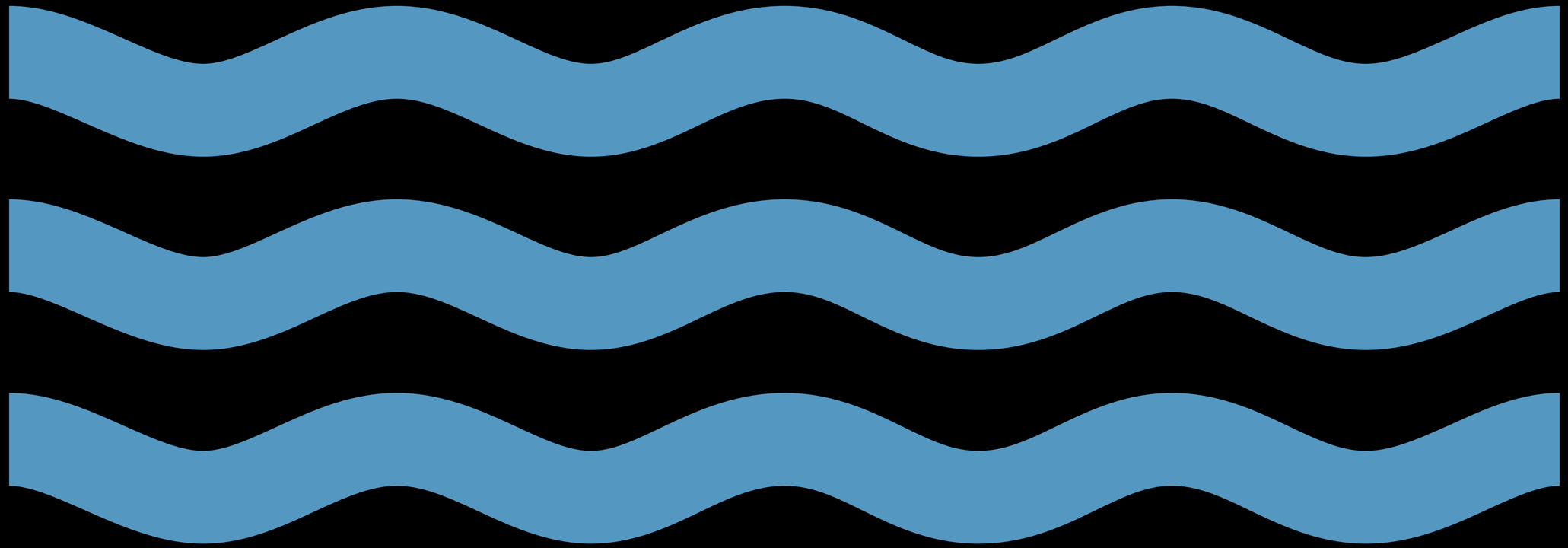
C++ interoperability with memory-safe languages

