Safe C++

Document #: D3390
Date: 2024-09-11
Project: Programming Language C++
Audience: SG23
Reply-to: Sean Baxter
<seanbax.circle@gmail.com>
Christian Mazakas
<christian.mazakas@gmail.com>

Contents

1 Introduction

1.1 The call for memory safety

Over the past two years, the United States Government has been issuing warnings about memory-unsafe programming languages with increasing urgency. Much of the country's critical infrastructure relies on software written in C and C++, languages which are very memory *unsafe*, leaving these systems more vulnerable to exploits by adversaries.

— Nov. 10, 2022 - **NSA Releases Guidance on How to Protect Against Software Memory Safety Issues**[nsa-guidance]

— Sep. 20, 2023 - **The Urgent Need for Memory Safety in Software Products**[cisa-urgent]

— Dec. 6, 2023 - **CISA Releases Joint Guide for Software Manufacturers: The Case for Memory Safe Roadmaps**[cisa-roadmaps]

— Feb. 26, 2024 - **Future Software Should Be Memory Safe**[white-house]

— May 7, 2024 - **National Cybersecurity Strategy Implementation Plan**[ncsi-plan]

The government papers are backed by industry research. Microsoft's bug telemetry reveals that 70% of its vulnerabilities would be stopped by memory safe languages.[ms-vulnerabilities] Google's research finds 68% of 0day exploits are related to memory corruption.[google-0day]

— Mar. 4, 2024 - **Secure by Design: Google's Perspective on Memory Safety**[secure-by-design]

Security professionals urge projects to migrate away from C++ and adopt memory safe languages. But the scale of the problem is daunting. C++ powers software that has generated trillions of dollars of value. There are many veteran C++ programmers and lots of C++ code. Given how wide-spread C++ is, what can industry really do to improve software quality and reduce vulnerabilities? What are the options for introducing new memory safe code into existing projects and hardening software that already exists?

There's only one popular systems level/non-garbage collected language that provides rigorous memory safety. That's the Rust language.[rust-language] Although they play in the same space, C++ and Rust have different designs with limited interop capability, making incremental migration from C++ to Rust a painstaking process.

Rust lacks function overloading, templates, inheritance and exceptions. C++ lacks traits, relocation and borrow checking. These discrepancies are responsible for an impedence mismatch when interfacing the two languages. Most code generators for inter-language bindings aren't able to represent features of one language in terms of the features of another. They typically identify a small number of special vocabulary types,[vocabulary-types] which have first-class ergonomics, and limit functionality of other constructs.

The foreignness of Rust for career C++ developers combined with the the friction of interop tools makes hardening C++ applications by rewriting critical sections in Rust difficult. Why is there no in-language solution to memory safety? *Why not a Safe C++?*

1.2 Extend C++ for safety

The goal of this proposal is to advance a superset of C++ with a *rigorously safe subset*. Begin a new project, or take an existing one, and start writing safe code in C++. Code in the safe context exhibits the same strong safety guarantees as code written in Rust.

Rigorous safety is a carrot-and-stick approach. The stick comes first. The stick is what security researchers and regulators care about. Safe C++ developers are prohibited from writing operations that may result in lifetime safety, type safety or thread safety undefined behaviors. Sometimes these operations are prohibited by the compiler frontend, as is the case with pointer arithmetic. Sometimes the operations are prohibited by static analysis in the compiler's middle-end; that stops use of uninitialized variables and use-after-free bugs, and it's the enabling technology of the *ownership and borrowing* safety model. The remainder of issues, like out-of-bounds array subscripts, are addressed with runtime panic and aborts.

The carrot is a suite of new capabilities which improve on the unsafe ones denied to users. The affine type system makes it easier to relocate objects without breaking type safety. Pattern matching, which is safe and expressive, interfaces with the extension's new choice types. Borrow checking,[borrow-checking] the most sophisticated part of the Safe C++, provides a new reference type that flags use-after-free and iterator invalidation defects at compile time.

What are the properties we're trying to deliver with Safe C++?

— A superset of C++ with a *safe subset*. Undefined behavior is prohibited from originating in the safe subset.
— The safe and unsafe parts of the language are clearly delineated. Users must explicitly leave the safe context to write unsafe operations.
— The safe subset must remain *useful*. If we get rid of a crucial unsafe technology, like unions and pointers, we should supply a safe alternative, like choice types and borrows. A safe toolchain is not useful if it's so inexpressive that you can't get your work done.

— The new system can't break existing code. If you point a Safe C++ compiler at existing C++ code, that code must compile normally. Users opt into the new safety mechanisms. Safe C++ is an extension of C++. It adds a robust safety model, but it's not a new language.

1.3 A safe program

iterator.cxx – (Compiler Explorer)

```
#feature on safety
#include <std2.h>

int main() safe {
  std2::vector<int> vec { 11, 15, 20 };

  for(int x : vec) {
    // Ill-formed. mutate of vec invalidates iterator in ranged-for.
    if(x % 2)
      mut vec.push_back(x);

    std2::println(x);
  }
}
```

```
$ circle iterator.cxx -I ../libsafecxx/single-header/
safety: during safety checking of int main() safe
  borrow checking: iterator.cxx:10:11
        mut vec.push_back(x);
            ^
  mutable borrow of vec between its shared borrow and its use
  loan created at iterator.cxx:7:15
    for(int x : vec) {
                ^
```

Consider this demonstration of Safe C++ that catches iterator invalidation, a kind of use-after-free bug. Let's break it down line by line:

Line 1: `#feature on safety` - Activate the safety-related keywords within this file. Other files in your translation unit are unaffected. This is how Safe C++ avoids breaking existing code–everything is opt-in, including the new keywords and syntax. The `[safety]` feature changes the object model for function definitions, enabling object relocation and deferred initialization. It lowers function definitions to mid-level intermediate representation (MIR),[mir] on which borrow checking is performed to flag use-after-free bugs on checked references.

Line 2: `#include <std2.h>` - Include the new safe containers and algorithms. Safety hardening is about reducing your exposure to unsafe APIs. The current Standard Library is full of unsafe APIs. The safe standard library in namespace `std2` will provide the same basic functionality, but with containers that are lifetime-aware and type safe.

Line 4: `int main() safe` - The new *safe-specifier* is part of a function's type, just like *noexcept-specifier*. `main`'s definition starts in a safe context. Unsafe operations such as pointer dereferences, which may raise undefined behavior, are disallowed. Rust's functions are safe by default. C++'s are unsafe by default. But that's now just a syntax difference. Once you enter a safe context in C++ by using the *safe-specifier*, you're backed by the same rigorous safety guarantees that Rust provides.

Line 5: `std2::vector<int> vec { 11, 15, 20 };` - List initialization of a memory-safe vector. This vector is aware of lifetime parameters, so borrow checking would extend to element types that have lifetimes. The vector's constructor doesn't use `std::initializer_list<int>`. That type is problematic for two reasons: first, users are given pointers into the argument data, and reading from pointers is unsafe; second, the

`std::initializer_list` *doesn't own* its data, making relocation impossible. For these reasons, Safe C++ introduces a `std2::initializer_list`, which supports our ownership object model.

Line 7: `for(int x : vec)` - Ranged-for on the vector. The standard mechanism returns a pair of iterators, which are pointers wrapped in classes. C++ iterators are unsafe. They come in begin and end pairs, and don't share common lifetime parameters, making borrow checking them impractical. The Safe C++ version uses slice iterators, which resemble Rust's `Iterator`.[rust-iterator] These safe iterators are implemented with lifetime parameters, making them robust against iterator invalidation defects.

Line 10: `mut vec.push_back(x);` - Push a value onto the vector. The `mut` token establishes a *mutable context* which enables standard conversions from lvalues to mutable borrows and references. When `[safety]` is enabled, *all mutations are explicit.* Explicit mutation lends precision when choosing between shared borrows and mutable borrows of an object. Rust doesn't feature function overloading, so it will bind whatever kind of reference it needs to a member function's object. C++, by contrast, has function overloading, so we'll need to be explicit in order to get the overload we want. Use `mut` to bind mutable borrows. Or don't use it and bind shared borrows.

If `main` checks out syntatically, its AST is lowered to MIR, where initialization and borrow checking takes place. The hidden `slice_iterator` that powers the ranged-for loop stays initialized over the duration of the loop. The `push_back` call *invalidates* that iterator, by mutating a place (the vector) that the iterator has a constraint on. When the value `x` is next loaded out of the iterator, the borrow checker raises an error: `mutable borrow of vec between its shared borrow and its use`. The borrow checker prevents Safe C++ from compiling a program that may exhibit undefined behavior. This analysis is done at compile time. It has no impact on your binary's size or execution speed.

This sample is only a few lines, but it introduces several new mechanisms and types. A comprehensive effort is needed to supply a superset of the language with a safe subset that has enough flexibility to remain expressive.

1.4 Memory safety as terms and conditions

Memory safe languages are predicated on a basic observation of programmer behavior: developers will try to use a library and only read documentation if their first few attempts don't seem to work. This is dangerous for software correctness, since seeming to work is not the same as working.

Many C++ functions have preconditions that you'd have to read the docs to understand. Violating preconditions, which is possible with benign-looking usage, could cause undefined behavior and open your software to attack. *Software safety and security should not be predicated on programmers following documentation.*

Here's the memory safety value proposition: language and library vendors make an extra effort to provide a robust environment so that users *don't have to read the docs.* No matter how they use the tooling, their actions will not raise undefined behavior and compromise their software to safety-related exploits. No system can guard against all misuse, and hastily written code may have plenty of logic bugs. But those logic bugs won't lead to memory-safety vulnerabilities.

Consider an old libc function, `std::isprint`,[isprint] that exhibits unsafe design. This function takes an `int` parameter. *But it's not valid to call `std::isprint` for all int arguments.* The preconditions state the function be called only with arguments between -1 and 255:

> Like all other functions from `<cctype>`, the behavior of `std::isprint` is undefined if the argument's value is neither representable as unsigned char nor equal to EOF. To use these functions safely with plain chars (or signed chars), the argument should first be converted to unsigned char. Similarly, they should not be directly used with standard algorithms when the iterator's value type is char or signed char. Instead, convert the value to unsigned char first.

It feels only right, in the year 2024, to pass Unicode code points to functions that are typed with `int` and deal with characters. But doing so may crash your application, or worse. While the mistake is the caller's for not reading the documentation and obeying the preconditions, it's fair to blame the design of the API. The safe context provided by memory safe languages prevents usage or authoring of functions like `std::isprint` which exhibit undefined behavior when called with invalid arguments.

Rust's approach to safety[safe-unsafe-meaning] centers on defining responsibility for enforcing preconditions. In a safe context, the user can call safe functions without compromising program soundness. Failure to read the docs may risk correctness, but it won't risk undefined behavior. When the user wants to call an unsafe function from a safe context, they *explicitly take responsibility* for sound usage of that unsafe function. The user writes the `unsafe` token as a kind of contract: the user has read the terms and conditions of the unsafe function and affirms that it's not being used in a way that violates its preconditions.

Who is to blame when undefined behavior is detected–the caller or the callee? Standard C++ does not address this. But Rust's safety model does: whoever typed out the `unsafe` token is to blame. Safe C++ adopts the same principle. Code is divided into unsafe and safe contexts. Unsafe operations may only occur in unsafe contexts. Dropping from a safe context to an unsafe context requires use of the `unsafe` keyword. This leaves an artifact that makes for easy audits: reviewers search for the `unsafe` keyword and focus their attention there first. Developers checking code into Rust's standard library are even required to write *safety comments*[safety-comments] before every unsafe block, indicating proper usage and explaining why it's sound.

Consider the design of a future `std2::isprint` function. If it's marked `safe`, it must be sound for all argument values. If it's called with an argument that is out of its supported range, it must fail in a deterministic way: it could return an error code, it could throw an exception or it could panic and abort. Inside the `std2::isprint` implementation, there's probably a lookup table with capabilities for each supported character. If the lookup table is accessed with a slice, an out-of-bounds access will implicitly generate a bounds check and panic and abort on failure. If the lookup table is accessed through a pointer, the implementer tests the subscript against the capacity of the lookup table and fails if it's out-of-range. Having enforced the preconditions, they write the `unsafe` keyword to drop to the unsafe context and fetch the data from the pointer. The `unsafe` keyword is the programmer's oath that the subsequent unsafe operations are sound.

In ISO C++, soundness bugs often occur because caller and callee don't know who should enforce preconditions, so neither of them do. In Safe C++, there's a convention backed up by the compiler, eliminating this confusion and improving software quality.

1.5 Categories of safety

It's instructive to break the memory safety problem down into four categories. Each of these is addressed with a different language technology.

1.5.1 Lifetime safety

How do we ensure that dangling references are never used? There are two mainstream lifetime safety technologies: garbage collection and borrow checking. Garbage collection is simple to implement and use, but moves object allocations to the heap, making it incompatible with manual memory manegement. It keeps objects initialized as long as there are live references to them, making it incompatible with C++'s RAII[raii] object model.

Borrow checking is an advanced form of live analysis. It keeps track of the *live references* at every point in the function, and errors when there's a *conflicting action* on a place associated with a live reference. For example, writing to, moving or dropping an object with a live shared borrow will raise a borrow check error. Pushing to a vector with a live iterator will raise an iterator invalidation error. This technology is compatible with manual memory management and RAII, making it a good fit for C++.

Borrow checking is a kind of local analysis. It avoids whole-program analysis by enforcing the *law of exclusivity*. Checked references (borrows) come in two flavors: mutable and shared, spelled `T^` and `const T^`, respectively. There can be one live mutable reference to a place, or any number of shared references to a place, but not both at once. Upholding this principle makes it easier to reason about your program. Since the law of exclusivity prohibits mutable aliasing, if a function is passed a mutable reference and some shared references, you can be certain that the function won't have side effects that, through the mutable reference, cause the invalidation of those shared references.

string_view.cxx – (Compiler Explorer)

```
#feature on safety
#include <std2.h>

using namespace std2;

int main() safe {
  // Populate the vector with views with a nice mixture of lifetimes.
  vector<string_view> views { };

  // string_view with /static lifetime.
  mut views.push_back("From a string literal");

  // string_view with outer scope lifetime.
  string s1("From string object 1");
  mut views.push_back(s1);

  {
    // string_view with inner scope lifetime.
    string s2("From string object 2");
    mut views.push_back(s2);

    // s2 goes out of scope. views now holds dangling pointers into
    // out-of-scope data.
  }

  // Print the strings. s2 already fell out of scope, so this should
  // be a borrowck violation. `views` now contains objects that hold
  // dangling pointers.
  println("Printing from the outer scope:");
  for(string_view sv : views)
    println(sv);
}
```

```
$ circle string_view.cxx -I ../libsafecxx/single-header/
safety: during safety checking of int main() safe
  borrow checking: string_view.cxx:30:24
    for(string_view sv : views)
                         ^
  use of views depends on expired loan
  drop of s2 between its shared borrow and its use
  s2 declared at string_view.cxx:19:12
      string s2("From string object 2");
             ^
  loan created at string_view.cxx:20:25
      mut views.push_back(s2);
                          ^
```

Borrow checking provides bullet-proof guarantees against lifetime safety defects when dealing with collections of references or views into objects with different scopes. Take a vector of `std2::string_view`. This string view is special. It's defined with a lifetime parameter that establishes a lifetime constraint on the string into which it points: that string must be in scope for all uses of the view.

Load the vector up with views into three different strings: a string constant with `/static` lifetime, a string in an outer scope and a string in an inner scope. Printing the contents of the vector from the outer scope raises a borrow checker error, because one of the function's lifetime constraints is violated: the vector depends on a loan

on `s2`, which is out-of-scope at the point of use.

Garbage collection solves this problem in a different way: `s1` and `s2` are stored on the GC-managed heap, and they're kept in scope as long as there are live references to them. That's an effective system, but it's not the right choice for C++, where deterministic destruction order is core to the language's design.

1.5.2 Type safety

> I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.
>
> – Tony Hoare[hoare]

The "billion-dollar mistake" is a type safety problem. Consider `std::unique_ptr`. It has two states: engaged and disengaged. The class presents member functions like `operator*` and `operator->` that are valid when the object is in the engaged state and *undefined* when the object is disengaged. `->` is the most important API for smart pointers. Calling it when the pointer is null? That's your billion-dollar mistake.

As Hoare observes, the problem comes from conflating two different things, a pointer to an object and an empty state, into the same type and giving them the same interface. Smart pointers should only hold valid pointers. Denying the null state eliminates undefined behavior.

We address the type safety problem by overhauling the object model. Safe C++ features a new kind of move: *relocation*, also called *destructive move*. The object model is called an *affine* or a *linear* type system. Unless explicitly initialized, objects start out *uninitialized*. They can't be used in this state. When you assign to an object, it becomes initialized. When you relocate from an object, it's value is moved and it's reset to uninitialized. If you relocate from an object inside control flow, it becomes *potentially uninitialized*, and its destructor is conditionally executed after reading a compiler-generated drop flag.

`std2::box` is our version of `unique_ptr`. It has no null state. There's no default constructor. Dereference it without risk of undefined behavior. If this design is so much safer, why doesn't C++ simply introduce its own fixed `unique_ptr` without a null state? Blame C++11 move semantics.

How do you move objects around in C++? Use `std::move` to select the move constructor. That moves data out of the old object, leaving it in a default state. For smart pointers, that's the null state. If `unique_ptr` didn't have a null state, it couldn't be moved in C++. This affine type system implements moves with relocation. That's type safe. Standard C++'s object model implements moves with move construction. That's unsafe.

box.cxx – (Compiler Explorer)

```
#feature on safety
#include <std2.h>

int main() safe {
  // p is uninitialized.
  std2::box<std2::string_view> p;

  // Error: p is uninitialized.
  println(*p);

  // p is definitely initialized.
  p = std2::box<std2::string_view>("Hello Safety");

  // Ok.
  println(*p);
```

```
  // p is moved into q. Now p is uninitialized again.
  auto q = rel p;

  // Error: p is uninitialized.
  println(*p);
}
```

```
$ circle box.cxx -I ../libsafecxx/single-header/
safety: during safety checking of int main() safe
  initialization analysis: box.cxx:9:12
    println(*p);
            ^
  cannot use uninitialized object p with type std2::box<std2::string_view>

  initialization analysis: box.cxx:21:12
    println(*p);
            ^
  cannot use uninitialized object p with type std2::box<std2::string_view>
```

The *rel-expression* names an object or subobject and relocates that into a new value. The old object becomes uninitialized. Using an uninitialized object generates a compiler error. Contrast the legacy object model with the relocation model: using a null `std::unique_ptr` results in runtime undefined behavior, but using an uninitialized `std2::box` raises a compile-time error.

We have to reimagine our standard library in the presence of relocation. Most kinds of resource handles include null states. These should all be replaced by safe versions to reduce exposure to unsafe APIs.

The compiler can only relocate local variables. How do we move objects that live on the heap, or for which we only have a pointer or reference? We need to use optional types.

The C++ Standard Library has an optional type, but it's not safe to use. The optional API is full of undefined behaviors: using `operator*` or `operator->` while the value is disengaged raises undefined behavior. `std::expected`, new to C++23, exhibits the same undefined behaviors for out-of-contract uses of its `operator*`, `operator->` and `error` APIs.

