SOME THINGS C++ DOES RIGHT

Patrice Roy

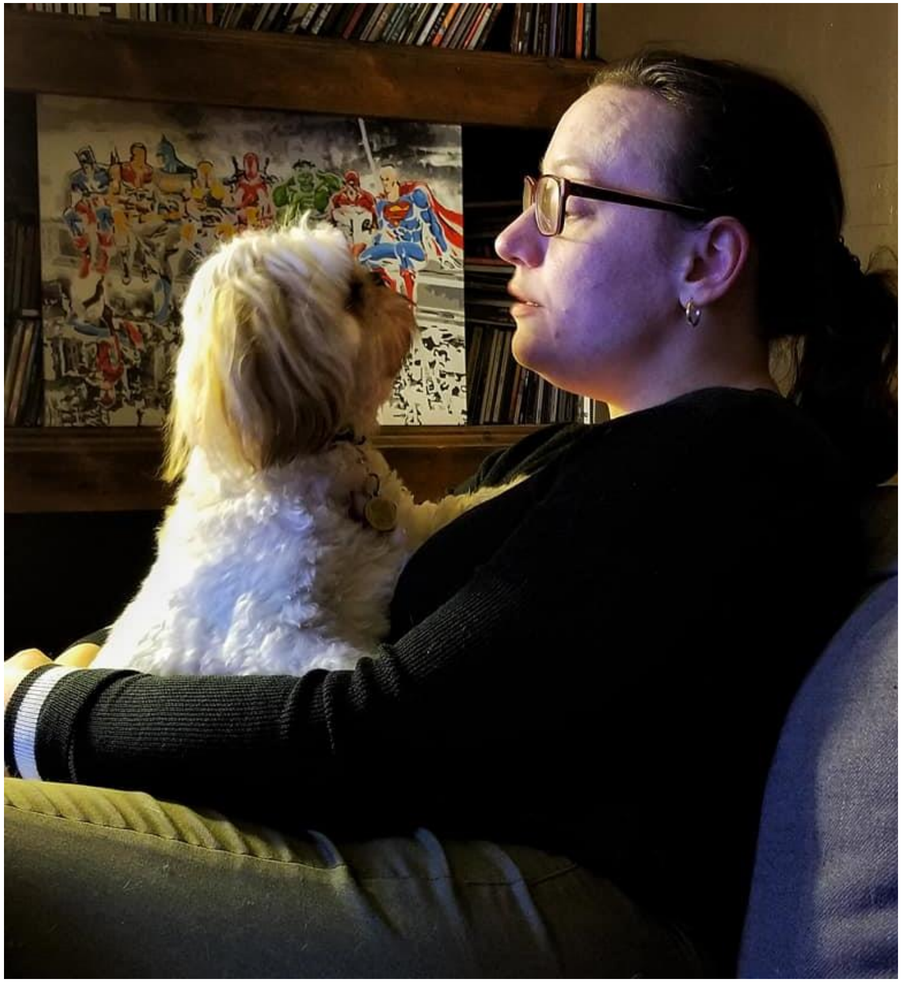
Patrice.Roy@USherbrooke.ca; Patrice.Roy@clg.qc.ca

CeFTI, Université de Sherbrooke; Collège Lionel-Groulx

Who am I?

- Father of five (four girls, one boy), ages 25 to 7
- 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…

There’s a rumor going on…



You might have heard…

- C++ is not memory-safe enough

You might have heard…

- C++ is not memory-safe enough
- C++ is not type-safe enough

You might have heard…

- C++ is not memory-safe enough
- C++ is not type-safe enough
- C++ is a language where some defaults are wrong

You might have heard…

- C++ is not memory-safe enough
- C++ is not type-safe enough
- C++ is a language where **~~some~~ all** defaults are wrong

You might have heard…

- C++ is not memory-safe enough
- C++ is not type-safe enough
- C++ is a language where ~~some~~ all defaults are wrong
- C++ is too expert-friendly

You might have heard…

- There's often a grain of truth in criticism, and C++ surely has a bit of each of these alleged warts

You might have heard…

- There's often a grain of truth in criticism, and C++ surely has a bit of each of these alleged warts
- It's a language that has history, obviously, and that has evolved organically over the years, and it has the imperfections we can expect for a tool used by millions to perform high-performance or safety-critical tasks in various application domains.

You might have heard…

- However, there are a **significant** number of things C++ does *right*

You might have heard…

- However, there are a **significant** number of things C++ does *right*
- There are a number of reasons why we love this language…

You might have heard…

- However, there are a **significant** number of things C++ does *right*
- There are a number of reasons why we love this language…
- …love it so much that we gather together to trade ideas, learn about it, understand it better… and enjoy it all!

You might have heard…

- This talk is about some of the things C++ does *right*

You might have heard…

- This ta

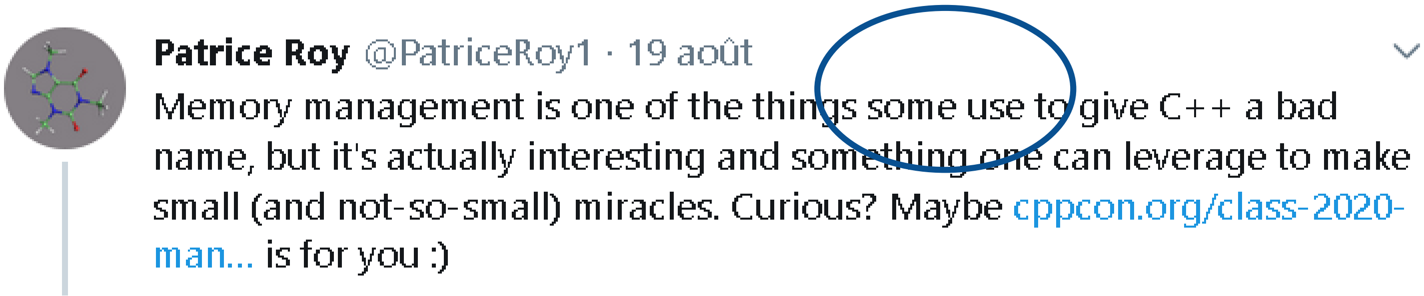
Me, promoting my CppCon 2020 class:
https://twitter.com/PatriceRoy1/status/1296153449235652609?s=20

Patrice Roy @PatriceRoy1 · 19 août

Memory management is one of the things some use to give C++ a bad name, but it's actually interesting and something one can leverage to make small (and not-so-small) miracles. Curious? Maybe cppcon.org/class-2020-man… is for you :)

You might have heard…

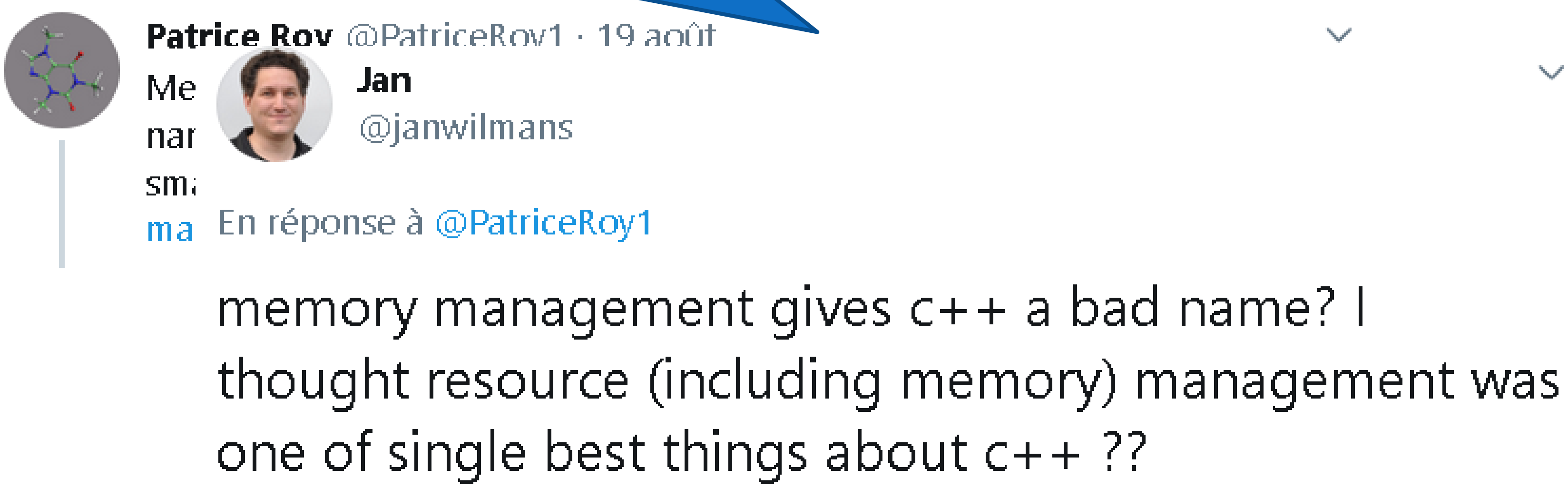
- This talk is about some of the things C++ does *right*



