CLASSES C++23 STYLE

Sebastian Theophil
stheophil@think-cell.com

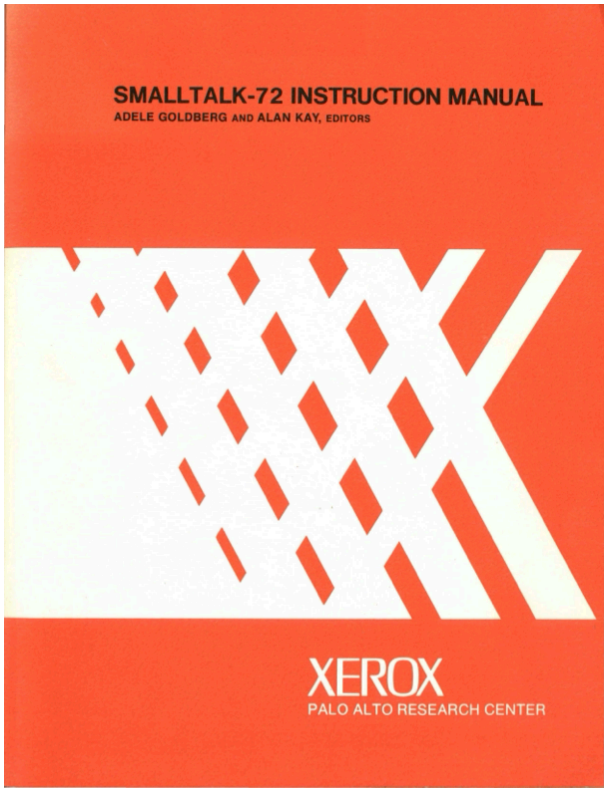
PRELIMINARIES

- All code samples compiled with some compiler
- Everything is `using std;` for brevity
- All code samples shown use UTF-8 encoding (although you cannot tell)
- Now officially supported source file encoding 👏

PRELIMINARIES

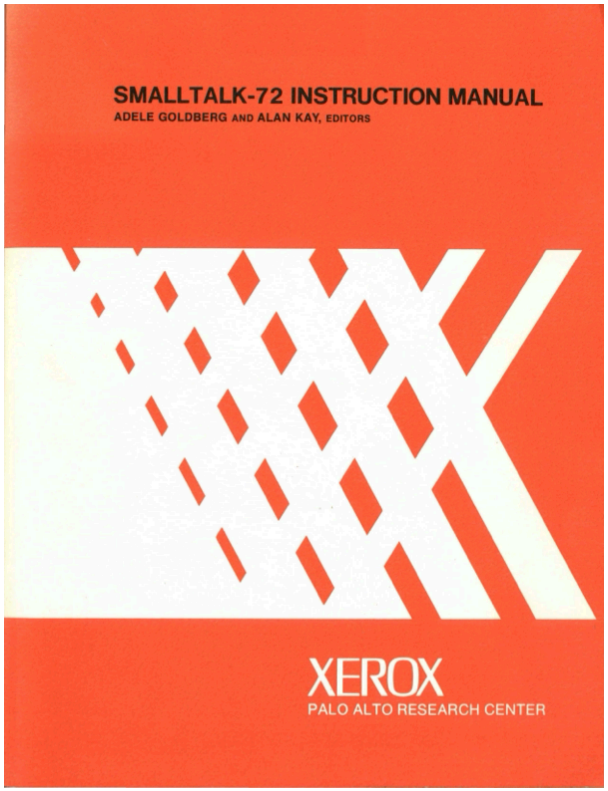
- All code samples compiled with some compiler
- Everything is `using std;` for brevity
- All code samples shown use UTF-8 encoding (although you cannot tell)
- Now officially supported source file encoding 👏
- Obviously the best C++23 feature

WHAT IS A CLASS?



Instead of dividing "computer stuff" into things each less strong than the whole -- like data structures, procedures, and functions [...] -- each Smalltalk ***object*** *is a recursion of the entire possibilities of the computer.*

Alan Kay, The Early History of Smalltalk
ACM SIGPLAN Notices, No.3 March 1993
https://tinyurl.com/3ssaxxb6



[the potential to] amplify human reach and bring new ways of thinking to a faltering civilization that desperately needed it.

Alan Kay, The Early History of Smalltalk
ACM SIGPLAN Notices, No.3 March 1993
https://tinyurl.com/3ssaxxb6

WHAT IS A CLASS?

- Unit of data and code
- In C++, it has constructors, destructors, base classes, access control, member functions template arguments. These are **syntax**.
- A class should have meaning

WHAT DOES IT REPRESENT?

- A simple value?
- A container?
- A resource?
- A semantic singleton?

WHAT DOES IT REPRESENT?

- A simple value?
- A container?
- A resource?
- A semantic singleton?

What lifecyle does the type have?

