::value, bool>::type
         close_enough(T a, T b) { return a == b; }
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T>
   constexpr
      typename std::enable_if<std::is_floating_point<T>::value, bool>::type
         close_enough(T a, T b) {
            return absolute(a - b) <= static_cast<T>(0.000001);
         }
```

C++11: adding enable_if

```
#include <type_traits>
template <class T>
   constexpr
      typename std::enable_if<!std::is_floating_point<T>::value, bool>::type
         close_enough(T a, T b) {
template <class T> constexpr T abs
template <class T>
   constexpr
      typename std::enable_if<std::is_floating_point<T>::value, bool>::type
         close_enough(T a, T b) {
            return absolute(a - b) <= static_cast<T>(0.000001);
         }
```

To the compiler's « eyes », the first version of close_enough<T>() only exists if T *is* a floating point type

C++11: adding enable_if

The way enable_if<cond,T> works is « if cond is true, then its ::type is T otherwise it doesn't exist and will be SFINAE'd away »

```
#include <type_traits>
template <class T>
   constexpr
      typename std::enable_if<!std::is_floating_point<T>::value, bool>::type
         close_enough(T a, T b) { return a == b; }
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T>
   constexpr
      typename std::enable_if<std::is_floating_point<T>::value, bool>::type
         close_enough(T a, T b) {
            return absolute(a - b) <= static_cast<T>(0.000001);
         }
```

C++11: adding enable_if

It's at once kind of cool, and very weird, and somewhat expert-friendly (should we say « new-to-C++-scary »?)

```
#include <type_traits>
template <class T>
   constexpr
      typename std::enable_if<!std::is_floating_point<T>::value, bool>::type
         close_enough(T a, T b) { return a == b; }
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T>
   constexpr
      typename std::enable_if<std::is_floating_point<T>::value, bool>::type
         close_enough(T a, T b) {
            return absolute(a - b) <= static_cast<T>(0.000001);
         }
```

C++11: adding enable_if

```
#include <type_traits>
template <class T>
   constexpr
      typename std::enable_if<!std::is_floating_point<T>::value, bool>::type
         close_enough(T a, T b) { return a == b; }
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T>
   constexpr
      typename std::enable_if<std::is_floating_point<T>::value, bool>::type
         close_enough(T a, T b) {
            return absolute(a - b) <= static_cast<T>(0.000001);
         }
```

Still, it works, and we got down to one function per algorithm.
https://godbolt.org/z/qsrhEPnKn

C++14

C++14: making things a bit easier

- It's been said that « C++11 was a big change, C++14 just saves you some typing »
  - It's not exactly true
  - Some things have really made a big difference in our lives
    - Generic lambdas
    - Extending the reaches of constexpr

C++14: making things a bit easier

- In our case, C++14 has mostly made writing code a bit easier by introducing the **_t** suffix for type traits expressing types
  - It's just a coding convention…
  - … but it really makes a difference
    - …especially with nested type traits, but that's not our story

C++14: making things a bit easier

- Something like the following trait :

```
template <class T> struct remove_const {
   using type = T;
};
template <class T> struct remove_const<const T> {
   using type = T;
};
```

- … would now be accompanied with :

```
template <class T>
   using remove_const_t = typename remove_const<T>::type;
```

C++14: making things a bit easier

- Something like the following trait :