You might have heard…

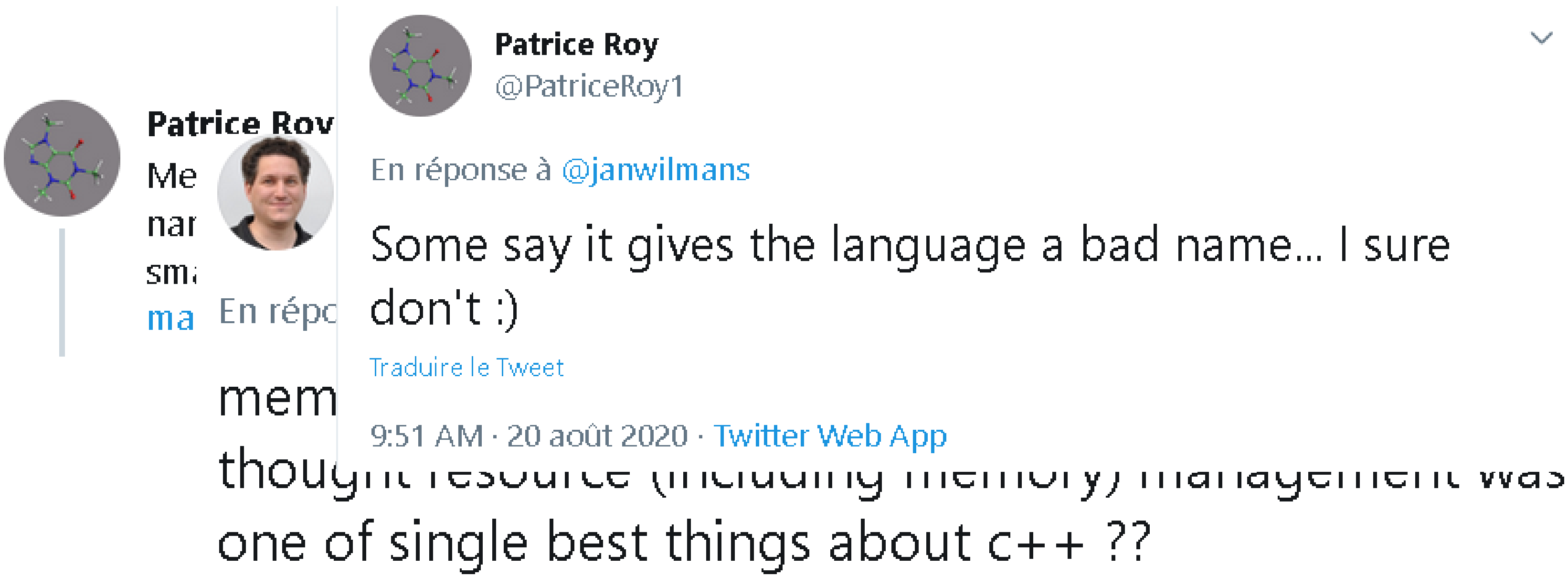
- This talk is about some of the things C++ does *right*

Immediate reaction:
https://twitter.com/janwilmans/status/1296183027731705856?s=20



You might have heard…

- This talk is about some of the things C++ does *right*



You might have heard…

- This talk is about some of the things C++ does *right*
  - It does not aim to provide an exhaustive list (far from it!)
  - It does not aim to throw arrows at other languages
    - Although there *will* be comparisons
  - It does not aim to offer an apologetic perspective on C++
    - That would be pointless, really

You might have heard…

- This talk is about some of those things one *misses* when using other languages
- It aims to remind us of some of those things that make C++ *beautiful*, *fun* and *efficient*

You might have heard…

- This talk is about some of those things one *misses* when using other languages
- It aims to remind us of some of those things that make C++ *beautiful*, *fun* and *efficient*

I'll be using C# for comparisons, but it's only because I have to use it often these days…

…and because it has a good reputation in some circles…

…and because *it makes me miss C++ quite a lot*!

You might hav

This is not a « C++ is better than C# » talk; it's a « C++ does some things right » talk.

C# is a *fine* language for what it aims to do, and has many strengths. The same goes for Java, JavaScript, Python, C, Haskell…

- This talk is about some of t
  other languages
- It aims to remind us of some of those things that m +
  beautiful, fun and efficient

I'll be using C# for comparisons, but it's only because I have to use it often these days…

…and because it has a good reputation in some circles…

…and because *it makes me miss C++ quite a lot*!

A word about beauty and elegance

A word about beauty and elegance

- Beauty is in the eye of the beholder
  - Beauty is cultural
- There’s beauty in most programming languages
  - C++ has warts, but it’s often beautiful

A word about beauty and elegance

- **Situation**: we have a game with all sorts of monsters
  - We made a class hierarchy managed within objects of a pImpl class named `Monster` (Do as the `int`s do!)
    - Write to me if you don't know how to do this (you can play with the pImpl idiom and add a few twists)
    - It's not difficult, but it would distract us from this talk
- There's a huge, carnage-style fight, after which we want to remove the dead monsters and thus, only keep the ones left alive

A word about beauty and elegance

```
// if we were in C#
static List<Monster>
  RemoveDead(List<Monster> lst)
{
  lst.RemoveAll(m => !m.IsAlive);
  return lst;
}
```

A word about beauty and elegance

```
// if we were in C#
static List<Monster>
  RemoveDead(List<Monster> lst)
{
  lst.RemoveAll(m => !m.IsAlive);
  return lst;
}
```

Simple and nice...

A word about beauty and elegance

```
// if we were in C#
static List<Monster>
  RemoveDead(List<Monster> lst)
{
  lst.RemoveAll(m => !m.IsAlive);
  return lst;
}
```

Simple and nice…

Note however that `lst` is not an object; it's a *reference* to an object, so we just modified what that `List<Monster>` refers to in the calling code!

A word about beauty and elegance

```
vector<Monster>
  remove_dead(vector<Monster> v) {
    auto p = remove_if(
      begin(v), end(v), [](auto &&m) {
        return !m.alive();
    });
    return { begin(v), p };
  }
```

A word about beauty and elegance

```
vector<Monster>
  remove_dead(vector<Monster> v) {
    auto p = remove_if(
      begin(v), end(v), [](auto &&m) {
        return !m.alive();
    });
    return { begin(v)
  }
```

Alexander Stepanov has brought much beauty to the language. We owe him and his team for some of the beauty in this case

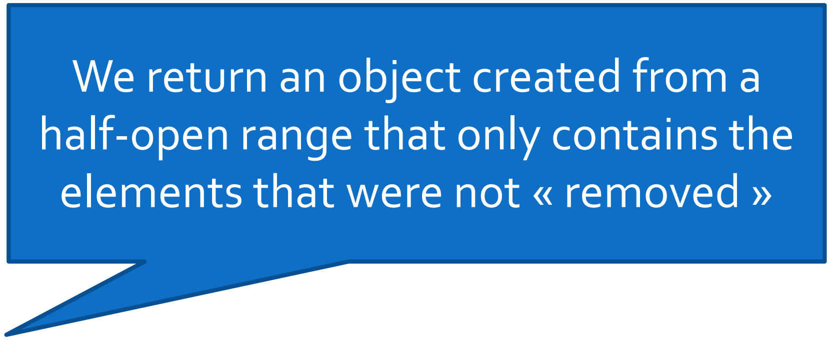
A word about beauty and elegance

```
vector<Monster>
  remove_dead(vector<Monster> v) {
    auto p = remove_if(
      begin(v), end(v), [](auto &&m) {
        return !m.alive();
    });
    return { begin(v), p };
  }
```

We have an algorithm (`remove_if`) that operates on a half-open range, and « removes » the elements that do not respect a predicate (here, a λ)

A word about beauty and elegance

```
vector<Monster>
  remove_dead(vector<Monster> v) {
    auto p = remove_if(
      begin(v), end(v), [
        return !m.alive()
    });
    return { begin(v), p };
  }
```



A word about beauty and elegance

C# goes for specifics

```
lst.RemoveAll(
  m => !m.IsAlive
);
return lst;
```

C++ goes for general

```
auto p = remove_if(
  begin(v), end(v),
  [](auto &&m) {
    return !m.alive();
});
return { begin(v), p };
```

A word about beauty and elegance

C# goes for specifics

```
lst.RemoveAll(
  m => !m.IsAlive
);
return lst;
```

Both languages generalize quite differently. C# code is smaller but relies on a specific collection type (e.g.: it won’t work with an array)

C++ goes for general

```
auto p = remove_if(
  begin(v), end(v),
  [](auto &&m) {
    return !m.alive();
});
return { begin(v), p };
```

A word about beauty and elegance

C# goes for specifics

```
lst.RemoveAll(
  m => !m.IsAlive
);
return lst;
```

C++ goes for general

```
auto p = remove_if(
  begin(v), end(v),
  [](auto &&m) {
}
return { begin(v), p };
```

Note that it’s two instructions in both cases, however

Both languages generalize quite differently. ***C# code is smaller*** but relies on a specific collection type (e.g.: it won’t work with an array)

A word about beauty and elegance

C# goes for specifics

```
lst.RemoveAll(
  m => !m.IsAlive
);
return lst;
```

C++ goes for general

```
auto p = remove_if(
  begin(v), end(v),
  [](auto &&m) {
    return !m.alive();
});
return { begin(v), p };
```

The C++ version works for a family of containers: `remove_if` works for an array; the return statement works for a family of containers

A word about beauty and elegance

- There is a form of beauty from being able to compose elegant and efficient solutions from a small, basic set of principles
  - That's one of the things C++ does well
  - We owe a lot to Alexander Stepanov and his team

A word about beauty and elegance

- **Situation**: you want to read all text from a text file and make a string out of it
- You hear from people working in C# that it offers a `File.ReadAllText(string path)` function which does precisely that
  - There's no such thing in C++'s standard library

A word about beauty and elegance

```
// if we were in C#
using System;
using System.IO;
using static System.Console; // recent C#
class Program
{
  static void Main()
  {
    Write(File.ReadAllText("z.cs"));
  }
}
```

A word about beauty and elegance

```
// if we were in C#
using System;
using System.IO;
using static System.Console; // recent C#
class Program
{
  static void Main()
  {
    Write(File.ReadAllText("z.cs"));
  }
}
```

Nice and simple… because someone else did it for us

A word about beauty and elegance

```
// if we were in C#
using System;
using System.IO;
using static System.Consol
class Program
{
  static void Main()
  {
    Write(File.ReadAllText("z.cs"));
  }
}
```

It’s non-trivial to write « by hand » in that language. Not *extremely* complex, but a bit tricky if we want to be efficient.