If we were to represent the null state by wrapping the safe `std2::box` in an `std::optional`, that would be just as unsafe as using `std::unique_ptr`. The `operator->` is unsafe either way. We need a new *sum type* that doesn't exhibit the union-like safety defects of `std::optional` and `std::expected`.

```
template<class T+>
choice optional
{
  default none,
  [[safety::unwrap]] some(T);

  template<class E>
  expected<T, E> ok_or(self, E e) noexcept safe {
    return match(self) -> expected<T, E> {
      .some(t) => .ok(rel t);
      .none    => .err(rel e);
    };
  }

  T expect(self, str msg) noexcept safe {
    return match(self) -> T {
      .some(t) => rel t;
```

```
      .none    => panic(msg);
    };
  }

  T unwrap(self) noexcept safe {
    return match(self) -> T {
      .some(t) => rel t;
      .none    => panic("{} is none".format(optional~string));
    };
  }
};
```

The new `std2::optional` is a *choice type*, a first-class discriminated union, accessible only with *pattern matching*. Pattern matching makes type safety violations on choice types impossible: we can't access the wrong state of the object.

match.cxx – (Compiler Explorer)

```
#feature on safety
#include <std2.h>

choice Value {
  i32(int),
  f32(float),
  f64(double),
  str(std2::string)
};

void print(Value val) safe {
  match(val) {
    // Type safety bugs are impossible inside the pattern match.
    // The alternatives listed must be exhaustive.
    .i32(i32) => std2::println(i32);
    .f32(f32) => std2::println(f32);
    .f64(f64) => std2::println(f64);
    .str(str) => std2::println(str);
  };
}

int main() safe {
  print(.i32(5));
  print(.f32(101.3f));
  print(.f64(3.15159));
  print(.str("Hello safety"));
}
```

Choice types are Safe C++'s type-safe offering. They're just like Rust's enums, one of the features most credited for that language's modern ergonomics. Accessing members of a choice object requires testing for the active type with a *match-expression*. If the match succeeds, a new declaration is bound to the corresponding payload, and that declaration is visible in the *match-body* following the `=>`.

The compiler also performs exhaustiveness testing on matches. Users must name all the alternatives, or use a wildcard `_` to default the unnamed ones.

Pattern matching and choice types aren't just a qualify-of-life improvement. They're a critical part of the memory safety puzzle.

1.5.3 Thread safety

A memory safe language should be robust against data races to shared mutable state. If one thread is writing to shared state, no other thread may access it. C++ is not a thread safe language. Its synchronization objects, such as `std::mutex`, are opt-in. If a user reads shared mutable state from outside of a mutex, that's a potential data race. It's up to users to coordinate that the same synchronization objects are locked before accessing the same shared mutable state.

Due to their non-deterministic nature, data race defects are notoriously difficult to debug. Safe C++ prevents them from occurring in the first place. Programs with potential data race bugs in the safe context are ill-formed at compile time.

The thread safety model uses send and sync interfaces, interior mutability and borrow checking to establish a system of constraints guaranteeing that shared mutable state is only accessed through synchronization primitives like `std2::mutex`.

thread_safety.cxx – (Compiler Explorer)

```
#feature on safety
#include <std2.h>
#include <chrono>

using namespace std2;

// mutex is sync, so arc<mutex<string>> is send.
void entry_point(arc<mutex<string>> data, int thread_id) safe {
  // Lock the data through the mutex.
  // When lock_guard goes out of scope, the mutex is released.
  auto lock_guard = data->lock();

  // Get a mutable borrow to the string data. When lock_guard goes out of
  // scope, this becomes a dangling pointer. The borrow checker prevents
  // us from accessing through dangling pointers.
  string^ s = mut lock_guard.borrow();

  // Append a fire and print the new shared state.
  s.append(" ");

  // Drop the lock before printing the shared state. This makes the borrow
  // `s` a "dangling pointer," in the sense that it depends on an expired
  // lifetime. That will raise a borrowck error on `println(*s)`, which
  // attempts to access shared state outside of the lock.
  // drp lock_guard;

  // Drop the data before printing shared state. This decrements arc's
  // reference count and could potentially free the string data. It
  // raises a borrowck error on `println(*s)` because the borrow on `data`
  // from `data->lock()` is kept live by the use of `println(*s)` via the
  // borrow `mut lock_guard.borrow()`.
  // drp data;

  // Print the updated shared state.
  println(*s);

  // Sleep 1s before returning.
  unsafe { std::this_thread::sleep_for(std::chrono::seconds(1)); }
}
```

```
int main() safe {
  // arc - Shared ownership.
  // mutex - Shared mutable access.
  arc<mutex<string>> shared_data(string("Hello world - "));

  // Launch 10 threads.
  vector<thread> threads { };
  for(int i : 10)
    mut threads.push_back(thread(entry_point, cpy shared_data, i));

  // The consuming into_iterator produced by `rel threads` lets us relocate
  // elements out of the vector for the consuming thread::join call.
  for(thread t : rel threads)
    t rel.join();
}
```

```
Hello world -
Hello world -
Hello world -
Hello world -
Hello world -
Hello world -
Hello world -
Hello world -
Hello world -
Hello world -
```

We spawn ten threads which append a fire emoji to a shared string. The string is stored in an `std2::mutex` which is owned by an `arc`, which stands for "atomic reference count." The `arc` provides *shared ownership* of the data. The `mutex` provides *shared access* to it. C++ programmers often think that `std::shared_ptr` pointer provides safe shared access to objects. It does not. It only provides shared ownership.

`arc`'s accessor `operator->` returns const-qualified borrows to the owned data. You can't mutate through most const-qualified types. You can only mutate through const-qualified types that encapsulate `unsafe_cell`, such as `cell`, `ref_cell`, `mutex` and `shared_mutex`. This is how interior mutability implements shared mutable access. The safe standard library provides `mutex` and `shared_mutex` which satisfy the `send` and `sync` interfaces. Only types satisfying `send` may be copied through the `std2::thread` constructor.

Inside the worker thread, we `lock` the mutex to initialize a lock guard object. The lock guard is an RAII type: on its construction the mutex is locked and on its destruction the mutex is unlocked. We call `borrow` on the lock guard to gain a mutable borrow to the string it contains. It's only correct to use the reference while the lock guard is in scope, that is, while the thread has the mutex locked. Now we have exclusive access to the string inside the mutex and append the fire emoji without risking a data race.

But the thread safety isn't yet demonstrated: the claim isn't that we *can* write thread safe software; the claim is that it's *ill-formed* to write thread unsafe software.

Let's sabotage our own design. Uncomment the `drp lock_guard` line. The lock guard is destroyed and unlocks the mutex. The next statement prints the string outside of the mutex, which is a data race, because one of the other nine threads may at that instant be appending to the string.

```
safety: during safety checking of void entry_point(std2::arc<std2::mutex<std2::string>>, int) safe
  borrow checking: thread_safety.cxx:35:12
    println(*s);
            ^
  use of s depends on expired loan
```

```
  drop of lock_guard between its mutable borrow and its use
  invalidating operation at thread_safety.cxx:25:3
    drp lock_guard;
    ^
  loan created at thread_safety.cxx:16:19
    string^ s = mut lock_guard.borrow();
                ^
```

Fortunately the borrow checker refuses to compile this function. We're informed that our use of `s` in `println(*s)` depends on an expired loan, which is the loan on the lock guard that gave us the string borrow in `mut lock_guard.borrow()`. That loan was invalidated when we dropped the lock guard: dropping a place with a live borrow on it is a borrow checker error.

This time uncomment `drp data` to destroy the thread's copy of the `arc` object. This decrements the ref count and potentially frees the string data. This is undesirable, as the subsequent statement prints that same string, and we don't want a use-after-free bug.

```
safety: during safety checking of void entry_point(std2::arc<std2::mutex<std2::string>>, int) safe
  borrow checking: thread_safety.cxx:32:3
    drp data;
    ^
  drop of data between its shared borrow and its use
  borrow kept live by drop of lock_guard declared at thread_safety.cxx:11:8
    auto lock_guard = data->lock();
         ^
  loan created at thread_safety.cxx:11:21
    auto lock_guard = data->lock();
                      ^
```

The borrow checker stops compilation. We're dropping `data`, which is the thread's copy of the `arc`, between its shared borrow and its use. The expired borrow is created by `data->lock()`, which is our lock on the mutex: it creates a lifetime constraint on `data`. That borrow is kept live after the `drp data` by the lock guard's destructor, which unlocks the mutex in the `arc`. The borrow checker won't let us drop `data`, because that conflicts with an in-scope loan on that same object. The use-after-free defect is avoided.

This is lifetime safety with an additional level of indirection compared to the previous borrow checker violation. The beauty of borrow checking is that, unlike lifetime safety based on heuristics, it's robust for any complicated set of constraints and control flow. The thread safety it enables is superior concurrency technology than what Standard C++ provides.

1.5.4 Runtime checks

Safe operations that can't be checked for soundness with static analysis must be checked at runtime. An operation may return an error code, throw an exception or panic and abort. Panics terminate the program, which is far preferable from a security standpoint than entering undefined behavior.

- Builtin arrays and slices panic on out-of-bounds subscripts.
- Integer division panics when the divisor is zero or when the numerator is INT_MIN and the divisor is -1.
- `std2::vector` panics on out-of-bounds subscripts.
- `std2::ref_cell` panics when requesting a mutable borrow and the inner object is borrowed or when requesting a shared borrow and the inner object is mutably borrowed.

subscript_array.cxx

```
#feature on safety
#include <cstdint>

int main() safe {
```

```
  int array[4] { 1, 2, 3, 4 };
  size_t index = 10;

  // Panic on out-of-bounds array subscript.
  int x = array[index];
}
```

```
$ circle subscript_array.cxx
$ ./subscript_array
subscript_array.cxx:9:17
int main() safe
subscript is out-of-range of type int[4]
Aborted (core dumped)
```

The compiler emits panics for out-of-bounds subscripts on builtin arrays and slice types. The runtime can be elided with unsafe subscripts.

subscript_vector.cxx

```
#feature on safety
#include <std2.h>

int main() safe {
  std2::vector<int> vec { 1, 2, 3, 4 };
  size_t index = 10;

  // Panic on out-of-bounds vector subscript.
  int x = vec[index];
}
```

```
$ circle subscript_vector.cxx -I ../libsafecxx/single-header/
$ ./subscript_vector
../libsafecxx/single-header/std2.h:1684:39
std2::vector<int>::operator[]
vector subscript is out-of-bounds
Aborted (core dumped)
```

Out-of-contract use of user-defined types should panic with a similar message. Unlike Standard C++, our containers always perform bounds checking. As with builtin types, the runtime check can be elided with unsafe subscripts or disabled for the entire translation unit with the `-no-panic` compiler switch. This is a drastic measure and is intended for profiling the effects of runtime checks.

```
template<typename T+>
class vector {
public:
  const value_type^ operator[](const self^, size_type i) noexcept safe {
    if (i >= self.size()) panic_bounds("vector subscript is out-of-bounds");
    unsafe { return ^self.data()[i]; }
  }
  ...
};
```

The safety model establishes rules for where library code must insert panic calls. If a function is marked safe but is internally unsound for some values of its arguments, it should check those arguments and panic before executing the unsafe operation. Unsafe functions generally don't panic because its the responsibility of their callers to observe the preconditions of the function.

2 Design overview

2.1 The `safe` context

Operations in the safe context are guaranteed not to cause undefined behavior. Some operations linked to undefined behavior can't be vetted by the frontend, during MIR analysis or with panics at runtime. Attempting to use them in the safe context makes the program ill-formed. These operations are:

— Dereference of pointers and legacy references. This may result in use-after-free undefined behaviors. Prefer using borrows, which exhibit lifetime safety thanks to the borrow checker.
— Pointer offsets. Advancing a pointer past the end of its allocation is undefined behavior. Prefer using slices, which include bounds information.
— Pointer difference. Taking the difference of pointers into different allocations is undefined behavior.
— Pointer relational operators <, <=, > and >=. Comparing pointers into different allocations is undefined behavior.
— Accessing fields of unions. Legacy unions present a type safety hazard. Prefer using choice types.
— Naming mutable objects with static storage duration. This is a data race hazard, as different users may be writing to and reading from the same memory simultaneously. This is even a hazard with `thread_local` storage, as the law of exclusivity cannot be upheld for shared mutable access within a single thread.
— Inline ASM. The compiler generally isn't equipped to determine if inline ASM is safe, so its usage in the safe context is banned.
— Calling unsafe functions. This is banned because the unsafe function may involve any of the above operations.

The compiler lowers function definitions to mid-level IR (MIR) and performs initialization analysis and borrow checking. These issues are found by data flow analysis, making your program ill-formed, regardless of the safe context:

— Use of uninitialized, partially initialized or potentially initialized objects is ill-formed. This is checked by initialization analysis.
— A conflicting action on an overlapping place with an in-scope loan is a borrow checker error. This includes use-after-free and iterator invalidation issues. The law of exclusivity is enforced as part of this check.
— Free region errors. The borrow checker upholds that lifetimes on function parameters do not outlive the constraints defined on the function's declaration. This ensures that the caller and callee agree on the lifetimes of arguments and result objects. It enables what is essentially inter-procedural live analysis without attempting very expensive whole-program analysis.

Some operations are potentially unsound, but can be checked at runtime. If the check fails, the program panics and aborts. Runtime checks are enforced regardless of the safe context.

2.1.1 *safe-specifier*

As with the *noexcept-specifier*, function types and declarations may be marked with a *safe-specifier*. Position this after the *noexcept-specifier*. Types and functions without the *noexcept-specifer* are assumed to be potentially throwing. Similarly, types and functions without the *safe-specifier* are assumed to be unsafe.

```
// `safe` is part of the function type.
using F1 = void(int);
using F2 = void(int) safe;
using F3 = void(int) noexcept;
using F4 = void(int) noexcept safe;
```

As with `noexcept`, `safe` is part of the function's type, so types with different *safe-specifiers* always compare differently:

```
// `safe` is part of the function type.
static_assert(F1 != F2);
static_assert(F3 != F4);
```

Just like `noexcept`, the safeness of a function's type can be stripped during standard conversion of function pointers. It's unsound to add a *safe-specifier* during conversions, as it is with `noexcept`, so that's prohibited.

```
// You can strip off `safe` in function pointers.
static_assert(std::is_convertible_v<F2*, F1*>);
static_assert(std::is_convertible_v<F4*, F3*>);

// You can strip off both `noexcept` and `safe`.
static_assert(std::is_convertible_v<F4*, F1*>);

// You can't add safe. That's unsafe.
static_assert(!std::is_convertible_v<F1*, F2*>);
static_assert(!std::is_convertible_v<F3*, F4*>);
```

Declaring functions with value-dependent *safe-specifiers* is supported. Query the safeness of an expression in an unevaluated context with the *safe-operator*. It's analagous to the existing *noexcept-operator*. It's useful when paired with *requires-clause*, as it lets you constrain inputs based on the safeness of a callable.

safe.cxx

```
#feature on safety

template<typename F, typename... Args>
void spawn(F f, Args... args) safe requires(safe(f(args...)));

struct Foo {
  // The int overload is safe.
  void operator()(const self^, int) safe;

  // The double overload is unsafe.
  void operator()(const self^, double);
};

int main() safe {
  Foo obj { };
  spawn(obj, 1);   // OK
  spawn(obj, 1.1); // Ill-formed. Fails requires-clause.
}
```

```
$ circle safe.cxx
error: safe.cxx:17:8
  spawn(obj, 1.1); // Ill-formed. Fails requires-clause.
       ^
error during overload resolution for spawn
  instantiation: safe.cxx:4:6
  void spawn(F f, Args... args) safe requires(safe(f(args...)));
       ^

  template arguments: [
    F = Foo
      class Foo declared at safe.cxx:6:1
    Args#0 = double
  ]
    constraint: safe.cxx:4:45
    void spawn(F f, Args... args) safe requires(safe(f(args...)));
                                                ^
```

```
    spawn fails requires-clause (safe(f(args...)))
```

Consider a `spawn` function that takes a callable `f` and a set of arguments `args`. The function is marked `safe`. Transitively, the callable, when invoked with the provided arguments, must also be a safe operation. But we can't stipulate `safe` on the type `F`, because it may not be a function type. Here it's a class with two overloaded call operators.

When the user provides an integer argument, the *requires-clause* substitutes to `safe(f(1))`, which is true, because the best viable candidate for the function call is `void operator()(const self^, int) safe;`. That's a safe function.

When the user provides a floating-point argument, the *requires-clause* substitutes to `safe(f(1.1))`, which is false, because the best viable candidate is `void operator()(const self^, double);`. That's not a safe function.

2.1.2 *unsafe-block*

The `unsafe` token escapes the safe context, permitting operations for which soundness cannot be guaranteed by the compiler. The primary unsafe escape is the *unsafe-block*. At the statement level, write `unsafe { }` with the unsafe operations inside the braces. Unlike in Rust, *unsafe-blocks* do *not* open new lexical scopes.

```
#feature on safety

int func(const int* p) safe {
  // Most pointer operations are unsafe. Use unsafe-block to perform them in
  // a safe function.

  // Pointer offset is unsafe.
  unsafe { ++p; }

  // Put multiple statements in an unsafe-block.
  unsafe {
    // Pointer difference is unsafe.
    ptrdiff_t diff = p - p;

    // unsafe-blocks do not open lexical scopes. Declarations will be
    // visible below.
    int x = p[4];
  }

  // Can use unsafe-block like compound-statement inside control flow.
  if(x == 1) unsafe { return p[5]; }

  return x;
}
```

Unsafe blocks are appear in different forms in a few places:

- Before subobject *mem-initializer*.
- Before *condition* expressions.
- Before *match-body* in a *match-clause*.

```
#feature on safety

// Unsafe function.
bool func();

struct Foo {
```

```
  // This constructor is safe, but the mem-initializer is unsafe.
  // Use an unsafe-block here.
  Foo() safe : unsafe b(func()) { }
  bool b;
};

int main() safe {
  // main() is safe, but the if-condition is unsafe.
  // Use an unsafe-block.
  if(unsafe func()) { }

  int x = match(1) {
    // Use an unsafe block in the match-body.
    _ => unsafe func();
  };
}
```

In all cases, `unsafe` is an auditable token where the user expressly takes responsibility for sound execution.

2.1.3 The `unsafe` type qualifier

The Rust ecosystem was built from the bottom-up prioritizing safe code. Consequently, there's so little unsafe code that the *unsafe-block* is generally sufficient for interfacing with it. By contrast, there are many billions of lines of unsafe C++. The *unsafe-block* isn't powerful enough to interface our safe and unsafe assets, as we'd be writing *unsafe-blocks* everywhere, making a noisy mess. Worse, we'd be unable to use unsafe types from safe function templates, since the template definition wouldn't know it was dealing with unsafe template parameters. Because of the ecosystem difference, Rust does not provide guidance for this problem, and we're left to our own devices.

Safe C++'s answer to safe/unsafe interoperability is to make safeness part of the type system.

Standard C++ has `const` and `volatile` type qualifiers. Safe C++ adds the `unsafe` type qualifier. Declare an object or data member with the `unsafe` qualifier and use it freely *even in safe contexts*. The `unsafe` token means the same thing here as it does with *unsafe-blocks*: the programmer is assuming responsibility for upholding the requirements of the unsafe-qualifed type. If something is unsound, blame lies with the `unsafe` wielder.

Naming an unsafe object yields an lvalue expression of the unsafe type. What are the effects of the unsafe qualifier on an expression? In a safe context:

- Calling unsafe member functions on unsafe-qualified objects is permitted.
- Calling unsafe functions where a function argument is unsafe-qualified is permitted.
- Unsafe constructors may initialize unsafe types.

Expressions carry `noexcept` and `safe` states, which is how *noexcept-operator* and *safe-specifier* are implemented. Those states are separate from the type of the expression. Why make `unsafe` a type qualifier, which represents a significant change to the type system, rather than a property of an expresion like those `noexcept` and `safe` states?