Gábor Horváth

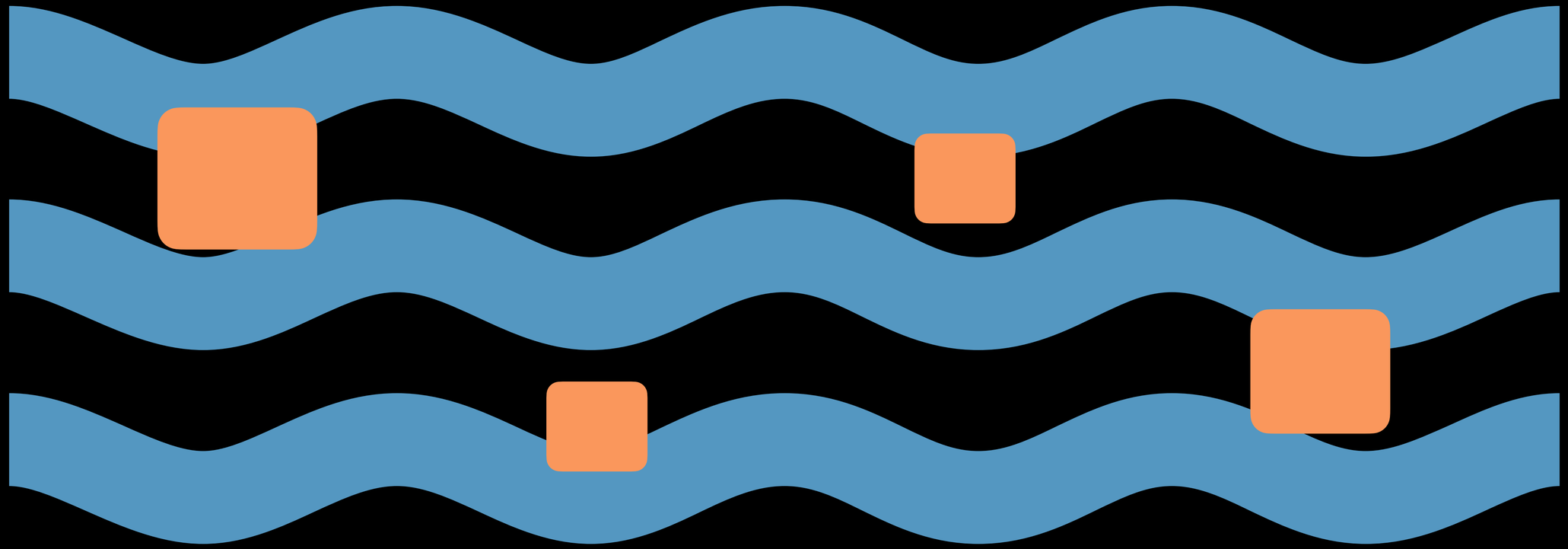
Improving safety

Improving safety



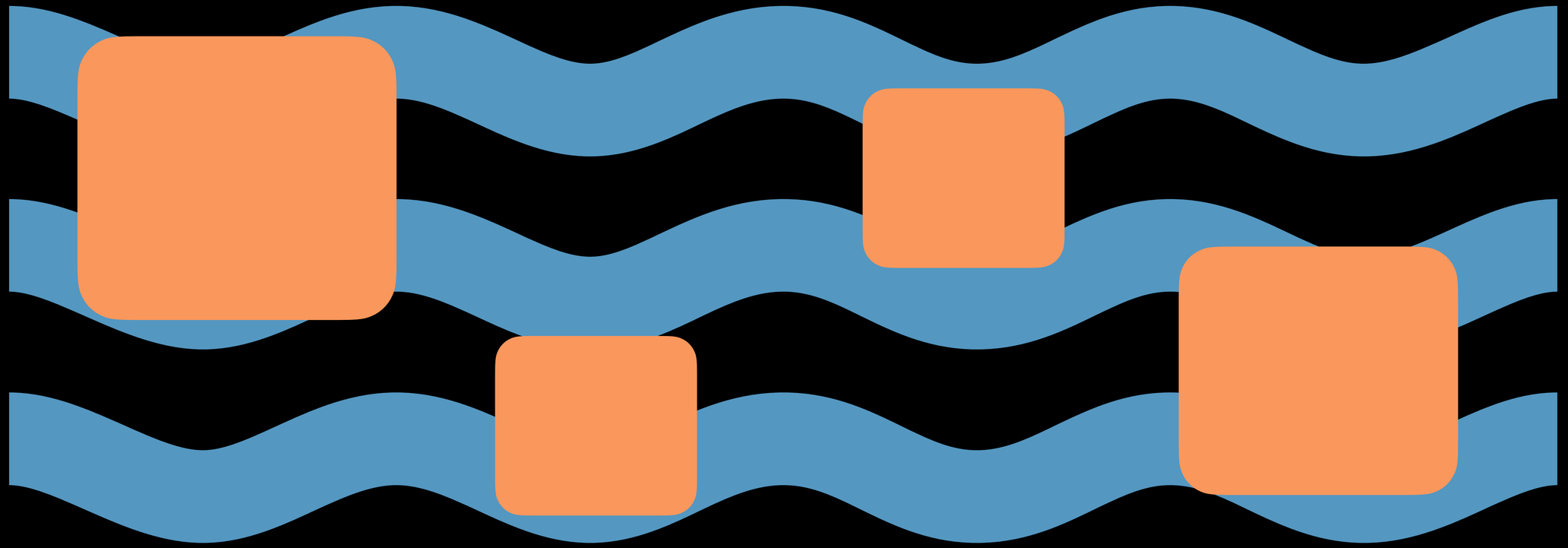
https://security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html

Improving safety



https://security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html

Improving safety



https://security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html

Improving safety

https://rustfoundation.org/brand-guide/#logos
https://github.com/isocpp/logos

Improving safety

https://rustfoundation.org/brand-guide/#logos
https://github.com/isocpp/logos

Improving safety

https://rustfoundation.org/brand-guide/#logos
https://github.com/isocpp/logos

Dimensions of memory safety

Lifetime safety

- Lifetime annotations

Bounds safety

- Bounds safety annotations, hardened libc++

Type safety

- Typed allocation, type sanitizer

Initialization safety

- Automatic variable initialisation, memory sanitizer

Thread safety

- Thread safety analysis, thread sanitizer

Dimensions of memory safety

Lifetime safety

- Lifetime annotations

Bounds safety

- Bounds safety annotations, hardened libc++

Type safety

- Typed allocation, type sanitizer

Initialization safety

- Automatic variable initialisation, memory sanitizer

Thread safety

- Thread safety analysis, thread sanitizer

What is lifetime safety?