A word about beauty and elegance

```
// ...
string
  read_all_text(const string &name) {
    ifstream in{ name };
    return {
      istreambuf_iterator<char>{ in },
      istreambuf_iterator<char>{}
    };
  }
int main() {
   cout << read_all_text("z.cpp");
}
```

A word about beauty and elegance

```
// ...
string
  read_all_text(const string &name) {
    ifstream in{ name };
    return {
      istreambuf_iterator<
      istreambuf_iterator<
    };
  }
int main() {
   cout << read_all_text("z.cpp");
}
```

That's it. We only need a `using` (for `namespace std`) and a few standard includes (`iostream`, `fstream`, `string`, `iterator`)

A word about beauty and elegance

```
// ...
string
  read_all_text(const string &name) {
    ifstream in{ name };
    return {
      istreambuf_iterator<char>{ in },
      istreambuf_iterator<cha
    };
  }
int main() {
   cout << read_all_text("z.c
}
```

There's no need for special case functions; C++ requires you to know some basic principles (iterators, half-open ranges) and one can build from these

A word about beauty and elegance

```
// ...
int main() {
    cout << "z.cpp"_file;
}
```

A word about beauty and elegance

```
// ...
int main() {
   cout << "z.cpp"_file;
}
```

Some will like it, some will not. There’s a « coolness factor » to it, at least

A word about beauty and elegance

```
// ...
string read_all_text(const string &name) {
  ifstream in{ name };
  return {
    istreambuf_iterator<char>{ in },
    istreambuf_iterator<char>{}
  };
}
auto operator "" _file(const char *s, size_t) {
  return read_all_text(s);
}
int main() {
  cout << "z.cpp"_file;
}
```

A word about beauty and elegance

```
// ...
string read_all_text(const string &name) {
  ifstream in{ name };
  return {
    istreambuf_iterator<char>{ in },
    istreambuf_iterator<char>{}
  };
  }
auto operator "" _file(const char *s, size_t) {
  return read_all_text(s);
}
int main() {
  cout << "z.cpp"_file;
}
```

That's it. We need nothing more than what we needed for `read_all_text`.

A word about beauty and elegance

- There is a form of beauty from being able to compose elegant and efficient solutions from a small, basic set of principles… without waiting for someone else to fix it for us
  - There is clearly value in frameworks and class libraries to « get the job done »…
  - … but in terms of beauty and elegance, it’s nice to know C++ provides such rich and fecund abstractions to build from

A word about beauty and elegance

- **Situation**: you want to apply a composition of functions `f` and `g` to each argument `x` in a range of values

A word about beauty and elegance

- The intent is to replace something like

```
transform(begin(v), end(v), begin(v), g);
transform(begin(v), end(v), begin(v), f);
```

- … which does two passes through `v`, with

```
transform(begin(v), end(v), begin(v),
          f_g_x(f,g));
```

- … which only does a single pass
  - … and might at the same time benefit from the return value optimization (RVO) better

A word about beauty and elegance

```
template <class F, class G>
  auto f_g_x(F f, G g) {
    return [=](auto && x) {
      return f(g(x));
    };
  }
```

A word about beauty and elegance

```
template <class F, class G>
  auto f_g_x(F f, G g) {
    return [=](auto && x) {
      return f(g(x));
    };
  }
```

I like this one. It creates the functional composition of expression `f(g(x))` as a short and simple one-liner

A word about beauty and elegance

```
template <class F, class G>
  auto f_g_x(F f, G g) {
    return [=](auto && x) {
      return f(g(x));
    };
  }
```

We are creating a λ that copies `f` and `g`, then returns something that will return `f(g(x))` when called for some `x` in whatever type or form. It even works with overload sets

A wor
elega

```
templ
  aut
    ret
      re
    };
  }
```

Before C++14, this was… unpleasant

```
template <class F, class G>
   class f_g_x_impl {
      F f; G g;
   public:
      f_g_x_impl(F f, G g) : f{ f }, g{ g } {
      }
      template <class T>
         auto operator()(T && x) const {
            return f(g(x));
         }
   };
template <class F, class G>
   f_g_x_impl<F,G> f_g_x(F f, G g) {
      return { f, g };
   }
```

A wo
elega

```
templ
  aut
    ret
      re
    };
  }
```

Before C++14, this was… unpleasant

```
template <class F, class G>
   class f_g_x_impl {
      F f; G g;
   public:
      f_g_x_impl(F f, G g) : f{ f }, g{ g } {
      }
      template <class T>
         auto operator()(T && x) const {
            return f(g(x));
         }
   };
   te <class F, class G>
   _x_impl<F,G> f_g_x(F f, G g) {
      return { f, g };
   }
```

In C++11 without `auto` return types, this is *particularly* painful

A word about beauty and elegance

```
template <class F, class G>
  auto f_g_x(F f, G g) {
    return [=](auto && x) {
      return f(g(x));
    };
  }
```

In C#, we'd have something like this

```
static Func<T,R>
   F_G_X<T,U,R>(Func<U,R> f, Func<T,U> g) =>
      x => f(g(x));
```

A word about elegance

```
template <class F, c
  auto f_g_x(F f, G
    return [=](auto
      return f(g(x)
    };
  }
```

In C#, we'd have something like this

```
static Func<T,R>
   F_G_X<T,U,R>(Func<U,R> f, Func<T,U> g) =>
      x => f(g(x));
```

… but the calling code has to explicitly state types `T`, `U` and `R`:

```
public static void Main()
{
   var f = F_G_X<int,int,int>
   (
      n => -n, n => n * 2
   );
   Console.WriteLine(f(3));
}
```

A word about beauty and elegance

```
// C++
template <class F, class G>
  auto f_g_x(F f, G g) {
    return [=](auto && x) {
      return f(g(x));
    };
  }
// C#
static Func<T,R>
  F_G_X<T,U,R>(Func<U,R> f, Func<T,U> g)
     => x => f(g(x));
```

A word about beauty and elegance

```
// C++
template <class F, cl
  auto f_g_x(F f, G g) {
    return [=](auto && x
      return f(g(x));
    };
  }
// C#
static Func<T,R>
  F_G_X<T,U,R>(Func<U,R> f, Func<T,U> g)
     => x => f(g(x));
```

```
auto f = f_g_x(
  [](auto x) { return -x; },
  [](auto x) { return x * 2; }
);
```

```
var f = F_G_X<int,int,int>
(
  n => -n,
  n => n * 2
);
```

A word about beauty and elegance

```
template <class F, class G>
  auto f_g_x(F f, G g) {
    return [=](auto && x) {
      return f(g(
        std::forward<decltype(x)>(x)
      ));
    }
  }
```

A word about beauty and elegance

```
template <class F, class G>
  auto f_g_x(F f, G g) {
    return [=](auto && x) {
      return f(g(
        std::forward<decltype(x)>(x)
      ));
    }
  }
```

C++ gives you control…

A word about beauty and elegance

```
template <class F, class G>
  auto f_g_x(F f, G g) {
    return [=](auto && x) {
      return f(g(
        std::forward<decltype(x)>(x)
      ));
    }
  }
```

C++ gives you control…

Is it a wart or something beautiful?

A word about beauty and elegance

- **Situation**: we want to apply a function `f` in sequence to each argument of a function
  - Note that this is not from me, but it made its way from twitter to isocpp.org when it came out
  - It actually fits in a tweet!

A word about beauty and elegance

```
template <class F,
          class... Args>
   void for_each_argument
      (F f, Args&&... args) {
    [](...){}
       ((f(std::forward<Args>(args)),
         0)...);
   }
```

A word about beauty and elegance

```
template <class F,
          class... Args>
   void for_each_argument
      (F f, Args&&... args) {
    [](...){}
      ((f(std::forward<Args>(args)),
        0)...);
   }
```

This beautiful piece of work is Sean Parent's (see https://isocpp.org/blog/2015/01/for-each-argument-sean-parent)

A word about beauty and elegance

```
template <class F,
          class... Args>
   void for_each_argument
      (F f, Args&&... args) {
    [](...){}
       ((f(std::forward<Args>(args)),
         0)...);
   }
```

We have a void lambda that takes *any number of arguments of any type* and does *nothing* with them…

A word about beauty and elegance

```
template <class F,
          class... Args>
   void for_each_argument
      (F f, Args&&... args) {
    [](...){}
      ((f(std::forward<Args>(args)),
        0)...);
   }
```

... and we pass it a sequence of zeros, but after applying `f` to each argument in sequence.

A fold expression over `operator,()` ...

A word about beauty and elegance

```
template <class F,
          class... Args>
   void for_each_argument
      (F f, Args&&... args) {
    [](...){}
       ((f(std::forward<Args>(args)),
         0)...);
   }
```

Beautiful? Clever?
...both?

