Circle evolves C++

Make the language safer and more productive

Sean Baxter
circle-lang.org
 @seanbax
circle-lang.org/bloomberg-2023.pdf

How to turn C++ into a language research platform

Sean Baxter
circle-lang.org
@seanbax
circle-lang.org/bloomberg-2023.pdf

Or, how to incentivize investment in language technology

Sean Baxter
circle-lang.org
@seanbax
circle-lang.org/bloomberg-2023.pdf

Or, a no-consensus model for C++ governance

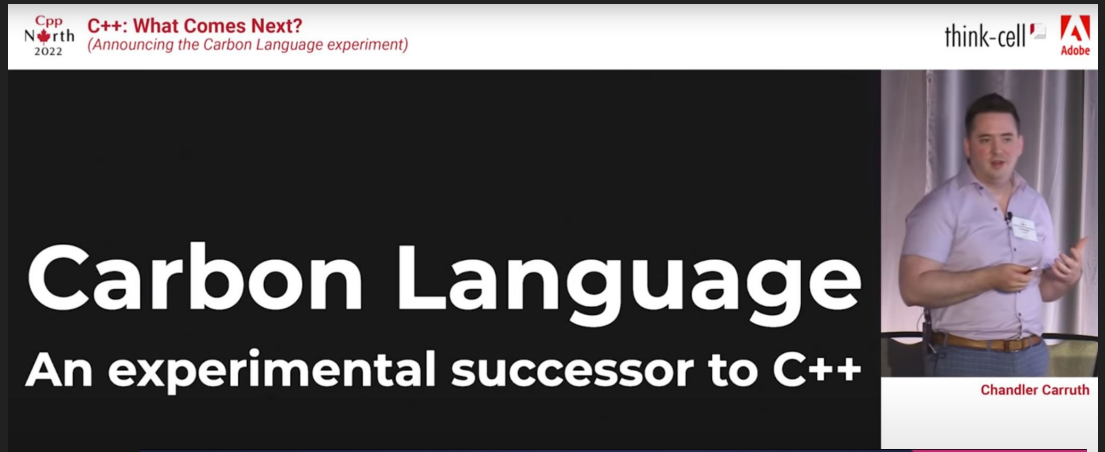
Sean Baxter
circle-lang.org
 @seanbax
circle-lang.org/bloomberg-2023.pdf

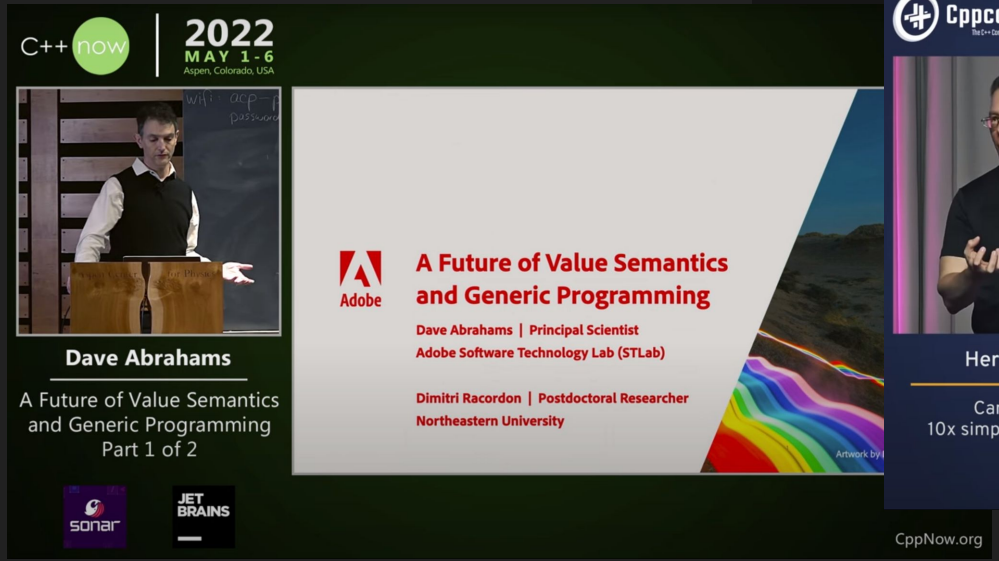
My favorite: one simple trick to unshackle creativity in C++ language evolution

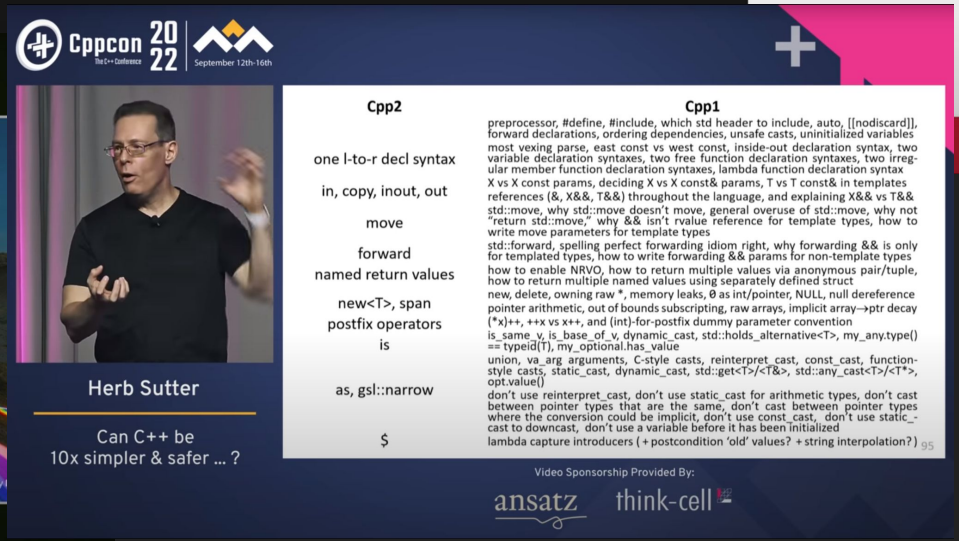
Sean Baxter
circle-lang.org
 @seanbax
circle-lang.org/bloomberg-2023.pdf