```
template <class T> struct remove_const {
    using type = T;
};
template <class T> struct remove
    using type = T;
};
```

- … would now be accompanied with :

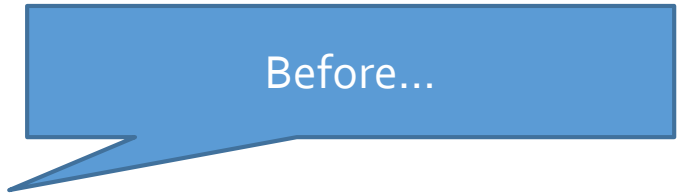
```
template <class T>
    using remove_const_t = typename remove_const<T>::type;
```

These aliases free user code from typing typename … ::type here and there. It's quite a syntactical win in practice

C++14: making things a bit easier

- This allows user code to replace this :
  ```
  typename remove_const<T>::type x;
  ```
- … with this :
  ```
  remove_const_t<T> x;
  ```

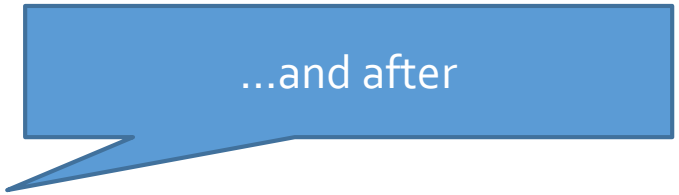
C++14: making things a bit easier




```
#include <type_traits>
template <class T>
   constexpr
      typename std::enable_if<!std::is_floating_point<T>::value, bool>::type
         close_enough(T a, T b) { return a == b; }
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T>
   constexpr
      typename std::enable_if<std::is_floating_point<T>::value, bool>::type
         close_enough(T a, T b) {
            return absolute(a - b) <= static_cast<T>(0.000001);
         }
```

C++14: making things a bit easier

```
#include <type_traits>
template <class T>
   constexpr
      std::enable_if_t<!std::is_floating_point<T>::value, bool>
         close_enough(T a, T b) { return a == b; }
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T>
   constexpr
      std::enable_if_t<std::is_floating_point<T>::value, bool>
         close_enough(T a, T b) {
            return absolute(a - b) <= static_cast<T>(0.000001);
         }
```



C++14: making things a bit easier

https://godbolt.org/z/GWvar11zv

```
#include <type_traits>
template <class T>
   constexpr
      std::enable_if_t<!std::is_floating_point<T>::value, bool>
         close_enough(T a, T b) { return a == b; }
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T>
   constexpr
      std::enable_if_t<std::is_floating_point<T>::value, bool>
         close_enough(T a, T b) {
            return absolute(a - b) <= static_cast<T>(0.000001);
         }
```

C++14: making things a bit easier

- C++14 also introduces **generic variables**
- This makes it possible for us to express values of generic types
  - The canonical example is :

```
template <class T>
  constexpr auto pi =
    static_cast<T>(3.14159265358979323846264338327);
// ...
cout << pi<float> << ' ' // a low-precision pi
     << pi<long double> << ' ' // a high-precision pi
     << pi<int>; // a ridiculous pi
```

C++14: making things a bit easier

```
#include <type_traits>
template <class T>
   constexpr std::enable_if_t<!std::is_floating_point<T>::value, bool>
      close_enough(T a, T b) { return a == b; }
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T> constexpr auto threshold = static_cast<T>(0.000001);
template <class T>
   constexpr std::enable_if_t<std::is_floating_point<T>::value, bool>
      close_enough(T a, T b) {
          return absolute(a - b) <= threshold<T>;
      }
```

C++14: making things a bit easier

```
#include <type_traits>
template <class T>
   constexpr std::enable_if_t<!std::is_floating_point<T>::value, bool>
      close_enough(T a, T b) { return a == b; }
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T> constexpr auto threshold = static_cast<T>(0.000001);
template <class T>
   constexpr std::enable_if_t<std::is_floating_point<T>::value, bool>
      close_enough(T a, T b) {
          return absolute(a - b) <= threshold<T>;
      }
```

https://godbolt.org/z/9jTr7d5rG

C++17

C++17: we're making progress

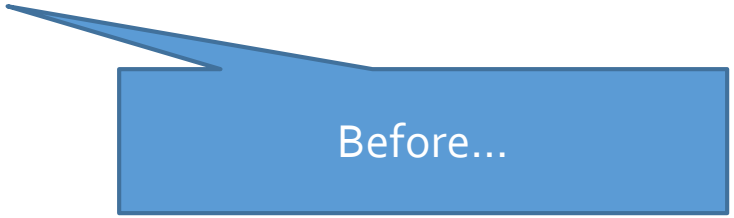
- C++17 was not the revolution many hoped for
  - Some big features such as concepts or coroutines did not make it
- However, that standard contains quite a bit of improvements for programmers' everyday lives

C++17: we're making progress

- C++17 has made writing code a bit easier by introducing the **_v** suffix for type traits expressing values
  - It's just a coding convention…
  - … but it makes a difference
    - …not as much as the _t convention did, but still

C++17: we're making progress