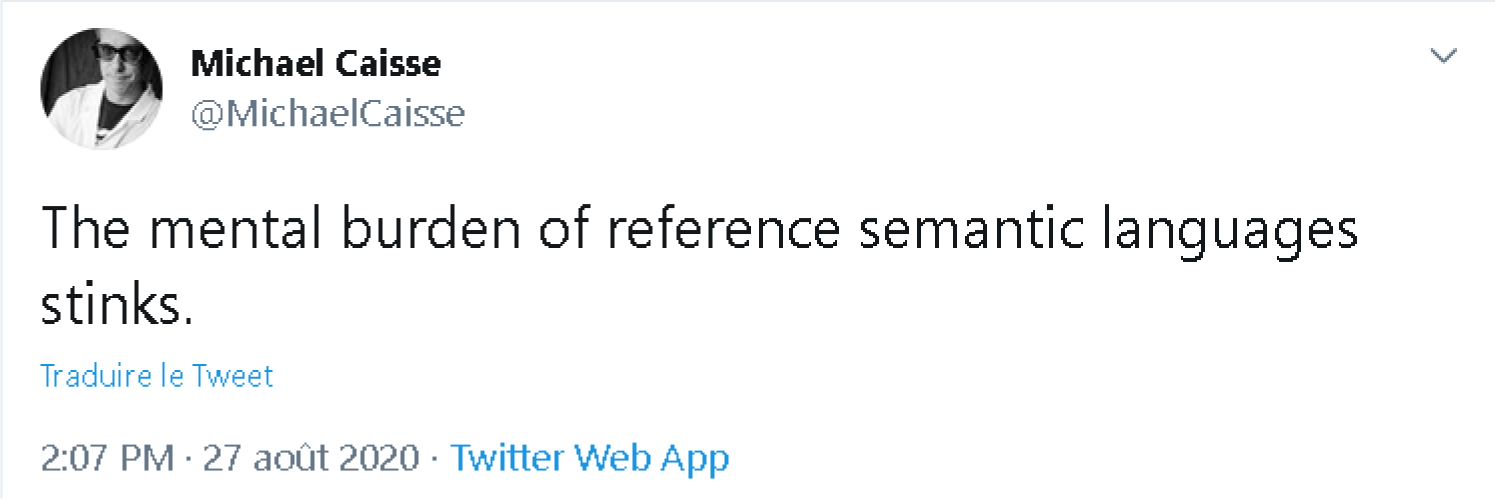
A word about beauty and elegance

```
template <class F,
          class... Args>
   void for_each_argument
      (F f, Args&&... args) {
    [](...){}
       ((f(std::forward<Args>(args)),
         0)...);
   }
```

I don't have a C# example... As far as I know, it's not in the realm of things-that-can-be-done in that language

Programming with a value-based language

Programming with a value-based language



- https://twitter.com/MichaelCaisse/status/1299045933603098624

Programming with a value-based language

- C++ favors values and direct access over indirections
  - Values are what we get from the syntax unless we make some effort to obtain something else
  - This influences the way we think and code, including how our objects are initialized

Programming with a value-based language

- We can (and do!) criticize initialization in C++
  - There are so many ways…
  - … but that's true in other languages too, and (I think) despite all the criticisms, C++ does the fundamentals right

Programming with a value-based language

- We can (and do!) criticize initialization in C++
  - There are so many ways…
  - … but that's true in other languages too, and (I think) despite all the criticisms, C++ does the fundamentals right

```
int x0; // ?
int x1 = 0; // limited
int x2 = int(); // 0
int x3(); // oops!
int x4 = {}; // 0
int x5{}; // 0
auto x6 = 0; // literal 0 is of type int
auto x7 = int(0); // 0
auto x8 = int{}; // 0
auto x9 = int(); // 0
```

Programming with a value-based language

- We can (and do!) criticize initialization in C++
  - There are so many ways…
  - … but that's true in other languages too, and (I think) despite all the criticisms, C++ does the fundamentals right

```
string s0; // empty string
string s1 = ""; // limited
string s2 = string(); // empty string
string s3(); // oops!
string s4 = {}; // empty string
string s5{}; // empty string
auto s6 = ""; // oops!
auto s7 = ""s; // empty string
auto s8 = string{}; // empty string
auto s9 = string(); // empty string
```

Programming with a value-based language

- Initialization is complicated elsewhere too
  - Maybe less so, but…

```
// C#
string s0; // null, but considered not initialized
string s1 = null; // null
string s2 = ""; // empty string
string s3 = default; // null
var s4 = new string(); // does not compile
var s5 = new string(""); // empty string
```

Programming with a value-based language

- Initialization is complicated elsewhere too
  - Maybe less so, but…

```
// C#
string s = null;
Console.WriteLine(s.Length); // boom!
```

Programming with a value-based language

- Initialization is complicated elsewhere too
  - Maybe less so, but…

```
// C#
string s = null;
s += "Yo"; // fine
Console.WriteLine(s.Length); // 2
```

Programming with a value-based language

- Initialization is complicated elsewhere too
  - Maybe less so, but…

```
// C#
string s = null;
s += "Yo"; // fine
e.W iteLine(s.Length); // 2
```

This is because, in C#, `a+=b` is rewritten as `a=a+b` and the `null` operand to `operator+` is considered to be an empty string

Programming with a value-based language

- Initialization is complicated elsewhere too
  - Maybe less so, but…

```
// C#
string s = null;
s += null; // fine
Console.Write(s.Length); // compiles, outputs 0
```

Programming with a value-based language

- Initialization is complicated elsewhere too
  - Maybe less so, but…

```
// C#
string s = null;
s += null; // fine
Console.Write(s.Length); // compiles, outputs 0
```

Icky…

Programming with a value-based language

- Through all the criticism, C++ initialization has a saner model than many will give it credit for

Programming with a value-based language

```
class Integral {
   int value_ = create_value();
public:
   static int create_value() {
      std::cout << "in create_value\n";
      return {};
   }
   int value() const {
      return value_;
   }
   Integral() = default;
   Integral(int value) : value_{ value } {
   }
};
```

Programming with a value-based language

```
class Integral {
   int value_ = create_value();
public:
   static int create_valu
      std::cout << "in cr
      return {};
   }
   int value() const {
      return value_;
   }
   Integral() = default;
   Integral(int value) : value_{ value } {
   }
};
```

```
int main() {
   [[maybe_unused]] auto i0 = Integral{};
   [[maybe_unused]] auto i1 = Integral{ 3 };
}
```

Programming with a value-based language

```
class Integral {
   int value_ = create_va
public:
   static int create_valu
      std::cout << "in cr
      return {};
   }
   int value() const {
      return value_;
   }
   Integral() = default;
   Integral(int value) : value_{ value } {
   }
};
```

```
int main() {
   [[maybe_unused]] auto i0 = Integral{};
   [[maybe_unused]] auto i1 = Integral{ 3 };
}
```

Displays "in create_value" only once, for `i0`

Programming with a value-based language

```
class Integral
{
    static int CreateValue()
    {
        Console.WriteLine("In CreateValue");
        return default;
    }
    public int Value { get; } = CreateValue();
    public Integral() {} // Ok, will call CreateValue
    public Integral(int value)
    {
        Value = value; // Oops!
    }
}
```

Programming with a value-based language

```
class Integral
{
   static int CreateValue()
   {
      Console.WriteLine("In CreateVa
      return default;
   }
   public int Value { get; } = CreateValue();
   public Integral() {} // Ok, will call CreateValue
   public Integral(int value)
   {
      Value = value; // Oops!
   }
}
```

```
public static void Main()
{
   var i0 = new Integral();
   var i1 = new Integral(3);
}
```

Programming with a value-based language

```
class Integral
{
   static int CreateValue()
   {
      Console.WriteLine("In CreateVa
      return default;
   }
   public int Value { get; } = CreateValue();
   public Integral() {} // Ok, will call Cr
   public Integral(int value)
   {
      Value = value; // Oops!
   }
}
```

```
public static void Main()
{
   var i0 = new Integral();
   var i1 = new Integral(3);
}
```

Displays "in CreateValue" twice!

Programming with a value-based language

- In C++, our objects are values by default
  - We have to make an effort (add syntax) to get indirect access
  - An object actually construct its data members by calling their constructors before beginning its own construction

Programming with a value-based language

```
class Person {
   string name_;
public:
   // default-constructs name_
   Person(const string &name) {
      name_ = name; // inefficient
   }
};
```

Programming with a value-based language

```
class Person {
    string name_;
public:
    // copy-constructs name_
    Person(const string &name)
        : name_{ name } { // much better!
    }
};
```

Programming with a value-based language

- In a reference-based language like C# or Java, the mindset is different
  - One gets indirect access by default for class instances
  - Objects are zeroed by default by `new`, so references start `null`

Programming with a value-based language

```
class Person
{
    string Name{ get; }
    // Name is null initially
    public Person(string name)
    {
        Name = name; // copies a pointer
    }
}
```

Programming with a value-based language

```
class Person
{
    static string GenerateName()
    {
        // some involved code goes here
    }
    string Name{ get; } = GenerateName();
    // Name is initialized by GenerateName
    public Person(string name)
    {
        // this follows initialization
        // (wasteful call to GenerateName)
        Name = name;
    }
}
```

Programming with a value-based language

```
class Person
{
    static string GenerateName()
    {
        // some involved code goes h
    }
    string Name{ get; } = GenerateName();
    // Name is initialized by GenerateName
    public Person(string name)
    {
        // this follows initialization
        // (wasteful call to GenerateName)
        Name = name;
    }
}
```

C++ avoids wasteful initializations in such cases

Programming with a value-based language

- Encapsulation is hard, *really* hard, when using a reference-based language
  - People underestimate this
  - This problem is exacerbated with generic code

Programming with a value-based language

```
// C#
class Bag<T>
{
   List<T> Contents{ get; }
   public T this[int n]
   {
      get => Contents[n];
   }
   public Bag(List<T> src)
   {
      Contents = src.Count <= 5?
         src : throw new Exception("Too big");
   }
}
```

Programming with a value-based language

The absence of a 'set' here means it can only be modified in the constructor

```
// C#
class Bag<T>
{
   List<T> Contents{ get; }
   public T this[int n]
   {
      get => Contents[n];
   }
   public Bag(List<T> src)
   {
      Contents = src.Count <= 5?
         src : throw new Exception("Too big");
   }
}
```

Programming with a value-based language

```
// C#
class Bag<T>
{
   List<T> Contents{ get; }
   public T this[int n]
   {
      get => Contents[n];
   }
   public Bag(List<T> src)
   {
      Contents = src.Count <= 5?
         src : throw new Exception("Too big");
   }
}
```

This is an indexer, by the way. It's just C#'s version of `operator[]`

We only wrote the 'get' part (not the 'set' part) because we don't want the elements of a `Bag<T>` to mutate