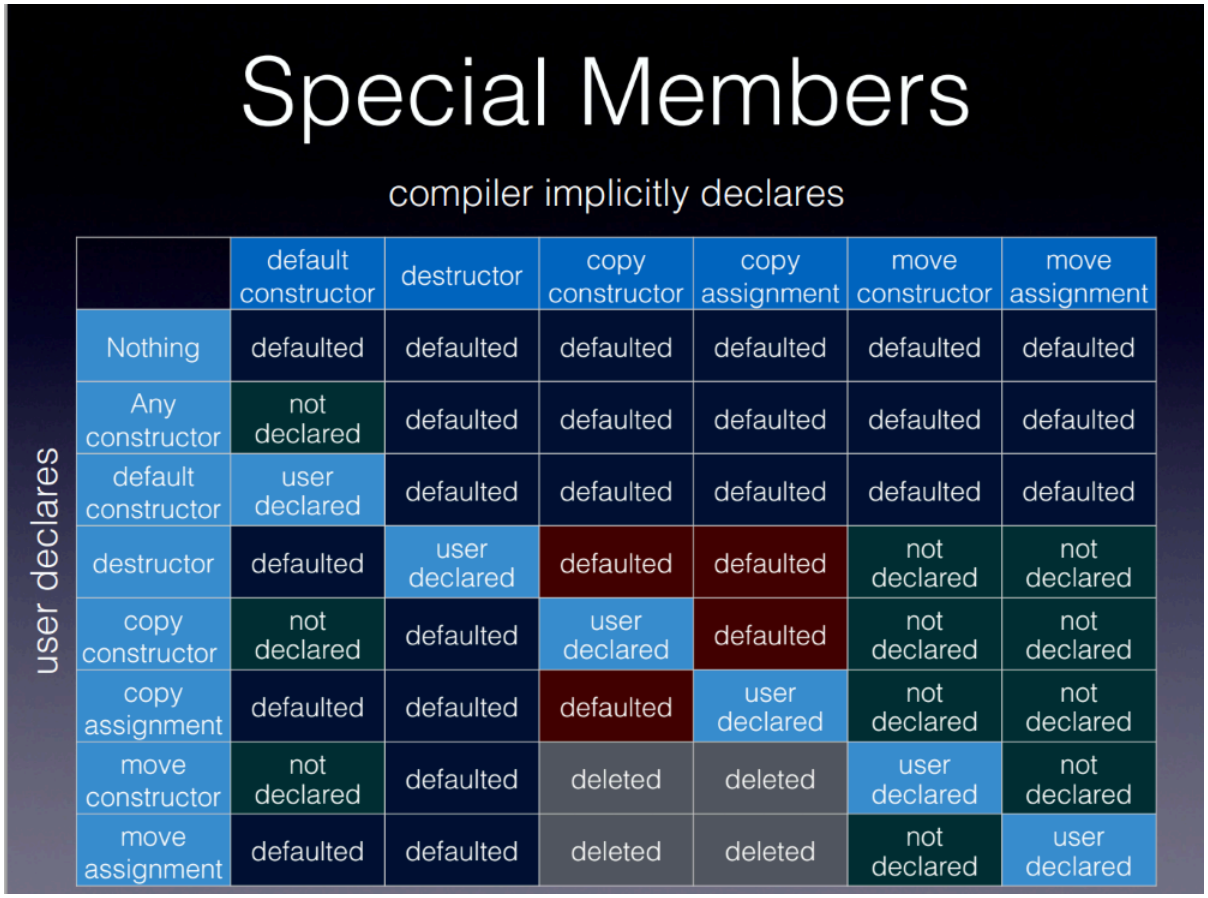
WHAT DOES IT REPRESENT?

- A simple value?
- A container?
- A resource?
- A semantic singleton?

What lifecyle does the type have?

How to implement constructors? Destructor?

CLASS LIFECYCLE



user declares	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

Everything You Ever Wanted to Know About Move Semantics
Howard Hinnant, Accu 2014

#1: SIMPLE VALUE TYPES

Definition: Behave like an `int`.

SIMPLE VALUE TYPES

"Fundamentals of Generic Programming"

A. Stepanov

https://stepanovpapers.com/DeSt98.pdf

"Regular, Revisited"

V. Ciura

https://www.youtube.com/watch?v=PFI_rpboj8U

SIMPLE VALUE TYPES

- default-constructible
- copyable
- movable
- assignable
- comparable

SIMPLE VALUE TYPES

- default-constructible
- copyable
- movable
- assignable
- comparable

This is a **regular type**.