The answer is that template specialization works on types and it doesn't work on these other states of an expression. A template argument with an unsafe type qualifier instantiates a template with an unsafe type qualifier on the corresponding template parameter. The unsafe qualifier drills through templates in a way that other language entities don't.

unsafe2.cxx – (Compiler Explorer)

```
#feature on safety
#include <std2.h>
#include <string>
```

```
int main() safe {
  // Requires unsafe type specifier because std::string's dtor is unsafe.
  std2::vector<unsafe std::string> vec { };

  // Construct an std::string from a const char* (unsafe)
  // Pass by relocation (unsafe)
  mut vec.push_back("Hello unsafe type qualifier!");

  // Append Bar to the end of Foo (unsafe)
  mut vec[0] += "Another unsafe string";

  std2::println(vec[0]);
}
```

We want to use the new memory safe vector with the legacy string type. The new vector is borrow checked, eliminating use-after-free and iterator invalidation defects. It presents a safe interface. But the old string is pre-safety. All its member functions are unsafe. We want to specialize the new vector on the old string, so we mark it `unsafe`.

The unsafe type qualifier propagates through the instantiated vector. The expressions returned from the `operator[]` accessor are unsafe qualified, allowing us to call unsafe member functions on the string, even in main's safe context.

Let's simplify the example above and study it in detail.

unsafe3.cxx

```
#feature on safety

template<typename T+>
struct Vec {
  Vec() safe;
  void push_back(self^, T obj) safe;
};

struct String {
  String(const char*); // Unsafe ctor.
};

int main() safe {
  // Vec has a safe constructor.
  Vec<unsafe String> vec { };

  // void Vec<unsafe String>::push_back(self^, unsafe String) safe;
  // Copy initialization of the `unsafe String` function parameter is
  // permitted.
  mut vec.push_back("A string");
}
```

In this example, the unsafe `String` constructor is called in the safe `main` function. That's permitted because substitution of `unsafe String` into Vec's template parameter creates a `push_back` specialization with an `unsafe String` function parameter. Safe C++ allows unsafe constructors to initialize unsafe-qualified types in an safe context.

Permitting unsafe operations with unsafe qualifier specialization is less noisy and exposes less of the implementation than using conditional *unsafe-specifiers* on the class template's member functions. More importantly, we

want to keep the new vector's interface safe, even when it's specialized with unsafe types. This device allows member functions to remain safe without resorting to *unsafe-blocks* in the implementations. There's a single use of the `unsafe` token, which makes for simple audits during code review.

Placing the unsafe token on the *template-argument-list*, where the class template gets used, is far preferable to enclosing operations on the template parameter type in *unsafe-blocks* inside the template. In the former case, the user of the container can read its preconditions and swear that the precondidions are met. In the latter case, the template isn't able to make any statements about properly using the template type, because it doesn't know what that type is. The `unsafe` token should go with the caller, not the callee.

unsafe4.cxx

```
#feature on safety

template<typename T+>
struct Vec {
  Vec() safe;
  void push_back(self^, T obj) safe;
};

struct String {
  String(const char*); // Unsafe ctor.
};

int main() safe {
  // Vec has a safe constructor.
  Vec<unsafe String> vec { };

  // void Vec<unsafe String>::push_back(self^, unsafe String) safe;
  // This is ill-formed. We can't invoke the unsafe String constructor
  // to initialize an unsafe type.
  mut vec.push_back(String("A string"));
}
```

```
$ circle unsafe4.cxx
error: unsafe4.cxx:20:27
  mut vec.push_back(String("A string"));
                           ^
cannot call unsafe constructor String::String(const char*) in safe context
see declaration at unsafe4.cxx:10:3
```

This code is ill-formed. It's permissible to invoke an unsafe constructor when copy-initializing into the `push_back` call, since its function parameter is `unsafe String`. Dut direct initialization of `String` is not allowed. The constructor chosen for direct initialization is unsafe, but the type it's initializing is not. The type is just `String`. The compiler is right to reject this program because the user is plainly calling an unsafe constructor in a safe context, without a mitigating *unsafe-block* or unsafe qualifier.

unsafe5.cxx

```
#feature on safety

template<typename T+>
struct Vec {
  Vec() safe;
  void push_back(self^, T obj) safe;

  template<typename... Ts>
```

```
  void emplace_back(self^, Ts... obj) safe {
    // Direct initialization with the String(const char*) ctor.
    // This compiles, because T is unsafe-qualified.
    self.push_back(T(rel obj...));
  }
};

struct String {
  String(const char*); // Unsafe ctor.
};

int main() safe {
  // Vec has a safe constructor.
  Vec<unsafe String> vec { };

  // void Vec<unsafe String>::emplace_back(self^, const char*) safe;
  mut vec.emplace_back("A string");
}
```

This program is well-formed. As with the previous example, there's a direct initialization of a String object using its unsafe constructor. This time it's allowed, because the type being initialized is `T`, which is substituted with `unsafe String`: unsafe constructors are permitted to initialize unsafe types.

The unsafe type qualifier is a powerful mechanism for incorporating legacy code into new, safe templates. But propagating the qualifier through all template parameters may be too permissive. C++ templates won't be expecting the `unsafe` qualifier, and it may break dependencies. Functions that are explicitly instantiated won't have `unsafe` instantiations, and that would cause link errors. It may be prudent to get some usage experience, and limit this type qualifier to being deduced only by type template parameters with a certain token, eg `typename T?`. That way, `typename T+?` would become a common incantation for containers: create template lifetime parameters for this template parameter *and* deduce the `unsafe` qualifier for it.

To be more accommodating when mixing unsafe with safe code, the `unsafe` qualifier has very liberal transitive properties. A function invoked with an unsafe-qualified object or argument, or a constructor that initializes an unsafe type, are *exempted calls*. When performing overload resolution for exempted calls, function parameters of candidates become unsafe qualified. This permits copy initialization of function arguments into parameter types when any argument is unsafe qualified. The intent is to make deployment of the `unsafe` token more strategic: use it less often but make it more impactful. It's not helpful to dilute its potency with many trivial *unsafe-block* operations.

2.1.4 unsafe subscripts

There's one more prominent use of the `unsafe` token: it suppresses runtime bounds checks in subscript operations on both builtin and user-defined types. For applications where nanoseconds matter, developers may want to forego runtime bounds checking. In Safe C++, this is straight forward. Just write `; unsafe` in your array, slice or vector subscript.

unsafe_bounds.cxx – (Compiler Explorer)

```
#feature on safety
#include <std2.h>

void subscript_array([int; 10] array, size_t i, size_t j) safe {
  array[i; unsafe] += array[j; unsafe];
}

void subscript_slice([int; dyn]^ slice, size_t i, size_t j) safe {
```

```
  slice[i; unsafe] += slice[j; unsafe];
}

void subscript_vector(std2::vector<int> vec, size_t i, size_t j) safe {
  mut vec[i; unsafe] += vec[j; unsafe];
}
```

The unsafe subscript indicates that runtime bounds checks is relaxed, while keeping a consistent syntax with ordinary checked subscripts. It doesn't expose any operations to an unsafe context. The use of the `unsafe` token is the programmer's way of taking responsibility for correct behavior.

unsafe_bounds.rs

```
fn subscript_array(mut array: [i32; 10], i: usize, j: usize) {
  unsafe { *array.get_unchecked_mut(i) += *array.get_unchecked(j); }
}

fn subcript_slice(slice: &mut [i32], i: usize, j: usize) {
  unsafe { *slice.get_unchecked_mut(i) += *slice.get_unchecked(j); }
}

fn subscript_vector(mut vec: Vec<i32>, i: usize, j: usize) {
  unsafe { *vec.get_unchecked_mut(i) += *vec.get_unchecked(j); }
}
```

Rust's unchecked subscript story is not elegant. You have to use the separately named functions `get_unchecked` and `get_unchecked_mut`. These are unsafe functions, so your calls have to be wrapped in *unsafe-blocks*. That exposes other operations to the unsafe context. Since we're invoking functions directly, as opposed to using the `[]` operator, we don't get automatic dereference of the returned borrows.

```
template<class T+>
class vector
{
  value_type^ operator[](self^, size_type i) noexcept safe {
    if (i >= self.size()) panic_bounds("vector subscript is out-of-bounds");
    unsafe { return ^self.data()[i]; }
  }
  value_type^ operator[](self^, size_type i, no_runtime_check) noexcept {
    return ^self.data()[i];
  }

  const value_type^ operator[](const self^, size_type i) noexcept safe {
    if (i >= self.size()) panic_bounds("vector subscript is out-of-bounds");
    unsafe { return ^self.data()[i]; }
  }
  const value_type^ operator[](const self^, size_type i, no_runtime_check) noexcept {
    return ^self.data()[i];
  }
  ...
};
```

The unsafe subscript works with user-defined types by introducing a new library type `std2::no_runtime_check`. Append this as the last parameter in a user-defined `operator[]` declaration, including multi-dimensional subscript operators, to enable unsafe subscript binding. The compiler default-initializes a `no_runtime_check` argument and attempts to pass that during overload resolution. `operator[]`s with `no_runtime_check` parameters are *unsafe functions*, since they aren't sound for all valid inputs. However, when invoked as part of an unsafe

subscript, they can still be used in safe contexts. That's because the user wrote the `unsafe` token at the point of use, exempting this function call from the unsafe check.

2.2 Borrow checking

There's one widely deployed solution to lifetime safety: garbage collection. In GC, the scope of an object is extended as long as there are live references to it. When there are no more live references, the system is free to destroy the object. Most memory safe languages use tracing garbage collection.[tracing-gc] Some, like Python and Swift, use automatic reference counting,[arc] a flavor of garbage collection with different tradeoffs.

Garbage collection requires storing objects on the *heap*. But C++ is about *manual memory management*. We need to track references to objects on the *stack* as well as on the heap. As the stack unwinds objects are destroyed. We can't extend their duration beyond their lexical scopes. Borrow checking[borrow-checking] is a kind of compile-time analysis that prevents using a reference after an object has gone out of scope. That is, it solves use-after-free and iterator invalidation bugs.

2.2.1 Use-after-free

`std::string_view` was added to C++17 as a safer alternative to passing character pointers around. Unfortunately, its rvalue-reference constructor is so dangerously designed that its reported to *encourage* use-after-free bugs.[string-view-use-after-free]

string_view0.cxx – (Compiler Explorer)

```
#include <iostream>
#include <string>
#include <string_view>

int main() {
  std::string s = "Hellooooooooooooooo ";
  std::string_view sv = s + "World\n";
  std::cout << sv;
}
```

```
$ circle string_view0.cxx
$ ./string_view
  :    ooo World
```

`s` is initialized with a long string, bypassing the short-string optimization and storing it on the heap. The `string::operator+` returns a temporary `std::string` object, also with its data on the heap. `sv` is initialized by calling the `string::operator string_view()` conversion function on the temporary. The temporary string goes out of scope at the end of that statement, its storage is returned to the heap, and the user prints a string view with dangling pointers!

This design is full of sharp edges. It should not have made the ISO Standard, but C++ as a language doesn't provide great alternatives. Borrow checking is the technology that flags these problems. Safe C++ provides a lifetime-aware `string_view` that ends the risk of dangling views.

string_view1.cxx – (Compiler Explorer)

```
#feature on safety
#include <std2.h>

int main() safe {
  std2::string s("Hellooooooooooooooo ");
  std2::string_view sv = s + "World\n";
  println(sv);
}
```

```
$ circle string_view1.cxx -I ../libsafecxx/single-header/
safety: during safety checking of int main() safe
  borrow checking: string_view1.cxx:7:11
    println(sv);
            ^
  use of sv depends on expired loan
  drop of temporary object std2::string between its shared borrow and its use
  loan created at string_view1.cxx:6:28
    std2::string_view sv = s + "World\n";
                               ^
```

The compiler flags the use of the dangling view: `println(sv)`. It marks the invalidating action: the drop of the temporary string. And it indicates where the loan was created: the conversion to `string_view` right after the string concatenation. See the error reporting section for details on lifetime diagnostics.

The lifetime mechanics are evident from the declaration of the conversion function on `std2::string` which produces a `std2::string_view`. This is gets called during `sv`'s initialization.

```
class basic_string_view/(a) {
public:
  basic_string_view(const [value_type; dyn]^/a str, no_utf_check) noexcept
    : p_(str) { }
};

class basic_string {
public:
  basic_string_view<value_type> str(self const^) noexcept safe {
    using no_utf_check = typename basic_string_view<value_type>::no_utf_check;
    unsafe { return basic_string_view<value_type>(self.slice(), no_utf_check{}); }
  }

  operator basic_string_view<value_type>(self const^) noexcept safe {
    return self.str();
  }
  ...
};
```

String concatenation forms a temporary string. The temporary string doesn't have any lifetime parameters. It owns its data and has *value semantics*. We form a `std2::string_view` from the temporary string. `basic_string_view` declares a named lifetime parameter, `/a`, which models the lifetime of of the view. The view has *reference semantics*, and it needs a lifetime for the purpose of live analysis. The lifetime of this view will originate with a loan on a string, which owns the data, and is kept live when using the view. Mutating or dropping the string while there's a live loan on it makes the program ill-formed. This is how borrow checking protects against use-after-free errors. Keep in mind that the lifetime parameter is part of the *view*, not part of the data it refers to.

Initializing `sv` invokes the conversion function on that temporary. The conversion function *borrows* self, creating a *loan* the temporary. The lifetime argument isn't spelled out in the conversion function's declaration; rather, it's assigned during lifetime elision, which is part of normalization. The return type of the conversion function is a `string_view`. Since `string_view` is a lifetime binder, elision invents a lifetime argument for the return type and constrains the lifetime of the `self` reference to outlive it. This is all established in the declaration. Callers don't look inside function definitions during borrow checking. Both the caller and callee agree on the function's lifetime contracts, entirely from information in the function declaration. This establishes a chain of constraints that relate all uses of a reference back to its original loan.

The `str` accessor provides the implementation. It constructs a `basic_string_view` and bypasses the runtime

UTF-8 check by passing a `no_utf_check` argument. Like Rust, Safe C++ strings guarantee their contents are well-formed UNICODE code points.

When we go to print the view, the compiler raises a use-after-free error. The call to `println` *uses* the lifetime arguments associated with the function argument `sv`. Middle-end analysis solves the constraint equation, and puts the original loan on the string temporary object *in scope* at all program points between the string concatenation and the `println` call. The borrow checker pass looks for invalidating actions on places that overloap in-scope loans. There's a drop of the temporary string at the end of the full expression containing the string concatenation. This isn't anything special to Safe C++, this is ordinary RAII cleanup of objects as they fall out of lexical scope. The drop is a kind of *shallow write*, and it invalidates the shared borrow taken on the temporary when its conversion operator to `string_view` was called.

Unlike previous attempts at lifetime safety[P1179R1], borrow checking is absolutely robust. It does not rely on heuristics that can fail. It allows for any distance between the use of a borrow and an invalidating action on an originating loan, with any amount of complex control flow between. MIR analysis will solve the constraint equation, run the borrow checker, and issue a diagnostic.

2.2.2 Iterator invalidation

Let's take a closer look at the iterator invalidation example.

iterator.cxx – (Compiler Explorer)

```
#feature on safety
#include <std2.h>

int main() safe {
  std2::vector<int> vec { 11, 15, 20 };

  for(int x : vec) {
    // Ill-formed. mutate of vec invalidates iterator in ranged-for.
    if(x % 2)
      mut vec.push_back(x);

    std2::println(x);
  }
}
```

```
$ circle iterator.cxx -I ../libsafecxx/single-header/
safety: during safety checking of int main() safe
  borrow checking: iterator.cxx:10:11
        mut vec.push_back(x);
            ^
  mutable borrow of vec between its shared borrow and its use
  loan created at iterator.cxx:7:15
    for(int x : vec) {
                ^
```

The *ranged-for* creates an iterator on the vector. The iterator stays initialized for the whole duration of the loop. This puts a shared borrow on `vec`, because the iterator provides read-only access to that container. Inside the loop, we mutate the container. The `mut` keyword enters the mutable context which enables binding mutable borrows to lvalues in the standard conversion during overload resolution for the `push_back` call. Now there's a shared borrow that's live, for the iterator, and a mutable borrow that's live, for the `push_back`. That violates exclusivity and the borrow checker raises an error.

To users, iterator invalidation looks like a different phenomenon than the use-after-free defect in the previous section. But to the compiler, and hopefully to library authors, they can be reasoned about similarly, as they're both borrow checker violations.

> The static analysis based on the proposed lifetime annotations cannot catch all memory safety problems in C++ code. Specifically, it cannot catch all temporal memory safety bugs (for example, ones caused by iterator invalidation), and of course lifetime annotations don't help with spatial memory safety (for example, indexing C-style arrays out of bounds). See the comparison with Rust below for a detailed discussion.
>
> – [RFC] Lifetime annotations for C++ Clang Frontend[clang-lifetime-annotations]

Clang's lifetime annotations project doesn't implement borrow checking. For that project, iterator invalidation is out of scope, because it really is a different phenomenon than the lifetime tracking *heuristics* employed in detecting other use-after-free defects.

```
interface iterator {
  typename item_type;
  optional<item_type> next(self^) safe;
};

interface make_iter {
  typename iter_type;
  typename iter_mut_type;
  typename into_iter_type;

  iter_type      iter(const self^) safe;
  iter_mut_type  iter(self^) safe;
  into_iter_type iter(self) safe;
};

template<class T>
class slice_iterator/(a);

template<class T>
impl vector<T> : make_iter {
  using iter_type = slice_iterator<T const>;
  using iter_mut_type = slice_iterator<T>;
  using into_iter_type  = into_iterator<T>;

  iter_type iter(self const^) noexcept safe override {
    return slice_iterator<const T>(self.slice());
  }

  iter_mut_type iter(self^) noexcept safe override {
    return slice_iterator<T>(self.slice());
  }

  into_iter_type iter(self) noexcept safe override {
    auto p = self^.data();
    auto len = self.size();
    forget(rel self);
    unsafe { return into_iter_type(p, p + len); }
  }
};
```

To opt into safe ranged-for iteration, containers implement the `std2::make_iter` interface. They can provide const iterators, mutable iterators or *consuming iterators* which take ownership of the operand's data and destroy its container in the process.

Because the *range-initializer* of the loop is an lvalue of `vector` that's outside the mutable context, the const

iterator overload of `vector<T>::iter` gets chosen. That returns an `std2::slice_iterator<T const>` into the vector's contents. Here's the first borrow: lifetime elision invents a lifetime parameter for the `self const^` parameter, which is also used for the named lifetime argument `/a` of the `slice_iterator` return type. As with the use-after-free example, the `vector<T>::iter` function declaration establishes an *outlives-constraint*: the lifetime on the `self const^` operand must outlive the lifetime on the result object.

```
template<class T>
class slice_iterator/(a)
{
  T* unsafe p_;
  T* end_;
  T^/a __phantom_data;

public:
  slice_iterator([T; dyn]^/a s) noexcept safe
    : p_((*s)~as_pointer), unsafe end_((*s)~as_pointer + (*s)~length) { }

  optional<T^/a> next(self^) noexcept safe {
    if (self->p_ == self->end_) { return .none; }
    return .some(^*self->p_++);
  }
};
```

The compiler drains the iterator by calling `next` until the `std2::optional` it returns is `.none`. Each in-range invocation of `slice_iterator::next` forms a borrow to the current element and advances `p_` to the next element. `next`'s object parameter is written `self^`, an abbreviated form of `slice_iterator/a^ self`. The borrow returned in `optional<T^/a>` is outlived by the implied lifetime of the `self` parameter. This transitively means that the underlying `vector` must outlive the reference in the `optional` returned from `slice_iterator::next`. All these lifetime constraints feed the constraint equation to enable inter-procedural live analysis.