Programming with a value-based language

```
// C#
class Bag<T>
{
   List<T> Contents{ get; }
   public T this[int n]
   {
      get => Contents[n];
   }
   public Bag(List<T> src)
   {
      Contents = src.Count <= 5?
         src : throw new Exception("Too big");
   }
}
```

We want a `Bag<T>` to be small (no more than five elements)

Programming with a value-based language

This code is *quite* broken…

```
// C#
class Bag<T>
{
   List<T> Contents{ get; }
   public T this[int n]
   {
      get => Contents[n];
   }
   public Bag(List<T> src)
   {
      Contents = src.Count <= 5?
         src : throw new Exception("Too big");
   }
}
```

Programmi
language

```
// C#
class Bag<T>
{
   List<T> Contents{ g
   public T this[int n
   {
      get => Contents[n];
   }
   public Bag(List<T> src)
   {
      Contents = src.Count <= 5?
         src : throw new Exception("Too big");
   }
}
```

```
static void Evil()
{
   var lst = new List<int>()
   {
      2,3,5,7,11 // five elements
   };
   var bag = new Bag<int>(lst); // fine
   lst.Add(-1); // oops! Broken invariant
}
```

Programming with a value-based language

```
// C#
class Bag<T>
{
    List<T
    public T
    {
        get => Con
    }
    public Bag(Li
    {
        Contents =
            src : t
    }
}
```

```
class Thing
{
    public int Val { get; set; }
    public Thing(int val)
    {
        Val = val;
    }
}
static void Sad()
{
    var bag = new Bag<Thing>(new List<Thing>()
    {
        new Thing(2), new Thing(3), new Thing(5)
    });
    bag[2].Val *= -1; // oops! Mutated element
}
```

Programming with a value-based language

- Reference semantics with mutable state makes it really hard to reason locally
  - Pointers, pointers everywhere
  - Does not make concurrency any simpler…
- C++ takes us away from reference semantic in many cases
  - At least, “modern C++” tends to do this
  - Even pointers are encouraged to have unique ownership

Programming with a value-based language

```
// C++
class too_big{};
template <class T> class Bag {
  vector<T> contents;
public:
  // we could return const T& too
  T operator[](int n) const {
    return contents[n];
  }
  Bag(const vector<T> &src)
    : contents{ src.size() <= 5? src : throw too_big{} } {
  }
}
```

Programming with a value-based language

Essentially the same code, but without the problems (and with actual encapsulation)

```
// C++
class too_big{};
template <class T> class Bag {
  vector<T> contents;
public:
  // we could return const T& too
  T operator[](int n) const {
    return contents[n];
  }
  Bag(const vector<T> &src)
    : contents{ src.size() <= 5? src : throw too_big{} } {
  }
}
```

Programming with a value-based language

Essentially the same code, but without the problems (and with actual encapsulation)

```
// C++
class too_big{};
template <class T> class Bag {
  vector<T> contents;
public:
  // we could return const T& too
  T operator[](int n) const {
    return contents[n];
  }
  Bag(const vector<T> &src)
    : contents{ src.size() <= 5? src : throw too_big{} } {
  }
}
```

By default, C++ lets us reason locally about code

The beauty of (free) functions

The beauty of (free) functions

- There once was a vogue of « everything should be in a class or in an object »
  - Think about classes that only exist to expose a main entrypoint to a program

The beauty of (free) functions

- Consider the following C# program
  - The situation in Java is essentially the same

```
using System;
class SuperMaths
{
   public static double Square(int n) =>
      Math.Pow(n,2);
}
class Program
{
   static void Main()
   {
      Console.Write(SuperMaths.Square(2));
   }
}
```

The beauty of (free) functions

- Consider the following C# prog
  - The situation in Java is essentiall

```
using System;
class SuperMaths
{
    public static double Squ      n) =>
        Math.Pow(n,2);
}
class Program
{
    static void Main()
    {
        Console.Write(SuperMaths.Square(2));
    }
}
```

The only reason why there are classes here is because… well, there's no good reason, to be honest (except maybe for grouping purposes), but they are required

The beauty of (free) functions

- At some point, `static` classes were introduced
  - These classes cannot be instantiated
  - They can only contain `static` members

The beauty of (free) functions

```
using System;
static class SuperMaths
{
   public static double Square(int n) =>
      Math.Pow(n,2);
}
class Program
{
   static void Main()
   {
      Console.Write(SuperMaths.Square(2));
   }
}
```

The beauty of (free) functions

- More recently, `using static` was introduced
  - This allows implicit usage of a `static` class' members

The beauty of (free) functions

```
using System;
using static SuperMaths;
static class SuperMaths
{
   public static double Square(int n) => Math.Pow(n,2);
}
class Program
{
   static void Main()
   {
      Console.Write(Square(2));
   }
}
```

The beauty of (free) functions

```
using static System.Console;
using static SuperMaths;
static class SuperMaths
{
   public static double Square(int n) => Math.Pow(n,2);
}
class Program
{
   static void Main()
   {
      Write(Square(2));
   }
}
```

The beauty of (free) functions

```
using static System.Console;
using static System.Math;
using static SuperMaths;
static class SuperMaths
{
   public static double Square(int n) => Pow(n,2);
}
class Program
{
   static void Main()
   {
      Write(Square(2));
   }
}
```

The beauty of (free) functions

```
using static System.Console;
using static System.Math;
using static SuperMaths;
static class SuperMaths
{
   public static double Square(int
}
class Program
{
   static void Main()
   {
      Write(Square(2));
   }
}
```

```
#include <cmath>
#include <iostream>
using namespace std;
auto square(int n) {
   return pow(n,2);
}
int main() {
   cout << square(2);
}
```

The beauty of (free) functions

```
using static System.Console;
using static System.Math;
using static SuperMaths;
static class SuperMaths
{
   public static double Square(int n) => Pow(n,2);
}
class Program
{
   static void Main()
   {
      Write(Square(2));
   }
}
```

Sometimes, a function's just the right thing to write

The beauty of const

The beauty of const

- I love `const`
- I love `const`-correctness
  - The capacity C++ provides to expose such an idea as « this member function shall leave 'this' object unchanged » is a brilliant idea

The beauty of const

- Not everyone agrees

The beauty of const

- Not everyone agrees

"The reason that `const` *works in C++ is because you can cast it away. If you couldn't cast it away, then your world would suck. If you declare a method that takes a* `const` *Bla, you could pass it a non-*`const` *Bla. But if it's the other way around you can't. If you declare a method that takes a non-*`const` *Bla, you can't pass it a* `const` *Bla. So now you're stuck. So you gradually need a* `const` *version of everything that isn't* `const`*, and you end up with a shadow world. In C++ you get away with it, because as with anything in C++ it is purely optional whether you want this check or not. You can just whack the* `const`*ness away if you don't like it."*

The beauty of const

- Not everyone agrees

can cast it away. If you couldn't cast it away, then your world would suck. If you declare a method that takes a `const` Bla, you could pass it a non-`const` Bla. But if it's the other way around you can't. ***If you declare a method that takes a non-`const` Bla, you can't pass it a `const` Bla. So now you're stuck. So you gradually need a `const` version of everything that isn't `const`, and you end up with a shadow world****. In C++ you get away with it, because as with anything in C++ it is purely optional whether you want this check or not. You can just whack*

The beauty of const

- Not everyone agrees

"The reason that `const` works in C++ is because you can cast it away. If you couldn't cast it away, then your world would suck. If you declare a method that takes a `const` Bla, you could pass it a non-`const` Bla. But if it's the other way around you can't. ***If you declare a method that takes a non-`const` Bla, you can't pass it a `const` Bla. So now you're stuck. So you gradually need a `const` version of everything that isn't `const`, and you end up with a shadow world****. In C++ you get away with it, because as with anything in C++ it is purely optional whether you want this check or not. You can just whack the `const`ness away if you don't like it."*

I'll let you spot the problem here…

The beauty of const

- Not everyone agrees

Anders Hejlsberg, in a 2004 interview https://www.artima.com/intv/choices.html

"The reason that `const` works in C++ is because you can cast it away. If you couldn't cast it away, then your world would suck. If you declare a method that takes a `const` Bla, you could pass it a non-`const` Bla. But if it's the other way around you can't. If you declare a method that takes a non-`const` Bla, you can't pass it a `const` Bla. So now you're stuck. So you gradually need a `const` version of everything that isn't `const`, and you end up with a shadow world. In C++ you get away with it, because as with anything in C++ it is purely optional whether you want this check or not. You can just whack the `const`ness away if you don't like it."