SIMPLE VALUE TYPES

- Built-in types are regular
- STL types are regular, e.g., `string` or `vector`

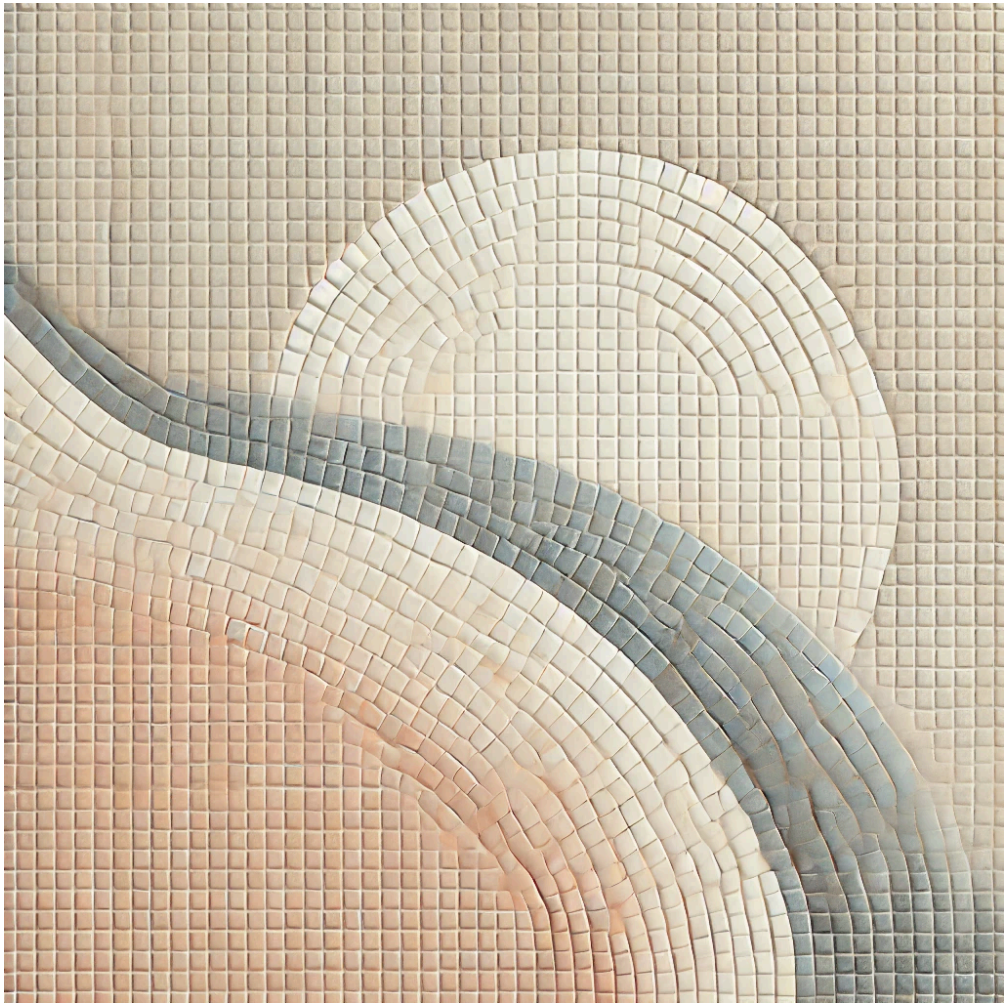
SIMPLE VALUE TYPES

C.11: Make concrete types regular

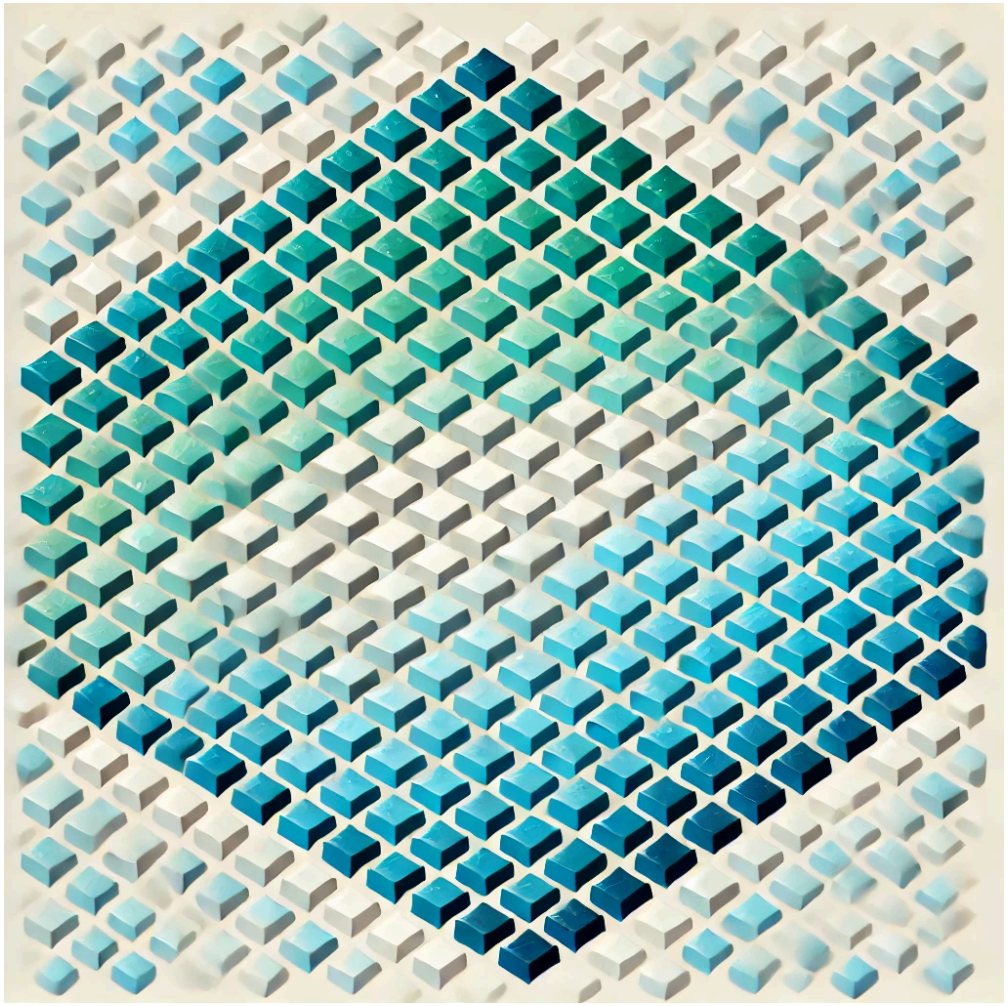
Regular types are easier to understand and reason about than types that are not regular (irregularities requires extra effort to understand and use).

C++ Core Guidelines

SIMPLICITY



CONSISTENCY



SIMPLE VALUE TYPES IN C++

SIMPLE VALUE TYPES IN C++

```
1 struct Contact {
2   string name = "Sebastian";
3   optional<int> age;
4
5   vector<string> emails;
6   string address;
7   variant<int, string> zipcode;
8 };
9 static_assert(semiregular<Contact>);
```

SIMPLE VALUE TYPES IN C++

```
1 struct Contact {
2   string name = "Sebastian";
3   optional<int> age;
4
5   vector<string> emails;
6   string address;
7   variant<int, string> zipcode;
8 };
9 static_assert(semiregular<Contact>);
```

This is the Rule of Zero.

Also see @foonathan's blog.

SIMPLE VALUE TYPES IN C++

```cpp
// https://godbolt.org/z/MnKjEKEEz
struct Address {
  string str;
};
struct Contact {
  string name = "Sebastian";
  optional<int> age;

  vector<string> emails;
  Address address;
  variant<int, string> zipcode;
};
static_assert(semiregular<Contact>);
```

This is the Rule of Zero.

Also see @foonathan's blog.

#2: CONTAINERS

```
 1 struct container {
 2   container() = default;
 3
 4   container(container const& other) noexcept;
 5   container& operator=(container const& other) noexcept;
 6
 7   container(container&& other) noexcept;
 8   container& operator=(container&& other) noexcept;
 9   ~container();
10 };
```

This is the Rule of Five/Six.