It's the periodic call into `next` that keeps the original borrow on `vec` live. Even though the use of `next` is lexically before the `push_back` (it's at the top of the loop, rather than inside the loop), control flow analysis shows that it's also *downstream* of the push_back. MIR analysis follows the backedge from the end of the body of the loop back to its start. That establishes the liveness of the shared borrow on `vec`.

```
template<class T+>
class vector {
  void push_back(self^, T t) safe {
    if (self.capacity() == self.size()) { self.grow(); }

    __rel_write(self->p_ + self->size_, rel t);
    ++self->size_;
  }
  ...
};
```

The user enters the mutable context with the `mut` token and calls `push_back`. This enables binding a mutable borrow to `vec` in a standard conversion during overload resolution to `push_back`. Since the shared borrow that was taken to produce `slice_iterator` is still in scope, the new mutable borrow violates exclusivity and the program is ill-formed.

Borrow checking is attractive because it's a unified treatment for enforcing dependencies. Compiler engineers aren't asked to develop some heuristics for use-after-free, different ones for iterator invalidation, and still more clever ones for thread safety. Developers simply use the borrow types and the compiler enforces the attendant constraints.

2.2.3 Initializer lists

Almost every part of the C++ Standard Library will exhibit unsound behavior even when used in benign-looking ways. Much of it was designed without consideration of memory safety. This goes even to the most core types, like `std::initializer_list`.

initlist0.cxx

```
#include <vector>
#include <string_view>
#include <iostream>

using namespace std;

int main() {
  initializer_list<string_view> initlist1;
  initializer_list<string_view> initlist2;

  string s = "Hello";
  string t = "World";

  // initializer lists holds dangling pointers into backing array.
  initlist1 = { s, s, s, s };
  initlist2 = { t, t, t, t };

  // Prints like normal.
  vector<string_view> vec(initlist1);
  for(string_view sv : vec)
    cout<< sv<< "\n";

  // Catastrophe.
  vec = initlist2;
  for(string_view sv : vec)
    cout<< sv<< "\n";
}
```

```
$ clang++ initlist0.cxx -o initlist0 -stdlib=libc++ -O2
$ ./initlist0
Hello
Hello
Hello
Hello
__cxa_guard_release%s failed to broadcast__cxa_guard_abortunexpected_handler une
xpectedly returnedterminate_handler unexpectedly returnedterminate_handler unexp
ectedly threw an exceptionPure virtual function called!Deleted virtual function
called!std::exceptionstd::bad_exceptionstd::bad_allocbad_array_new_lengthSt9exce
ptionSt13bad_exceptionSt20bad_array_new_lengthSt9bad_alloc a  a b b 0b St12
domain_errorSt11logic_errorSt16invalid_argumentSt12length_errorSt12out_of_rangeS
t11range_errorSt13runtime_errorSt14overflow_errorSt15underflow_errorstd::bad_cas
tstd::bad_typeidSt9type_infoSt8bad_castSt10bad_typeidlibc++abi: reinterpret_cast
<size_t>(p + 1) % RequiredAlignment == 0/home/sean/projects/llvm-new/libcxxabi/s
rc/fallback_malloc.cppvoid *(anonymous namespace)::fallback_malloc(size_t)reinte
rpret_cast<size_t>(ptr) % RequiredAlignment == 0N10__cxxabiv116__shim_type_infoE
N10__cxxabiv117__class_type_infoEN10__cxxabiv117__pbase_type_infoEN10__cxxabiv11
9__pointer_type_infoE
```

This example declares two initializer lists of string views, `initlist1` and `initlist2`. It declares two strings objects and assigns views of the strings into those initializer lists. Printing their contents is undefined behavior. It may go undetected. But compiling with `clang++ -O2` and targeting libc++ manifests the defect: the program is printing uncontrollably from some arbitrary place in memory. This is the kind of safety defect that the NSA and corporate researchers have been warning industry about.

The defect is perplexing because the string objects `s` and `t` *are still in scope*! This is a use-after-free bug, but not with any object that the user declared. It's a use-after-free of implicit backing stores that C++ generates when lowering initializer list expressions.

```
  // initializer lists holds dangling pointers into backing array.
  initlist1 = { s, s, s, s };
  initlist2 = { t, t, t, t };
```

Each initializer list expression provisions a 4-element array of `string_view` as a backing store. An initializer list object is simply a pointer into the array and a length. It's the pointer into the backing store that's copied into `initlist1` and `initlist2`. At the end of each full expression, those backing stores fall out of scope, leaving dangling pointers in the initializer lists that the user declared. When compiled with aggressive local storage optimizations, the stack space that hosted the backing store for `{ t, t, t, t }` gets reused for other objects. Some subsequent initialization overwrites the pointer and length fields of the string views in that reclaimed backing store. Printing from the `initlist2` declaration prints from the smashed pointers.

initlist1.cxx – (Compiler Explorer)

```
#feature on safety
#include <std2.h>

using namespace std2;

int main() safe {
  initializer_list<string_view> initlist;

  string s{"Hello"};
  string t{"World"};

  // initializer lists holds dangling pointers into backing array.
  initlist = { s, t, s, t };

  // Borrow checker error. `use of initlist depends on expired loan`
  vector<string_view> vec(rel initlist);
  for(string_view sv : vec)
    println(sv);
}
```

```
$ circle initlist1.cxx -I ../libsafecxx/single-header/
safety: during safety checking of int main() safe
  borrow checking: initlist1.cxx:16:31
    vector<string_view> vec(rel initlist);
                                ^
  use of initlist depends on expired loan
  drop of temporary object std2::string_view[4] while mutable borrow of
    temporary object std2::string_view[4][..] is in scope
  loan created at initlist1.cxx:13:14
    initlist = { s, t, s, t };
               ^
```

Safe C++ includes a new `std2::initializer_list` which sits side-by-side with the legacy type. The compiler

provisions a backing store to hold the data, and the new initializer list also keeps pointers into that. The backing store expires at the end of the assignment state. But borrow checking prevents the use-after-free defect that troubles `std::initializer_list`. An error is filed where the initializer list constructs a vector: `use of initlist depends on expired loan`.

The Safe C++ replacement for initializer lists takes the *ownership and borrowing* model to heart by both owning and borrowing. It borrows the storage its elements are held in, but it also owns the elements and calls their destructors. We want to support relocation out of the initializer list. That means the backing store doesn't call the element destructors, because those elements may have been relocated out by the user. `std2::initializer_list` is more a vector, in that it destroys un-consumed objects when dropped, but also like a view, in that sees into another object's storage.

```
template<class T+>
class initializer_list/(a) {
  // Point to byte data on the stack.
  T* unsafe _cur;
  T* _end;
  T^/a __phantom_data;

  explicit
  initializer_list([T; dyn]^/a data) noexcept safe :
    _cur((*data)~as_pointer),
    unsafe _end((*data)~as_pointer + (*data)~length) { }

public:

  ~initializer_list() safe requires(T~is_trivially_destructible) = default;

  [[unsafe::drop_only(T)]]
  ~initializer_list() safe requires(!T~is_trivially_destructible) {
    std::destroy_n(_cur, _end - _cur);
  }

  optional<T> next(self^) noexcept safe {
    if(self->_cur != self->_end)
      return .some(__rel_read(self->_cur++));
    else
      return .none;
  }

  T* data(self^) noexcept safe {
    return self->_cur;
  }

  const T* data(const self^) noexcept safe {
    return self->_cur;
  }

  std::size_t size(const self^) noexcept safe {
    return (std::size_t)(self->_end - self->_cur);
  }

  // Unsafe call to advance. Use this after relocating data out of
  // data().
  void advance(self^, std::size_t size) noexcept {
```

```
    self->_cur += static_cast<std::ptrdiff_t>(size);
  }

  ...
};
```

`std2::initializer_list` has a named lifetime parameter, `/a`, which puts a constraint on the backing store. The private explicit constructor is called when lowering the braced initializer expression–that's compiler magic. The argument to the constructor is a slice borrow to the backing store. Note the lifetime argument on the borrow is the class's named lifetime parameter. This means the backing store outlives the initializer list.

This class has a relatively rich interface. Of course you can query for the pointer to the data and its length. But users may also call `next` to pop off an element at a time and use it much like an iterator. But for element types that are trivially relocatable, users can call `data` and `size` to relocate them in bulk, and then `advance` by the number of consumed elements. When the `initializer_list` goes out of scope, it destructs all unconsumed elements.

2.2.4 Scope and liveness

Key to one's understanding of lifetime safety is the distinction between scope and liveness. Consider lowering your function to MIR instructions which are indexed by program points. The set of points at which an object is initialized is its *scope*. In normal C++, this corresponds to its lexical scope. In Safe C++, due to relocation/destructive move, there are points in the lexical scope where the object may not be initialized, making the scope a subset of the lexical scope.

The compiler lowers AST to MIR and runs *initialization analysis*, a form of forward dataflow analysis that computes the scope of all local variables. If a local variable has been relocated or dropped, and is then named in an expression, the scope information helps the compiler flag this as an illegal usage.

Liveness is a different property than scope, but they're often confused: end users speak of lifetime to mean initialization or scope, while backend engineers speak of lifetime to mean liveness. Borrow checking is concerned with liveness. That's the set of points where the value stored in a variable (i.e. a specific bit pattern) is subsequently used.

```
void f(int);

int main() {
  int x = 1;  // x is live due to use of 1 below.
  f(x);       // not live, because 1 isn't loaded out again.

  x = 2;      // not live, because 2 isn't loaded out.

  x = 3;      // live because 3 is used below.
  f(x);       // still live, because 3 is used below.
  f(x);       // not live, because 3 isn't used again.
}
```

Live analysis is a reverse dataflow computation. Start at the return instruction of the control flow graph and work your way up to the entry point. When you encounter a load instruction, that variable becomes live. When you encounter a store instruction, that variable is marked dead.

The liveness property is useful in register alloction: you only care about representing a variable in register while it's holding a value that has an upcoming use. But we're solving lifetime safety, we're not doing code generation. Here, we're only concerned with liveness as a property of *borrows*.

```
#feature on safety
void f(int);
```

```
int main() {
  int^ ref;   // An uninitialized borrow.
  {
    int x = 1;
    ref = ^x; // *ref is dereferenced below, so ref is live.
    f(*ref);  // now ref is dead.

    int y = 2;
    ref = ^y; // ref is live again
    f(*ref);  // ref is still live, due to read below.
  }

  f(*ref);    // ref is live but y is uninitialized.
}
```

Borrows are checked references. It's a compile-time error to use a borrow after the data it refers to has gone out of scope. Consider the set of all live references at each point in the program. Is there an invalidating action on a place referred to by one of these live references? If so, that's a contradiction that makes the program ill-formed. In this example, the contradiction occurs when `y` goes out of scope, because at that point, `ref` is a live reference to it. What makes `ref` live at that point?–Its use in the last `f(*ref)` expression in the program.

It's not enough to compute liveness of references. To determine the invalidating actions, it's important to know which place the live borrow refers to. `ref` is live until the end of the function, but `x` going out of scope is not an invalidating function because `ref` doesn't refer to `x` anymore. We need data structures that indicate not just when a borrow is live, but to which places it may refer.

2.2.5 Systems of constraints

NLL borrow checking[borrow-checking] tests invalidating actions against live borrows in the presence of control flow. The algorithm involves generating a system of lifetime constraints which relate regions at program points, growing region variables until all the constraints are satisfied, and then testing invalidating actions against all loans in scope.

A loan is the action that forms a borrow to a place. In the example above, there are two loans: `^x` and `^y`. Solving the constraint equation extends the liveness of loans `^x` and `^y` up until their last indirect uses through `ref`. When `y` goes out of scope, it doesn't invalidate the loan `^x`, because that's not live. But it does invalidate the loan `^y`, which is lifetime extended by the final `f(*ref)` expression.

Liveness is stored in bit vectors called *regions variables*. There are region variables for loans `^x` and `^y` and regions for objects with borrow types, such as `ref`. There are also regions for user-defined types with lifetime parameters, such as `string_view`.

A lifetime constraint `'R0 : 'R1 : P` reads "region 0 outlives region 1 at point P." The compiler emits constraints when encountering assignment and function calls involving types with regions.

```
    #feature on safety
    void f(int);

    int main() {
P0:   int^ ref;          // ref is 'R0
      {
P1:     int x = 1;
P2:     <loan R1> = ^x;  // ^x is loan 'R1
P3:     ref = <loan R1>; // 'R1 : 'R0 @ P3
P4:     f(*ref);

P5:     int y = 2;
```

```
P6:     <loan R2> = ^y;  // ^y is loan 'R2
P7:     ref = <loan R2>; // 'R2 : 'R0 @ P7
P8:     f(*ref);

P9:     drop y
P10:    drop x
      }

P11:  f(*ref);
    }
```

I've relabelled the example to show function points and region names of variables and loans. If we run live analysis on 'R0, the region for the variable `ref`, we see it's live at points 'R0 = { 4, 8, 9, 10, 11 }. These are the points where its subsequently used. We'll grow the loan regions 'R1 and 'R2 until their constraint equations are satisfied.

`'R1 : 'R0 @ P3` means that starting at P3, the 'R1 contains all points 'R0 does, along all control flow paths, as long as 'R0 is live. 'R1 = { 3, 4 }. Grow 'R2 the same way: 'R2 = { 7, 8, 9, 10, 11 }.

Now we can hunt for contradictions. Visit each point in the function and consider, "is there a read, write, move, drop or other invalidating action on any of the loans in scope?" The only potential invalidating actions are the drops of `x` and `y` where they go out of scope. At P9, the loan `^y` is in scope, because P9 is an element of its region 'R2. This is a conflicting action, because the loan is also on the variable `y`. That raises a borrow checker error. There's also a drop at P10. P10 is in the region for `^y`, but that is not an invalidating action, because the loan is not on a place that overlaps with with `x`, the operand of the drop.

The law of exclusivity is enforced at this point. A new mutable loan is an invalidating action on loans that are live at an overlapping place. A new shared loan is an invalidating action on mutable loans that are live at an overlapping place. Additionally, storing to variables is always an invalidating action when there is any loan, shared or mutable, on an overlapping place.

2.2.6 Lifetime error reporting

The borrow checker is concerned with invalidating actions on in-scope loans. There are three instructions at play:

- **(B)** The creation of the loan. This is the lvalue-to-borrow operation, equivalent to an addressof (&).
- **(A)** The action that invalidates the loan. That includes taking a mutable borrow on a place with a shared loan, or taking any borrow or writing to a place with a mutable borrow. These actions could lead to use-after-free bugs.
- **(U)** A use that extends the liveness of the borrow past the point of the invalidating action.

view.cxx – (Compiler Explorer)

```
#feature on safety
#include <std2.h>

int main() safe {
  std2::string s("Hello safety");

  // (B) - borrow occurs here.
  std2::string_view view = s;

  // (A) - invalidating action
  s = std2::string("A different string");

  // (U) - use that extends borrow
```

```
  std2::println(view);
}
```

```
$ circle view.cxx
safety: during safety checking of int main() safe
  borrow checking: view.cxx:14:11
    println(view);
            ^
  use of view depends on expired loan
  drop of s between its shared borrow and its use
  invalidating operation at view.cxx:11:5
    s = "A different string";
      ^
  loan created at view.cxx:8:28
    std2::string_view view = s;
                             ^
```

Circle tries to identify all three of these points when forming borrow checker errors. Usually they're printed in bottom-to-top order. That is, the first source location printed is the location of the use of the invalidated loan. Next, the invalidating action is categorized and located. Lastly, the creation of the loan is indicated.

The invariants that are tested are established with a network of lifetime constraints. It might not be the case that the invalidating action is obviously related to either the place of the loan or the use that extends the loan. More completely describing the chain of constraints could help users diagnose borrow checker errors. But there's a fine line between presenting an error like the one above, which is already pretty wordy, and overwhelming programmers with information.

2.2.7 Lifetime constraints on called functinos

Borrow checking is easiest to understand when applied to a single function. The function is lowered to a control flow graph, the compiler assigns regions to loans and borrow variables, emits lifetime constraints where there are assignments, iteratively grows regions until the constraints are solved, and walks the instructions, checking for invalidating actions on loans in scope. Within the definition of the function, there's nothing it can't analyze. The complexity arises when passing and receiving borrows through function calls.

Whole program analysis is not practical. In order to extend lifetime safety guarantees outside of single functions, we have to introduce *lifetime contracts* on function boundaries that are satisfied by both caller and callee. These contracts are noted by *lifetime parameters*.

get.cxx – (Compiler Explorer)

```
#feature on safety

auto get_x/(a, b)(const int^/a x, const int^/b y) -> const int^/a {
  return x;
}
auto get_y/(a, b)(const int^/a x, const int^/b y) -> const int^/b {
  return y;
}

int main() {
  const int^ ref1;
  const int^ ref2;
  int x = 1;
  {
    int y = 2;
    ref1 = get_x(x, y);
```

```
    ref2 = get_y(x, y);
  }
  int val1 = *ref1;  // OK.
  int val2 = *ref2;  // Borrow checker error.
}
```

```
$ circle get.cxx
safety: get.cxx:20:14
  int val2 = *ref2;
             ^
use of *ref2 depends on expired loan
drop of y between its shared borrow and its use
y declared at get.cxx:15:9
    int y = 2;
        ^
loan created at get.cxx:17:21
    ref2 = get_y(x, y);
                 ^
```

`get_x` takes two shared borrow parameters and returns a shared borrow. The return type is marked with the lifetime parameter `/a`, which corresponds with the lifetime argument on the returned value `x`. `get_y` is declared to return a shared borrow with a lifetime associated with the parameter `y`. Since we're not specifying an *outlives-constraint* between the lifetime parameters, the function bodies can't assume anything about how these lifetimes relate to each other. It would be ill-formed for `get_x` to return `y` or `get_y` to return `x`.

This code raises a borrow checker error when dereferencing `*ref2`. As a human with access to the definiton of `get_y`, it's obvious that the use of `ref2` is a use-after-free defect. We can see that `get_y` returns `y`, and `y` goes out of scope by the time `ref2` is used. But static analysis can't see this–when `main` is being analyzed, it doesn't look into the definitions of other functions. Instead, it's the lifetime parameterization on the `get_y` declaration which informs the borrow checker. The call generates a constraint: at the point of the call, the lifetime on the borrow of the second function parameter outlives the lifetime on the borrow of the result object. That means that the region on the loan `^y` in the `get_y` call *outlives* the region on the result object `ref2` at point of the call. As long as `ref2` is live, so is the loan on `y`. That's the transitive property and the result of solving the constraint equation. When `y` goes out of scope, the borrow checker raises an error, because a loan on it is still in scope.

```
    int main() {
P0:   const int^ ref1;        // ref1 is 'R0
P1:   const int^ ref2;        // ref2 is 'R1
P2:   int x = 1;
      {
        int y = 2;
P3:     <loan R2> = ^const x; // ^const x is 'R2
P4:     <loan R3> = ^const y; // ^const y is 'R3
P5:     ref1 = get_x(<loan R2>, <loan R3>);
        // /a is 'R4.
        // /b is 'R5.
        // 'R2 : 'R4 @ P5
        // 'R3 : 'R5 @ P5
        // 'R4 : 'R0 @ P5

P6:     <loan R6> = ^const x; // ^const x is 'R6
P7:     <loan R7> = ^const y; // ^const y is 'R7
P8:     ref2 = get_y(<loan R6>, <loan R7>);
        // /a is 'R8'.
        // /b is 'R9'.
```

```
        // 'R6 : 'R8 @ P8
        // 'R7 : 'R9 @ P8
        // 'R9 : 'R1 @ P8

P9:     drp y
      }
P10:  int val1 = *ref1;  // OK.
P11:  int val2 = *ref2;  // Borrow checker error.

P12:  drp x
    }
```

For every function call, the lifetime parameters of that function are assigned regions distinct from the regions of the arguments and result object. This is done to more cleanly support *outlives-constraints*. The regions of the function arguments outlive their corresponding lifetime parameters, and the lifetime parameters outlive their corresponding result object regions. This creates a chain of custody in from the arguments, through the function's lifetime parameters, and out through the result object. The caller doesn't have to know the definition of the function, because it upholds the constraints at the point of the call, and the callee upholds the constraints in the definition of the function.