Successor languages

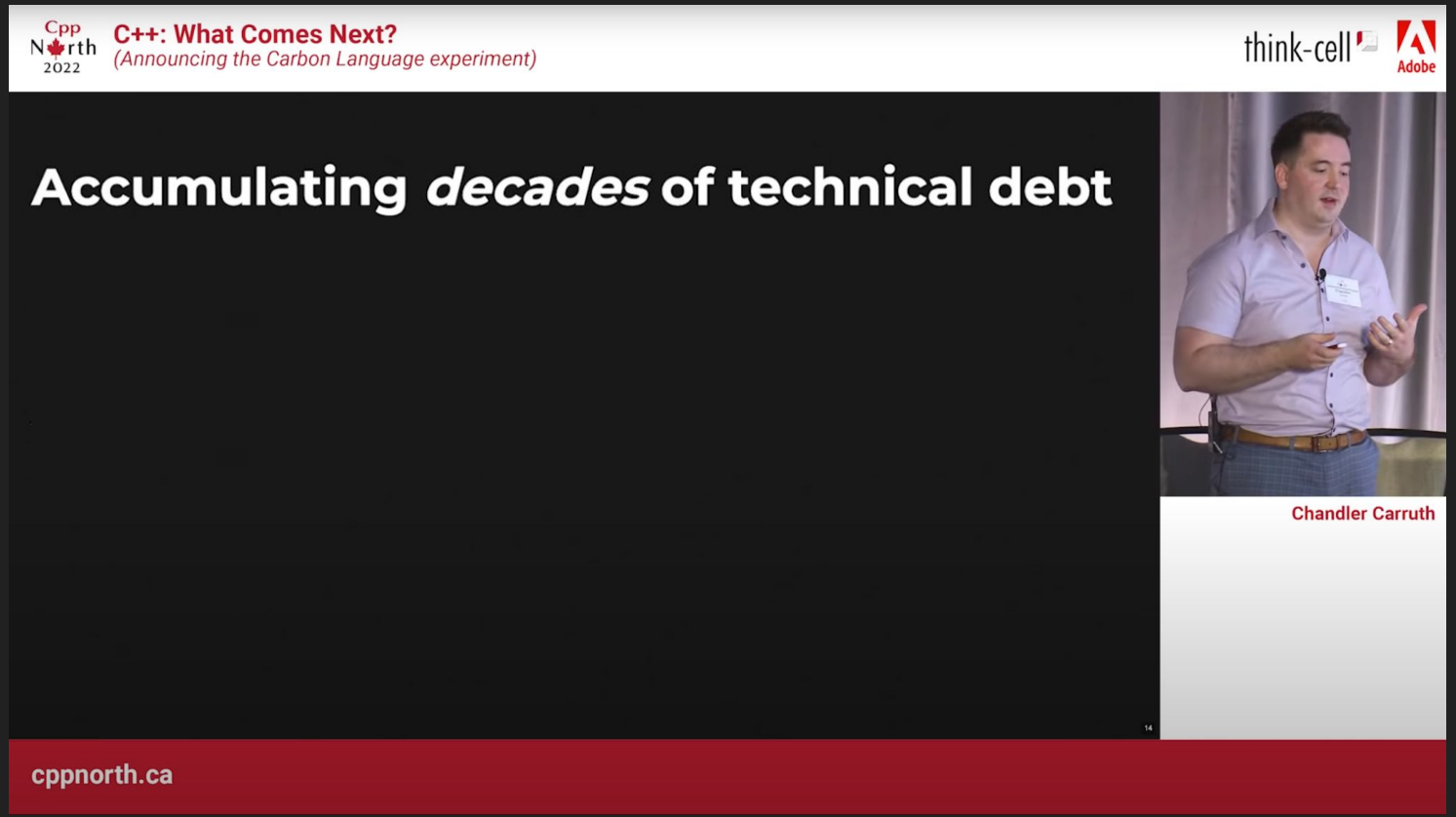
- Carbon
- Cpp2
- Val



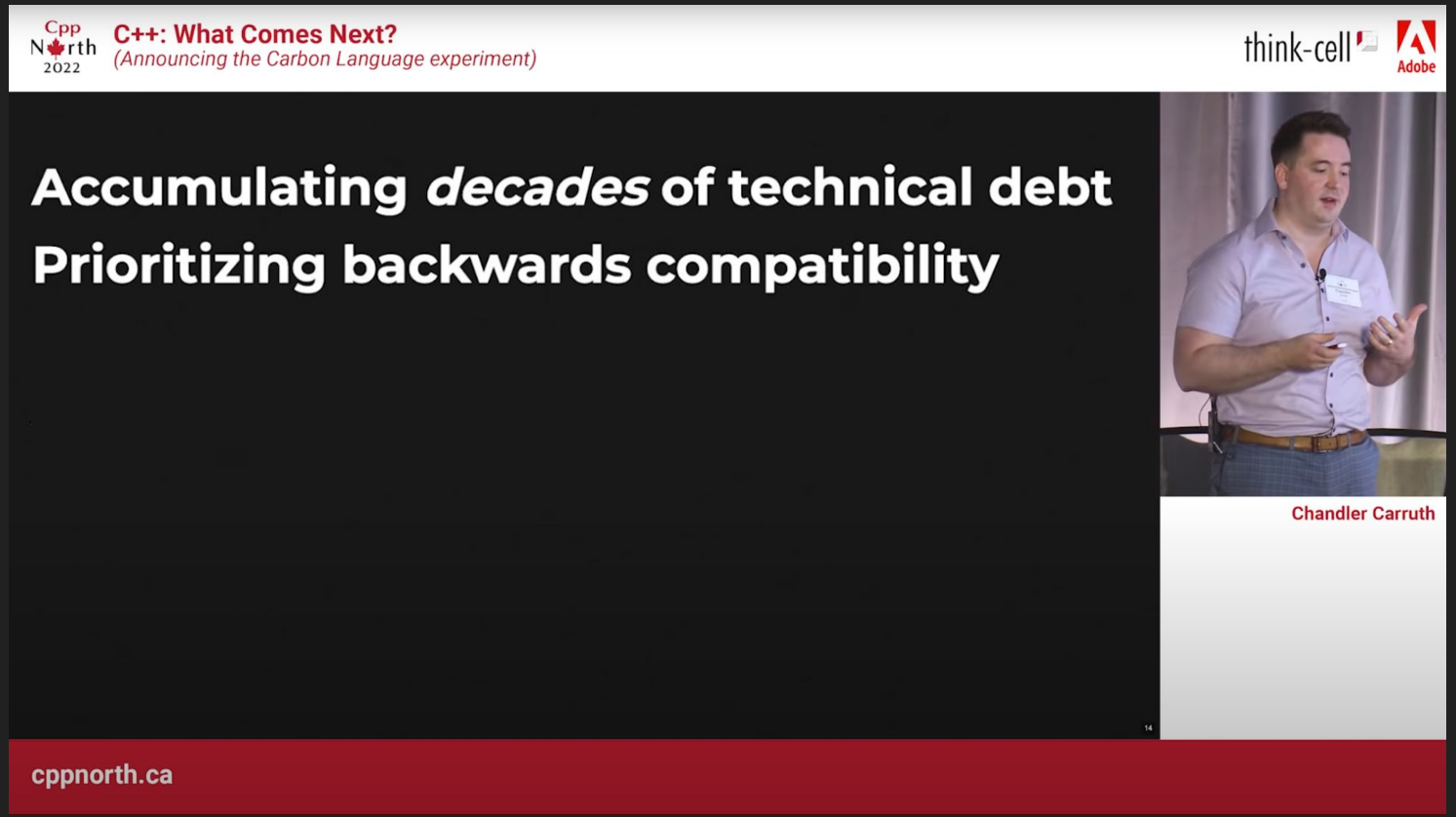




To err is human, to fix divine

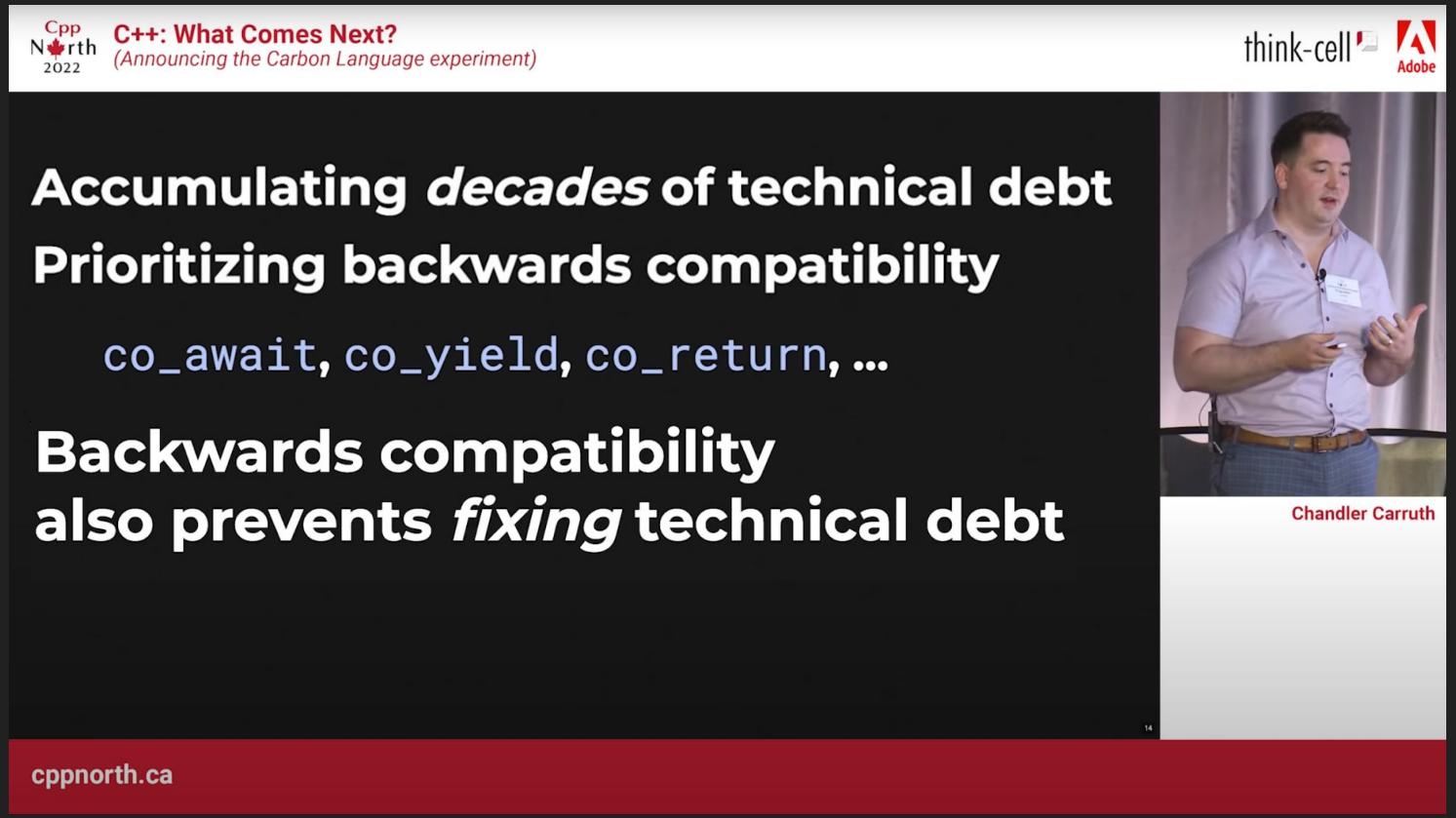


To err is human, to fix divine



To err is human, to fix divine

To err is human, to fix divine



Can't fix technical debt because we can't

"break or change the meaning of existing code"

Why build Carbon?

C++ remains the dominant programming language for performance-critical software, with massive and growing codebases and investments. However, it is struggling to improve and meet developers' needs, as outlined above, in no small part due to accumulating decades of technical debt. Incrementally improving C++ is extremely difficult, both due to the technical debt itself and challenges with its evolution process. The best way to address these problems is to avoid inheriting the legacy of C or C++ directly, and instead start with solid language foundations like modern generics system, modular code organization, and consistent, simple syntax.

Carbon project
https://github.com/carbon-language/carbon-lang/#why-build-carbon

Difficulties improving C++

C++ is the dominant programming language for the performance critical software our goals prioritize. The most direct way to deliver a modern and excellent developer experience for those use cases and developers would be to improve C++.