Again see @foonathan's blog.

The result is also a semiregular type.

#3: RESOURCE WRAPPERS

Holding a file, a database connection, etc.

#3: RESOURCE WRAPPERS

Holding a file, a database connection, etc.

These types are (almost always) move-only.

RESOURCE WRAPPERS

```
 1 struct file {
 2   FILE* fp = nullptr;
 3
 4   file() = default;
 5   file(char const* file);
 6   // defaulted copy ctor/assignment is wrong!
 7   ~file() {
 8     if(fp) fclose(fp);
 9   }
10 };
```

RESOURCE WRAPPERS

```
 1 struct file {
 2   // moved-from state
 3   FILE* fp = nullptr;
 4
 5   file() = default;
 6   file(char const* file);
 7
 8   file(file&& other) noexcept
 9   : fp{exchange(other.fp, nullptr)}
10   {} // other in moved-from state
11
12   ~file() {
13     if(fp) fclose(fp);
14   }
15 };
```

RESOURCE WRAPPERS

```
 1 struct file {
 2   // moved-from state
 3   FILE* fp = nullptr;
 4
 5   file() = default;
 6   file(char const* file);
 7
 8   file(file&& other) noexcept
 9   : fp{exchange(other.fp, nullptr)}
10   {} // other in moved-from state
11
12   file& operator=(file&& other) noexcept {
13     swap(fp, other.fp); // self-assignment!
14     return *this;
15   }
16
17   ~file() {
18     if(fp) fclose(fp);
19   }
20 };
```

RESOURCE WRAPPERS

```
1  struct noncopyable {
2    noncopyable(nonmovable&& other) = default;
3    noncopyable& operator=(noncopyable&& other) = default;
4  };
5
6  struct file : noncopyable {
7    // moved-from state
8    FILE* fp = nullptr;
9
10   file() = default;
11   file(char const* file);
12
13   file(file&& other) noexcept;
14   file& operator=(file&& other) noexcept;
15   ~file();
16 };
```

RESOURCE WRAPPERS

```
1  struct noncopyable {
2    noncopyable(nonmovable&& other) = default;
3    noncopyable& operator=(noncopyable&& other) = default;
4  };
5
6  struct file : noncopyable {
7    // moved-from state
8    FILE* fp = nullptr;
9
10   file() = default;
11   file(char const* file);
12
13   file(file&& other) noexcept;
14   file& operator=(file&& other) noexcept;
15   ~file();
16 };
```

New in C++23: `std::move_only_function`

RESOURCE WRAPPERS

```
1 file f; // move-only type
2 std::function<void ()> func =
3   [f = std::move(f)]() {
4     // a callback capturing a move only variable
5   };
```

Won't compile.

RESOURCE WRAPPERS

```
1 file f; // move-only type
2 std::move_only_function<void ()> func =
3   [f = std::move(f)]() {
4     // a callback capturing a move only variable
5   };
```

This will.

#4: SINGLETONS

```
1 struct nonmovable {
2   nonmovable(nonmovable const& other) = delete;
3   nonmovable& operator=(nonmovable const& other) = delete
4 };
```

e.g. mutexes, global singletons, some RAII wrappers.

No generated move either.

CLASS LIFECYCLE

- Simple values: Rule of Zero
- Containers: Rule of Five/Six
- Both should be at least semiregular
- Resource Wrappers: `noncopyable`
- Singleton objects: `nonmovable`

WE CANNOT CHECK IF TYPES ARE REALLY MOVED

- `std::semiregular` is lying
- `std::is_nothrow_move_constructible` is too

VALUE TYPES IN C++

```
1  // https://godbolt.org/z/Gvrnfcnvn
2  struct Address {
3    string str;
4    Address() = default;
5    Address(Address const& a) noexcept : str(a.str) {
6      println("Copied!"); // C++23
7    }
8    // move is disabled, will fallback to copy
9  };
10
11 static_assert(semiregular<Address>);
12 static_assert(is_nothrow_move_constructible<Address>);
13
14 Address a;
15 Address b = std::move(a);
```

VALUE TYPES IN C++

```cpp
// https://godbolt.org/z/Gvrnfcnvn
struct Address {
  string str;
  Address() = default;
  Address(Address const& a) noexcept : str(a.str) {
    println("Copied!"); // C++23
  }
  // move is disabled, will fallback to copy
};

static_assert(semiregular<Address>);
static_assert(is_nothrow_move_constructible<Address>);

Address a;
Address b = std::move(a);
```

⚠️ `Address` will be copied, not moved.

OVERVIEW

1. Defining regular types
2. Compile-time evaluation
3. Compile-time allocation
4. Making types formattable
5. Deducing **this**

VALUE TYPES IN C++

```
 1 // https://godbolt.org/z/ov5daroT5
 2 struct Address {
 3   string str;
 4 };
 5 struct Contact {
 6   string name = "Sebastian";
 7   optional<int> age;
 8
 9   vector<string> emails;
10   Address address;
11   variant<int, string> zipcode;
12 };
13 static_assert(semiregular<Contact>);
```

VALUE TYPES IN C++

```
 1 // https://godbolt.org/z/ov5daroT5
 2 struct Address {
 3   string str;
 4 };
 5 struct Contact {
 6   string name = "Sebastian";
 7   optional<int> age;
 8
 9   vector<string> emails;
10   Address address;
11   variant<int, string> zipcode;
12 };
13 static_assert(semiregular<Contact>);
```

VALUE TYPES IN C++

```cpp
// https://godbolt.org/z/ov5daroT5
struct Address {
  string str;
  weak_ordering operator<=>(Address const&) const&
    = default;
};
struct Contact {
  string name = "Sebastian";
  optional<int> age;

  vector<string> emails;
  Address address;
  variant<int, string> zipcode;
  weak_ordering operator<=>(Contact const&) const&
    = default;
};
static_assert(regular<Contact>);
```