```
#include <type_traits>
template <class T>
   constexpr std::enable_if_t<!std::is_floating_point<T>::value, bool>
      close_enough(T a, T b) { return a == b; }
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T> constexpr auto threshold = static_cast<T>(0.000001);
template <class T>
   constexpr std::enable_if_t<std::is_floating_point<T>::value, bool>
      close_enough(T a, T b) {
          return absolute(a - b) <= threshold<T>;
      }
```



C++17: we're making progress

```
#include <type_traits>
template <class T>
   constexpr std::enable_if_t<!std::is_floating_point_v<T>, bool>
      close_enough(T a, T b) { return a == b; }
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T> constexpr auto threshold = static_cast<T>(0.000001);
template <class T>
   constexpr std::enable_if_t<std::is_floating_point_v<T>, bool>
      close_enough(T a, T b) {
          return absolute(a - b) <= threshold<T>;
      }
```

C++17: we're making progress

```
#include <type_traits>
template <class T>
   constexpr std::enable_if_t<!std::is_floating_point_v<T>, bool>
      close_enough(T a, T b) { return a == b; }
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T> constexpr auto threshold = static_cast<T>(0.000001);
template <class T>
   constexpr std::enable_if_t<std::is_floating_point_v<T>, bool>
      close_enough(T a, T b) {
          return absolute(a - b) <= threshold<T>;
      }
```

https://godbolt.org/z/vx5fGeqd5

C++17: we're making progress

- In our case, a bigger improvement was the advent of **if constexpr**
  - if constexpr requires a compile-time-known condition
  - If the condition is true, then only the « if » branch exists
  - If the condition is false, then only the « else » branch (if any) exists
- if constexpr is not a branch, it's a code generation mechanism
- This new feature allowed us to let go of enable_if, and write code that's both efficient and easy to understand

C++17: we're making progress

```
#include <type_traits>
template <class T> constexpr T absolute(T val) {
   return val < 0? -val : val;
}
template <class T> constexpr auto threshold = static_cast<T>(0.000001);
template <class T>
   constexpr bool close_enough(T a, T b) {
      if constexpr(std::is_floating_point_v<T>)
         return absolute(a - b) <= threshold<T>;
      else
         return a == b;
   }
```

C++17: we're making progress

```
#include <type_traits>
template <class T> constexpr T absolute(T val
   return val < 0? -val : val;
}
template <class T> constexpr auto threshold = stat                e<T>(0.000001);
template <class T>
   constexpr bool close_enough(T a, T b) {
      if constexpr(std::is_floating_point_v<T>)
         return absolute(a - b) <= threshold<T>;
      else
         return a == b;
   }
```

The beauty of this feature is that essentially anyone can understand it, and use it. It's what most programmers would intuitively try… without the inefficiency of an actual branch

C++17: we're making progress

```
#include <type_traits>
template <class T> constexpr T absolute(T val) {
   return val < 0? -val : val;
}
template <class T> constexpr auto threshold = stat          <T>(0.000001);
template <class T>
   constexpr bool close_enough(T a, T b) {
      if constexpr(std::is_floating_point_v<T>)
         return absolute(a - b) <= threshold<T>;
      else
         return a == b;
   }
```

https://godbolt.org/z/bze17jovo

SHORT RECAP…

Short recap…

- Let’s pause for a moment to see how language evolution and ease-of-use evolution led us from here to there

C++98/03