The beauty of const

- There is some truth to that, of course
  - C++ provides pragmatic mechanisms to circumvent `const` (as it offers mechanisms to circumvent many other things)
    - `mutable`
    - `const_cast`
    - These « locally opt-out » mechanisms have their place in the language
  - Still, `const` *works*

The beauty of const

- Many popular languages get by without `const`
  - Java and C# among them
  - Java has `final`, however
    - For references, it makes the reference immutable, not what it refers to
  - C# has `readonly`
    - For references, it makes the reference immutable, not what it refers to
    - C# has `const`, but only for things known at compile-time (so some `string`s and some `struct`s)

The beauty of const

```
using System;
public class Program
{
    class Integral
    {
        public int Value { get; set; }
        public Integral(int value)
        {
            Value = value;
        }
        public override string ToString() => $"{Value}";
    }
    // ...
```

The beauty of const

```
// ...
class Point
{
   private readonly Integral x;
   private readonly Integral y;
   public Integral X{ get => x; }
   public Integral Y{ get => y; }
   public Point(int x, int y)
   {
      this.x = new Integral(x);
      this.y = new Integral(y);
   }
   public override string ToString() => $"{X},{Y}";
}
// ...
```

The beauty of const

```
    // ...
    public static void Main()
    {
        var pt = new Point(2,3);
        Console.WriteLine(pt); // 2,3
        pt.X.Value++;
        Console.WriteLine(pt); // 3,3
    }
}
```

The beauty of const

```
    // ...
    public static void Mai
    {
        var pt = new Point(
        Console.WriteLine(pt                ,3
        pt.X.Value++;
        Console.WriteLine(pt); // 3,3
    }
}
```

Note that the real problem here is that *value semantics are lacking* in this program. In C++, we could get into the same kind of issues with `const` pointers

It's just more visible when pointers are opt-in, not the default

The beauty of const

```
#include <ostream>
class Integral {
   int value_;
public:
   Integral(int value) : value_{ value } {
   }
   int value() const {
      return value_;
   }
};
std::ostream&
   operator<<(std::ostream &os, const Integral &i) {
      return os << i.value();
   }
// ...
```

The beauty of const

Idiomatic (value-based) design

```
#include <ostream>
class Integral { /* ... */ };
class Point {
   Integral x, y;
public:
   Point(int x, int y) : x{ x }, y{ y } {
   }
   auto X() const { return x; }
   auto Y() const { return y; }
};
std::ostream&
   operator<<(std::ostream &os, const Point &pt) {
      return os << pt.X() << ',' << pt.Y();
   }
// ...
```

The beauty of const

```
#include <ostream>
class Integral { /* ... */ };
class Point { /* ... */ };
#include <iostream>
int main() {
   Point pt{ 2,3 };
   // cannot modify pt.x here
   std::cout << pt << '\n';
}
```

The beauty of const

Unidiomatic (indirection-based) design. Requires effort… and makes you think « why am I doing this? »

```
#include <ostream>
class Integral { /* ... */ };
class Point {
   Integral *x, *y;
public:
   Point(int x, int y)
      : x{ new Integral{ x } }, y{ new Integral{ y } } { // might leak!
   }
   Point(const Point&) = delete;
   Point& operator=(const Point&) = delete;
   auto& X() const { return x; }
   auto& Y() const { return y; }
   ~Point() { delete y; delete x; }
};
std::ostream& operator<<(std::ostream &os, const Point &pt) {
   return os << *pt.X() << ',' << *pt.Y();
}
// ...
```

The beauty of const

```
#include <ostream>
class Integral { /* ... */ };
class Point { /* ... */ };
#include <iostream>
int main() {
   Point pt{ 2,3 };
   std::cout << pt << '\n'; // 2,3
   // works because Point::X() returns
   // a reference (crafted to make a
   // point; not nice here)
   *pt.X() = Integral{ 3 };
   std::cout << pt << '\n'; // 3,3
}
```

The beauty of const

Note that `Point::X()` is `const`, but `const` applies to `this`, thus to `pt.x` which is a pointer. That pointer's `const`ness is respected

```
#include <ostream>
class Integral { /* ... */ };
class Point { /* ... */ };
#include <iostream>
int main() {
   Point pt{ 2,3 };
   std::cout << pt << '\n'; // 2,3
   // works because Point::X() returns
   // a reference (crafted to make a
   // point; not nice here)
   *pt.X() = Integral{ 3 };
   std::cout << pt << '\n'; // 3,3
}
```

The beauty of const

```
#include <ostream>
class Integral { /* ... */ };
#include <memory>
class Point {
   std::unique_ptr<Integral> x, y;
public:
   Point(int x, int y)
      : x{ std::make_unique<Integral>(x) }, y{ std::make_unique<Integral>(y) } {
   }
   auto& X() const { return x; }
   auto& Y() const { return y; }
};
std::ostream& operator<<(std::ostream &os, const Point &pt) {
   return os << *pt.X() << ',' << *pt.Y();
}
// ...
```

Unidiomatic (indirection-based) design. Requires effort… and (again) makes you think « why am I doing this? »

The beauty of const

```
#include <ostream>
class Integral { /* ... */ };
class Point { /* ... */ };
#include <iostream>
int main() {
   Point pt{ 2,3 };
   std::cout << pt << '\n'; // 2,3
   // works because Point::X() returns
   // a reference (crafted to make a
   // point; not nice here)
   *pt.X() = Integral{ 3 };
   std::cout << pt << '\n'; // 3,3
}
```

The beauty of c

Note that `Point::X()` is `const`, but `const` applies to `this`, thus to `pt.x` which is a `unique_ptr<Integral>`. That object's `const`ness is respected (we could not call non-`const` member function `reset()` on it, for example)

```
#include <ostream>
class Integral { /* ... *
class Point { /* ... */ },
#include <iostream>
int main() {
   Point pt{ 2,3 };
   std::cout << pt << '\n'; // 2,3
   // works because Point::X() returns
   // a reference (crafted to make a
   // point; not nice here)
   *pt.X() = Integral{ 3 };
   std::cout << pt << '\n'; // 3,3
}
```

The beauty of const

- I love `const`
  - It « plays well » with value semantics
  - C++ is a value-based language

The beauty of const

- I love const
  - It « plays well » with value semantics
  - C++ is a valu

Th

CppCon 2019: Tony Van Eerd Objects vs Values: Value Oriented Programming in an Object Oriented World

https://www.youtube.com/watch?v=2JGH_SWURrI&feature=emb_rel_pause

- I love const
  - It « plays well » with valu tics
  - C++ is a valu

The beauty of const

- I love const
  - It « plays well » with value semantics
  - C++ is a val

Tony presents objects as reference types, distinct from values, but I disagree. By default, C++ objects *are* values (and we can opt out)

The beauty of const

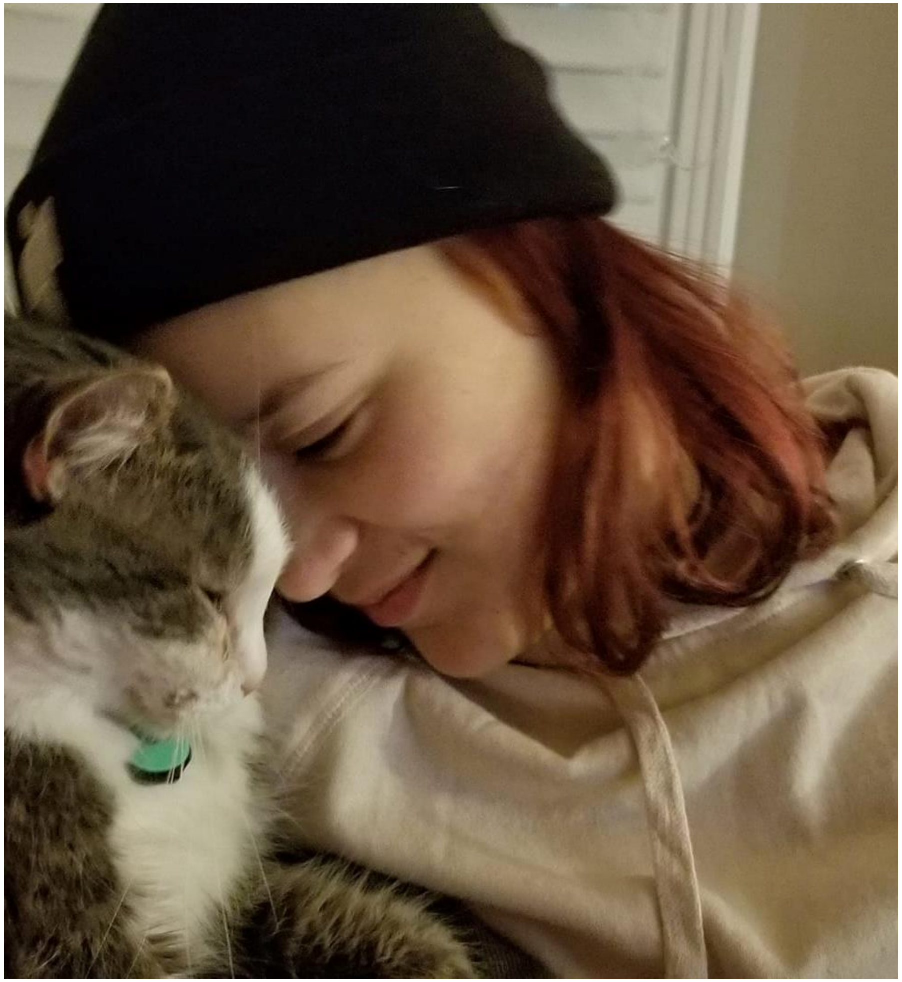
- I love const
  - It « plays well » with value semantics
  - C++ is a valu

Still, Tony gives great talks!