DEFINING EQUALITY

- Can be difficult
- Value types may contain references, pointers or handles
- Need to be compared by value, not by reference
- (This may make them non-copyable and thus non-regular)
- Only implement comparisons if you are sure of semantics
- http://stepanovpapers.com/DeSt98.pdf

DEFINING EQUALITY (BADLY)

```
 1 // https://godbolt.org/z/x3686fqMW
 2 struct T {
 3   int data;
 4   unique_ptr<T> sibling; // may be nullptr
 5
 6   friend weak_ordering operator<=>
 7     (T const& lhs, T const& rhs) noexcept
 8   {
 9     if(lhs.sibling && rhs.sibling) {
10       auto const o = *lhs.sibling<=>*rhs.sibling;
11       if(weak_ordering::equivalent!=o) return o;
12     }
13     return lhs.data <=> rhs.data;
14   }
15 };
16 static_assert(totally_ordered<T>);
```

DEFINING EQUALITY (BADLY)

```
1  // https://godbolt.org/z/x3686fqMW
2  struct T {
3    int data;
4    unique_ptr<T> sibling;
5  };
6  static_assert(totally_ordered<T>);
7
8  int main() {
9    auto A = T{3, nullptr};
10   auto B = T{0, make_unique<T>(5, nullptr)};
11   auto C = T{5, make_unique<T>(3, nullptr)};
12
13   // 🚨 B < A < C < B 🚨
14   assert(B<A);
15   assert(A<C);
16   assert(C<B);
17   return 0;
18 }
```

DEFINING EQUALITY (CORRECTLY)

All types T

DEFINING EQUALITY (CORRECTLY)

All types T

sibling

!sibling

DEFINING EQUALITY (CORRECTLY)

All types T

sibling

!sibling

by *sibling

by data

DEFINING EQUALITY (CORRECTLY)

All types T

sibling

!sibling

by *sibling

by data

by data

DEFINING EQUALITY (CORRECTLY)

DEFINING EQUALITY (CORRECTLY)

```
1  // https://godbolt.org/z/GvTTKn8Pc
2  friend weak_ordering operator<=>
3    (T const& lhs, T const& rhs) noexcept
4  {
5    if(auto const o = !!lhs.sibling<=>!!rhs.sibling;
6      weak_ordering::equivalent!=o
7    ) {
8      return o;
9    }
10   assert(!!lhs.sibling==!!rhs.sibling);
11
12   if(lhs.sibling) {
13     if(auto const o = *lhs.sibling<=>*rhs.sibling;
14       weak_ordering::equivalent!=o
15     ) {
16       return o;
17     }
18   }
19   return lhs.data <=> rhs.data;
20 }
```

BACK TO VALUES!

```
 1 // https://godbolt.org/z/Edo9Tb6j3
 2 struct Address {
 3   // ...
 4 };
 5 struct Contact {
 6   string name = "Sebastian";
 7   optional<int> age;
 8   vector<string> emails;
 9   Address address;
10   weak_ordering operator<=>(Contact const&) const&
11     = default;
12 };
```

```cpp
// https://godbolt.org/z/Edo9Tb6j3
struct Address {
  // ...
};
struct Contact {
  string name = "Sebastian";
  optional<int> age;
  vector<string> emails;
  Address address;
  weak_ordering operator<=>(Contact const&) const&
    = default;
};

constexpr Contact c;
constexpr auto c2 = c;
constexpr auto b = c==c2;
constexpr auto n = std::ranges::fold_left(
  c2.emails,
  0,
  [](int i, auto const& s) { return i+size(s); }
); // Since C++23 too!
```

```cpp
// https://godbolt.org/z/Edo9Tb6j3
struct Address {
  // ...
};
struct Contact {
  string name = "Sebastian";
  optional<int> age;
  vector<string> emails;
  Address address;
  weak_ordering operator<=>(Contact const&) const&
    = default;
};

constexpr Contact c;
constexpr auto c2 = c;
constexpr auto b = c==c2;
constexpr auto n = *c2.age;
```

COMPILE-TIME UB CHECKS

```
<source>:32:16: error: constexpr variable 'n' must be
initialized by a constant expression
   32 | constexpr auto n = *c2.age;
      |                ^   ~~~~~~~
<source>:32:20: note: read of member '__val_' of union
with active member '__null_state_' is not allowed in a
constant expression
   32 | constexpr auto n = *c2.age;
      |                    ^
1 error generated.
Compiler returned: 1
```

CONSTEXPR

- Make everything `constexpr` as a habit
- Including functions if possible
- It does not cost anything
- Your future self will thank you
- Compile-time evaluation means no UB, no lifetime issues

CONSTEXPR, CONSTINIT, CONST

	const	guaranteed compile-time init	constexpr dtor

CONSTEXPR, CONSTINIT, CONST

	const	guaranteed compile-time init	constexpr dtor
`const`	x		

CONSTEXPR, CONSTINIT, CONST

	const	guaranteed compile-time init	constexpr dtor
`const`	x		
`constinit`		x	

CONSTEXPR, CONSTINIT, CONST

	const	guaranteed compile-time init	constexpr dtor
`const`	x		
`constinit`		x	
`constinit const`	x	x	

CONSTEXPR, CONSTINIT, CONST

	const	guaranteed compile-time init	constexpr dtor
`const`	x		
`constinit`		x	
`constinit const`	x	x	
`constexpr`	x	x	x

CONSTEXPR, CONSTINIT, CONST

	const	guaranteed compile-time init	constexpr dtor
`const`	x		
`constinit`		x	
`constinit const`	x	x	
`constexpr`	x	x	x