Let's solve for the regions of the four loans:

```
'R2 = { 4, 5, 6, 7, 8, 9, 10 }
'R3 = { 5 }
'R6 = { 7, 8, }
'R7 = { 8, 9, 10, 11 }
```

The drops of `x` and `y` are the potentially invalidating actions. `y` goes out of scope at P9, and the loans with regions `R2` and `R7` are live at P9 (because they have 9 in their sets). The 'R2 loan borrows variable `x`, which is non-overlapping with the drop operand `y`, so it's not an invalidating action. The 'R7 loan borrows variable `y`, which is overlapping with the drop operand `y`, so we get a borrow checker error. The drop of `x` is benign, since no loan is live at P12.

2.2.8 Lifetime parameters

On function types and function declarators, the *lifetime-parameters-list* goes right after the *declarator-id*.

```
// A function type declarator.
using F1 = void/(a, b)(int^/a, int^/b) safe;

// A function declaration with the same type
void f1/(a, b)(int^/a, int^/b) safe;
```

Borrows are first-class *lifetime binders*. You can also declare class and class templates that bind lifetimes by putting *lifetime-parameters-lists* after their *declarator-ids*.

```
template<typename char_type>
class basic_string_view/(a);
```

All types with lifetime binders named in function declarations and data members must be qualified with lifetime arguments. This helps define the lifetime contract between callers and callees. Often it's necessary to specify requirements between lifetime parameters. These take the form of *outlives-constraints*. They're specified in a *where-clause* inside the *lifetime-parameter-list*:

```
using F2 = void/(a, b where a : b)(int^/a x, int^/b y) safe;
```

Rust chose a different syntax, mixing lifetime parameters and type parameters into the same parameter list. Coming from C++, where templates supply our generics, I find this misleading. Lifetime parameters are

nothing like template parameters. A function with lifetime parameters isn't really a generic function. Lifetime parameters are never substituted with any kind of concrete lifetime argument. Instead, the relationship between lifetime parameters, as deduced from *outlives-constraints*, implied bounds and variances of function parameters, establishes constraints that are used by the borrow checker at the point of call.

```
// A Rust declaration.
fn f<'a, 'b, T>(x: &'a mut T, y: &'b T) where 'a : 'b { }
```

```
// A C++ declaration.
template<typename T>
void f/(a, b where a : b)(T^/a x, const T^/b y) { }
```

Rust uses single-quotes to introduce lifetime parameters and lifetime arguments. That's not a workable choice for us, because C supports multi-character literals. This cursed feature, in which literals like `'abcd'` evaluate to constants of type `int`, makes lexing Rust-style lifetime arguments very messy.

```
template<typename T>
void f/(a, b where a : b)(T^/a x, const T^/b y) { }

int main() {
  int x = 1;
  int y = 2;
  f(^x, y);
}
```

```
    int main() {
P0:   int x = 1;
P1:   int y = 2;
P2:   <loan R0> = ^x;
P3:   <loan R1> = ^const y;
P4:   f(<loan R0>, <loan R1>);
      // /a is 'R2
      // /b is 'R3
      // 'R2 : 'R3 @ P4 - `where a : b`
      // 'R0 : 'R2 @ P4
      // 'R1 : 'R3 @ P4
    }
```

The *where-clause* establishes the relationship that `/a` outlives `/b`. Does this mean that the variable pointed at by `x` goes out of scope later than the variable pointed at by `y`? No, that would be too straight-forward. This declaration emits a *lifetime-constraint* at the point of the function call. The regions of the arguments already constrain the regions of the lifetime parameters. The *where-clause* constrains the lifetime parameters to each other. `f`'s outlives constraint is responsible for the lifetime constraint `'R2 : 'R3 @ P4`.

Lifetime parameters and *where-clauses* are a facility for instructing the borrow checker. The obvious mental model is that the lifetimes of references are connected to the scope of the objects they point to. But this is not accurate. Think about lifetimes as defining rules that can't be violated, with the borrow checker looking for contradictions of these rules.

2.2.9 Destructors and phantom data

Safe C++ includes an operator to call the destructor on local objects.

— `drp x` - call the destructor on an object and set as uninitialized.

Local objects start off uninitialized. They're initialized when first assigned to. Then they're uninitialized again when relocated from. If you want to *destruct* an object prior to it going out of scope, use *drp-expression*. Unlike Rust's `drop` API,[drop] this works even on objects that are pinned or are only potentially initialized (was

uninitialized on some control flow paths) or partially initialized (has some uninitialized subobjects). This is useful feature for the affine type system. But it's presented in the borrow checking section. Why? Because the operand of a *drp-expression* isn't a normal use, in the borrow checking sense. It's a special *drop use*.

drop.rs – (Compiler Explorer)

```
fn main() {
  let mut v = Vec::<&i32>::new();
  {
    let x = 101;
    v.push(&x);
  }

  // Error: `x` does not live long enough
  drop(v);
}
```

```
$ rustc drop.rs
error[E0597]: `x` does not live long enough
 --> drop.rs:5:12
  |
4 |     let x = 101;
  |         - binding `x` declared here
5 |     v.push(&x);
  |            ^^ borrowed value does not live long enough
6 |   }
  |   - `x` dropped here while still borrowed
...
9 |   drop(v);
  |        - borrow later used here
```

Rust's `drop` API is a normal function. It performs the usual non-drop-use of all lifetimes on the operand. A vector with dangling lifetimes would pass the normal drop check, but when used as an argument to `drop` call, makes the program ill-formed. The borrow checker in `main` doesn't know that the body of the `drop` call won't load or store a value through that dangling reference.

drop.cxx – (Compiler Explorer)

```
#feature on safety
#include <std2.h>

int main() safe {
  std2::vector<const int^> vec { };
  {
    int x = 101;
    mut vec.push_back(x);
  }

  // Okay. vec has dangling borrows, but passes the drop check.
  drp vec;
}
```

The *drp-expression* in Safe C++ isn't implemented with a function call. It's a first-class feature and only performs a *drop use* of the operand's lifetimes. Why is it safe to drop a `std2::vector` that holds dangling borrows? That's the result of careful interplay between `__phantom_data` and `drop_only`.

A user-defined destructor uses the lifetimes associated with all class data members. However, container types that store data in heap memory, such as box and vector, don't store their data as data members, but instead

keep pointers into keep memory. In order to expose data for the drop check, container types must have phantom data[phantom-data] to declare a sort of phantom data.

```
template<typename T+>
struct Vec {
  ~Vec() safe;

  // Pointer to the data. Since T* has a trivial destructor, template
  // lifetime parameters of T are not used by Vec's dtor.
  T* _data;

  // For the purpose of drop check, declare a member of type T.
  // This is a phantom data member that does not effect data
  // layout or initialization.
  T __phantom_data;
};
```

Including a `__phantom_data` member informs the compiler that the class may use objects of type `T` inside the user-defined destructor. The drop check expects user-defined destructors to be maximally permissive with its data members, and that it can use any of the class's lifetime parameters. For a `Vec<const int^>` type, the destructor could dereference its elements and print out their values before freeing their storage. The `__phantom_data` declares that a type will be used like a data member in the user-defined destructor. Without this annotaton, the destructor could load from a dangling reference and cause a use-after-free bug.

In order to permit *dangling references* in containers, destructors opt into the `[[unsafe::drop_only(T)]]` attribute. It's available in Rust as the `#[may_dangle]` attribute.[may-dangle] This attribute is a promise that the user-defined destructor will only destroy the type in attribute's operand. Permitting dangling references in a *drop use* is a crucial feature. Without it, objects may squabble over destruction order, resulting in code that fights the borrow checker.

drop2.cxx – (Compiler Explorer)

```
#feature on safety

template<typename T+, bool DropOnly>
struct Vec {
  Vec() safe { }

  [[unsafe::drop_only(T)]] ~Vec() safe requires(DropOnly);
                           ~Vec() safe requires(!DropOnly);

  void push(self^, T rhs) safe;

  // T is a phantom data member of Vec. The non-trivial dtor will
  // use the lifetimes of T, raising a borrow checker error if
  // T is not drop_only.
  T __phantom_data;
};

template<typename T, bool DropOnly>
void test() safe {
  Vec<const T^, DropOnly> vec { };
  {
    T x = 101;
    mut vec.push(^x);
  }
```

```
  // Use of vec is ill-formed due to drop of x above.
  // int y = 102;
  // mut vec.push(^y);

  // Should be ill-formed due to drop check.
  drp vec;
}

int main() safe {
  test<int, true>();
  test<int, false>();
}
```

```
$ circle drop2.cxx
safety: during safety checking of void test<int, false>() safe
  borrow checking: drop2.cxx:31:3
    drp vec;
    ^
  use of vec depends on expired loan
  drop of x between its mutable borrow and its use
  x declared at drop2.cxx:22:7
      T x = 101;
        ^
  loan created at drop2.cxx:23:18
      mut vec.push(^x);
                   ^
```

This example explores the effects of the `[[unsafe::drop_only(T)]]` attribute. As of C++ 20, class templates can overload destructors based on *requires-specifier*. We want to provide both normal and `drop_only` destructors to test borrow checking.

A `T __phantom_data` member indicates that `T` will be destroyed as part of the user-defined destructor. The user-defined destructor is conditionally declared with the `[[unsafe::drop_only(T)]]` attribute, which means that the destructor body will only use `T`'s lifetime parameters as part of a drop, and not any other use.

The point of this mechanism is to make drop order less important by permitting the drop of containers that hold dangling references. Declare a `Vec` specialized on a `const int^`. Push a shared borrow to a local integer declaration, then make the local integer go out of scope. The `Vec` now holds a dangling borrow. This should compile, as long as we don't use the template lifetime parameter associated with the borrow. That would produce a use-after-free borrow checker error.

When `DropOnly` is true, the destructor overload with the `[[unsafe::drop_only]]` attribute is instantiated. The *drp-expression* in `test` doesn't automatically use the lifetimes of its operand. Instead, the drop check considers lifetime parameters of the operand that are drop used. Due to the `[[unsafe::drop_only(T)]]` attribute, the data member is only a *drop use* rather than a normal use. And the drop use of a borrow is nothing: it's a trivial destructor. We can don't care about lifetimes of borrows when dropping them, because we're doing operations that can cause soundness problems. The *drp-expression* destroys the `Vec` without a borrow checker violation. The programmer is making an unsafe promise not to do anything with `T`'s template lifetime parameters inside `~Vec` other than use it to drop `T`.

When `DropOnly` is false, the destructor overload without the attribute is instantiated. In this case, the user-defined destructor is assumed to use all of the class's lifetime parameters outside. If the `Vec` held a borrow to out-of-scope data, as it does in this example, loading that data through the borrow would be a use-after-free defect. To prevent this unsound behavior, the compiler uses all the class's lifetime parameters when its user-defined destructor is called. This raises the borrow checker error, indicating that the *drp-expression* depends on an expired loan on `x`.

The `drop_only` attribute is currently `unsafe` because it depends on the programmer following through with a promising not to do anything with the operand other than to destruct it. We plan to make this attribute safe. The compiler should be able to monitor the user-defined destructor for use of lifetimes associated with `T` outside of drops and raise borrow checker errors. However, that may require extending `drop_only` to all functions, not just destructors. `std::destruct_at` involves a non-drop use of the parameter type. But a `drop_only`-attributed `std2::destruct_at` would only permit and qualify as a drop use.

2.2.10 Lifetime canonicalization

```
#feature on safety

// Two distinct lifetimes with no constraint.
using F1 = void/(a, b)(int^/a, int^/b) safe;

// These are the same.
using F2 = void/(a, b where a : b)(int^/a, int^/b) safe;
using F3 = void/(a, b where b : a)(int^/b, int^/a) safe;
static_assert(F2 == F3);

// They differ from F1, due to the outlives-constraint.
static_assert(F1 != F2);

// These are the same.
using F4 = void/(a, b where a : b, b : a)(int^/a, int^/b) safe;
using F5 = void/(a)                      (int^/a, int^/a) safe;
static_assert(F4 == F5);

// They differ from F2, due to the constraint going both directions.
static_assert(F2 != F4);
```

Lifetime parameterizations are part of the function's type. But different textual parameterizations may still result in the same type! `F1` and `F2` have different parameterizations and are different types. `F2` and `F3` have different parameterizations yet are the same type. Likewise, `F4` and `F5` are the same type, even though `F4` has two lifetime parameters and two outlives constraints.

The compiler maps all non-dependent types to canonical types. When comparing types for equality, it compares the pointers to their canonical types. This is necessary to support typedefs and alias templates that appear in functions–we need to strip away those inessential details and get to the canonical types within. Lifetime parameterizations the user declares also map to canonical pameterizations.

Think about lifetime parameterizations as a directed graph. Lifetime parameters are the nodes and outlives constraints define the edges. The compiler finds the strongly connected components[scc] of this graph. That is, it identifies all cycles and reduces them into SCC nodes. In `F4`, the `/a` and `/b` lifetime parameters constrain one another and are collapsed into the same strongly connected component. The canonical function type is encoded using SCCs as lifetime parameters. After lifetime canonicalization both `F4` and `F5` map to the same canonical type and therefore compare the same.

During the type relation pass that generates lifetime constraints for function calls in the MIR, arguments and result object regions are constrained to regions of the canonical type's SCCs, rather than the lifetime parameters of the declared type. This reduces the number of regions the borrow checker solves for. But the big reason for this process is to permit writing compatible functions when dealing with the quirks of lifetime normalization.

2.2.11 Lifetimes and templates

Templates are specially adapted to handle types with lifetime binders. It's important to understand how template lifetime parameters are invented in order to understand how borrow checking fits with C++'s late-checked

generics.

In the current implementation, class templates feature a variation on the type template parameter: `typename T+` is a lifetime binder parameter. The class template invents an implicit *template lifetime parameter* for each lifetime binder of the argument's type.

```
template<typename T0+, typename T1+>
struct Pair {
  T0 first;
  T1 second;
};

class string_view/(a);
```

The `Pair` class template doesn't have any named lifetime parameters. `string_view` has one named lifetime parameter, which constrains the pointed-at string to outlive the view. The specialization `Pair<string_view, string_view>` invents a template lifetime parameter for each view's named lifetime parameters. Call them `T0.0.0` and `T1.0.0`. `T0.0` is for the 0th parameter pack element of the template parameter `T0`. `T0.0.0` is for the 0th lifetime binder in that type. If `string_view` had a second named parameter, the compiler would invent two more template lifetime parameters: `T0.0.1` and `T1.0.1`.

Consider the transformation when instantiating the definition of `Pair<string_view/a, string_view/b>`:

```
template<>
struct Pair/(T0.0.0, T1.0.0)<string_view/T0.0.0, string_view/T1.0.0> {
  string_view/T0.0.0 first;
  string_view/T1.0.0 second;
};
```

The compiler ignores the lifetime arguments `/a` and `/b` and replaces them with the template lifetime parameters `T0.0.0` and `T1.0.0`. This transformation deduplicates template instantiations. We don't want to instantiate class templates for every lifetime argument on a template argument type. That would be an incredible waste of compute and result in enormous code bloat. Those lifetime arguments don't carry data in the same way as integer or string types do. Instead, lifetime arguments define constraints on region variables between different function parameters and result objects. Those constraints are an external concern to the class template being specialized.

Since we replaced the named lifetime arguments with template lifetime parameters during specialization of `Pair`, you have to wonder, what happened to `/a` and `/b`? They're factored out and stuck to the outside of the class template specialization: `Pair<string_view, string_view>/a/b`. Since this specialized `Pair` has two lifetime binders (those two template lifetime parameters), it needs to bind two lifetime arguments. Safe C++ replaces lifetime arguments on template arguments with invented template lifetime parameters and reattaches the lifetime arguments to the specialization.

A class template's instantiation doesn't depend on the lifetimes of its users. `std2::vector<string_view/a>` is transformed to `std2::vector<string_view/T0.0.0>/a`. `T0.0.0` is the implicitly declared lifetime parameter, which becomes the lifetime argument on the template argument, and `/a` is the user's lifetime argument that was hoisted out of the *template-argument-list* and appended to the class specialization.

In the current safety model, this transformation only occurs for bound lifetime template parameters with the `typename T+` syntax. It's not done for all template parameters, because that would interfere with C++'s partial and explicit specialization facilities.

```
template<typename T0, typename T1>
struct is_same {
  static constexpr bool value = false;
};
template<typename T>
struct is_same<T, T> {
```

```
  static constexpr bool value = true;
};
```

Should `std::is_same<string_view, string_view>::value` be true or false? Of course it should be true. But if we invent template lifetime parameters for each lifetime binder in the *template-argument-list*, we'd get something like `std::is_same<string_view/T0.0.0, string_view/T1.0.0>::value`. Those arguments are different types, and they wouldn't match the partial specialization.

When specializing a `typename T` template parameter, lifetimes are stripped from the template argument types. They become unbound types. When specializing a `typename T+` parameter, the compiler creates fully-bound types by implicitly adding placeholder arguments `/_` whenever needed. When a template specialization is matched, the lifetime arguments are replaced with the specialization's invented template lifetime parameters and the original lifetime arguments are hoisted onto the specialization.

You might want to think of this process as extending an electrical circuit. The lifetime parameters outside of the class template are the source, and their usage on the template arguments are the return. When we instantiate a class template, the invented template lifetime parameters become a new source, and their use as lifetime arguments in the data members of the specialization are a new return. The old circuit is connected to the new one, where the outer lifetime arguments lead into the new template lifetime parameters.

```
Foo<string_view/a>
  Source: /a
  Return: string_view

transforms to:

Foo<string_view/T0.0.0>/a
  Source: /a
  Return: T0.0.0 (user-facing)
  Source: T0.0.0 (definition-facing)
  Return: string_view
```

Because Rust doesn't support partial or explicit specialization of its generics, it has no corresponding distinction between type parameters which bind lifetimes and type parameters which don't. There's nothing like `is_same` that would be confused by lifetime arguments in its operands.

Deciding when to use the `typename T+` parameter kind on class templates will hopefully be straight-forward. If the class template is a container and the parameter represents a thing being contained, use the new syntax. If the class template exists for metaprogramming, like the classes found in `<type_traits>`, it's probably uninterested in bound lifetimes. Use the traditional `typename T` parameter kind.

The mechanism for generating template lifetime parameters feels right, but its presentation to the user is still in flux. An alternative treatment would optionally permit bound lifetimes in `typename T` template parameters, but require users to explicitly bind a placeholder lifetime argument `/_` if they want bound types:

```
// Types are unbound
static_assert(std::is_same_v<int^, int^> == true);

// Template arguments have bound placeholder lifetimes. These get replaced
// with template lifetime parameters, and the resulting types compare
// differently.
static_assert(std::is_same_v<int^/_, int^/_> == false);

template<typename T>
struct Obj {
  // Ill-formed if T has bound lifetimes.
  static_assert(std::is_same_v<T, int^>);    // #1
```

```
  // /0 strips lifetime arguments from types.
  static_assert(std::is_same_v<T/0, int^>);  // #2
};

Obj<int^/_> obj1;  // Ill-formed at #1. Well-formed at #2.
Obj<int^>   obj2;  // Well-formed at #1 and #2.
```

2.2.12 Lifetime normalization

Lifetime normalization conditions the lifetime parameters and lifetime arguments on function declarations and function types. After normalization, function parameters and return types have fully bound lifetimes. Their lifetime arguments always refer to lifetime parameters of the function, and not to those of any other scope, such as the containing class.