The beauty of const

- Does C++ have the wrong default? Should `const` be opt-out instead of opt-in?
  - From a 2020 perspective: maybe
  - From a 1979 perspective: unclear
    - The power of hindsight…
  - This does not prevent us from using `const` wisely

The beauty of friend



The beauty of friend

- In some circles (and in some books!), the idea of `friend` has been wildly misrepresented over the years as something dangerous or to be avoided
  - If used well, it does not damage encapsulation: it enforces it

The beauty of friend

- **Situation**: you have a type that has to be initialized in two steps before being ready to be used

The beauty of friend

```
class TwoStepInitThing {
public:
   TwoStepInitThing();
   void Init();
   void Use(); // hmm...
   // ...
};
int main() {
   // correct usage
   TwoStepInitThing tsit;
   tsit.Init();
   // Ok, now tsit is ready to be used
   tsit.Use();
}
```

The beauty of friend

- Obviously, if you leave it to programmers' discipline, *you'll get what you deserve*

The beauty of friend

```
class TwoStepInitThing {
public:
    TwoStepInitThing();
    void Init();
    void Use(); // hmm...
    // ...
};
int main() {
    TwoStepInitThing tsit;
    tsit.Use(); // oops! Not ready yet!
}
```

The beauty of friend

- Following the guidance of « literature from the ancients », you want to provide a factory for that type
  - The goal is to ensure that a call to `Init` always follows construction, and the object is only made available once both steps have been performed
  - This is not difficult do to

The beauty of friend

```
class TwoStepInitThing {
public:
    TwoStepInitThing();
    void Init();
    void Use(); // hmm...
    // ...
};
auto TwoStepInitFactory() {
  TwoStepInitThing tsit;
  tsit.Init();
  return tsit;
}
```

The beauty of friend

```
class TwoStepInitThi
public:
    TwoStepInitThing(
    void Init();
    void Use(); // hmm...
    // ...
};
auto TwoStepInitFactory() {
  TwoStepInitThing tsit;
  tsit.Init();
  return tsit;
}
```

Problem solved!

```
int main() {
   auto tsit = TwoStepInitFactory();
   tsit.Use(); // cool!
}
```

The beauty of friend

```
class TwoStepInitThi
public:
    TwoStepInitThing(
    void Init();
    void Use(); // hmm...
    // ...
};
auto TwoStepInitFactory() {
  TwoStepInitThing tsit;
  tsit.Init();
  return tsit;
}
```

Problem solved…?

```
int main() {
    // users can still do this...
    auto tsit = TwoStepInit{};
    tsit.Use(); // oops!
}
```

The beauty of friend

- If we don’t want to use `friend`, we still have options

The beauty of friend

- If we don’t want to use `friend`, we still have options
  - We can use an interface and make `TwoStepInitThing` only visible to `TwoStepInitFactory`

The beauty of friend

```
// exposed in some header file
struct Usable {
   virtual void Use() = 0;
   virtual ~Usable() = default;
};
unique_ptr<Usable> TwoStepInitFactory();
// hidden in a source file
class TwoStepInitThing : public Usable {
   // ...
};
unique_ptr<Usable> TwoStepInitFactory() {
   auto p = make_unique<TwoStepInitThing>();
   p->Init();
   return p;
}
```

The beauty of friend

- If we don't want to use `friend`, we still have options
  - We can use an interface and make `TwoStepInitThing` only visible to `TwoStepInitFactory`
  - We can use an interface, make `TwoStepInitFactory` a class and make `TwoStepInitThing` an inner class of `TwoStepInitFactory`

The beauty of friend

```
struct Usable {
   virtual void Use() = 0;
   virtual ~Usable() = default;
};
class TwoStepInitFactory {
   class TwoStepInitThing : public Usable {
      // ...
   };
public:
   unique_ptr<Usable> TwoStepInitFactory() const {
      auto p = make_unique<TwoStepInitThing>();
      p->Init();
      return p;
   }
};
```

The beauty of friend

- If we don't want to use `friend`, we still have options
  - We can use an interface and make `TwoStepInitThing` only visible to `TwoStepInitFactory`
  - We can use an interface, make `TwoStepInitFactory` a class and make `TwoStepInitThing` an inner class of `TwoStepInitFactory`
  - We can add a static member factory function to `TwoStepInitThing` and couple the class with its factory mechanism

The beauty of friend

```
class TwoStepInitThing {
   TwoStepInitThing();
   Init(); // private
public:
   static auto TwoStepInitFactory() {
      TwoStepInitThing tsit;
      tsit.Init();
      return tsit;
   }
};
```

The beauty of friend

- These solutions carry a cost
  - For the first two, we have to allocate
  - For the first two, calls become polymorphic and indirect
  - For the third one, we introduce a mechanism in our class that was not part of the original design
    - It's a minor issue, however
  - There are other ways still, e.g.: wrapping `TwoStepInitThing` in another class, doing the two-step init in the wrapper's constructor and delegating the calls afterward
- Using `friend` describes the localized privilege relationship we want to express

The beauty of friend

```
class TwoStepInitThing {
   TwoStepInitThing(); // private
   void Init();
public:
   void Use(); // hmm...
   // ...
   friend auto TwoStepInitFactory() {
      TwoStepInitThing tsit;
      tsit.Init();
      return tsit;
   }
};
```

The beauty of friend

```
class TwoStepInitThi
    TwoStepInitThing(
    void Init();
public:
    void Use(); // hmm...
    // ...
    friend auto TwoStepInitFactory() {
        TwoStepInitThing tsit;
        tsit.Init();
        return tsit;
    }
};
```

Problem solved

```
int main() {
    // Ok
    auto tsit = TwoStepInitFactory();
    tsit.Use();
    // auto nope = TwoStepInitThing{};
}
```

The beauty of friend

- Java has an access qualification that C++ does not support with *package private*
  - Only expressible implicitly
  - It's what you get if you don't qualify something as `private`, `protected` or `public`

The beauty of friend

- Java has an access qualification that C++ does not support with *package private*
  - Only expressible implicitly
  - It's what you get if you don't qualify something as `private` `protected` or `public`

Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

The beauty of friend

- C# has access qualifiers that C++ does not support with `internal`, `protected internal` and `private protected`

The beauty of friend

- C# has access qualifiers that C++ does not support with `internal`, `protected internal` and `private protected`
  - `internal` means accessible only to the class itself or to others in the same assembly (executable, .dll)

The beauty of friend

- C# has access qualifiers that C++ does not support with `internal`, `protected internal` and `private protected`
  - `internal` means accessible only to the class itself or to others in the same assembly (executable, .dll)
  - `protected internal` means accessible only to the class itself, to others in the same assembly or to derived classes

The beauty of friend

- C# has access qualifiers that C++ does not support with `internal`, `protected internal` and `private protected`
  - `internal` means accessible only to the class itself or to others in the same assembly (executable, .dll)
  - `protected internal` means accessible only to the class itself, to others in the same assembly or to derived classes
  - `private protected` means accessible to the class itself or to derived classes in the same assembly

The beauty of friend

- The `friend` qualification in C++ is much more precise, applying to a single class, a single function

The beauty of friend

- The `friend` qualification in C++ is much more precise, applying to a single class, a single function
  - …or, if that's what is desired, to a family thereof
  - For example:

```
class Popular {
   template <class T>
      friend struct Aficionado;
   int n = 0; // private
};
```

The beauty of friend

It's better than qualifying `Popular::n` as `public`, but be careful due to template specializations!

```
class Popular {
    template <class T>
        friend struct Aficionado;
    int n = 0;
};
template <class T> struct Aficionado {
    int fetch(const Popular &p) const {
        return p.n;
    }
};
int main() {
    Popular pop;
    return Aficionado<void>{}.fetch(pop);
}
```

Sometimes unnoticed upsides of iterators

Sometimes unnoticed upsides of iterators

- **Situation**: we want to write our own « reverse » in order to reverse the order of the elements in a range
  - C++17 style

Sometimes unnoticed upsides of iterators

```
template <class It>
   void rev(It b, It e) {
      using std::swap;
      while(b != e) {
         --e;
         if(b == e) return;
         swap(*b, *e);
         ++b;
      }
   }
```

Sometimes unnoticed … iterators

```
// CTAD!
vector v{ 2,3,5,7,11 };
rev(begin(v), end(v));
```

```
template <class It>
   void rev(It b, It e) {
      using std::swap;
      while(b != e) {
         --e;
         if(b == e) return;
         swap(*b, *e);
         ++b;
      }
   }
```

Sometimes unnoticed iterators

```
// CTAD!
list v{ 2,3,5,7,11 };
rev(begin(v), end(v));
```

```
template <class It>
   void rev(It b, It e) {
      using std::swap;
      while(b != e) {
         --e;
         if(b == e) return;
         swap(*b, *e);
         ++b;
      }
   }
```

Sometimes unnoticed … iterators

```
// CTAD!
deque v{ 2,3,5,7,11 };
rev(begin(v), end(v));
```

```
template <class It>
   void rev(It b, It e) {
      using std::swap;
      while(b != e) {
         --e;
         if(b == e) return;
         swap(*b, *e);
         ++b;
      }
   }
```

Sometimes unnoticed upsides of iterators

```
template <class It>
    void rev(It b, It e) {
        using std::swap;
        while(b != e) {
            --e;
            if(b == e) return;
            swap(*b, *e);
            ++b;
        }
    }
```

Our algorithm is in large part independent of the underlying container; it expects bidirectional iterators and that's about it

Sometimes unnoticed upsides of iterators

- **Situation**: we want to write our own « reverse » in order to reverse the order of the elements in a range
  - C# style

Sometimes unnoticed upsides of iterators

```
static void Rev<T>(IEnumerable<T> rng)
{
   // ... hmm ...
}
```

Sometimes unnoticed upsides of iterators

```
static void Rev<T>(IEnum
{
    // ... hmm ...
}
```

From an `IEnumerable<T>` we can get an `IEnumerator<T>`...

Sometimes unnoticed upsides of iterators

```
static void Rev<T>(IEnum
{
   // ... hmm ...
}
```

From an `IEnumerable<T>` we can get an `IEnumerator<T>`...

`IEnumerator<T>` lets one move to the next element (`MoveNext`), obtain the current element (`Current`) and restart from the beginning of the `IEnumerable<T>` (`Reset`)

Sometimes unnoticed upsides of iterators

```
static void Rev<T>(IEnum
{
    // ... hmm ...
}
```

From an IEnumerable<T> we can get an IEnumerator<T>...

IEnumerator<T> lets one move to the next element (MoveNext), obtain the current element (Current) and restart from the beginning of the IEnumerable<T> (Reset)

The upside is that it can model an infinite sequence. The downside is... how can we write our algorithm?