(for variables with static storage duration)

https://godbolt.org/z/MThq1b1sW

COMPILE-TIME ASSERTS

```
 1 // https://godbolt.org/z/Ps19Ge4os
 2 struct T {
 3   string str = "Hello";
 4   weak_ordering operator<=>(T const&) const&
 5       = default;
 6 };
 7
 8 constexpr T t;
 9 constexpr T t2{ .str = "World" };
10
11 static_assert(t==t2);
```

COMPILE-TIME ASSERTS

```
<source>:24:16: error: static assertion failed
   24 | static_assert(t==t2);
      |               ~^~~~
Compiler returned: 1
```

COMPILE-TIME ASSERTS

```
1  // https://godbolt.org/z/Ps19Ge4os
2  struct T {
3    string str = "Hello";
4    weak_ordering operator<=>(T const&) const&
5        = default;
6  };
7
8  constexpr T t;
9  constexpr T t2{ .str = "World" };
10
11 template<typename ...Args>
12 constexpr auto PoorStdFormat(Args const& ...args) {
13   return (args + ... + string(""));
14 }
15
16 static_assert(t==t2, PoorStdFormat(t.str,"!=",t2.str));
```

COMPILE-TIME ASSERTS

```
<source>:24:16: error: static assertion failed:
                  Hello!=World
   24 | static_assert(t==t2, Message(t.str, "!=", t2.str));
      |               ~^~~~
Compiler returned: 1
```

COMPILE-TIME ASSERTS

Constant expression `msg` satisfying all following conditions:

- `msg.size()` is implicitly convertible to `std::size_t`.
- `msg.data()` is implicitly convertible to `const char*`.

https://en.cppreference.com/w/cpp/language/static_assert

COMPILE-TIME ALLOCATION

```
 1 // https://godbolt.org/z/s7bbarcPn
 2 struct S {
 3   int i = 0;
 4 };
 5
 6 struct T {
 7   unique_ptr<S> ptr; // constexpr in C++23
 8 };
 9
10 constexpr T t; // this is fine
11 constexpr auto n = t.ptr->i; // this isn't
```

COMPILE-TIME ALLOCATION

```
 1 // https://godbolt.org/z/s7bbarcPn
 2 struct S {
 3   int i = 0;
 4 };
 5
 6 struct T {
 7   unique_ptr<S> ptr; // constexpr in C++23
 8 };
 9
10 constexpr T t; // this is fine
11 constexpr auto n = t.ptr->i; // this isn't
```

<source>:21:27: error: dereferencing a null pointer

COMPILE-TIME ALLOCATION

```
1  // https://godbolt.org/z/z1TvxnoPW
2  struct S {
3    int i = 0;
4  };
5
6  struct T {
7    unique_ptr<S> ptr = make_unique<S>();
8  };
9
10 constexpr T t;
11 constexpr auto n = t.ptr->i;
```

COMPILE-TIME ALLOCATION

```
1  // https://godbolt.org/z/z1TvxnoPW
2  struct S {
3    int i = 0;
4  };
5
6  struct T {
7    unique_ptr<S> ptr = make_unique<S>();
8  };
9
10 constexpr T t;
11 constexpr auto n = t.ptr->i;
```

```
<source>:21:27: error: 'T()' is not a constant
  expression because it refers to a result of
                'operator new'
```

COMPILE-TIME ALLOCATION

```
 1 // https://godbolt.org/z/z1TvxnoPW
 2 struct S {
 3   int i = 0;
 4 };
 5
 6 struct T {
 7   unique_ptr<S> ptr = make_unique<S>();
 8 };
 9
10 constexpr auto n = T{}.ptr->i;
```

COMPILE-TIME ALLOCATION

```
1  // https://godbolt.org/z/Gc3nnvh81
2  struct S {
3    int i = 0;
4  };
5
6  struct T {
7    unique_ptr<S> ptr = make_unique<S>();
8  };
9
10 constexpr int Calculate() {
11   T t;
12   // do more stuff with t
13   return t.ptr->i;
14 }
15 constexpr auto n = Calculate();
```

WHAT ELSE HAVE WE GOT?

```
1  // https://godbolt.org/z/16z3Gnc9T
2  struct Address {
3    // ...
4  };
5  struct Contact {
6    string name = "Sebastian";
7    optional<int> age;
8    vector<string> emails;
9    Address address;
10   weak_ordering operator<=>(Contact const&) const&
11     = default;
12 };
13 static_assert(regular<Contact>);
14 constexpr Contact c;
```

WHAT ELSE HAVE WE GOT?

```
1  // https://godbolt.org/z/16z3Gnc9T
2  struct Address {
3    // ...
4  };
5  struct Contact {
6    string name = "Sebastian";
7    optional<int> age;
8    vector<string> emails;
9    Address address;
10   weak_ordering operator<=>(Contact const&) const&
11     = default;
12 };
13 static_assert(regular<Contact>);
14 constexpr Contact c;
```

WHAT ELSE HAVE WE GOT?

```
1  // https://godbolt.org/z/16z3Gnc9T
2  struct Address {
3    // ...
4  };
5  struct Contact {
6    string name = "Sebastian";
7    optional<int> age;
8    vector<string> emails;
9    Address address;
10   weak_ordering operator<=>(Contact const&) const&
11     = default;
12 };
13 static_assert(regular<Contact>);
14 constexpr Contact c;
15
16 void main() {
17   println("{}", c);
18 }
```

WHAT ELSE HAVE WE GOT?

```
1  // https://godbolt.org/z/16z3Gnc9T
2  struct Address {
3    // ...
4  };
5  struct Contact {
6    string name = "Sebastian";
7    optional<int> age;
8    vector<string> emails;
9    Address address;
10   weak_ordering operator<=>(Contact const&) const&
11     = default;
12 };
13 static_assert(regular<Contact>);
14 static_assert(formattable<Contact>); // C++23
15 constexpr Contact c;
16
17 void main() {
18   println("{}", c);
19 }
```