Lifetime elision assigns lifetime arguments to function parameters and return types with unbound lifetimes. * If the type with unbound lifetimes is the enclosing class of a member function, the lifetime parameters of the enclosing class are used for elision. * If a function parameter has unbound lifetimes, *elision lifetime parameters* are invented to fully bind the function parameter. * If a return type has unbound lifetimes and there is a `self` parameter, it is assigned the lifetime argument on the `self` parameter, if there is just one, or elision is ill-formed. * If a return type has unbound lifetimes and there is one lifetime parameter for the whole function, the return type is assigned that, or elision is ill-formed.

In addition to elision, lifetime normalization substitutes type parameters in *outlives-constraints* with all corresponding lifetime arguments of the template arguments.

```
template<typename T1, typename T2, typename T3>
void func/(where T1:static, T2:T3)(T1 x, T2 y, T3 Z);

&func<int^, const string_view^, double>;
```

During normalization of this function specialization, the *outlives-constraint* is rebuilt with the substituted lifetime arguments on its type parameters. The template lifetime parameter on `int^` outlives `/static`. Both lifetime parameters bound on `const string_view^` (one for the outer borrow, one on the string_view) outlive the lifetime parameters bound on the `double`. But `double` isn't a lifetime binder. Since it contributes no lifetime parameters, the `T2:T3` type constraint is dropped during normalization.

2.3 Explicit mutation

Reference binding convention is important in the context of borrow checking. Const and non-const borrows differ by more than just constness. By the law of exclusivity, users are allowed multiple live shared borrows, but only one live mutable borrow. C++'s convention of always preferring non-const references would tie the borrow checker into knots, as mutable borrows don't permit aliasing. This is one reason why there's no way to borrow check existing C++ code: standard conversions are too permissive and contribute to mutable aliasing.

Unlike in Standard C++, expressions in this object model can have reference types. Naming a reference object yields an lvalue expression with reference type, rather than implicitly dereferencing the reference and giving you an lvalue to the pointed-at thing. Dereferencing legacy references is unsafe. If references were implicitly dereferenced, you'd never be able to use them in safe contexts. In Safe C++, references are more like pointers, and may be passed around in safe contexts, but not dereferenced.

Rather than binding the mutable overload of functions by default, Safe C++ prefers binding const overloads. It prefers binding shared borrows to mutable borrows. Shared borrows are less likely to bring borrow checker errors. To improve reference binding precision, the relocation object model takes a new approach to references. Standard conversions bind const borrows and const lvalue references to lvalues of the same type, as they always have. But standard conversions won't bind mutable borrows and mutable lvalue references. Those require an opt-in.

```
struct Obj {
  const int& func() const;   // #1
        int& func();         // #2
};

void func(Obj obj) {
  // In ISO C++, calls overload #2.
  // In Safe C++, calls overload #1.
  obj.func();

  // In Safe C++, these all call overload #2.
  (&obj).func();
  obj&.func();
  mut obj.func();
}
```

In Safe C++, the standard conversion will not bind a mutable borrow or mutable lvalue reference. During overload resolution for `obj.func()`, candidate #2 fails, because the compiler can't bind the object parameter type `Obj&` to the object expression `lvalue Obj`. But candidate #1 is viable, because the standard conversion can still bind the object parameter type `const Obj&` to the object expression `lvalue Obj`.

The subsequent statements call candidate #1 by explicitly requesting mutation. `&obj` is syntax for creating a `prvalue Obj&` from the `lvalue Obj` operand. We can call the `func` member function on that reference. `obj&.func()` uses a special postfix syntax that alleviates the need for parentheses. Finally, the `mut` keyword puts the remaining operators of the *cast-expression* into the *mutable context*. In the mutable context, standard conversions to mutable borrows and mutable lvalue references are enabled. Overload resolution finds both candidates #1 and #2 viable, and chooses #2 because the mutable reference outranks the const reference.

Here's a list of *unary-operators* for taking borrows, lvalue and rvalue references, and pointers to lvalues.

- `^x` - mutable borrow to `x`
- `^const x` - shared borrow to `x`
- `&x` - lvalue reference to `x` (convertible to pointer)
- `&const x` - const lvalue reference to `x`
- `&&x` - rvalue reference to `x`
- `addr x` - pointer to `x`
- `addr const x` - const pointer to `x`

While the motivation of this design is to pacify the borrow checker by preferring shared reference binding, a desirable side effect is that **all mutations are explicit**. You don't have to wonder about side-effects. If you're passing arguments to a function, and you don't see `mut`, `^`, `&` or `&&` before it, you know the argument won't be modified by that function.

```
#feature on safety
#include <iostream>

void f1(int^);
void f2(const int^);

void f3(int&);
void f4(const int&);

void f5(int&&);

int main() {
  int x = 1;
```

```
  // Explicit ^ for mut borrow required.
  f1(^x);
  f1(mut x);

  // Standard conversion or explicit ^const for shared borrow.
  f2(x);
  f2(^const x);

  // Explicit & for lvalue ref required.
  f3(&x);
  f3(mut x);

  // Standard conversion or explicit &const for const lvalue ref.
  f4(x);
  f4(&const x);

  // Explicit rvalue ref.
  f5(&&x);

  // Explicit & for non-const lvalue reference operands.
  &std::cout<< "Hello mutation\n";
}
```

In Rust, types and declarations are const by default. In C++, they're mutable by default. But by supporting standard conversions to const references, reference binding in Safe C++ is less noisy than the Rust equivalent, while being no less precise.

```
// Rust:
f(&mut x);    // Pass by mutable borrow.
f(&x);        // Pass by shared borrow.

// C++:
f(^x);        // Pass by mutable borrow.
f(x);         // Pass by shared borrow.
f(^const x);  // Extra verbose -- call attention to it.

f(&x);        // Pass by mutable lvalue ref.
f(x);         // Pass by const lvalue ref.
f(&const x);  // Extra verbose -- call attention to it.
```

2.3.1 The mutable context

The mutable context is the preferred way to express mutation. In a sense it returns Safe C++ to legacy C++'s default binding behavior. Use it at the start of a *cast-expression* and the mutable context lasts for all subsequent higher-precedence operations. In the mutable context, standard conversion may bind mutable borrows and mutable lvalue references to lvalue operands.

mut.cxx – (Compiler Explorer)

```
#feature on safety
#include <std2.h>

int main() safe {
  std2::vector<size_t> A { }, B { };
```

```
  // Get a shared borrow to an element in B.
  // The borrow is live until it is loaded from below.
  const size_t^ b = B[0];

  // A[0] is in the mutable context, so A[0] is the mutable operator[].
  // B[0] is outside the mutable context, so B[0] is the const operator[].
  // No borrow checking error.
  mut A[0] += B[0];

  // Keep b live.
  size_t x = *b;
}
```

Write the `mut` token before the *cast-expression* you want to mutate. *cast-expressions* include high-precedence unary expressions. Lower-precedence binary expressions aren't in the scope of the mutable context. In this example, the mutable context applies only to the left-hand side of the assignment. `A[0]` chooses the mutable overload of `vector::operator[]` and `B[0]` chooses the const overload of `vector::operator[]`. The example takes a shared borrow on `B` and uses it at the end of `main` in order to trap a mutable borrow to `B` at the statement with the compound assignment.

`mut A[0] += B[0]` mutable binds `A` to get the mutable overload of `operator[]`. `B` is on the right-hand side of the assignment, which is outside the scope of the mutable context.

mut2.cxx – (Compiler Explorer)

```
#feature on safety
#include <std2.h>

int main() safe {
  std2::vector<size_t> A { }, B { };

  // Get a shared borrow to an element in B.
  // The borrow is live until it is loaded from below.
  const size_t^ b = B[0];

  // A is in the mutable context for its operator[] call.
  // B is not in the mutable conetxt for its operator[] call.
  // No borrow checker error.
  mut A[B[0]] += 1;

  // Keep b live.
  size_t x = *b;
}
```

The mutable context is entered at the start of a *cast-expression* with the `mut` token. It exists for subsequent high-precedence *unary-expression*, *postfix-expression* and *primary-expression* operations. But it doesn't transer into subexpressions of these. The index operand of the subscript operator matches the *expression-list* production, which is lower precedence than *cast-expression*, so it's not entered with the mutable context. `B[0]` is bound with a shared borrow. One would have to write `mut A[mut B[0]] += 1;` to bind a mutable borrow on both objects.

The mutable context explicitly marks points of mutation while letting standard conversions worry about the details. Its intent is to reduce cognitive load on developers compared with using the reference operators.

2.4 Relocation object model

A core enabling feature of Safe C++ is its new object model. It supports relocation/destructive move of local objects, which is necessary for satisfying type safety.

In Rust, objects are *relocated by default*. Implicit relocation is too surprising for C++ users. We're more likely to have raw pointers and legacy references tracking objects, and you don't want to pull the storage out from under them, at least not without some clear token in the source code. That's why Safe C++ includes *rel-expression* and *cpy-expression*.

— `rel x` - relocate `x` into a new value. `x` is set as uninitialized.
— `cpy x` - copy construct `x` into a new value. `x` remains initialized.

In line with C++'s goals of *zero-cost abstractions*, we want to make it easy for users to choose the more efficient option between relocaton and copy. If the expression's type is trivially copyable and trivially destructible, it'll initialize a copy from an lvalue. Otherwise, the compiler prompts for a `rel` or `cpy` token to resolve the copy initialization. You're not going to accidentally hit the slow path or the mutable path. Opt into mutation. Opt into non-trivial copies. opy, or do you want to relocate?

You've noticed the nonsense spellings for some of these keywords. Why not call them `relocate`, `copy` and `drop`? Alternative token spelling avoids shadowing these common identifiers and improves results when searching code or the web.

move.cxx – (Compiler Explorer)

```
#feature on safety
#include <std2.h>

int main() safe {
  // Objects start off uninitialized. A use of an uninitialized
  // object is ill-formed.
  std2::string s;

  // Require explicit initialization.
  s = std2::string("Hello ");

  // Require explicit mutation.
  mut s.append("World");

  // Require explicit relocation.
  std2::string s2 = rel s;  // Now s is uninitialized.

  // Require explicit non-trivial copy.
  std2::string s3 = cpy s2;

  // `Hello World`.
  println(s3);
}
```

Initialization, copy, move and mutation are explicit. Deleting the `rel` token in this example breaks translation and prompts for guidance:

```
  std2::string s2 = s;  // Now s is uninitialized.
                    ^
implicit copy from lvalue std2::string not allowed
specify cpy or rel or add the [[safety::copy]] attribute
```

A standard conversion from `lvalue std2::string` to `std2::string` is found. But without the user's help, the compiler doesn't know how satsify the conversion: does it call the copy constructor, which is expensive, or does it perform relocation, which is cheap but changes the operand by ending its scope?

2.4.1 Tuples, arrays and slices

Using an uninitialized or potentially uninitialized object raises a compile-time error. But initialization analysis only accounts for the state of local variables that are directly named and not part of a dereference. Local objects and their subobjects are *owned places*. Initialization analysis allocates flags for owned places, so that it can track their initialization status and error when there's a use of a place that's not *definitely initialized*. It's not possible for the compiler to chase through references and function calls to find initialization data for objects potentially owned by other functions–that's a capability of *whole program analysis*, which is beyond of the scope of this proposal.

rel1.cxx – (Compiler Explorer)

```
#feature on safety
#include <tuple>

struct Pair {
  int x, y;
};

Pair g { 10, 20 };

int main() {
  // Relocate from an std::tuple element.
  auto tup = std::make_tuple(5, 1.619);
  int x = rel *get<0>(&tup);

  // Relocate from runtime subscript.
  int data[5] { };
  int index = 1;
  int y = rel data[index];

  // Relocate from subobject of a global.
  int gy = rel g.y;
}
```

```
$ circle rel1.cxx
safety: rel1.cxx:13:15
  int x = rel *get<0>(&tup);
              ^
rel operand does not refer to an owned place
an owned place is a local variable or subobject of a local variable
the place involves a dereference of int&

safety: rel1.cxx:18:19
  int y = rel data[index];
                  ^
rel operand does not refer to an owned place
an owned place is a local variable or subobject of a local variable
the place involves a subscript of int[5]

safety: rel1.cxx:21:17
  int gy = rel g.y;
               ^
rel operand does not refer to an owned place
an owned place is a local variable or subobject of a local variable
g is a non-local variable declared at rel1.cxx:8:6
```

```
Pair g { 10, 20 };
     ^
```

This example shows that you can't relocate through the reference returned from `std::get`. You can't relocate through a dynamic subscript of an array. You can't relocate the subobject of a global variable. You can only relocate or drop *owned places*.

If we can't relocate through a reference, how do we relocate through elements of `std::tuple`, `std::array` or `std::variant`? Unless those become magic types with special compiler support, you can't. Those standard containers only provide access to their elements through accessor functions which return references. Subobjects behind references are not *owned places*.

We address the defects in C++'s algebraic types by including new first-class tuple, array and choice types. Safe C++ is still compatible with legacy types, but due to their non-local element access, relocation from their subobjects is not currently implemented. Relocation is important to type safety, because many types prohibit default states, making C++-style move semantics impossible. Either relocate your object, or put it in an `optional` from which it can be unwrapped.

tuple1.cxx – (Compiler Explorer)

```
#feature on safety
#include <std2.h>          // Pull in the definition of std2::tuple.

using T0 = ();             // Zero-length tuple type.
using T1 = (int, );        // One-length tuple type.
using T2 = (int, double);  // Longer tuples type.

// Nest at your leisure.
using T3 = (((int, double), float), char);

int main() {
   // Zero-length tuple expression.
  auto t0 = (,);
  static_assert(T0 == decltype(t0));

  // One-length tuple expression.
  auto t1 = (4, );
  static_assert(T1 == decltype(t1));

  // Longer tuple expression.
  auto t2 = (5, 3.14);
  static_assert(T2 == decltype(t2));

  // Nest tuples.
  auto t3 = (((1, 1.618), 3.3f), 'T');
  static_assert(T3 == decltype(t3));

  // Access the 1.618 double field:
  auto x = t3.0.0.1;
  static_assert(double == decltype(x));
}
```

The new first-class tuple type is syntaxed with comma-separated lists of types inside parentheses. Tuple expressions are noted with comma-separated lists of expressions inside parentheses. You can nest them. You can access elements of tuple expressions by chaining indices together with dots. Tuple fields are accessed with the customary special tuple syntax: just write the element index after a dot, eg `tup.0`.

Use `circle -print-mir` to dump the MIR of this program.

```
15  Assign _4.1 = 84
16  Commit _4 (((int, double), float), char)
17  InstStart _5 double
18  Assign _5 = use _4.0.0.1
19  InstEnd _5
20  InstEnd _4
```

The assignment `t3.0.0.1` lowers to `_4.0.0.1`. This is a place name of a local variable. Importantly, it doesn't involve dereferences, unlike the result of an `std::get` call. It's an *owned place* which the compiler is able to relocate out of.

C++'s native array decays to pointers and doesn't support pass-by-value semantics. `std::array` encapsulates arrays to fix these problems and provides an `operator[]` API for consistent subscripting syntax. But `std::array` is broken for our purposes. Since `operator[]` returns a reference, the `std::array`'s elements are not *owned places* and can't be relocated out of.

Safe C++ introduces a first-class pass-by-value array type `[T; N]` and a first-class slice companion type `[T; dyn]`. Subobjects of both the old and new array types with constant indices are *owned places* and support relocation.

Slices have dynamic length and are *incomplete types*. You may form borrows, references or pointers to slices and access through those. These are called *fat pointers* and are 16 bytes on 64-bit platforms. The data pointer is accompanied by a length field.

The new array type, the slice type and the legacy builtin array type panic on out-of-bounds subscripts. They exhibit bounds safety in the new object model. Use unsafe subscripts to suppress the runtime bounds check.

Making `std::pair`, `std::tuple` and `std::array` magic types with native support for relocation is on the short list of language improvements. We hope to incorporate this functionality for the next revision of this proposal. In the meantime, the first-class replacement types provide us with a convenient path forward for developing the safe standard library.

2.4.2 `operator rel`

Safe C++ introduces a new special member function, the *relocation constructor*, written `operator rel(T)`, for all class types. Using the *rel-expression* invokes the relocation constructor. The relocation constructor exists to bridge C++11's move semantics model with Safe C++'s relocation model. Relocation constructors can be:

- User defined - manually relocate the operand into the new object. This can be used for fixing internal addresses, like those used to implement sentinels in standard linked lists and maps.
- `= trivial` - Trivially copyable types are already trivially relocatable. But other types may be trivially relocatable as well, like `box`, `unique_ptr`, `rc`, `arc` and `shared_ptr`.
- `= default` - A defaulted or implicitly declared relocation constructor is implemented by the compiler with one of three strategies: types with safe destructors are trivially relocated; aggregate types use member-wise relocation; and other types are move-constructed into the new data, and the old operand is destroyed.
- `= delete` - A deleted relocation constructor *pins* a type. Objects of that type can't be relocated. A `rel-expression` is a SFINAE failure. Rust uses its `std::Pin`[pin] pin type as a container for structs with with address-sensitive states. That's an option with Safe C++'s deleted relocation constructors. Or, users can write user-defined relocation constructors to update address-sensitive states.

Relocation constructors are always noexcept. It's used to implement the drop-and-replace semantics of assignment expressions. If a relocation constructor was throwing, it might leave objects involved in drop-and-replace in illegal uninitialized states. An uncaught exception in a user-defined or defaulted relocation constructor will panic and terminate.

[P1144R11] proposes tests to automatically classify types that should exhibit trivial relocation based on other properties. The Safe C++ model will need to work through the same issues, but with a richer set of language features to consider.

2.5 Choice types

The new `choice` type is a type safe discriminated union. It's equivalent to Rust's `enum` type. Choice types contain a list of alternatives, each with an optional payload type. `choice` does not replace C++'s `enum` type. It's a different type with different capabilities.

> For an enumeration whose underlying type is fixed, the values of the enumeration are the values of the underlying type. Otherwise, the values of the enumeration are the values representable by a hypothetical integer type with minimal width M such that all enumerators can be represented. The width of the smallest bit-field large enough to hold all the values of the enumeration type is M. *It is possible to define an enumeration that has values not defined by any of its enumerators.*
>
> – dcl.enum

It's not possible to safely exhaustiveness checking on C++ enums, because enums may hold values that aren't among their enumerators.

`choice` types are more rigorously defined. They may only hold values that are specified alternatives. That makes them much easier to reason about, and permits the compiler perform exhaustiveness checking. There's no safe conversion between choice types and their underlying types; programmers must use choice initializers to create new choice objects and *match-expressions* to interrogate them.

choice.cxx

```
#feature on safety

// An enum-like choice that has alternatives but no payloads.
// Unlike enums, it is not permitted to cast between C1 and its underlying
// integral type.
choice C1 {
  A, B, C,
};

// A choice type with payloads for each alternative.
choice C2 {
  i8(int8_t),
  i16(int16_t),
  i32(int32_t),
  i64(int64_t),
};

// A choice type with mixed payload and no-payload alternatives.
choice C3 {
  // default allowed once per choice, on a no-payload alternative.
  default none,
  i(int),
  f(float),
};

int main() safe {
  // choice-initializer uses scope resolution to name the alternative.
  C1 x1 = C1::B();

  // abbreviated-choice-name infers the choice type from the lhs.
  // If there's no payload type, using () is optional.
  C1 x2 = .B();
  C1 x3 = .B;
```

```
  // Create choice objects with initializers.
  C2 y1 = C2::i32(55);
  C2 y2 = .i32(55);

  // If there's a defaulted alternative, the choice type has a default
  // initializer that makes an instance with that value.
  C3 z1 { };
  C3 z2 = C3();
  C3 z3 = C3::none();
  C3 z4 = .none;
}
```

Construct a `choice` type using the qualified *choice-initializer* syntax `ChoiceName::Alternative(args)`. In the context of copy initialization, which include return statements, *mem-initializer*, function arguments, and so on, use the *abbreviated-choice-name*, `.Alternative` or `.Alternative(args)` to create a choice object with less typing. The choice type is inferred from type being initialized.

```
template<class T+, class E+>
choice expected {
  [[safety::unwrap]] ok(T),
  err(E);

  T unwrap(self) noexcept safe {
    return match(self) -> T {
      .ok(t)  => rel t;
      .err(e) => panic("{} is err".format(expected~string));
    };
  }
};

template<class T+>
choice optional
{
  default none,
  [[safety::unwrap]] some(T);

  template<class E>
  expected<T, E> ok_or(self, E e) noexcept safe {
    return match(self) -> expected<T, E> {
      .some(t) => .ok(rel t);
      .none    => .err(rel e);
    };
  }

  T expect(self, str msg) noexcept safe {
    return match(self) -> T {
      .some(t) => rel t;
      .none    => panic(msg);
    };
  }

  T unwrap(self) noexcept safe {
    return match(self) -> T {
      .some(t) => rel t;
```

```
      .none    => panic("{} is none".format(optional~string));
    };
  }

  ...
};
```

Rust uses traits to extend data types with new member functions outside of their definitions. Safe C++ doesn't have that capability, so it's important that choice types support member functions directly. These member functions may internalize pattern matching on their `self` parameters, improving the ergonomics of these type-safe variants.