```
#include <cmath>

class exact {};

class floating {};

template <class T> bool close_enough(T a, T b, exact) { return a == b; }

template <class T> bool close_enough(T a, T b, floating) { return std::abs(a - b) <= static_cast<T>(0.000001); }

struct false_type { enum { value = false }; };

struct true_type { enum { value = true }; };

template <class> struct is_floating : false_type { };

template <> struct is_floating<float> : true_type { };

template <> struct is_floating<double> : true_type { };

template <> struct is_floating<long double> : true_type { };

template <bool, class T, class F> struct static_if_else;

template <class T, class F> struct static_if_else<true, T, F> { typedef T type; };

template <class T, class F> struct static_if_else<false, T, F> { typedef F type; };

template <class T>

  bool close_enough(T a, T b) { return close_enough(a, b, typename static_if_else<is_floating<T>::value, floating, exact>::type{});   }
```

C++11

```
#include <type_traits>
template <class T>
   constexpr
      typename std::enable_if<!std::is_floating_point<T>::value, bool>::type
         close_enough(T a, T b) { return a == b; }
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T>
   constexpr
      typename std::enable_if<std::is_floating_point<T>::value, bool>::type
         close_enough(T a, T b) {
            return absolute(a - b) <= static_cast<T>(0.000001);
         }
```

C++14

```
#include <type_traits>
template <class T>
   constexpr std::enable_if_t<!std::is_floating_point<T>::value, bool>
      close_enough(T a, T b) { return a == b; }
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T> constexpr auto threshold = static_cast<T>(0.000001);
template <class T>
   constexpr std::enable_if_t<std::is_floating_point<T>::value, bool>
      close_enough(T a, T b) {
          return absolute(a - b) <= threshold<T>;
      }
```

C++17

```
#include <type_traits>
template <class T> constexpr T absolute(T val) {
   return val < 0? -val : val;
}
template <class T> constexpr auto threshold = static_cast<T>(0.000001);
template <class T>
   constexpr bool close_enough(T a, T b) {
      if constexpr(std::is_floating_point_v<T>)
         return absolute(a - b) <= threshold<T>;
      else
         return a == b;
   }
```

C++20

C++20: adding elegance and beauty

- The C++17 approach to this problem is a significant upgrade over our starting point
- However, it still lacks elegance in many ways
  - It's « manual »
  - It's not well-suited to comparing to integrals of different types, or floating points of different types

C++20: adding elegance and beauty

- C++20 adds concepts
- Once one adds concepts to the mix, the trick gains in beauty and elegance

C++20: adding elegance and beauty

```
#include <type_traits>
#include <concepts>
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T> constexpr auto threshold = static_cast<T>(0.000001);
template <class T> requires std::integral<T>
   constexpr bool close_enough(T a, T b) {
      return a == b;
   }
template <class T> requires std::floating_point<T>
   constexpr bool close_enough(T a, T b) {
      return absolute(a - b) <= threshold<T>;
   }
```

C++20: adding elegance and beauty

```
#include <type_traits>
#include <concepts>
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T> constexpr auto threshold = static_cast<T>(0.000001);
template <class T> requires std::integral<T>
   constexpr bool close_enough(T a, T b) {
      return a == b;
   }
template <class T> requires std::floating_point<T>
   constexpr bool close_enough(T a, T b) {
      return absolute(a - b) <= threshold<T>;
   }
```

https://godbolt.org/z/srcazncrK

C++20: adding elegance and beauty

```
#include <type_traits>
#include <concepts>
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T> constexpr auto threshold = static_cast<T>(0.000001);
template <class T> requires std::integral<T>
   constexpr bool close_enough(T a, T b) {
      return a == b;
   }
template <class T> requires std::floating_point<T>
   constexpr bool close_enough(T a, T b) {
      return absolute(a - b) <= threshold<T>;
   }
```

Note that our C++17 version would have accepted close_enough("hi"sv,"ho"sv) whereas this one would not (we could add an overload for this if we wanted, of course)

C++20: adding elegance and beauty