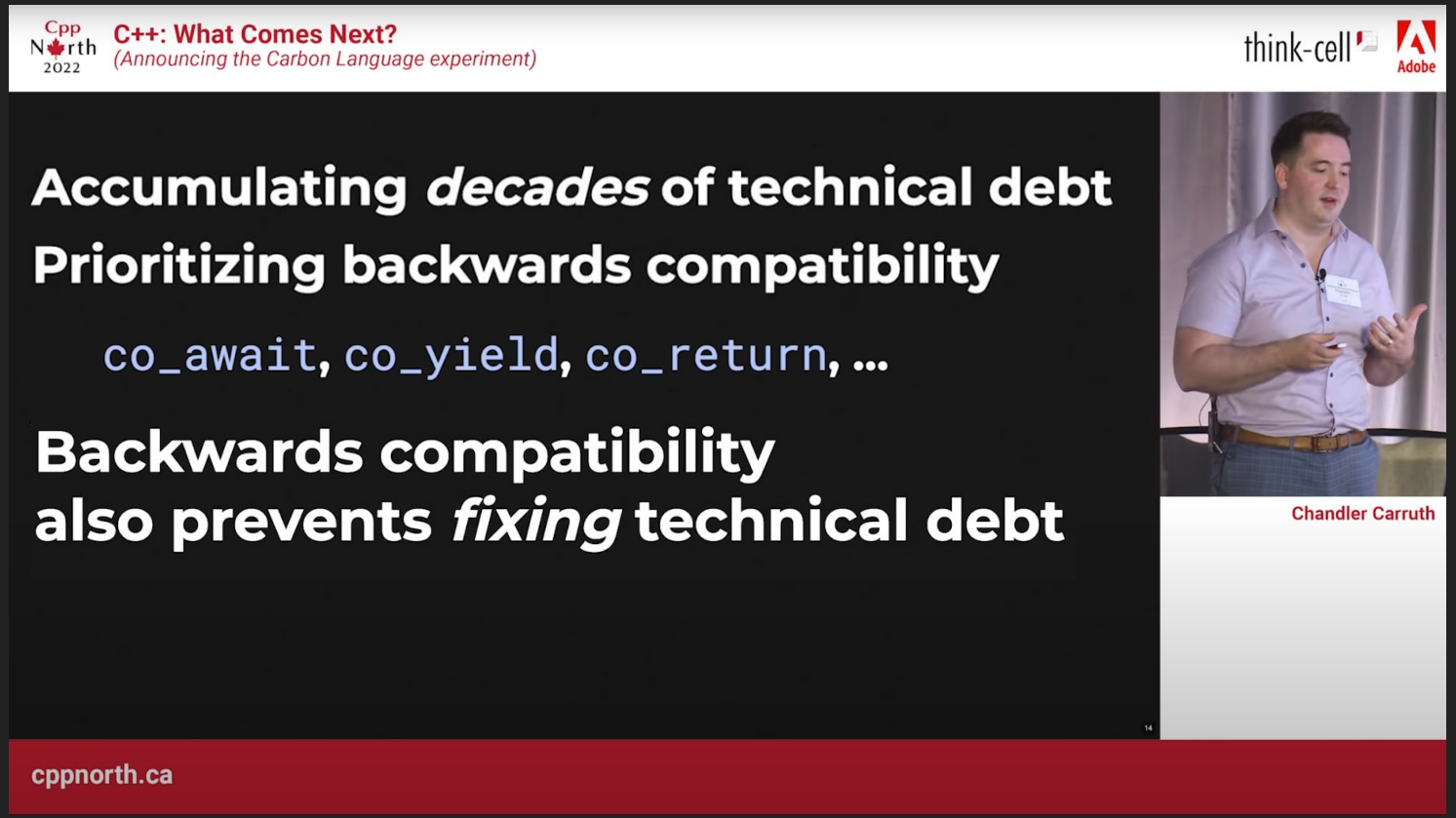
Improving C++ to deliver the kind of experience developers expect from a programming language today is difficult in part because **C++ has decades of technical debt** accumulated in the design of the language. It inherited the legacy of C, including textual preprocessing and inclusion. At the time, this was essential to C++'s success by giving it instant and high quality access to a large C ecosystem. However, over time this has resulted in significant technical debt ranging from integer promotion rules to complex syntax with "the most vexing parse".

Carbon project
https://github.com/carbon-language/carbon-lang/blob/trunk/docs/project/difficulties_improving_cpp.md

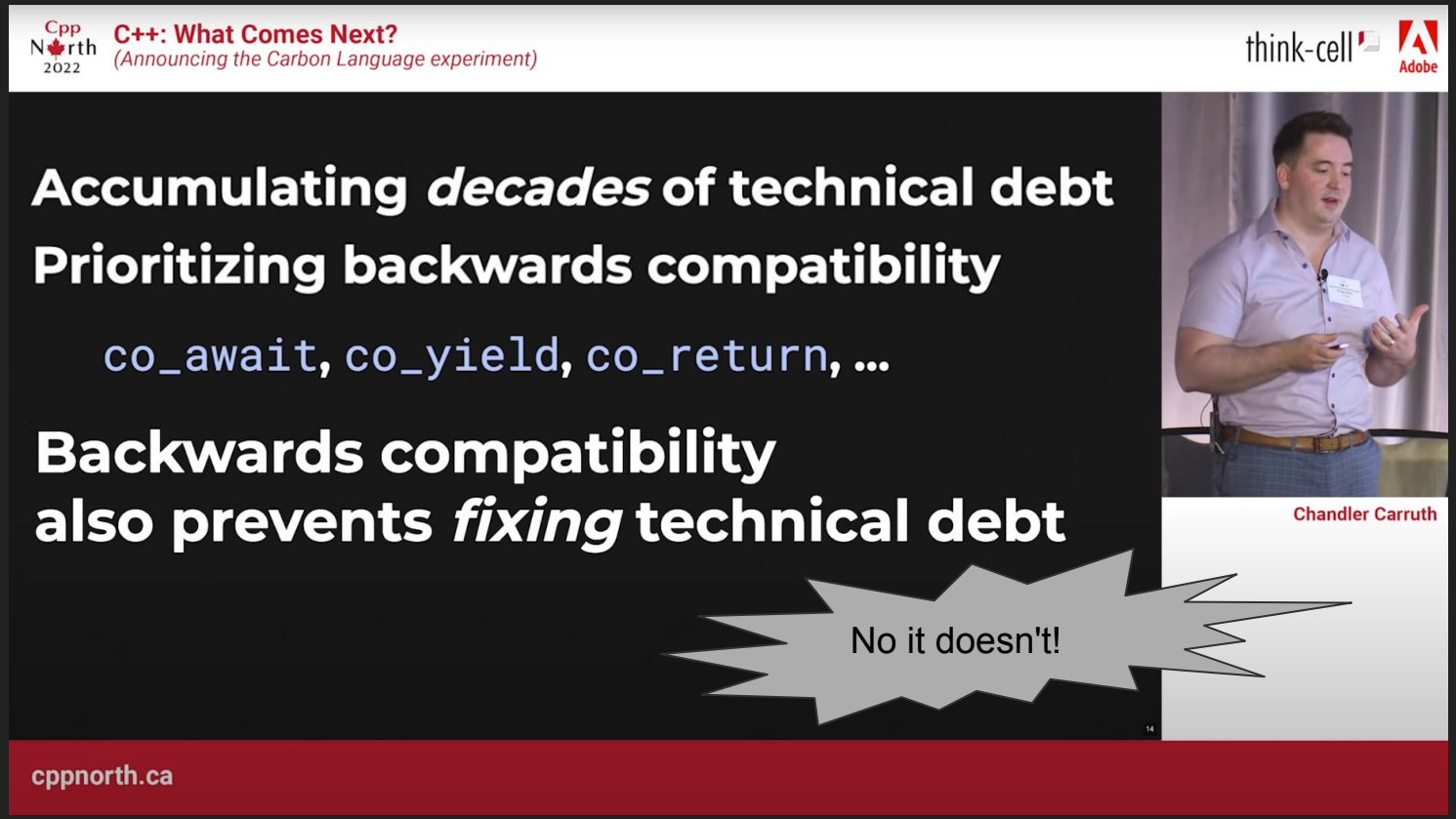
C++'s technical debt

- C++ has some "wrong defaults."
  - Uninitialized automatic variables.
  - Integral promotions.
  - Implicit narrowing conversions.
  - Switches should break rather than fallthrough.
  - Operator precedence is complicated and wrong.
  - Hard-to-parse declarations/most vexing parse.
  - Template brackets < > are a nightmare to parse.
  - `std::forward` is error prone.
  - Braced initializers can choose the wrong constructor.
  - `0` shouldn't be a null pointer constant.
  - `this` shouldn't be a pointer.
- A system that can't fix its mistakes is an unhealthy system.

To err is human, to fix divine



To err is human, to fix divine



It's easy to fix C++

To err is human, to fix divine

- C++ has some "wrong defaults."
  - `[default_value_initialization]` - Uninitialized automatic variables.
  - `[no_integral_promotions]` - Integral promotions.
  - `[no_implicit_integral_narrowing]` - Implicit narrowing conversions.
  - `[switch_break]` - Switches should break rather than fallthrough.
  - `[simpler_precedence]` - Operator precedence is complicated and wrong.
  - `[new_decl_syntax]` - Hard-to-parse declarations/most vexing parse.
  - `[template_brackets]` - Template brackets < > are a nightmare to parse.
  - `[forward]` - `std::forward` is error prone.
  - `[safer_initializer_list]` - Braced initializers can choose the wrong constructor.
  - `[no_zero_nullptr]` - `0` shouldn't be a null pointer constant.
  - `[self]` - `this` shouldn't be a pointer.
- We can fix our mistakes.

It's easy to evolve C++

Successor languages?

```
1  #pragma feature edition_carbon_2023
2  #include <string>
3  #include <iostream>
4
5  using String = std::string;
6
7  choice IntResult {
8    Success(int32_t),
9    Failure(String),
10   Cancelled,
11 }
12
13 fn ParseAsInt(s: String) -> IntResult {
14   var r : int32_t = 0;
15   for(var c in s) {
16     if(not isdigit(c)) {
17       return .Failure("Invalid character");
18     }
19
20     // Accumulate the digits as they come in.
21     r = 10 * r + c - '0';
22   }
23
24   return .Success(r);
25 }
26
```

```
27 fn TryIt(s: String) {
28   var result := ParseAsInt(s);
29   match(result) {
30     .Success(var x)   => std::cout<< "Read integer "<< x<< "\n";
31     .Failure(var err) => std::cout<< "Failure '"<< err<< "'\n";
32     .Cancelled        => std::terminate();
33   };
34 }
35
36 fn main() -> int {
37   TryIt("12345");
38   TryIt("12x45");
39   return 0;
40 }
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
  Read integer 12345
  Failure 'Invalid character'
```

https://godbolt.org/z/qja4jn49K

```cpp
#pragma feature edition_carbon_2023
#include <string_view>
#include <iostream>

using str = std::string_view;

interface IAnimal {
  fn static make_new(name: str) -> Self;

  fn name() const -> str;
  fn noise() const -> str;

  // Traits can provide default method definitions.
  fn talk() const {
    std::cout<< name()<< " says "<< noise()<< " \n";
  }
}

struct Sheep {
  var naked : bool;
  var name : str;

  fn is_naked() const noexcept -> bool { return naked; }
  fn shear() noexcept {
    if(naked) {
      std::cout<< name<< " is already naked...\n";
    } else {
      std::cout<< name<< " gets a haircut!\n";
      naked = true;
    }
  }
}
```

https://godbolt.org/z/9xjbP69M6

```cpp
impl Sheep : IAnimal {
  fn static make_new(name: str) -> Sheep {
    return { .naked=false, .name=name };
  }

  fn name() const -> str {
    return self.name;
  }

  fn noise() const -> str {
    return self.naked ?
      "baaaaah?" :
      "baaaaah!";
  }

  // Default trait methods can be overriden.
  fn talk() const {
    std::cout<< name()<< " pauses briefly... "<< noise()<< "\n";
  }
}

fn main() -> int {
  // Create a Sheep instance through IAnimal's static method.
  var dolly := impl!<Sheep, IAnimal>::make_new("Dolly");

  // Put the Sheep : IAnimal impl in scope so member lookup works.
  using impl Sheep : IAnimal;

  // Call a mix of interface methods and class member functions.
  dolly.talk();
  dolly.shear();
  dolly.talk();
}
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
 Dolly pauses briefly... baaaaah!
 Dolly gets a haircut!
 Dolly pauses briefly... baaaaah?
```

Bubbles of new code

The "successor language" idea

Goals and history

My goal is to explore whether there's a way we can evolve C++ itself to become 10x simpler, safer, and more toolable. If we had an alternate C++ syntax, it would give us a "bubble of new code that doesn't exist today" where we could make arbitrary improvements (e.g., change defaults, remove unsafe parts, make the language context-free and order-independent, and generally apply 30 years' worth of learnings), free of backward source compatibility constraints.