`std2::expected` and `std2::optional` are choice types important to the safe standard library. They implement a suite of member functions as a convenience.

2.5.1 Pattern matching

The *match-expression* in Safe C++ offers much the same capabilities as the pattern matching implemented in C# and Swift and proposed for C++ in [P2688R1]. But the requirements of borrow checking and relocation make it most similar to Rust's feature. Pattern matching is the only way to access alternatives of choice types. Pattern matching important in achieving Safe C++'s type safety goals.

match1.cxx – (Compiler Explorer)

```
#feature on safety
#include <std2.h>

choice Primitive {
  i8(int8_t),
  i16(int16_t),
  i32(int32_t),
  i64(int64_t),
  i64_2(int64_t),
  i64_3(int64_t),
  pair(int, int),
  s(std2::string)
};

int test(Primitive obj) noexcept safe {
  return match(obj) {
    // Match 1, 2, 4, or 8 for either i32 or i64.
    .i32(1|2|4|8) | .i64(1|2|4|8)   => 1;

    // Match less than 0. ..a is half-open interval.
    .i32(..0)                       => 2;

    // Match 100 to 200 inclusive. ..= is closed interval.
    .i32(100..=200)                 => 3;

    // variant-style access. Match all alternatives with
    // a `int64_t` type. In this case, i64, i64_2 or i64_3
    // matches the pattern.
    {int64_t}(500 | 1000..2000)     => 4;

    // Match a 2-tuple/aggregate. Bind declarations x and y to
    // the tuple elements. The match-guard passes when x > y.
```

```
    .pair([x, y]) if (x > y)        => 5;

    // Match everything else.
    // Comment the wildcard for an exhaustiveness error.
    _                               => 6;
  };
}
```

A *match-expression*'s operand is an expression of class, choice, array, slice, arithmetic, builtin vector or builtin matrix type. The *match-specifier* is populated with a set of *match-clauses*. Each *match-clause* has a *pattern* on the left, an optional *match-guard* in the middle, and a *match-body* after the `=>` token.

A match is rich in the kind of patterns it supports:

- **structured pattern** `[p1, p2]` - Matches subobjects of aggregates and C++ types that implement the `tuple_size`/`tuple_element` customization point. These are useful when destructuring Safe C++'s first-class tuple and array types. Nest the patterns to destructure multiple levels of subobjects. The syntax of this pattern corresponds to structured bindings and aggregate initializers.
- **designated pattern** `[x: p1, y: p2]` - Matches subobjects of aggregates by member name rather than ordinal. The syntax of this pattern corresponds to designated bindings and designated initializers.
- **choice pattern** `.alt` or `.alt(p)` - Matches choice alternatives. If the choice alternative has a payload type, the `.alt(p)` syntax opens a pattern on its payload.
- **variant pattern** `{type}` or `{type}(p)` - Matches all choice alternatives with a payload type of *type*. The `{type}(p)` syntax opens a pattern on the payload.
- **wildcard pattern** `_` Matches any pattern. These correspond to the `default` case of a *switch-statement* and may be required to satisfy a *match-expression*'s exhaustiveness requirement.
- **rest pattern** - `..` Matches any number of elements. Use inside structured patterns to produce patterns on the beginning or end elements. Also used to supports patterns on slice operands.
- **binding declaration** *binding-mode* `decl` - Bind a declaration to the pattern's operand. There are six binding modes: *default*, `cpy`, `rel`, `^` for mutable borrows, `^const` for shared borrows and `&` for lvalue references.
- **test pattern** - Name a constant literal or constant expression inside parentheses. These can be formed into range expressions with an operand on either or both sides of `..` or `..=`. The current operand of the match must match the test pattern to go to the body.
- **disjunction pattern** `p1 | p2` - Separate patterns with the disjunction operator `|`.

This example uses choice, test, variant, binding, disjunction and wildcard patterns. The wildcard pattern at the end proves exhaustiveness.

match2.cxx – (Compiler Explorer)

```
#feature on safety
#include <std2.h>

using namespace std2;

choice Primitive {
  i8(int8_t),
  u8(uint8_t),
  i16(int16_t),
  u16(uint16_t),
  i32(int32_t),
  u32(uint32_t),
  i64(int64_t),
  u64(uint64_t),
  s(string);
```

```
  bool is_signed(const self^) noexcept safe {
    // concise form. Equivalent to Rust's matches! macro.
    return match(*self; .i8 | .i16 | .i32 | .i64);
  }

  bool is_unsigned(const self^) noexcept safe {
    return match(*self; .u8 | .u16 | .u32 | .u64);
  }

  bool is_string(const self^) noexcept safe {
    return match(*self; .s);
  }
};

int main() safe {
  println(Primitive::i16(5i16).is_signed());
  println(Primitive::u32(100ui32).is_unsigned());
  println(Primitive::s("Hello safety").is_string());
}
```

The *concise-match-expression* is an abbreviated syntax for pattern matching that evaluates a pattern and a *match-guard*. This maps directly to Rust's [matches] facility, although instead of being a macro, this is a first-class language feature.

We're working on better specifying the binding modes for match declarations and their interactions with the relocation object model and with the borrow checker. We hope to show more complex usage for the next revision.

2.6 Interior mutability

Recall the law of exclusivity, the program-wide invariant that guarantees a resource isn't mutated while another user aliases it. How does this square with the use of shared pointers, which enables shared ownership of a mutable resource? How does it support threaded programs, where access to shared mutable state is permitted between threads? Shared mutable access exists in this safety model, but the way it's enabled involves some trickery.

```
template<class T+>
class [[unsafe::sync(false)]] unsafe_cell
{
  T t_;

public:
  unsafe_cell() = default;

  explicit
  unsafe_cell(T t) noexcept safe
    : t_(rel t) { }

  T* get(self const^) noexcept safe {
    return const_cast<T*>(addr self->t_);
  }
};
```

Types with interior mutability implement *deconfliction* strategies to support shared mutation without the risk of data races or violating exclusivity. They encapsulate `std2::unsafe_cell`, which is based on Rust's `UnsafeCell`[unsafe-cell] struct. `unsafe_cell::get` is a blessed way of stripping away const. While the function

is safe, it returns a raw pointer, which is unsafe to dereference. Types encapsulating `unsafe_cell` must take care to only permit mutation through this const-stripped pointer one user at a time.

Our safe standard library currently offers four types with interior mutability:

— `std2::cell<T>`[cell] provides get and set methods to read out the current value and store new values into the protected resource. Since `cell` can't be used across threads, there's no risk of violating exclusivity.
— `std2::ref_cell<T>`[ref-cell] is a single-threaded multiple-read, single-write lock. If the caller requests a mutable reference to the interior object, the implementation checks its counter, and if the object is not locked, it establishes a mutable lock and returns a mutable borrow. If the caller requests a shared reference to the interior object, the implementation checks that there is no live mutable borrow, and if there isn't, increments the counter. When users are done with the borrow, they have to release the lock, which decrements the reference count. If the user's request can't be serviced, the `ref_cell` can either gracefully return with an error code, or it can panic and abort.
— `std2::mutex<T>`[mutex] provides mutable borrows to the interior data across threads. A mutex synchronization object deconflicts access, so there's only one live borrow at a time.
— `std2::shared_mutex<T>`[rwlock] is the threaded multiple-read, single-write lock. The interface is similar to `ref_cell`'s, but it uses a mutex for deconfliction, so clients can sit on the lock until their request is serviced.

```
template<class T+>
class
[[unsafe::send(T~is_send), unsafe::sync(T~is_send)]]
mutex
{
  using mutex_type = unsafe_cell<std::mutex>;

  unsafe_cell<T> data_;
  box<mutex_type> mtx_;

public:
  class lock_guard/(a)
  {
    friend class mutex;
    mutex const^/a m_;

    lock_guard(mutex const^/a m) noexcept safe
      : m_(m) { }

    public:
    ~lock_guard() safe {
      unsafe { mut m_->mtx_->get()->unlock(); }
    }

    T^ borrow(self^) noexcept safe {
      unsafe { return ^*self->m_->data_.get(); }
    }

    ...
  };

  explicit mutex(T data) noexcept safe
    : data_(rel data)
    , unsafe mtx_(box<mutex_type>::make()) { }

  mutex(mutex const^) = delete;
```

```
  lock_guard lock(self const^) safe {
    unsafe { mut self->mtx_->get()->lock();}
    return lock_guard(self);
  }
};
```

```
void entry_point(arc<mutex<string>> data, int thread_id) safe {
  auto lock_guard = data->lock();
  string^ s = mut lock_guard.borrow();
  s.append(" ");
  println(*s);
}
```

Let's examine how `std2::mutex` implements interior mutability to obey the law of exclusivity while permitting mutation through const borrows. We've stripped the comments from the example in the thread safety section. `data` is an `arc` and `data->` returns a `const mutex<string>^`. Normally this is immutable. But `std2::mutex::lock` binds a const `self`. Its `mtx_` data member has type `box<unsafe_cell<std::mutex>>`. `std::mutex` is non-movable, so we're hosting it in a box on the heap to make `std2::mutex` movable. `mut_->` returns `const unsafe_cell<std::mutex>^` because the `self` reference is const. But `mut_->get()` returns `std::mutex*`! The const was stripped off by `unsafe_cell::get`. This is the pivot in interior mutability that allows the system to work. We have mutable pointer to `std::mutex` and simply lock it.

The `lock_guard` is a view into the `std2::mutex` with a named lifetime parameter. When the `lock_guard` goes out of scope, it calls `unlock` on the system mutex. The guard's lifetime parameter `/a` keeps the mutex in scope as long as the `lock_guard` is in scope, or the borrow checker errors.

Lifetime safety also guarantees that the `lock_guard` is in scope (meaning the mutex is locked) whenever the reference into the protected resource is used. `lock_guard::borrow` connects the lifetime on the return borrow `T^` with the lifetime on self. If the lock guard goes out of scope while the returned borrow is live, that's a borrow checker error.

Interior mutability is a legal loophole around exclusivity. You're still limited to one mutable borrow or any number of shared borrows to an object. Types with a deconfliction strategy use `unsafe_cell` to safely strip the const off shared borrows, allowing users to mutate the protected resource.

Safe C++ and Rust and conflate exclusive access with mutable borrows and shared access with const borrows. It's is an economical choice, because one type qualifier, `const` or `mut`, also determines exclusivity. But the cast-away-const model of interior mutability is an awkward consequence. But this design is not the only way: The Ante language[ante] experiments with separate `own mut` and `shared mut` qualifiers. That's really attractive, because you're never mutating something through a const reference. This three-state system doesn't map onto C++'s existing type system as easily, but that doesn't mean the const/mutable borrow treatment, which does integrate elegantly, is the most expressive. A `shared` type qualifier merits investigation during the course of this project.

- `T^` - Exclusive mutable access. Permits standard conversion to `shared T^` and `const T^`.
- `shared T^` - Shared mutable access. Permits standard conversion to `const T^`. Only types that enforce interior mutability have overloads with shared mutable access.
- `const T^` - Shared constant access.

2.7 send and sync

Thread safety is perhaps the most remarkable guarantee made by Rust's memory safety model. Central to its implementation are the `send` and `sync` traits:

> Some types allow you to have multiple aliases of a location in memory while mutating it. Unless these types use synchronization to manage this access, they are absolutely not thread-safe. Rust captures this through the Send and Sync traits.

— A type is Send if it is safe to send it to another thread.

— A type is Sync if it is safe to share between threads (T is Sync if and only if &T is Send).

– Rustnomicon[send-sync]

Safe C++ follows this model and implements `std2::send` and `std2::sync` as interfaces. Specify the interface values for a type using the `[[unsafe::send]]` and `[[unsafe::sync]]` attributes. Query the interface values with the `T~is_send` and `T~is_sync` member traits.

```
template<class T+>
class [[unsafe::send(false)]] rc;
```

`std2::rc` is the non-atomic reference-counted pointer. It permits shared ownership but only within a single thread. Its declaration marks it `send(false)`. It can't be copied to other threads, since the non-atomic reference counter would cause data races.

```
template<class T+>
class [[
  unsafe::send(T~is_send && T~is_sync),
  unsafe::sync(T~is_send && T~is_sync)
]] arc;
```

`std2::arc` is the atomic reference-counted pointer. If its inner type is both `send` and `sync`, then the `arc` specialization is also `send` and `sync`. Most types with *value semantics*, including builtin types, are `send` and `sync`. By the rules of *inherited mutability*, so are aggregate types built from `send` and `sync` subobjects. `std2::arc<int>` is `send`, permitting copy to other threads.

But `std2;:arc<int>` isn't an interesting case. `arc`'s interface only produces *const* borrows to the owned value: you can't have a data race if you're only reading from something. `arc` is intended to be used with types that implement *interior mutability*, permitting mutation through const references. `sync` characterizes the thread safety of hde deconfliction mechanisms of types with interior mutability. Is that deconfliction mechanism *single threaded* (`sync`=false) or *multi-threaded* (`sync`=true)?

```
template<class T+>
class [[unsafe::sync(false)]] cell;
```

`std2::cell` implements interior mutability, making it a candidate for use with `std2::arc`. It uses *transactions* for deconfliction: callers may only `set` or `get` the interior value. This system is robust within a thread, but unsound across threads: thread 1 could `set` a value while thread 2 `get`s it, producing a data race. `std2::cell` is marked `sync(false)`: it's unsafe to share borrows to a `cell` across threads due to risk of data race.

```
template<class T+>
class [[
  unsafe::send(T~is_send),
  unsafe::sync(T~is_send)
]] mutex;
```

`std2::mutex` is another candidate for use with `std2::arc`. This type is thread safe. As shown in the thread safety example, it provides threads with exclusive access to its interior data using a synchronization object. The borrow checker prevents the reference to the inner data from being used outside of the mutex's lock. Therefore, `std2::mutex` is `sync` if its inner type is `send`. Why make it conditional on `send` when the mutex is already providing threads with exclusive access to the inner value? This provides protection for the rare type with thread affinity. A type is `send` if it can both be copied to a different thread *and used* by a different thread.

`std2::arc<std2::mutex<T>>` is `send` if `std2::mutex<T>` is `send` and `sync`. `std2::mutex<T>` is `send` and `sync` if `T` is `send`. Since most types are `send` by construction, we can safely mutate shared state over multiple threads as long as its wrapped in a `std2::mutex` and that's owned by an `std2::arc`. The `arc` provides shared ownership. The `mutex` provides shared mutation.

```
class thread {
public:
  template<class F+, class ...Args+>
  thread/(where F: static, Args...: static)(F f, Args... args) safe
  requires(
    F~is_send &&
    (Args~is_send && ...) &&
    safe(mut f(rel args...)))
    : unsafe t_()
  { ... }
  ...
};
```

The `send` property is enforced by `std2::thread`'s constructor. If all the thread arguments are `send`, the *requires-clause* evaluates true and the constructor may be called. If any argument is `send`=false, the program is ill-formed. Data races are a runtime phenomenon, but our protection is guaranteed at compile time.

The `send` constraint against demonstrates the safety model's theme of responsibility: a `safe` function must be sound for all arguments. Before entering an *unsafe-block* and calling the unsafe system-level thread function, `thread`'s constructor must confirm that it's enforcing the safety guarantees of its contract. Only types that are `send` may be used across threads.

`std2::thread` is designed defensively with the safety promise that it won't produce undefined behavior no matter how it's used. Can we fool `thread` into producing a data race?

Pass a borrow to a value on the stack. There's no guarantee that the thread will join before the stack object is destroyed. Is that a potential use-after-free? No, because the thread has an *outlives-constraint* which checks that all function arguments outlive `/static`. An `std2::arc` doesn't have lifetime arguments (unless its inner type is a lifetime binder), so that checks out. But a shared or mutable borrow does have a lifetime argument, and if it refers to an object on the stack, it's not `/static`. Those are arguments are accepted by the *requires-clause* but are rejected by the borrow checker.

Pass a borrow to a global variable. If the global's type is not `sync`, then a borrow to it is not `send`, and that's a constraint violation. If the global variable is mutable, that could cause a data race. Fortunately it's ill-formed to name mutable global objects in a safe context. Otherwise, it's safe to share const global objects between threads.

It's the responsibility of a safe library to think through all possible scenarios of use and prevent execution that could result in soundness defects. After all, the library author is a specialist in that domain. This is a friendlier system than Standard C++, which places the all the weight of writing thread safe code on the shoulders of users.

2.8 Unresolved design issues

2.8.1 *expression-outlives-constraint*

C++ variadics don't convey lifetime constraints from a function's return type to its parameters. Calls like `make_unique` and `emplace_back` take parameters `Ts... args` and return an unrelated type `T`. This may trigger the borrow checker, because the implementation of the function will produce free regions with unrelated endpoints. It's not a soundness issue, but it is a serious usability issue.

We need an *expression-outlives-constraint*, a programmatic version of *outlives-constrant* `/(where a : b)`. It would consist of an *expression* in an unevaluated context, which names the actual function parameters and harvests the lifetime constraints implied by those expressions. We should name function parameters rather than declvals of their types, because they may be borrow parameters with additional constraints than their template lifetime parameters have.

In order to name the function parameters, we'll need a trailing *expression-lifetime-constraint* syntax. Something like,

```
template<typename T+, typename... Ts+>
box<T> make_box(Ts... args) safe where(T:T(rel args...));
```

There's a unique tooling aspect to this. To evaluate the implied constraints of the outlives expression, we have to lower the expression to MIR, create new region variables for the locals, generate constraints, solve the constraint equation, and propagate region end points up to the function's lifetime parameters.

2.8.2 Function parameter ownership

The C++ Standard does not specify parameter passing conventions. That's left to implementers. Unfortunately, different implementers settled on different conventions.

> If the type has a non-trivial destructor, the caller calls that destructor after control returns to it (including when the caller throws an exception).
>
> – Itanium C++ ABI: Non-Trivial Parameters[itanium-abi]

In the Itanium C++ ABI, callers destruct function arguments after the callee has returned. This isn't compatible with Safe C++'s relocation object model. If you relocate from a function parameter into a local object, then the object would be destroyed *twice*: once by the callee when the local object goes out of scope and once by the caller on the call returns. Safe C++ specifies this aspect of parameter passing: the callee is responsible for destroying its own function parameters. If a function parameter is relocated out of, that parameter becomes uninitialized and drop elaboration elides its destructor call.