```
#include <type_traits>
#include <concepts>
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T> constexpr auto threshold = static_cast<T>(0.000001);
template <std::integral T>
   constexpr bool close_enough(T a, T b) {
      return a == b;
   }
template <std::floating_point T>
   constexpr bool close_enough(T a, T b) {
      return absolute(a - b) <= threshold<T>;
   }
```

C++20: adding elegance and beauty

```
#include <type_traits>
#include <concepts>
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T> constexpr auto threshold = static_cast<T>(0.000001);
template <std::integral T>
   constexpr bool close_enough(T a, T b) {
      return a == b;
   }
template <std::floating_point T>
   constexpr bool close_enough(T a, T b) {
      return absolute(a - b) <= threshold<T>;
   }
```

https://godbolt.org/z/zYaebn4Kf

C++20: adding elegance and beauty

```
#include <type_traits>
#include <concepts>
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T> constexpr auto threshold = static_cast<T>(0.000001);
template <std::integral T>
   constexpr bool close_enough(T a, T b) {
      return a == b;
   }
template <std::floating_point T>
   constexpr bool close_enough(T a, T b) {
      return absolute(a - b) <= threshold<T>;
   }
```

Note that both arguments still have to be of the same type (two ints, two floats, two shorts…)

C++20: adding elegance and beauty

```
#include <type_traits>
#include <concepts>
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T> constexpr auto threshold = static_cast<T>(0.000001);
constexpr bool close_enough(std::integral auto a, std::integral auto b) {
   return a == b;
}
constexpr bool close_enough(std::floating_point auto a, std::floating_point auto b) {
   return absolute(a - b) <= threshold<
      std::common_type_t<decltype(a), decltype(b)>
   >;
}
```

C++20: adding elegance and beauty

```
#include <type_traits>
#include <concepts>
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T> constexpr auto threshold = static_cast<T>(0.000001);
constexpr bool close_enough(std::integral auto a, std::integral auto b) {
   return a == b;
}
constexpr bool close_enough(std::floating_point auto a, std::floating_point auto b) {
   return absolute(a - b) <= threshold<
      std::common_type_t<decltype(a), decltype(b)>
   >;
}
```

https://godbolt.org/z/nMW6GMzf7

C++20: adding elegance and beauty

```
#include <type_traits>
#include <concepts>
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T> constexpr auto threshold = static_cast<T>(0.000001);
constexpr bool close_enough(std::integral auto a, std::integral auto b) {
   return a == b;
}
constexpr bool close_enough(std::floating_point auto a, std::floating_point auto b) {
   return absolute(a - b) <= threshold<
      std::common_type_t<decltype(a), decltype(b)>
   >;
}
```

One goal of concepts was to « make generic programming normal programing »

C++20: adding elegance and beauty

```
#include <type_traits>
#include <concepts>
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <class T> constexpr auto threshold = static_cast<T>(0.000001);
constexpr bool close_enough(std::integral auto a, std::integral auto b) {
   return a == b;
}
constexpr bool close_enough(std::floating_point auto a, std::floating_point auto b) {
   return absolute(a - b) <= threshold<
      std::common_type_t<decltype(a), decltype(b)>
   >;
}
```

Note that we can now write close_enough(3,3L) as both arguments are integral

C++20: adding elegance and beauty

```
#include <type_traits>
#include <concepts>
template <class T> constexpr T absolute(T
template <class T> constexpr auto threshol
constexpr bool close_enough(std::integral
   return a == b;
}
constexpr bool close_enough(std::floating_point auto a, std::floating_point auto b) {
   return absolute(a - b) <= threshold<
      std::common_type_t<decltype(a), decltype(b)>
   >;
}
```

We can also write close_enough(3.0f,3.0) as both arguments are floating point numbers, but we have to make an effort inside to pick the right type for our threshold

C++20: adding elegance and beauty

```
#include <type_traits>
#include <concepts>
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <std::floating_point T> constexpr auto threshold = static_cast<T>(0.000001);
constexpr bool close_enough(std::integral auto a, std::integral auto b) {
   return a == b;
}
constexpr bool close_enough(std::floating_point auto a, std::floating_point auto b) {
   return absolute(a - b) <= threshold<
      std::common_type_t<decltype(a), decltype(b)>
   >;
}
```

C++20: adding elegance and beauty