Herb Sutter, CppFront
https://github.com/hsutter/cppfront#goals-and-history

Bubbles of new code

- I want "bubbles of new code."
- I don't want a new language!
- *Just declare that your code operates by new rules not the old rules.*

Features establish source domains

- Features have limited scope.
  - The *bubble of new code*.
- A feature changes the language within its source domain.
  - Richness
    - [as]
    - [choice]
    - [forward]
    - [interface]
    - [placeholder_keyword]
    - [template_brackets]
    - [tuple]
    - [self]
  - Safety
    - [default_value_initialization]
    - [no_implicit_floating_narrowing]
    - [no_implicit_integral_narrowing]
    - [no_implicit_pointer_to_bool]
    - [no_implicit_signed_to_unsigned]
    - [no_integral_promotions]
    - [no_zero_nullptr]
    - [safer_initializer_list]
    - [simpler_precedence]
    - [switch_break]

How feature pragmas work

- Each **textual file** has an active mask of features.
  - *Per-file*, not per translation unit.
  - These are not like command-line options.
- `#pragma feature X Y Z` - set features in the file mask.
- `#pragma feature_off X Y Z` - clear features in the file mask.
- `#pragma feature_off` - clear all features in the file mask.

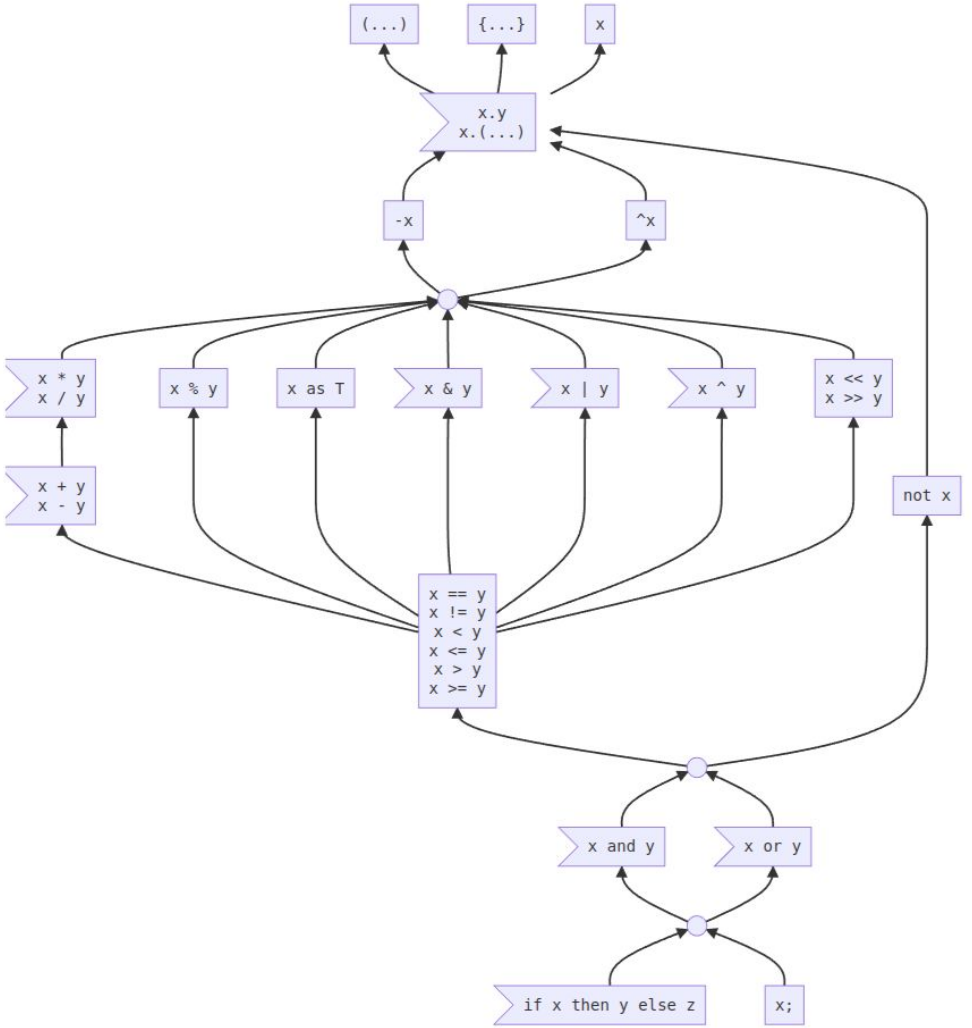
```cpp
#include <iostream>

// interface and impl are identifiers.
int interface = 0;
int impl = 1;

// Bring interface, impl, dyn, make_dyn and self in as keywords.
#pragma feature interface self

// interface is a keyword. Define an interface that allows implicit impls.
interface IPrint auto {
  void print() const default (Self~is_arithmetic) {
    std::cout<< self<< "\n";
  }
};

int main() {
  // Bring impl<int, IPrint> and impl<double, IPrint> into scope.
  using impl int, double : IPrint;

  // It looks like we're calling member functions on builtin types!
  // These are interface method calls. Name lookup finds them, because
  // we brought their impls into scope.
  int x = 101;
  double y = 1.618;
  x.print();
  y.print();

  // We can still access the interface and impl object declarations from
  // the top, using backtick identifiers.
  `interface` = 2;
  `impl` = 3;

  // Or turn off the feature entirely.
  #pragma feature_off interface
  interface = 4;
  impl = 5;
}
```

https://godbolt.org/z/Kqf4xeWMn

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
 101
 1.618
```

```
1  int main() {
2    const int mask  = 0x07;
3    const int value = 0x03;
4
5    static_assert(3 != mask & value);
6
7    #pragma feature simpler_precedence
8
9    static_assert(3 == mask & value);
10 }
```

https://godbolt.org/z/s5svdYrqc



https://github.com/carbon-language/carbon-lang/tree/trunk/docs/design/expressions#precedence

How feature pragmas work

- The active mask is written to the token stream *at each source location.*
- Circle takes feature-specific actions based on mask at source location:
  - `[interface]` - Promote identifiers like `interface` and `impl` to keywords.
  - `[simpler_precedence]` - Change *binary-expression* grammar.
  - `[no_integral_promotions]` - Change semantics around arithmetic.
- Templates don't create confusion.
  - Only consider the mask at each source location.
  - It doesn't matter who instantiated the template, only who defined it.

Files don't inherit features

- Application files (.cpp files):
  - Feature mask *has no effect* on files it includes.
- Library files (.h files):
  - Feature mask *has no effect* on files that include it.
- Files are independently featured.

Don't use the pragmas!

1. Make a `pragma.feature` file in your source folder.
2. List feature names/editions in the file.
3. These features get *picked up by every file in the folder*.

- A single point of definition for your folder.

How to deploy features at scale

- Modernize existing code with safety/security features.
  - Eg, disable narrowing conversions.

1. Create a file `pragma.feature`.
2. Add a feature to it.
3. Don't touch third party folders.
4. Build, test, fix issues.
   a. When something breaks, you *know why it broke*.
5. While more features remain, go to 2.
6. Deploy.

Extend the productive lifetime of C++ by decades

- Get rid of bad things:
  - Fix defects
  - Retire features we don't want
- Add good things:
  - Features we like
  - Different semantics that improve safety
  - Make core guidelines part of the language
- -> Don't break or change the meaning of existing code <-
  - 100% compatible with existing code
- ***Innovate without reaching consensus on any single issue!***

[no_integral_promotions]

- Builtins smaller than int are promoted to int before arithmetic.
- *Very surprising!*

```cpp
int main() {
  char           x = 1;
  unsigned char  y = 2;
  short          z = 3;
  unsigned short w = 4;

  // Integral types smaller than int are automatically promoted
  // to int before arithmetic.
  static_assert(int == decltype(x * x), "promote to int");
  static_assert(int == decltype(y * y), "promote to int");
  static_assert(int == decltype(z * z), "promote to int");
  static_assert(int == decltype(w * w), "promote to int");

  char8_t  a = 'a';
  char16_t b = 'b';
  char32_t c = 'c';
  wchar_t  d = 'd';
  static_assert(int      == decltype(a * a), "promote to int");
  static_assert(int      == decltype(b * b), "promote to int");
  static_assert(unsigned == decltype(c * c), "promote to unsigned");
  static_assert(int      == decltype(d * d), "promote to int");
}
```

https://godbolt.org/z/73487szMb

[no_integral_promotions]

- Turn off integral promotions.
- That was easy.

https://godbolt.org/z/M8bdd5M8o

```
1  int main() {
2    char           x = 1;
3    unsigned char  y = 2;
4    short          z = 3;
5    unsigned short w = 4;
6
7    // Turn off integral promotions
8    #pragma feature no_integral_promotions
9    static_assert(char           == decltype(x * x), "no promote to int");
10   static_assert(unsigned char  == decltype(y * y), "no promote to int");
11   static_assert(short          == decltype(z * z), "no promote to int");
12   static_assert(unsigned short == decltype(w * w), "no promote to int");
13
14   char8_t  a = 'a';
15   char16_t b = 'b';
16   char32_t c = 'c';
17   wchar_t  d = 'd';
18   static_assert(char8_t  == decltype(a * a), "no promote to int");
19   static_assert(char16_t == decltype(b * b), "no promote to int");
20   static_assert(char32_t == decltype(c * c), "no promote to unsigned");
21   static_assert(wchar_t  == decltype(d * d), "no promote to int");
22 }
```

[new_decl_syntax]

- Most vexing parse is a consequence of object and function declarations only being distinguished by their declarators.
- If the declarator can be parsed as a function, then it's a function declaration.