FORMATTABLE VALUES

```
 1 // https://godbolt.org/z/16z3Gnc9T
 2 struct Address {
 3   // ...
 4 };
 5 struct Contact {
 6   // ...
 7 };
 8 static_assert(formattable<Contact>); // C++23
 9 constexpr Contact c;
10
11 void main() {
12   println("{}", c);
13 }
```

FORMATTABLE VALUES

```cpp
// https://godbolt.org/z/16z3Gnc9T
template<>
struct formatter<Contact, char>
{
  template<class ParseContext>
  constexpr ParseContext::iterator parse(
    ParseContext& ctx
  ) {
    return ctx.begin();
  }

  template<class FmtContext>
  FmtContext::iterator format(
    Contact c, FmtContext& ctx
  ) const {
    return format_to(ctx.out(),
      "\"{}\" ({})\n{}",
      c.name, c.age.value_or(-1), c.emails
    );
  }
};
```

FORMATTABLE VALUES

```cpp
// https://godbolt.org/z/16z3Gnc9T
template<>
struct formatter<Contact, char>
{
  template<class ParseContext>
  constexpr ParseContext::iterator parse(
    ParseContext& ctx
  ) {
    return ctx.begin();
  }

  template<class FmtContext>
  FmtContext::iterator format(
    Contact c, FmtContext& ctx
  ) const {
    return format_to(ctx.out(),
      "\"{}\" ({})\n{}",
      c.name, c.age.value_or(-1), c.emails
    );
  }
};
```

FORMATTABLE VALUES

```cpp
// https://godbolt.org/z/16z3Gnc9T
template<>
struct formatter<Contact, char>
{
  template<class ParseContext>
  constexpr ParseContext::iterator parse(
    ParseContext& ctx
  ) {
    return ctx.begin();
  }

  template<class FmtContext>
  FmtContext::iterator format(
    Contact c, FmtContext& ctx
  ) const {
    return format_to(ctx.out(),
      "\"{}\" ({})\n{}",
      c.name, c.age.value_or(-1), c.emails
    );
  }
};
```

FORMATTABLE VALUES

```cpp
// https://godbolt.org/z/16z3Gnc9T
template<>
struct formatter<Contact, char>
{
  template<class ParseContext>
  constexpr ParseContext::iterator parse(
    ParseContext& ctx
  ) {
    return ctx.begin();
  }

  template<class FmtContext>
  FmtContext::iterator format(
    Contact c, FmtContext& ctx
  ) const {
    return format_to(ctx.out(),
      "\"{}\" ({})\n{}",
      c.name, c.age.value_or(-1), c.emails
    );
  }
};
```

FORMATTABLE VALUES

```cpp
// https://godbolt.org/z/16z3Gnc9T
template<>
struct formatter<Contact, char>
{
  template<class ParseContext>
  constexpr ParseContext::iterator parse(
    ParseContext& ctx
  ) {
    return ctx.begin();
  }
  // ...
};
```

FORMATTABLE VALUES

```
1  // https://godbolt.org/z/vr3n9sYMf
2  bool be_private = false;
3
4  template<class ParseContext>
5  constexpr ParseContext::iterator parse(
6    ParseContext& ctx
7  ) {
8    auto it = ctx.begin();
9    if('p'==*it) {
10     be_private = true;
11     ++it;
12   }
13
14   if(it!=ctx.end() && '}'!=*it) {
15     throw format_error("Unsupported format specifier");
16   }
17   return it;
18 }
```

FORMATTABLE VALUES

```
 1 // https://godbolt.org/z/vr3n9sYMf
 2 bool be_private = false;
 3
 4 template<class FmtContext>
 5 FmtContext::iterator format(
 6   Contact c, FmtContext& ctx) const {
 7
 8   return be_private
 9   ? format_to(ctx.out(), "\"{}\"", c.name)
10   : format_to(ctx.out(),
11     "\"{}\" ({})\n{}", c.name, c.age.value_or(-1), c.ema
12   );
13 }
```

FORMATTABLE VALUES

```cpp
// https://godbolt.org/z/vr3n9sYMf
bool be_private = false;

template<class FmtContext>
FmtContext::iterator format(
  Contact c, FmtContext& ctx) const {

  string str = "\"{}\"";
  if(!be_private) str+=" ({})\n{}";

  return format_to(
    ctx.out(),
    runtime_format(str), // C++26
    c.name, c.age.value_or(-1), c.emails
  );
}
```

FORMATTABLE VALUES

`std::basic_format_context`

https://en.cpprference.com/w/cpp/utility/format/basic_format_context

`std::basic_format_parse_context`

https://en.cpprference.com/w/cpp/utility/format/basic_format_parse_co

`std::pair/std::tuple` formatter

`std::range` formatter

`std::thread_id` formatter

`std::stacktrace_entry` formatter

FORMATTABLE VALUES

```
1  // https://godbolt.org/z/16z3Gnc9T
2  struct Address {
3    // ...
4  };
5  struct Contact {
6    string name = "Sebastian";
7    optional<int> age;
8    vector<string> emails;
9    Address address;
10   weak_ordering operator<=>(Contact const&) const&
11     = default;
12 };
13 static_assert(regular<Contact>);
14 static_assert(formattable<Contact>); // C++23
15 constexpr Contact c;
16
17 void main() {
18   println("{}", c);
19 }
```

ACCESSING VALUES

```
1 // https://godbolt.org/z/eEq7WvKnv
2 struct Contact {
3   vector<string> emails;
4 };
```

ACCESSING VALUES

```
 1 // https://godbolt.org/z/eEq7WvKnv
 2 struct Contact {
 3 private:
 4   vector<string> emails;
 5
 6 public:
 7   vector<string> const& Get() const {
 8     // maybe assert something too? log something?
 9     return emails;
10   }
11 };
```