Let's say all functions declared in the `[safety]` feature implement the *relocate calling convention*. Direct calls to them should be no problem. This includes virtual calls. Direct calls to the legacy ABI from the relocate ABI should be no problem. And calls going the other way, where the caller is in the legacy ABI and the callee implements the relocate ABI, is not an issue either.

The friction comes when forming function pointers or pointers-to-member functions for indirect calls. The pointer has to contain ABI information in its type, and a pointer to a relocate ABI function must have a different type than a pointer to the equivalent legacy ABI function. Ideally we'd have an undecorated function pointer type that can point to either legacy or relocate ABI functions, creating a unified type for indirect function calls.

Consider these three new calling conventions:

- `__relocate` - Relocate CC. Callee destroys parameters. This is opt-in for both legacy and `[safety]` modes.
- `__legacy` - Legacy CC. The system's default calling convention. For Itanium ABI, the caller destroys arguments.
- `__unified` - A unified CC that holds function pointers of either the above types. The implementer can use the most significant bit to store a discriminator between CCs: set=`__relocate`, cleared=`__legacy`. This is the default for the `[safety]` mode.

There's a standard conversion from both `__legacy` and `__relocate` function pointers to the `__unified` function pointer type. The latter is a trivial bit-cast. The former merely demands setting the most significant bit.

Surprisingly, we can also support standard conversions from a `__unified` function pointer to the `__legacy` and `__relocate` function poiner types. If it's a mismatch, the null pointer is returned. This is still memory safe, because dereferencing a pointer is the unsafe operation. However, standard conversions from `__unified` function references to `__legacy` and `__relocate` references are not supported, because references may not hold nullptr.

2.8.3 Non-static member functions with lifetimes

At this point in development, lifetime parameters are not supported for non-static member functions where the enclosing class has lifetime parameters, including including template lifetime parameters. Use the `self` parameter to declare an explicit object parameter. Non-static member functions don't have full object parameter types,

which makes it challenging for the compiler to attach lifetime arguments. As the project matures it's likely that this capability will be included.

Constructors, destructors and the relocation constructor don't take explicit `self` parameters. But that's less problematic because the language won't form function pointers.

```
struct Foo/a {
  // Self parameter syntax. Supported.
  void func1(Foo^/a self, int x) safe;

  // Equivalent abbreviated self parameter syntax. Supported.
  void func2(self^, int x) safe;

  // Possible non-static member function syntax.
  // Bind a mutable borrow.
  void func3(int x) ^ safe;

  // Bind a shared borrow.
  void func4(int x) const^ safe;

  // Bind a consuming object. cv-qualifiers are unsupported.
  void func5(int x) rel safe;
};
```

Supporting `^` and `rel` in the *ref-qualifier* on a function declarator is a prospective syntax for supporting borrows and relocation on the implicit object. However, inside the functions, you'd still need to use the `self` keyword rather than `this`, because `this` produces pointers and it's unsafe dereferencing pointers.

2.8.4 Relocation out of references

You can only relocate out of *owned places*, and owned places are subobjects of local variables. Dereferences of borrows are not owned places, so you can't relocate out of them. Niko Matsakis writes about a significant potential improvement in the ownership model, [unwinding-puts-limits-on-the-borrow-checker] citing situations where it would be sound to relocate out of a reference, as long as you relocate back into it before the function returns.

```
fn swap<T>(
    a: &mut T,
    b: &mut T,
) {
    let tmp = *a;
    *a = *b;
    *b = tmp;
}
```

The blog post considers the above swap function, which transliterates to the Safe C++ code below.

```
template<typename T>
void swap(T^ a, T^ b) safe {
  T tmp = rel *a;
  *a = rel *b;
  *b = rel tmp;
}
```

This code doesn't compile under Rust or Safe C++ because the operand of the relocation is a projection involving a reference, which is not an *owned place*. This defeats the abilities of initialization analysis.

In Rust, every function call is potentially throwing, including destructors. In some builds, panics are throwing,

allowing array subscript operations to exit a function on the cleanup path. In debug builds, integer arithmetic may panic to protect against overflow. There are many non-return paths out functions, and unlike C++, Rust lacks a *noexcept-specifier* to disable cleanup. Matsakis suggests that relocating out of references is not implemented because its use would be severely limited by the many unwind paths out of a function, making it rather uneconomical to support.

It's already possible to write C++ code that is less burdened by cleanup paths than Rust. If Safe C++ adopted the `throw()` specifier from the Static Exception Specification,[P3166R0] we could statically verify that functions don't have internal cleanup paths. It may be worthwhile to give `noexcept` interfaces to `vector` and similar containers. Exceptions are a poor way to signal out-of-memory states. If containers panicked on out-of-memory, we'd enormously reduce the cleanup paths in most functions. Reducing cleanup paths extends the supported interval between relocating out of a reference and restoring an object there, helping justify the cost of more complex initialization analysis.

This extended relocation feature is some of the ripest low-hanging fruit for improving the safety experience in Safe C++.

3 Implementation guidance

The intelligence behind the *ownership and borrowing* safety model resides in the compiler's middle-end, in its *MIR analysis* passes. The first thing compiler engineers should focus on when pursuing memory safety is to lower their frontend's AST to MIR. Several compiled languages already pass through a mid-level IR: Swift passes through SIL,[sil] Rust passes through MIR,[mir] and Circle passes through it's mid-level IR when targeting the new object model. There is an effort called ClangIR[clangir] to lower Clang to an MLIR dialect called CIR, but the project is in an early phase and doesn't have enough coverage to support the language or library features described in this document.

The AST->MIR and MIR->LLVM pipelines (or whatever codegen is used) fully replaces the compiler's old AST->LLVM codegen. It is more difficult to lower through MIR than directly emitting LLVM, but implementing new codegen is not a very large investment. You can look into Circle's MIR support with the `-print-mir` and `-print-mir-drop` cmdline options, which print the MIR before and after drop elaboration, respectively.

Once there is a MIR representation with adequete coverage, begin work on the MIR analysis. Borrow checking is a very intricate algorithm, but fortunately it's easy to interrogate. Print the MIR and the lifetime and outlives constraints as you lower and check functions to know where you're at. The NLL RFC[borrow-checking] is sufficient to get developers started on MIR analysis. By my count there are seven rounds of operations on MIR in the compiler's middle-end:

1. **Initialization analysis** - Perform forward dataflow analysis on the control flow graph, finding all program points where a place is initialized or uninitialized. This can be computed efficiently with gen-kill analysis,[gen-kill] in which the gen and kill states within a basic block are computed once and memoized into a pair of bit vectors. An iterative fixed-point solver follows the control flow graph and applies the gen-kill vectors to each basic block. After establishing the initialization for each place, raise errors if any uninitialized, partially initialized or potentially initialized place is used.
2. **Live analysis** - Invent new region variables for each lifetime binder on each local variable in the function. Perform reverse dataflow analysis on the control flow graph to find all program points where a region is live or dead. A region is live when the borrow it is bound to may be dereferenced at a subsequent point in the program. This also has a gen-kill solution, but it's computed in the opposite direction.
3. **Variance analysis** - Inter-procedural live analysis solves the *constraint equation*. The constraints go one way: `'R1 : 'R2 @ P3` reads "R1 outlives R2 starting from the point P3." Variance[variance] relates the lifetime parameterizations of function parameters and return types to these one-way constraints. It's necessary to examine the definitions of classes with lifetime binders and recursively examine their data member types to solve for the variance[taming-the-wildcards] for each function call. Failure to appropriately solve for variance can result in soundness holes, as illustrated by cve-rs[cve-rs], which drew attention to this advanced aspect of the safety model.

4. **Type relation** - Emit lifetime constraints to relate assignments from one object with a region variable to another object with a region variable. The variance of lifetime parameters determines the directionality of lifetime constraints.
5. **Solve the constraint equation** - Iteratively grow region variables until all lifetime constraints emitted during type relation are satisfied. We now have *inter-procedural live analysis*: we know the full set of live borrows, *even through function calls*.
6. **Borrow checking** - Visit all instructions in the control flow graph. For all *loans in scope* (i.e. the set of loans live at each program point) test for invalidating actions.[how-the-borrow-check-works] If there's a read or write on an object with a mutable borrow in scope, or a write to an object with a shared borrow in scope, that violates exclusivity, and a borrowck errors is raised. The borrow checker also detects invalid end points in free region variables.
7. **Drop elaboration** - The relocation object model may leave objects uninitialized, partially initialized or potentially initialized at the point where they go out of scope. Drop elaboration erases drops on uninitialized places, replaces drops on partially initialized places with drops only on the initialized subobjects, and gates drops on potentially initialized places behind drop flags. Drop elaboration changes your program, and is the reason why MIR isn't just an analysis representation, but a transformation between the AST and the code generator.

This is challenging work for implementers. However, I found the intelligibility of the MIR between every phase to be useful for making steady progress.

The most demanding frontend work involves piping information from the language syntax down to the MIR. Lifetime parameters on functions and classes and necessitate a comprehensive change to the compiler's type system. Wherever there's a type, you may now have a *lifetime-qualified type*. (That is, a type with bound lifetimes.)

Template specialization involves additional deduplication work, where lifetime arguments on template argument types are replaced by proxy lifetime arguments. This helps canonicalize specializations while preserving the relationship established by its lifetime parameterization. Lifetime normalization and canonicalization is also very challenging to implement, partly because there is no precedent for lifetime language entities in C++. In my experience this aspect of the frontend work was the most frustrating and error-prone work I faced when implementing memory safety. It suffers by comparison to the dataflow analysis work, because it lacks that system's elegant MIR representation.

Implementing the *safe-specifier* and *safe-operator* is easily achieved by duplicating the *noexcept* code paths. These features are implemented in the frontend with similar means, and their noexceptness/safeness are both conveyed through flags on expressions.

Supporting the `unsafe` type qualifier is more challenging, because the `const` and `volatile` qualifiers offer less guidance. Unlike the cv-qualifiers, the unsafe qualifier is ignored when examining pairs of types for ref-relation, but is attached to prvalues that are the result of lvalue-to-rvalue conversions, pointer adjustments, subobject access and the like.

There are some new syntax requirements. Circle chose the `#feature` directive to accommodate deep changes to the grammar and semantics of the language with a per-file scope, rather than per-translation unit scope. The directive is implemented by notionally including a mask of active features for each token in the translation unit. During tokenization, identifier tokens check their feature masks for the [safety] feature, which enables promotion of the `safe`, `unsafe`, `cpy`, `rel`, `match`, `mut` tokens as keywords. The frontend can simply match against these new keywords when parsing the input and building the AST. Certain aspects of the relocation object model, such as using drop-and-replace instead of invoking assignment operators, don't involve new keywords and are implemented by testing the object model mode of the current function. The mode is established by the state of the `[safety]` flag at the start of the function's definition.

In addition to the core safety features, there are many new types that put a demand on engineering resources: borrows, choice types and pattern matching to use them, first-class tuples, arrays and slices. Interfaces and interface templates are a new language mechanism that provide customization points to C++, making it much easier to author libraries that are called directly by the language. Examples in Safe C++ are the iterator support for ranged-for statements, `send`/`sync` for thread safety and the upcoming `fmt` interfaces for f-strings.

4 Conclusion

The US Government is telling industry to stop using C++ for reasons of national security. Academia is turning away in favor of languages like Rust and Swift which are built on modern technology. Tech executives are pushing their organizations to move to Rust.[russinovich] All this dilutes the language's value to novices. That's trouble for companies which rely on a pipeline of new C++ developers to continue their operations.

Instead of being received as a threat, we greet the safety model developed by Rust as an opportunity to strengthen C++. The Rust community has spent a decade generating *soundness knowledge*, which is the tactics and strategies (interior mutability, send/sync, borrow checking, and so on) for achieving memory safety without the overhead of garbage collection. Their investment in soundness knowledge informs our design of Safe C++. Adopting the same *ownership and borrowing* safety model that security professionals have been pointing to is the sensible and timely way to keep C++ viable for another generation.

Safe C++ must provide safe alternatives to everything in today's Standard Library. This proposal is a healthy beginning but it's not comprehensive treatment. Adoption will look daunting to teams that maintain large applications. However, users aren't compelled to switch everything over at once. If you need to stick with some legacy types, that's fine. The compiler can't enforce sound usage of that code, but that's always been the case. As developers incorporate more of the safe standard library, their safety coverage increases. This is not an all-or-nothing system. Some unsafe code doesn't mean that your whole project is unsafe. A project with 50% safe code should have half as many undetected soundness bugs as a project with no safe code. A project with 99% safe code, as many Rust applications have, should have 1% as many undetected soundness bugs. Rather than focusing on the long tail of difficult use cases, we encourage developers to think about the bulk of code that is amenable to the safety improvements that a mature Safe C++ toolchain will offer.

We're co-designing the Safe C++ standard library along with the language extensions. Visit our repository to follow our work. You can access all the examples included in this document. Or visit our Slack channel to get involved in the effort:

Github: https://github.com/cppalliance/safe-cpp
Slack #safe-cpp channel: https://cpplang.slack.com/
Circle download: https://www.circle-lang.org/
Working draft: https://cppalliance.org/safe-cpp/draft.html

Everything in this proposal took about 18 months to design and implement in Circle. With participation from industry, we could resolve the remaining design questions and in another 18 months have a language and standard library robust enough for mainstream evaluation. While Safe C++ is a large extension to the language, the cost of building new tooling is not steep. If C++ continues to go forward without a memory safety strategy, that's because institutional users are choosing not to pursue it; it's not because memory safe tooling is "impossible" to build.

An earlier version of this work was presented to SG23 at the St Louis 2024 ISO meeting, with the closing poll "We should promise more committee time on borrow checking?" — SF: 20, WF: 7, N: 1, WA: 0, SA: 0.

5 References

[ante] Ante Shared Interior Mutability.
https://antelang.org/blog/safe_shared_mutability/#shared-interior-mutability

[arc] Automatic reference counting.
https://docs.swift.org/swift-book/documentation/the-swift-programming-language/automaticreferencecounting/

[borrow-checking] The Rust RFC Book - Non-lexical lifetimes.
https://rust-lang.github.io/rfcs/2094-nll.html

[cell] Cell.
https://doc.rust-lang.org/std/cell/struct.Cell.html

[cisa-roadmaps] CISA Releases Joint Guide for Software Manufacturers : The Case for Memory Safe Roadmaps.
https://www.cisa.gov/news-events/alerts/2023/12/06/cisa-releases-joint-guide-software-manufacturers-case-memory-safe-roadmaps

[cisa-urgent] The Urgent Need for Memory Safety in Software Products.
https://www.cisa.gov/news-events/news/urgent-need-memory-safety-software-products

[clangir] Upstreaming ClangIR.
https://discourse.llvm.org/t/rfc-upstreaming-clangir/76587

[clang-lifetime-annotations] Lifetime annotations for the C++ Clang Frontend.
https://discourse.llvm.org/t/rfc-lifetime-annotations-for-c/61377

[cve-rs] cve-rs.
https://github.com/Speykious/cve-rs

[drop] drop in std::ops.
https://doc.rust-lang.org/std/ops/trait.Drop.html

[gen-kill] Data-flow analysis.
https://en.wikipedia.org/wiki/Data-flow_analysis#Bit_vector_problems

[google-0day] 0day "In the Wild."
https://googleprojectzero.blogspot.com/p/0day.html

[hoare] Null References: The Billion Dollar Mistake.
https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/

[how-the-borrow-check-works] How the borrow check works.
https://rust-lang.github.io/rfcs/2094-nll.html#layer-5-how-the-borrow-check-works

[isprint] std::isprint.
https://en.cppreference.com/w/cpp/string/byte/isprint

[itanium-abi] Itanium C++ ABI: Non-Trivial Parameters.
https://itanium-cxx-abi.github.io/cxx-abi/abi.html#non-trivial-parameters

[matches] matches! macro.
https://doc.rust-lang.org/std/macro.matches.html

[may-dangle] dropck_eyepatch.
https://rust-lang.github.io/rfcs/1327-dropck-param-eyepatch.html

[mir] The MIR (Mid-level IR).
https://rustc-dev-guide.rust-lang.org/mir/index.html

[ms-vulnerabilities] We need a safer systems programming language.
https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming-language

[mutex] Mutex.
https://doc.rust-lang.org/std/sync/struct.Mutex.html

[ncsi-plan] National Cybersecurity Strategy Implementation Plan.
https://www.whitehouse.gov/wp-content/uploads/2024/05/NCSIP-Version-2-FINAL-May-2024.pdf

[nsa-guidance] NSA Releases Guidance on How to Protect Against Software Memory Safety Issues.
https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/3215760/nsa-releases-guidance-on-how-to-protect-against-software-memory-safety-issues/

[P1144R11] Arthur O'Dwyer. 2024-05-15. std::is_trivially_relocatable.
https://wg21.link/p1144r11

[P1179R1] Herb Sutter. 2019-11-22. Lifetime safety: Preventing common dangling.
https://wg21.link/p1179r1

[P2688R1] Michael Park. 2024-02-15. Pattern Matching: ‘match‘ Expression.
https://wg21.link/p2688r1

[P3166R0] Lewis Baker. 2024-03-16. Static Exception Specifications.
https://wg21.link/p3166r0

[phantom-data] PhantomData.
https://doc.rust-lang.org/nomicon/phantom-data.html

[pin] Module std::pin.
https://doc.rust-lang.org/std/pin/index.html

[raii] Resource acquisition is initialization.
https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization

[ref-cell] RefCell.
https://doc.rust-lang.org/std/cell/struct.RefCell.html

[russinovich] It’s time to start halting any new projects in C/C++ and use Rust.
https://x.com/markrussinovich/status/1571995117233504257?lang=en

[rust-iterator] Iterator in std::iter.
https://doc.rust-lang.org/std/iter/trait.Iterator.html

[rust-language] The Rust Programming Language.
https://doc.rust-lang.org/book/

[rwlock] RwLock.
https://doc.rust-lang.org/std/sync/struct.RwLock.html

[safety-comments] Safety comments policy.
https://std-dev-guide.rust-lang.org/policy/safety-comments.html

[safe-unsafe-meaning] How Safe and Unsafe Interact.
https://doc.rust-lang.org/nomicon/safe-unsafe-meaning.html

[scc] Strongly connected component.
https://en.wikipedia.org/wiki/Strongly_connected_component

[secure-by-design] Secure by Design : Google’s Perspective on Memory Safety.
https://research.google/pubs/secure-by-design-googles-perspective-on-memory-safety/

[send-sync] Rustnomicon – Send and Sync.
https://doc.rust-lang.org/nomicon/send-and-sync.html

[sil] Swift Intermediate Language (SIL).
https://github.com/swiftlang/swift/blob/main/docs/SIL.rst

[string-view-use-after-free] std::string_view encourages use-after-free; the Core Guidelines Checker doesn’t complain.
https://github.com/isocpp/CppCoreGuidelines/issues/1038

[taming-the-wildcards] Taming the Wildcards: Combining Definition- and Use-Site Variance.
https://yanniss.github.io/variance-pldi11.pdf

[tracing-gc] Tracing garbage collection.
https://en.wikipedia.org/wiki/Tracing_garbage_collection

[unsafe-cell] UnsafeCell.
https://doc.rust-lang.org/std/cell/struct.UnsafeCell.html

[unwinding-puts-limits-on-the-borrow-checker] Unwinding puts limits on the borrow checker.
https://smallcultfollowing.com/babysteps/blog/2024/05/02/unwind-considered-harmful/#unwinding-puts-limits-on-the-borrow-checker

[variance] RFC 0738 - variance.
https://rust-lang.github.io/rfcs/0738-variance.html

[vocabulary-types] CXX — safe interop between Rust and C++.
https://cxx.rs/bindings.html

[white-house] Future Software Should Be Memory Safe.
https://www.whitehouse.gov/oncd/briefing-room/2024/02/26/press-release-technical-report/