```
1  struct a_t { };
2
3  struct b_t {
4    b_t(a_t);
5    int x;
6  };
7
8  int main() {
9    // Most vexing parse:
10   // This is not an object declaration.
11   // It's a function declaration.
12   b_t obj(a_t());
13
14   // Error: can't access obj.x, because obj
15   // isn't an object, it's a function name.
16   obj.x = 1;
17 }
```

[new_decl_syntax]

- `fn` declares functions.
- `var` declares objects and members.
- Most vexing parse is resolved. ✅

```
1  #pragma feature new_decl_syntax
2
3  struct a_t { }
4
5  struct b_t {
6    fn b_t(a : a_t);
7    var x : int;
8  }
9
10 fn main() -> int {
11   // The most vexing parse has been resolved.
12   // This is explicitly, not a function declaration.
13   var obj : b_t = a_t();
14
15   // Ok. obj really is an object.
16   obj.x = 1;
17 }
```

https://godbolt.org/z/r58756KeW

Difficulties improving C++

C++ is the dominant programming language for the performance critical software our goals prioritize. The most direct way to deliver a modern and excellent developer experience for those use cases and developers would be to improve C++.

Improving C++ to deliver the kind of experience developers expect from a programming language today is difficult in part because **C++ has decades of technical debt** accumulated in the design of the language. It inherited the legacy of C, including textual preprocessing and inclusion. At the time, this was essential to C++'s success by giving it instant and high quality access to a large C ecosystem. However, over time this has resulted in significant technical debt ranging from integer promotion rules to complex syntax with "the most vexing parse".

- `[no_integral_promotions]` - no integral promotions.
- `[new_decl_syntax]` - no most vexing parse.

[edition_2023]

- Group features together into editions – keep people on the same page.
- Editions bend the language to your core guidelines.

- Richness
  - [as]
  - [choice]
  - [forward]
  - [interface]
  - [placeholder_keyword]
  - [template_brackets]
  - [tuple]
  - [self]

- Safety
  - [default_value_initialization]
  - [no_implicit_floating_narrowing]
  - [no_implicit_integral_narrowing]
  - [no_implicit_pointer_to_bool]
  - [no_implicit_signed_to_unsigned]
  - [no_integral_promotions]
  - [no_zero_nullptr]
  - [safer_initializer_list]
  - [simpler_precedence]
  - [switch_break]

[carbon_edition_2023]

- Richness
  - [as]
  - [choice]
  - [forward]
  - [interface]
  - [placeholder_keyword]
  - [template_brackets]
  - [tuple]
  - [self]
- Safety
  - [default_value_initialization]
  - [no_implicit_floating_narrowing]
  - [no_implicit_integral_narrowing]
  - [no_implicit_pointer_to_bool]
  - [no_implicit_signed_to_unsigned]
  - [no_integral_promotions]
  - [no_zero_nullptr]
  - [safer_initializer_list]
  - [simpler_precedence]
  - [switch_break]
- Simplicity
  - [new_decl_syntax]
  - [require_control_flow_braces]
  - [no_function_overloading]
  - [no_multiple_inheritance]
  - [no_user_constructors]
  - [no_virtual_inheritance]

Vic□or

@vzverovich

BREAKING: @seanbax implemented every programming language as a set of pragmas for circle.

10:10 AM · Dec 12, 2022

Thinking C++ is doomed is easy.

Thinking C++ is ripe with potential?
That takes independence.

Be creative

Be creative

ISO brain

Features I like: necessary.

Features other people like: the language is already too complex.

We don't see progress in any human pursuit

when this is what people jump to.

Try everything

2. Feature catalog
   i. `[adl]`
   ii. `[as]`
   iii. `[choice]`
      - Pattern matching
      - Structured binding patterns
      - Test patterns
      - Designated binding patterns
      - Other ways to access choice objects
      - Choice type requirements
   iv. `[default_value_initialization]`
   v. `[forward]`
      - std::forward is bad
      - First-class forwarding
   vi. `[interface]`
      - Interface definitions
      - Impls
      - Interface name lookup
      - Interfaces in templates
      - Type erasure and `dyn`
      - Type erasure and the heap
      - Value semantics containers
   vii. `[new_decl_syntax]`
      - Function declarations
      - Object declarations
   viii. `[no_function_overloading]`
   ix. `[no_implicit_ctor_conversions]`
   x. `[no_implicit_enum_to_underlying]`
   xi. `[no_implicit_floating_narrowing]`
   xii. `[no_implicit_integral_narrowing]`
   xiii. `[no_implicit_pointer_to_bool]`
   xiv. `[no_implicit_signed_to_unsigned]`
   xv. `[no_implicit_user_conversions]`
   xvi. `[no_implicit_widening]`
   xvii. `[no_integral_promotions]`
   xviii. `[no_multiple_inheritance]`
   xix. `[no_signed_overflow_ub]`
   xx. `[no_user_defined_ctors]`
   xxi. `[no_virtual_inheritance]`
   xxii. `[no_zero_nullptr]`
   xxiii. `[placeholder_keyword]`
   xxiv. `[require_control_flow_braces]`
   xxv. `[safer_initializer_list]`
   xxvi. `[self]`
   xxvii. `[simpler_precedence]`
   xxviii. `[switch_break]`
   xxix. `[template_brackets]`
      - Abbreviated template arguments
   xxx. `[tuple]`

https://github.com/seanbaxter/circle/blob/master/new-circle/README.md

Starting with a working compiler builds confidence in your ideas

Successor languages vs evolver languages

It's good to leverage an existing toolchain!	Lines of code	Has compiler
Cppfront	12,000	❌
Val	27,000	❌
Carbon	44,000	❌
Circle	284,000	✅

```
[as]
[no_implicit_floating_narrowing]
[no_implicit_integral_narrowing]
[no_implicit_enum_to_underlying]
[no_implicit_pointer_to_bool]
[no_implicit_ctor_conversions]
[no_implicit_user_conversions]
```

- Disable implicit conversions.
  - For narrowing-related, permit when non-narrowing is confirmed at compile time.
- ***expr*** `as` ***type-id*** - *as-expression* cast, a nicer `static_cast`.
- ***expr*** `as _` enables implicit conversion sequences.
  - Borrowed from Rust.
- Follows *principle of least surprise.*
  - Compiler *considers* implicit conversions. If one is chosen during overload resolution, the program is ill-formed.
  - Compiler prompts you for an explicit conversion or `as _`.

```
[as]
[no_implicit_floating_narrowing]
[no_implicit_integral_narrowing]
```

```cpp
#pragma feature as no_implicit_integral_narrowing no_implicit_floating_narrowing

void f_short(short x);
void f_int(int x);
void f_float(float x);

int main() {
  int x = 100;
  f_short(x);           // Error
  f_short(x as short);  // OK
  f_short(x as _);      // OK

  double y = 100;
  f_float(y);           // Error
  f_float(y as float);  // OK
  f_float(y as _);      // OK

  f_int(y);             // Error
  f_int(y as int);      // OK
  f_int(y as _);        // OK
}
```

```
error: example.cpp:9:11
[no_implicit_integral_narrowing]: no implicit conversion from int to short
  f_short(x);           // Error
          ^

error: example.cpp:14:11
[no_implicit_floating_narrowing]: no implicit conversion from double to float
  f_float(y);           // Error
          ^

error: example.cpp:18:9
[no_implicit_floating_narrowing]: no implicit conversion from double to int
  f_int(y);             // Error
        ^
```

https://godbolt.org/z/zKzvWMb18

[adl]

Meeting C++: What feature would you remove from C++, if you could?
Sean Parent: It would probably be argument-dependent lookup.

- Only permit calls to ADL functions with the `adl` keyword.
- ADL still runs, but errors when an ADL candidate is called.

https://godbolt.org/z/qr5hbMWYT

```
1  #pragma feature adl
2  #include <tuple>
3
4  int main() {
5    auto tup = std::make_tuple(1, 2.2, "Three");
6
7    // Ok.
8    auto x1 = std::get<0>(tup);
9
10   // Ok.
11   auto x2 = adl get<0>(tup);
12
13   // Error.
14   auto x3 = get<0>(tup);
15 }
```

```
error: example.cpp:14:19
[adl]: adl candidate called without adl token before unqualified name
  best viable candidate int& std::get(std::tuple<int, double, const char*>&) noexcept was chosen by adl
  function declared at /opt/compiler-explorer/gcc-10.3.0/include/c++/10.3.0/tuple:1299:5
  auto x3 = get<0>(tup);
                   ^
```

[forward]

- Why does T&& mean *forwarding parameter*? It shouldn't!
- `forward` is a keyword.
- `forward` is a parameter directive.
  - `void func(forward auto param);`
- `forward` is an expression.
  - `func2(forward param);`
- Can't be used incorrectly. *forward-expression* must name a *forwarding parameter*.
- Can't inadvertently use `std::forward` since `forward` is now a keyword.
  - The keyword shadows the identifier.
  - You could still write `std::`forward`.

bad forward

- Naming subobjects in std::forward *does the wrong thing*.
- We want an lvalue but it gives an rvalue.

```
Program returned: 0
  Pass by lvalue:
    int&&  double&&
    int&  double&
  Pass by rvalue:
    int&&  double&&
    int&&  double&&
```

```cpp
#include <iostream>

#define CPP2_FORWARD(x) std::forward<decltype(x)>(x)

struct pair_t {
  int first;
  double second;
};

void print(auto&&... args) {
  std::cout<< "  " + decltype(args)~string ...;
  std::cout<< "\n";
}

void func(auto&& obj) {
  print(CPP2_FORWARD(obj.first), CPP2_FORWARD(obj.second));    // BAD!
  print(CPP2_FORWARD(obj).first, CPP2_FORWARD(obj).second);    // GOOD!
}

int main() {
  std::cout<< "Pass by lvalue:\n";
  pair_t obj { 1, 2.2 };
  func(obj);

  std::cout<< "Pass by rvalue:\n";
  func(pair_t { 3, 4.4 });
}
```

https://godbolt.org/z/bf1Kcd7xr

good forward

- `forward` is a parameter directive.
- `forward` is an expression.
- A *forward-expression* can only name a forward parameter.
- Subobject access works.