ACCESSING VALUES

```
1  // https://godbolt.org/z/eEq7WvKnv
2  struct Contact {
3  private:
4    vector<string> emails;
5
6  public:
7    vector<string> const& Get() const {
8      return emails;
9    }
10 };
11
12 Contact MakeContact();
13
14 int main() {
15   // Unnecessary copy
16   auto const vec = MakeContact().Get();
17   // 🧨 dangling reference
18   auto const& vec2 = MakeContact().Get();
19 }
```

ACCESSING VALUES

```
1  // https://godbolt.org/z/eEq7WvKnv
2  struct Contact {
3  private:
4    vector<string> emails;
5
6  public:
7    vector<string> const& Get() const {
8      return emails;
9    }
10 };
11 using C = Contact;
12
13 // Member access preserves rvalue-ness
14 // declval<C&>().emails -> vector<string>&
15 // declval<C const&>().emails -> vector<string> const&
16 // declval<C>().emails -> vector<string>&&
17 // declval<C&&>().emails -> vector<string>&&
18 // declval<C const>().emails -> vector<string> const&&
```

ACCESSING VALUES

```
1  // https://godbolt.org/z/eEq7WvKnv
2  struct Contact {
3  private:
4    vector<string> emails;
5
6  public:
7    vector<string> const& Get() const {
8      return emails;
9    }
10 };
11 using C = Contact;
12
13 // const functions bind to all values
14 // declval<C&>().Get() -> vector<string> const&
15 // declval<C const&>().Get() -> vector<string> const&
16 // declval<C>().Get() -> vector<string> const&
17 // declval<C&&>().Get() -> vector<string> const&
18 // declval<C const>().Get() -> vector<string> const&
```

ACCESSING VALUES

```
1  // https://godbolt.org/z/eEq7WvKnv
2  struct Contact {
3  private:
4    vector<string> emails;
5
6  public:
7    // ref-qualifiers!
8    vector<string> const& Get() const& {
9      return emails;
10   }
11 };
12 using C = Contact;
13
14 // const& functions bind to all values
15 // declval<C&>().Get() -> vector<string> const&
16 // declval<C const&>().Get() -> vector<string> const&
17 // declval<C>().Get() -> vector<string> const&
18 // declval<C&&>().Get() -> vector<string> const&
19 // declval<C const>().Get() -> vector<string> const&
```

ACCESSING VALUES

```
1  // https://godbolt.org/z/eEq7WvKnv
2  struct Contact {
3  private:
4    vector<string> emails;
5
6  public:
7    // ref-qualifiers!
8    vector<string>& Get() &;
9    vector<string> const& Get() const&;
10   vector<string>&& Get() &&;
11   vector<string> const&& Get() const&&;
12 };
13 using C = Contact;
14
15 // declval<C&>().Get() -> vector<string>&
16 // declval<C const&>().Get() -> vector<string> const&
17 // declval<C>().Get() -> vector<string>&&
18 // declval<C&&>().Get() -> vector<string>&&
19 // declval<C const>().Get() -> vector<string> const&&
```

ACCESSING VALUES

```cpp
// https://godbolt.org/z/eEq7WvKnv
struct Contact {
private:
  vector<string> emails;

public:
  // explicit object parameter or 'deducing this'
  template<typename Self>
  [[nodiscard]] decltype(auto) Get(this Self&& s) {
    return (std::forward<Self&&>(s).emails);
  }
};
using C = Contact;

// declval<C&>().Get() -> vector<string>&
// declval<C const&>().Get() -> vector<string> const&
// declval<C>().Get() -> vector<string>&&
// declval<C&&>().Get() -> vector<string>&&
// declval<C const>().Get() -> vector<string> const&&
```

ACCESSING VALUES

```cpp
// https://godbolt.org/z/eEq7WvKnv
struct Contact {
private:
  vector<string> emails;

public:
  template<typename Self>
  [[nodiscard]] decltype(auto) Get(this Self&& s) {
    return (std::forward<Self&&>(s).emails);
  }
};

Contact MakeContact();

int main() {
  // Now we get a move!
  auto const vec = MakeContact().Get();
}
```

ACCESSING VALUES

```cpp
// https://godbolt.org/z/eEq7WvKnv
struct Contact {
private:
  vector<string> emails;

public:
  template<typename Self>
  [[nodiscard]] decltype(auto) Get(this Self&& s) {
    return (std::forward<Self&&>(s).emails);
  }
};

Contact MakeContact();

int main() {
  // Lifetime extension
  auto const& c = MakeContact();
  // No lifetime extension: Compiles and dangles!
  auto const& vec = MakeContact().Get();
}
```

ACCESSING VALUES

```cpp
// https://godbolt.org/z/eEq7WvKnv
struct Contact {
private:
  vector<string> emails;

public:
  template<typename Self>
  [[nodiscard]] decltype(auto) Get(this Self&& s) {
    return (std::forward<Self&&>(s).emails);
  }
};

Contact MakeContact();

int main() {
  // https://github.com/think-cell/think-cell-library
  // vec is a value for rvalues
  // and a reference for lvalues
  tc_auto_cref(vec, MakeContact().Get());
}
```

ACCESSING VALUES

```
 1 // https://godbolt.org/z/eEq7WvKnv
 2 struct Contact {
 3 private:
 4   vector<string> emails;
 5
 6 public:
 7   vector<string> const& Get() const& {
 8     return emails;
 9   }
10   auto Get() && = delete;
11   auto Get() const&& = delete;
12 };
13
14 Contact MakeContact();
15
16 int main() {
17 }
```

SUMMARY

- Follow simplest lifecycle best-practices
- Make value types regular
- Make resource wrappers move-only types
- Make everything constexpr or constinit
- Specialize `std::formatter`
- Write accessors by deducing this

THANK YOU

stheophil@think-cell.com