```
#include <type_traits>
#include <concepts>
template <class T> constexpr T absolute(T val) { return val < 0? -val : val; }
template <std::floating_point T> constexpr auto threshold = static_cast<T>(0.000001);
constexpr bool close_enough(std::integral auto a, std::integral auto b) {
   return a == b;
}
constexpr bool close_enough(std::floating_point auto a, std::floating_point auto b) {
   return absolute(a - b) <= threshold<
      std::common_type_t<decltype(a), decltype(b)>
   >;
}
```

While we're at it :)

C++20: adding elegance and beauty

```
#include <type_traits>
#include <concepts>
#include <numeric>
template <class T> constexpr T absolute(T 
template <std::floating_point T> constexpr
constexpr bool close_enough(std::integral
   return a == b;
}
constexpr bool close_enough(std::floating_point auto a, std::floating_point auto b) {
   return absolute(std::midpoint(a,b)) <= threshold<
      std::common_type_t<decltype(a), decltype(b)>
   >;
}
```

~~Oh, and we can now (at last!) be correct here~~

I made a mistake adding this at the last minute before my talk just to be « cute ». What we would need is a difference(a,b) function that would probably use midpoint() within, but the code I presented is incorrect (my apologies)

C++20: adding elegance and beauty

```
// ...
template <class T, class ... Ts>
   constexpr bool all_close_enough(T val, Ts ... vals) {
      return (close_enough(val, vals) && ...);
   }
```

C++20: adding elegance and beauty

```
// ...
template <class T, class ... Ts>
   constexpr bool all_close_enough(T val, Ts ... vals) {
      return (close_enough(val, vals) && ...);
   }
int main() {
   static_assert(all_close_enough(3,3L,3u), "");
   static_assert(all_close_enough(3.0,3.000000001,3.0f));
   // static_assert(all_close_enough(3,3.000000001,3.0f));
}
```

C++20: adding elegance and beauty

```
// ...
template <class T, class ... Ts>
   constexpr bool all_close_enough(T val, Ts ... vals) {
      return (close_enough(val, vals) && ...);
   }
int main() {
   static_assert(all_close_enough(3,3L,3u), "");
   static_assert(all_close_enough(3.0,3.000000001,3.0f));
   // static_assert(all_close_enough(3,3.000000001,3.0f));
}
```

Concept mismatch (integral and floating point)

LOOKING BACK…

Looking back…

- C++ has always been expressive
- We could do efficiently in 1998 what many languages cannot do efficiently today

Looking back…

```
// C# (example), generic
class Integral : IEquatable<Integral>
{
    public int Value{ get; init; }
    // ...
    public bool Equals(Integral other) =>
        other != null && Value == other.Value;
    public override bool Equals(object other) =>
        other is Integral n && Equals(n);
}
```

Looking back…

```
// C# (example), generic
// ...
static bool CloseEnough<T>(T a, T b)
   where T : IEquatable<T> =>
      a.Equals(b);
```

Looking back…

```
// C# (example), generic
// ...
static bool CloseEnough<T>(T a, T b)
    where T : IEquatable<T> =>
        a.Equals(b);
```

I don't think there's an actual clean solution for CloseEnough<T,U> in that language (but if I'm wrong, write to me!)

Looking back…

```
// C# (example), generic
// ...
static bool CloseEnough<T>(T a, T b)
   where T : IEquatable<T> =>
      a.Equals(b);
```

With C++, it mostly just falls out of the type system!

Looking back…

- C++ has always been expressive
- We could do efficiently in 1998 what many languages cannot do efficiently today
- There's beauty in efficiency
- There's beauty in generality

Looking back…

- C++ has always been expressive
- We could do efficiently in 1998 what many languages cannot do efficiently today
- There's beauty in efficiency
- There's beauty in generality
- There's beauty in C++

THANK YOU