```
Program returned: 0
  Pass by lvalue:
    pair_t<int, double>&
    int&  double&
  Pass by rvalue:
    pair_t<char, unsigned short>&&
    char&&  unsigned short&&
```

```cpp
#pragma feature forward
#include <iostream>

void consume(forward auto... args) {
  std::cout<< "  " + decltype(forward args)~string ...;
  std::cout<< "\n";
}

void func(forward auto pair) {
  consume(forward pair);
  consume(forward pair.first, forward pair.second);
}

template<typename T1, typename T2>
struct pair_t {
  T1 first;
  T2 second;
};

int main() {
  std::cout<< "Pass by lvalue:\n";
  pair_t pair { 100, 200.2 };
  func(pair);

  std::cout<< "Pass by rvalue:\n";
  func(pair_t { 1i8, 2ui16 });
}

```

https://godbolt.org/z/b5jcG17MG

[simpler_precedence]

Operator precedence

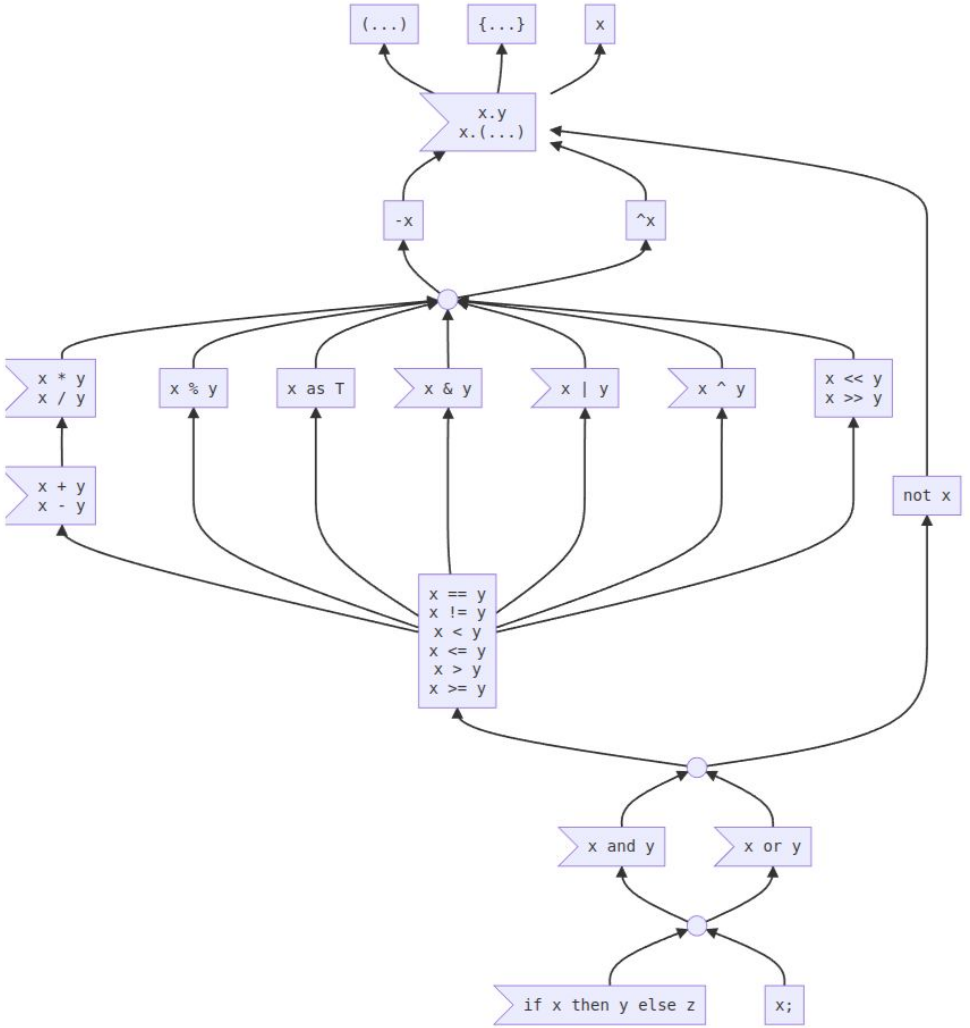
- Too many levels.
- Bitwise ops are in the wrong place. 🤢
- Confusing logical operator precedence makes for very hard-to-diagnose bugs.

`.*` `->*`	Pointer-to-member
`a*b` `a/b` `a%b`	Multiplication, division, and remainder
`a+b` `a-b`	Addition and subtraction
`<<` `>>`	Bitwise left shift and right shift
`<=>`	Three-way comparison operator (since C++20)
`<` `<=` `>` `>=`	For relational operators < and ≤ and > and ≥ respectively
`==` `!=`	For equality operators = and ≠ respectively
`a&b`	Bitwise AND
`^`	Bitwise XOR (exclusive or)
`|`	Bitwise OR (inclusive or)
`&&`	Logical AND
`||`	Logical OR
`a?b:c`	Ternary conditional[note 2]
`throw`	throw operator
`co_yield`	yield-expression (C++20)
`=`	Direct assignment (provided by default for C++ classes)
`+=` `-=`	Compound assignment by sum and difference
`*=` `/=` `%=`	Compound assignment by product, quotient, and remainder
`<<=` `>>=`	Compound assignment by bitwise left shift and right shift
`&=` `^=` `|=`	Compound assignment by bitwise AND, XOR, and OR

[simpler_precedence]

- Carbon's most attractive language simplification.
- Follow the path from bottom to top.
- Can't mix operators from different paths - use ( ).
- Can't accidentally get precedences wrong!

https://github.com/carbon-language/carbon-lang/tree/trunk/docs/design/expressions#precedence



[simpler_precedence]

```cpp
#pragma feature simpler_precedence

int main() {
  int x = 0, y = 1, z = 2;

  auto r1 = x || y && z;          // Error
  auto r2 = (x || y) && z;        // Ok

  auto r3 = y ^ 3 | z;            // Error
  auto r4 = y ^ (3 | z);          // Ok

  auto r5 = x<< y + z;            // Error
  auto r6 = x<< (y + z);          // Ok

  auto r7 = x & y == z & y;       // Ok
  auto r8 = (x & y) == (z & y); // Same as above.
}
```

https://godbolt.org/z/GExx6ohfd

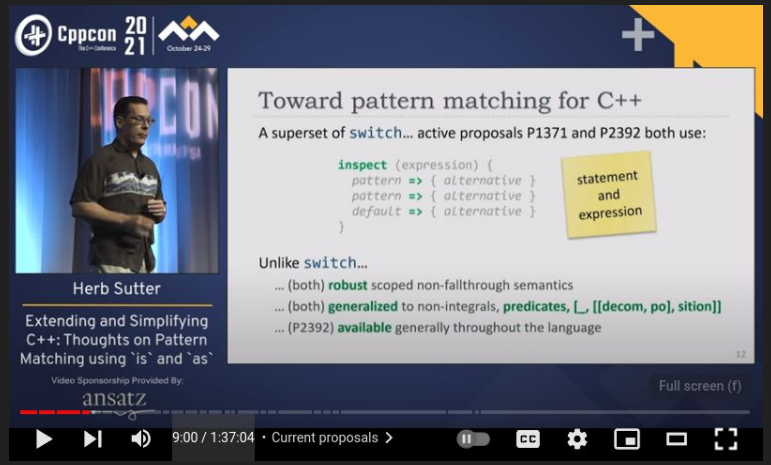
```
error: example.cpp:6:20
[simpler_precedence]: cannot follow operator|| with operator&&
  auto r1 = x || y && z;        // Error
                   ^

error: example.cpp:9:19
[simpler_precedence]: cannot follow operator^ with operator|
  auto r3 = y ^ 3 | z;          // Error
                  ^

error: example.cpp:12:19
[simpler_precedence]: cannot follow operator<< with operator+
  auto r5 = x<< y + z;          // Error
                  ^
```

[choice]

[choice]

- `choice` is a type safe discriminated union.
  - Just like Rust/Swift's enum!
  - Better than std::variant - no valueless_by_exception state.
  - Convenient dotted initializers.
- `match` is a pattern match.
  - Most closely resembles C#'s switch.
  - match patterns back-ported to C++ structured bindings.
  - Consider match in two minutes instead of 1h37m.

```
template<typename T>
choice Option {
  None,
  Some(T),
};

template<typename T, typename E>
choice Result {
  Ok(T),
  Err(E),
};
```



[choice]

- Match alternatives
- Destructure tuples
- Bind variables
- Test expressions

```
#pragma feature choice new_decl_syntax
#include <string>
#include <tuple>
#include <iostream>

choice MyChoice {
  MyTuple(int, std::string),   // The payload is a tuple.
  MyArray(double[3]),          // The payload is an array.
  MyScalar(short),             // The payload is a scalar.
}

fn test(obj : MyChoice) {
  // You can pattern match on tuples and arrays.
  match(obj) {
    .MyTuple([1, var b])    => std::cout<< "The tuple int is 1\n";
    .MyArray([var a, a, a]) => std::cout<< "All array elements are "<< a<< "\n";
    .MyArray(var [a, b, c]) => std::cout<< "Some other array\n";
    .MyScalar(> 10)         => std::cout<< "A scalar greater than 10\n";
    .MyScalar(var x)        => std::cout<< "The scalar is "<< x<< "\n";
    _                       => std::cout<< "Something else\n";
  };
}

fn main() -> int {
  test(.MyTuple{1, "Hello choice tuple"});
  test(.MyArray{10, 20, 30});
  test(.MyArray{50, 50, 50});
  test(.MyScalar{100});
  test(.MyScalar{6});
  test(.MyTuple{2, "Foo"});
}
```

[choice]

- Expressive tests.
- Scalars
- Ranges (x … y)
- bool-valued functions
- || alternatives

```
1  #pragma feature choice new_decl_syntax
2  #include <iostream>
3  #include <concepts>
4
5  fn even(x : std::integral auto) noexcept -> bool {
6    return 0 == x % 2;
7  }
8
9  fn func(arg : auto) {
10   match(arg) {
11     5                => std::cout<< "The arg is five.\n";
12     10 ... 20        => std::cout<< "The arg is between 10 and 20.\n";
13     even             => std::cout<< "The arg is even.\n";
14     1 || 3 || 7 || 9 => std::cout<< "The arg is special.\n";
15     _                => std::cout<< "The arg is not special.\n";
16   };
17 }
18
19 fn main() -> int {
20   func(5);
21   func(13);
22   func(32);
23   func(7);
24   func(21);
25 }
```

https://godbolt.org/z/fcM38f8G3

```
Program returned: 0
  The arg is five.
  The arg is between 10 and 20.
  The arg is even.
  The arg is special.
  The arg is not special.
```

[interface]

[interface]

- `interface` - Like Rust traits and Swift protocols.
  - A 3rd way of organizing functions by "receiver type."
- interface templates - Parameterize interfaces by template parameters.
  - Pass interfaces and interface templates as template arguments.
- `impl` - Partial or explicit specializations of a complete impl.
  - Associate types and interfaces.
  - Type-specific implementations of interface methods.
- `dyn` - First-class dynamic type erasure.
  - External polymorphism. Access through `dyn<IFace>*`.
  - Compiler generates a "dyntable" - like a virtual table for external polymorphism.
- `make_dyn` - Convert an object pointer to a `dyn<IFace>*`.

[interface]

- 3 ways to organization functions:

1. Class member functions.
   - - Internal declarations.
   - + No overloading.
2. Free functions.
   - + External declarations.
   - - Requires overloading.
3. Interface methods.
   - + External declarations.
   - + No overloading.