Sometimes unnoticed upsides of iterators

- **Situation**: we want to write our own « reverse » in order to reverse the order of the elements in a range
  - C# style
  - The quick answer is… you cannot

Sometimes unnoticed upsides of iterators

- **Situation**: we want to write our own « reverse » in order to reverse the order of the elements in a range
  - C# style
  - The quick answer is… you cannot
  - The C# solution is to provide, through extension methods in the `System.Linq` namespace, a set of case-by-case implementations of various services such as this one
    - It works, in the end

Sometimes unnoticed upsides of iterators

- C++ iterators have elegance
  - They allow the expression of efficient algorithms in a general form, rather than the case-by-case solution other languages often provide
  - … and they do so without imposing the cost of virtual function calls or the intrusiveness of an interface

Sometimes unnoticed upsides of iterators

- **Situation**: you want to make a `List<int>` from an `int[]` in C#

Sometimes unnoticed upsides of iterators

- **Situation**: you want to make a `List<int>` from an `int[]` in C#
  - Of course, you will write

    ```
    int [] arr = new int[]{ 2,3,5,7,11 };
    List<int> lst = arr.ToList();
    ```

  - … and if you want to go in the other direction, you will of course write

    ```
    arr = lst.ToArray();
    ```

Sometimes unnoticed upsides of iterators

- **Situation**: you want to make a `List<int>` from an `int[]` in C#
  - Of course, you will write

    ```
    int [] arr = new int[]{ 2,3,5,7,11 };
    List<int> lst = arr.ToList();
    ```

  - … and if you want to go in the other direction, you will of course write

    ```
    arr = lst.ToArray();
    ```

If you use Linq, you will also have `ToDictionary` and `ToHashSet`.

It's on a case-by-case basis…

Sometimes unnoticed upsides of iterators

- **Situation**: you want to make a `vector<int>` from an `int[]` in C++

Sometimes unnoticed upsides of iterators

- **Situation**: you want to make a `vector<int>` from an `int[]` in C++
  - You will write

```
int arr []{ 2,3,5,7,11 };
vector<int> v{ begin(arr), end(arr) };
```

  - Now, if you want to make a `list<double>` from this vector, you will write

```
list<double> lst{ begin(v), end(v) };
```

Sometimes unnoticed upsides of iterators

- **Situation**: you want to make a `vector<int>` from an `int[]` in C++
  - You will write

```
int arr []{ 2,3,5,7,11 };
vector<int> v{ begin(arr), end(arr) };
```

  - Now, if you want to make a `list<double>` from this vector, you will write

```
list<double> lst{ begin(v), end(v) };
```

Standard containers have sequence constructors. It’s elegant, homogeneous… and efficient. Oh, and you have to write only one per container

The beauty of variadics

The beauty of variadics

- **Situation**: we want to serialize a series of objects to standard output

The beauty of variadics

```
// Java
public static void print(Object ... args) {
    // works by calling toString on
    // each object obj, because Object
    // exposes virtual method toString
    for(Object obj : args) {
        System.out.print(obj + " ");
    }
}
static class X{}
public static void example() {
    print(3, 3.14159, "Hi", new X());
}
```

The beauty of variadics

```
// Java
public static void print(Object
    // works by calling toString
    // each object obj, because
    // exposes virtual method to
    for(Object obj : args) {
        System.out.print(obj + "
    }
}
static class X{}
public static void example() {
    print(3, 3.14159, "Hi", new X());
}
```

Synthesizes an array from the arguments, that must all be of the same type. Here, they're all ultimately of type `Object` once boxing has occurred for the `int` and the `double`

The beauty of variadics

```
// C#
public static void Print(params object [] args)
{
   // works by calling ToString on
   // each object obj, because object
   // exposes virtual method ToString
   foreach(var obj in args)
   {
      Console.Write($"{obj} ");
   }
}
class X{}
public static void Example()
{
   Print(3, 3.14159, "Hi", new X());
}
```

The beauty of variadics

```
// C#
public static void Print(params o
{
   // works by calling ToString o
   // each object obj, because ob
   // exposes virtual method ToSt
   foreach(var obj in args)
   {
      Console.Write($"{obj} ");
   }
}
class X{}
public static void Example()
{
   Print(3, 3.14159, "Hi", new X());
}
```

Synthesizes an array from the arguments, that must all be of the same type. Here, they're all ultimately of type `object` once boxing has occurred for the `int` and the `double`

The beauty of variadics

```
// C++ (naïve)
template <class T>
   void print(const T &arg) {
      cout << arg << ' ';
   }
template <class T, class ... Ts>
   void print(const T &arg, Ts ... args) {
      print(arg);
      print(args...);
   }
void example() {
   print(3, 3.14159, "Hi"s);
}
```

The beauty of variadics

```
// C++ (naïve)
template <class T>
   void print(const T &arg) {
      cout << arg << ' ';
   }
template <class T, class ... Ts>
   void print(const T &arg, Ts ... args) {
      print(arg);
      print(args...);
   }
void example() {
   print(3, 3.14159, "Hi"s);
}
```

Arguments can be of any serializable type. No conversion to `string`. No array is being synthesized.

The beauty of variadics

```
// C++ (less naïve)
template <class T>
   void print(T &&arg) {
      cout << arg << ' ';
   }
template <class T, class ... Ts>
   void print(T &&arg, Ts &&... args) {
      print(std::forward<T>(arg));
      print(std::forward<Ts>(args)...);
   }
void example() {
   print(3, 3.14159, "Hi"s);
}
```

The beauty of variadics

```
// C++ (less naïve)
template <class T>
   void print(T &&arg) {
      cout << arg << ' ';
   }
template <class T, class ... Ts>
   void print(T &&arg, Ts &&... args) {
      print(std::forward<T>(arg));
      print(std::forward<Ts>(args)...);
   }
void example() {
   print(3, 3.14159, "Hi"s);
}
```

Difficult to write due to syntax (uses perfect forwarding, forwarding references), sadly…

The beauty of variadics

```
// C++ (less naïve)
template <class T>
   void print(T &&arg) {
      cout << arg << ' ';
   }
template <class T, class ... Ts>
   void print(T &&arg, Ts &&... args) {
      print(std::forward<T>(arg));
      print(std::forward<Ts>(args)...);
   }
void example() {
   print(3, 3.14159, "Hi"s);
}
```

... but remains very easy to use

The beauty of variadics

```
// C++ (C++17 fold expressions)
template <class T>
   void print_one(T &&arg) {
      cout << arg << ' ';
   }
template <class ... Ts>
   void print(Ts &&... args) {
      (print_one(std::forward<Ts>(args)), ...);
   }
void example() {
   print(3, 3.14159, "Hi"s);
}
```

The beauty of variadics

```
// C++ (C++17 fold expressions)
template <class T>
   void print_one(T &&arg) {
      cout << arg << ' ';
   }
template <class ... Ts>
   void print(Ts &&... args) {
      (print_one(std::forward<Ts>(args)), ...);
   }
void example() {
   print(3, 3.14159, "Hi"s);
}
```

Admittedly, requires some « getting used to »

The beauty of variadics

- Variadics are part of what makes the language beautiful
  - Yes, the syntax is not immediately obvious to everyone
    - Depends on the background of each individual
  - … but it makes the language so much more expressive

The beauty of variadics

```
template <int ... Ns>
   int useless_but_fun() {






   }
```

Let's write a function template on a variadic number of `ints`

The beauty of variadics

```
template <int ... Ns>
   int useless_but_fun() {
      int arr[sizeof...(Ns)];



   }
```

Let's make an array big enough to hold them all

The beauty of variadics

```
template <int ... Ns>
    int useless_but_fun() {
        int arr[sizeof...(Ns)]{ Ns... };



    }
```

Let’s initialize the array with these `int`s (note that we don’t need to explicitly write the size anymore)

The beauty of variadics

```
template <int ... Ns>
   int useless_but_fun() {
      int arr[sizeof...(Ns)]{ Ns... };
      for(int n : arr)
         cout << n << ' ';



   }
```

We can of course perform operations on the elements of our array

The beauty of variadics

```
template <int ... Ns>
    int useless_bu
        int arr[size
        for(int n :
            cout << n
        for(int n : { Ns... })
            cout << n << ' ';

    }
```

Or just operate on an `initializer_list` made from those values (yes, even `initializer_list` are cool sometimes!)

The beauty of variadics

```
template <int ... Ns>
   int useless_but_fun() {
      int arr[sizeof...(Ns)]{ Ns... };
      for(int n : arr)
         cout << n << ' ';
      for(int n : { Ns... })
         cout << n << ' ';
      return (Ns + ...);
   }
```

Want the sum of those values? No need for a loop when you can express that sum directly

A word about the fun of programming

A word about the fun of programming

- I like programming
  - The act of programming
  - The *Praxis*
  - The problem solving

A word about the fun of programming

- I like programming
  - The act of programming
  - The *Praxis*
  - The problem solving
- Building solutions from sound principles is intellectually stimulating to me
  - Of course, that does not mean I’m against libraries and frameworks

A word about the fun of programming

- Compare…

A word about the fun of programming

- Compare…

```
static string ReadFile(string path) =>
    File.ReadAllText(path);
```

- … with …

```
string read_file(const string &path) {
    ifstream in{ path };
    return {
        istreambuf_iterator<char>{ in },
        istreambuf_iterator<char>{}
    };
}
```

A word about the fun of programming

- Compare…

```
static string[]
    ReadLines(string path) =>
        File.ReadAllLines(path);
```

- … with …

```
vector<string>
    read_lines(const string &path) {
        ifstream in{ path };
        return {
            istream_iterator<string>{ in },
            istream_iterator<string>{}
        };
    }
```

A word about the fun of programming

- Compare…

```
static List<T>
  MakeList<T>(T [] arr) =>
    arr.ToList();
```

- … with …

```
template <class T, int N>
  vector<T>
    make_vec(const array<T,N> &src) {
      return { begin(src), end(src) };
    }
```

A word about the fun of programming

- Compare…

```
// no real solution in C#, except maybe
// through some Linq interface
```

- … with …

```
template <class D, class S>
  D convert_container(const S &src) {
    return { begin(src), end(src) };
  }
```

A word about the fun of programming

- Generality sometimes implies there’s a price to pay up front…
  - …and there’s a reward that comes along with it

Questions?