```
1  // Classes: methods are bound with data.
2  struct A1 { void print() const; };
3  struct B1 { void print() const; };
4
5  // Overloading: free functions are overloaded by
6  // receiver type.
7  struct A2 { }; struct B2 { };
8
9  void print(const A2& a);
10 void print(const B2& b);
11
12 // Interfaces: types externally implement
13 // interfaces.
14 // Receiver type is implicit in the impl.
15 #pragma feature interface
16
17 interface IPrint { void print() const; };
18
19 struct A3 { };
20 struct B3 { };
21
22 impl A3 : IPrint { void print() const; };
23 impl B3 : IPrint { void print() const; };
```

https://godbolt.org/z/Yd3efGhGo

[interface]

- New `interface` language entity.
- Optional default method implementation.
- *using-impl-specifier* brings impls into scope.
- *External polymorphism* extends the interface of any object type.

```
1  #pragma feature interface self
2  #include <iostream>
3
4  // Define an interface that allows implicit impls.
5  interface IPrint auto {
6    void print() const default (Self~is_arithmetic) {
7      std::cout<< self<< "\n";
8    }
9  };
10
11 int main() {
12   // Bring IPrint impls into scope.
13   using impl int, double : IPrint;
14
15   // Call interface methods on the impls in scope.
16   int x = 101;
17   double y = 1.618;
18   x.print();
19   y.print();
20 }
```

https://godbolt.org/z/x9nxT8eTr

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
 101
 1.618
```

[interface]

- Type parameters are constrained by interfaces.
- Supports multiple:
  - IFace1 & IFace2 & IFace3
- Interfaces are implicitly brought into scope for name lookup.

```
1  #pragma feature interface self
2  #include <iostream>
3
4  // Define an interface that allows implicit impls.
5  interface IPrint auto {
6    void print() const default (Self~is_arithmetic) {
7      std::cout<< self<< "\n";
8    }
9  };
10
11 template<typename T : IPrint>
12 void func(T obj) {
13   obj.print();    // Performs name lookup into IPrint.
14 }
15
16 int main() {
17   func(101);
18   func(1.618);
19 }
```

https://godbolt.org/z/bo16Gjcdq

```
Program returned: 0
 101
 1.618
```

[interface]

- Explicit impls are specializations of a complete `impl` template.
- Normal partial deduction and partial ordering rules apply.
  - Don't invent when you can reuse.

```
1  #pragma feature interface self
2  #include <iostream>
3  #include <type_traits>
4
5  interface IPrint {
6    void print() const;
7  };
8
9  impl double : IPrint {
10   void print() const {
11     std::cout<< "double: "<< self<< "\n";
12   }
13 };
14
15 template<typename T> requires(T~is_arithmetic)
16 impl T : IPrint {
17   void print() const {
18     std::cout<< T~string + ": "<< self<< "\n";
19   }
20 };
21
22 template<typename T : IPrint>
23 void func(T obj) {
24   obj.print();
25 }
26
27 int main() {
28   func(100u);
29   func(1.618);
30 }
```

https://godbolt.org/z/684KnYoec

```
Program returned: 0
 unsigned: 100
 double: 1.618
```

C++ now

Polymorphism À La Carte

Eduardo Madrid &
Phil Nash

JET BRAINS Bloomberg Engineering

Virtual functions can be tricky to get right, especially in large class heirarchies.

- We've all seen lots of these problems in real code.
- C++11's `override` and `final` help.
- There is frequently a question of whether or not a given type's virtual function implementation should call its base class's version of the same function.

PRAGMATIC TYPE ERASURE: SOLVING CLASSIC OOP PROBLEMS WITH AN ELEGANT DESIGN PATTERN
Zach Laine

Type erasure…
Okay?

Cppcon The C++ Conference | Breaking Dependencies: Type Erasure - A Design Analysis

Klaus Iglberger

21

Breaking Dependencies:
Type Erasure - A Design Analysis

KLAUS IGLBERGER

Cppcon The C++ Conference

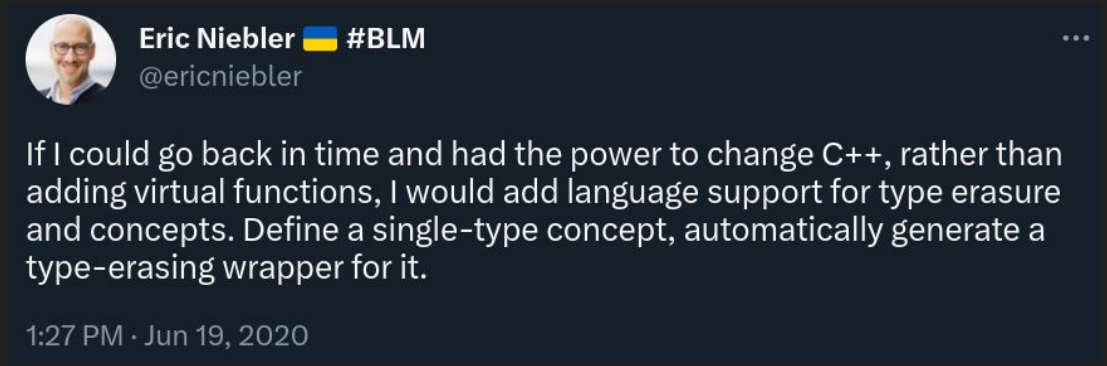
2021 October 24-29

…don }
16-20 Janua… 2017

Inspiring Software Developers since 2008

Type erasure

- Type erasure is *external polymorphism*.
- `dyn<IFace>` is an object type-erased behind an interface.
- `dyn<IFace>*` is a *fat pointer.*
  - 8 bytes for data pointer
  - 8 bytes for dyntable pointer.
- `make_dyn<IFace>(p)` generates a fat pointer from an object pointer.



[interface]

- Only lines 25 and down are new!
- Circle automatically generates a *dyntable*.
- The dyntable has pointers to impl methods.
- It's just like virtual functions, but external to the class, not internal to it.
- Interface/impl is like *external inheritance*.

```
1  #pragma feature interface self
2  #include <iostream>
3
4  interface IFace {
5    void print() const;
6  };
7
8  impl int : IFace {
9    void print() const {
10     std::cout<<"int.print = "<< self<< "\n";
11   }
12 };
13
14 impl std::string : IFace {
15   void print() const {
16     std::cout<<"string.print = "<< self<< "\n";
17   }
18 };
19
20 int main() {
21   int x = 100;
22   std::string s = "My nifty string";
23
24   // Type erase the objects.
25   dyn<IFace>* p1 = make_dyn<IFace>(&x);
26   dyn<IFace>* p2 = make_dyn<IFace>(&s);
27
28   // Invoke the interface methods.
29   p1->print();
30   p2->print();
31 }
```

https://godbolt.org/z/oazzWa3Tz

```
Program returned: 0
  int.print = 100
  string.print = My nifty string
```

Nine kinds of template parameters

```
#pragma feature interface
#include <iostream>
#include <concepts>
#include <vector>

template<
                auto      nontype,
                typename  type,
  template<...> typename  type_template,
  template<...> auto      var_template,
  template<...> concept   concept_,
                interface interface_,
  template<...> interface interface_template,
                namespace namespace_,
  template      auto      universal
> void f() {
  std::cout<< "nontype           = {}\n".format(nontype~string);
  std::cout<< "type              = {}\n".format(type~string);
  std::cout<< "type_template     = {}\n".format(type_template~string);
  std::cout<< "var_template      = {}\n".format(var_template~string);
  std::cout<< "concept           = {}\n".format(concept_~string);
  std::cout<< "interface         = {}\n".format(interface_~string);
  std::cout<< "interface_template = {}\n".format(interface_template~string);
  std::cout<< "namespace         = {}\n".format(namespace_~string);
  std::cout<< "universal         = {}\n".format(universal~string);
}
```

https://godbolt.org/z/o3z6eofP8

```

interface IPrint { };

template<interface IBase>
interface IClone : IBase { };

int main() {
  f<
    5,                  // non-type
    char[3],            // type
    std::basic_string,  // type template
    std::is_signed_v,   // variable template
    std::integral,      // concept
    IPrint,             // interface
    IClone,             // interface template
    std,                // namespace
    void                // universal
  >();
}
```

```
Program returned: 0
  nontype           = 5
  type              = char[3]
  type_template     = std::basic_string
  var_template      = std::is_signed_v
  concept           = std::integral
  interface         = IPrint
  interface_template = IClone
  namespace         = std
  universal         = void
```

[interface]

- Achieve value semantics with a clone interface.
- Generate clone functions *automatically* for types with copy constructors.
- `IClone<>` is parameterized over the interface we want for type erasure.
- It's really good to take interfaces as template parameters!

https://godbolt.org/z/YMhzEcx8a

```
1  #pragma feature interface forward self template_brackets
2  #include <memory>
3  #include <iostream>
4
5  // This is all goes into a library.
6
7  // Create a unique_ptr that wraps a dyn.
8  template<typename Type, interface IFace>
9  std::unique_ptr!dyn!IFace make_unique_dyn(forward auto... args) {
10   return std::unique_ptr!dyn!IFace(make_dyn<IFace>(new Type(forward args...)));
11 }
12
13 // Implicitly generate a clone interface for copy-constructible types.
14 template<interface IFace>
15 interface IClone auto : IFace {
16   // The default-clause causes SFINAE failure to protect the program from
17   // being ill-foremd if IClone is attempted to be implicitly instantiated
18   // for a non-copy-constructible type.
19   std::unique_ptr!dyn!IClone clone() const
20   default(Self~is_copy_constructible) {
21     // Pass the const Self lvalue to make_unique_dyn, which causes copy
22     // construction.
23     return make_unique_dyn!<Self, IClone>(self);
24   }
25 };
```

Incentivizing investment in language technology

Principal scientists, architects, staff researchers

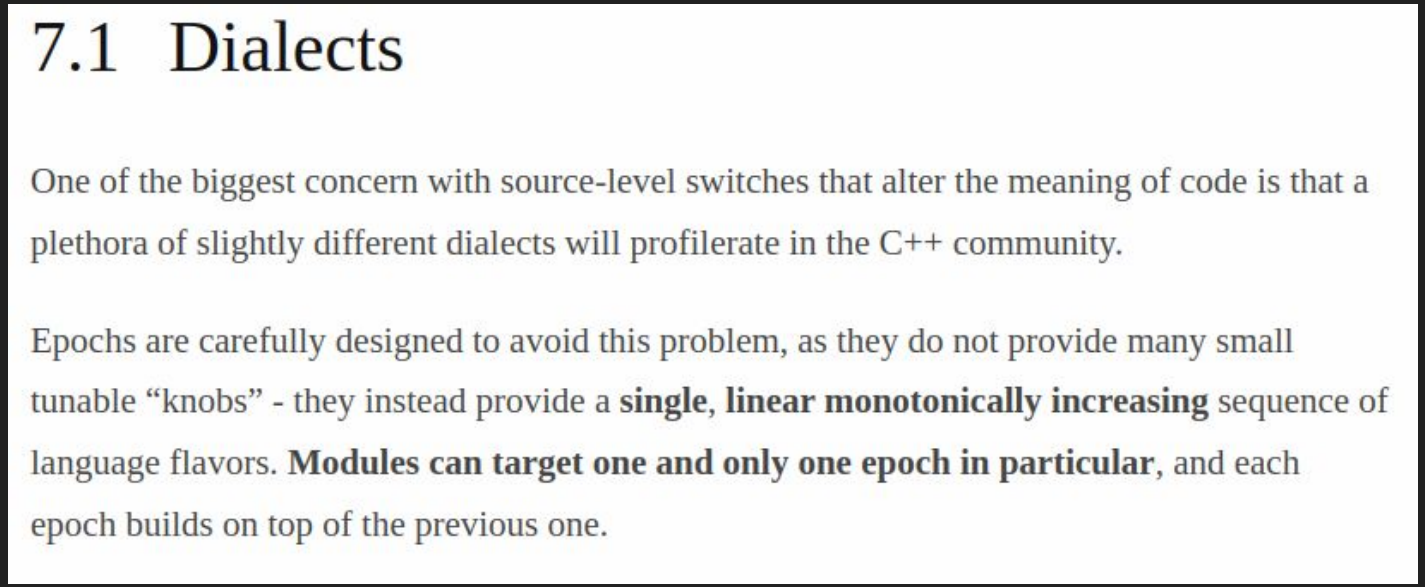
- Institutional users employ *professional language opiners*.
- Language opiner *conceptualizes* a feature:
  - Does a study. Consults accounting.
  - Feature saves organization $100 million / 10 years.
- Language opiner *takes action* to land the feature:
  - Flies to six conferences a year.
  - Submits four proposals a year.
  - Joins ISO study group.
  - Repeats for five years.
- Spend lots of time and money.
- No feature!
- We are leaving money on the table.

What business is it of mine what you do with your organization?

1. Conceptualize
2. Build
3. Test
4. Deploy
5. Upstream

Incentivize compiler development

- Everything in this talk was built by one guy over six months.
- The feature system encourages rapid creation.
- How does this change the value proposition of supporting language development?
  - Contracts, language allocators, etc.
- A path to get your changes into a compiler.
- I want companies to turn *institutional knowledge* into technology.
- The consensus model isn't the only way:
  - Work with the vendors.
  - Let vendors talk to one another.



7.1 Dialects

One of the biggest concern with source-level switches that alter the meaning of code is that a plethora of slightly different dialects will profilerate in the C++ community.

Epochs are carefully designed to avoid this problem, as they do not provide many small tunable “knobs” - they instead provide a **single**, **linear monotonically increasing** sequence of language flavors. **Modules can target one and only one epoch in particular**, and each epoch builds on top of the previous one.

Vittorio Romeo - P1881R0 "Epochs"

- A single design is great… *if you're the designer!*
- Fine granularity avoids the need for consensus.

Research features

Rust envy

- Some C++ people have a bad case of Rust envy.
- Don't eat your heart out.
- Identify the things you like.
- Implement them in C++.
- C++ keeps up while supporting on money-making projects.

Rust envy

- `[interface]`
  - Rust **trait** implemented as Circle `interface`.
  - Type erasure - Rust `dyn` is Circle `dyn`.
- `[choice]`
  - Rust enums implemented as Circle *choice types*.
  - Pattern matching.
- `[new_decl_syntax]`
  - Modernized declaration syntax.
- `[as]`
  - Rust's *as-expression* implemented as Circle *as-expression*.
- `[relocate]` (in development)
  - Ownership semantics and destructive move.
- `[borrow_checking]` (in development)
  - `ref` and `refmut` types and expressions.

[relocate]

- Establish *ownership semantics* for C++.
- You can't *borrow* something if nobody *owns* it.
- *Relocate* transfers ownership.
- `relocate` constructor is new special member function.
  - Almost always implicitly generated.
- *relocate-expression* atomically:
  - Constructs new prvalue from the rhs object.
  - Ends the lifetime of the rhs object.

```
1  template<typename T>
2  struct unique_ptr {
3    // Three options for user-declared relocation.
4    // 1. Implicitly-defined relocation constructor.
5    relocate(unique_ptr rhs) = default;
6
7    // 2. Deleted relocation constructor.
8    relocate(unique_ptr rhs) = delete;
9
10   // 3. User-defined relocation constructor.
11   relocate(unique_ptr rhs) {
12     m_p = relocate rhs.m_p;
13   }
14
15   T* m_p;
16 };
17
18 int main() {
19   // Create a new object.
20   unique_ptr<double> p1;
21
22   // relocate-expression creates a prvalue and
23   // destroys the rhs.
24   auto p2 = relocate p1;
25
26   p1->call();  // Error: p1 has been relocated.
27   p2->call();  // Ok.
28 }
```

[borrow_checking]

- `ref` and `refmut` borrow types.
- `ref` and `refmut` borrow expressions.
- Allow multiple refs to an object.
- Or, allow one refmut to an object.
- Lower to mid-level IR:
  - Lifetime analysis of borrows.
- Just copy Rust. 😉
- Develop borrow checking independently of unrelated features:
  - lexer, parser, overloading, pattern matching, interfaces, generics, irrelevant!

[argument_directives]

- The future of C++?
- Evolve towards making object lifetime operations *explicit* at the point of arguments.
- `copy` and `move` create new objects.
- `relocate` is the default.
- `ref` and `refmut` borrow.

```
1  void func(obj_t obj);         // #1
2  void func(obj_t ref obj);     // #2
3  void func(obj_t refmut obj); // #3
4
5  void call(obj_t obj) {
6    // Copy-construct and call #1.
7    func(copy obj);
8
9    // Move-construct and call #1.
10   func(move obj);
11
12   // Implicitly relocate and call #1.
13   func(obj);
14
15   // Pass a non-mutable ref to #2.
16   func(ref obj);
17
18   // Pass a mutable ref to #3.
19   func(refmut obj);
20 }
```

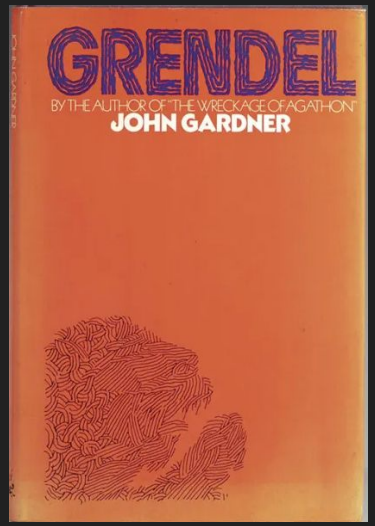
Be creative

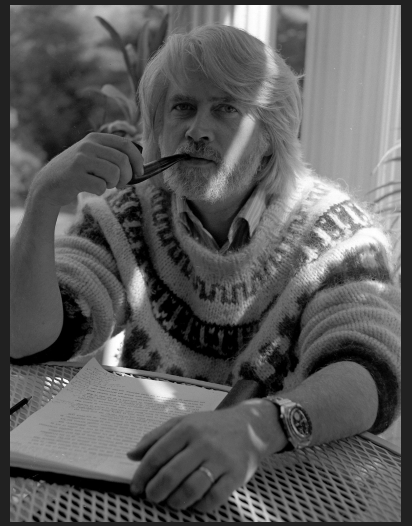
Book minute

Grendel - John Gardner 1971

- Beowulf - Dated 700-1000 CE
- Beowulf feigns sleep, tussles with Grendel and pulls his arm off.
- Most significant Old English epic.
- Not that good - Vikings* weren't bookish.
- Important to thinkers who looked to the North rather than to Greece. (Tolkien, Wagner)

* Ok, technically Goths.





- Beowulf told from Grendel's perspective.
- The dragon explains Grendel's role as historical antagonist.
  - Challenge men to force them to form common defense, societies and government.
- Grendel has many monologues about this responsibility.
- A French existentialist novel told by an actual monster.

No questions

Go make language evolution

Sean Baxter
circle-lang.org
@seanbax
circle-lang.org/bloomberg-2023.pdf