What is lifetime safety?

```
const char* p = std::string("Hello").data();
```

What is lifetime safety?

```
const char* p = std::string("Hello").data();
```

What is lifetime safety?

```
const char* p = std::string("Hello").data();
```

```
SomeType* p = myObject.get();
```

What is lifetime safety?

```
const char* p = std::string("Hello").data();
```

```
SomeType* p = myObject.get();
```

```
auto it = vec.begin();
vec.push_back(42);
*it += 1729;
```

What is lifetime safety?

```
const char* p = std::string("Hello").data();
```

```
SomeType* p = myObject.get();
```

```
auto it = vec.begin();
vec.push_back(42);
*it += 1729;
```

```
// C++
std::vector<int> getVecOfInt();

// Swift
let vec = getVecOfInt()
let begin = vec.__beginUnsafe()
let val = begin.pointee
```

What is lifetime safety?

```cpp
const char* p = std::string("Hello").data();
```

```cpp
SomeType* p = myObject.get();
```

```cpp
auto it = vec.begin();
vec.push_back(42);
*it += 1729;
```

```cpp
// C++
std::vector<int> getVecOfInt();
```

```swift
// Swift
let vec = getVecOfInt()
let begin = vec.__beginUnsafe()
let val = begin.pointee
```

C and C++ APIs are missing lifetime information

Adding the missing information has a huge design space

Different tradeoffs for the ease of adoption

C and C++ APIs are missing lifetime information

Adding the missing information has a huge design space

Different tradeoffs for the ease of adoption

C and C++ APIs are missing lifetime information

Adding the missing information has a huge design space

Different tradeoffs for the ease of adoption

C and C++ APIs are missing lifetime information

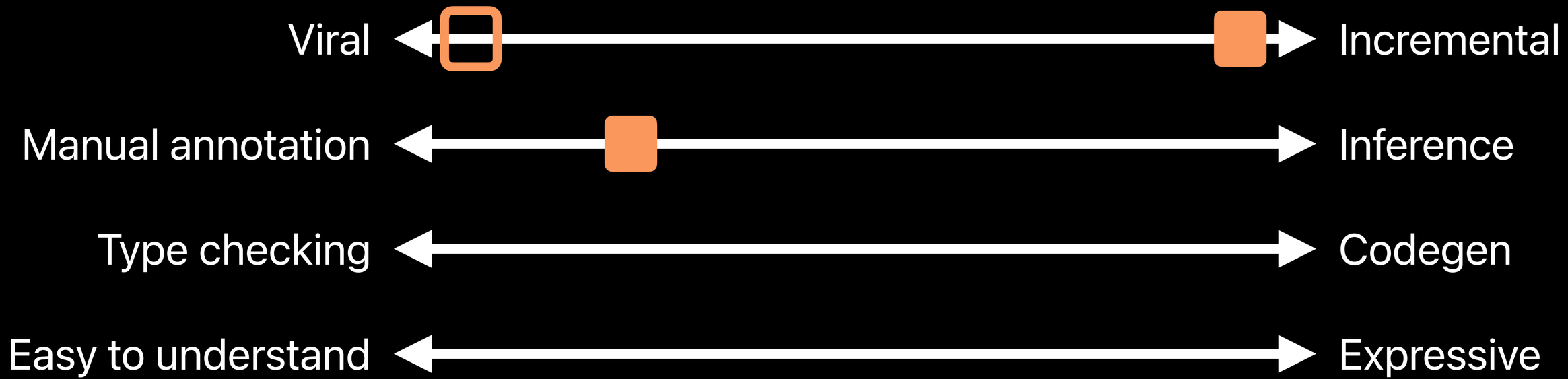
Adding the missing information has a huge design space

Different tradeoffs for the ease of adoption

C and C++ APIs are missing lifetime information

Adding the missing information has a huge design space

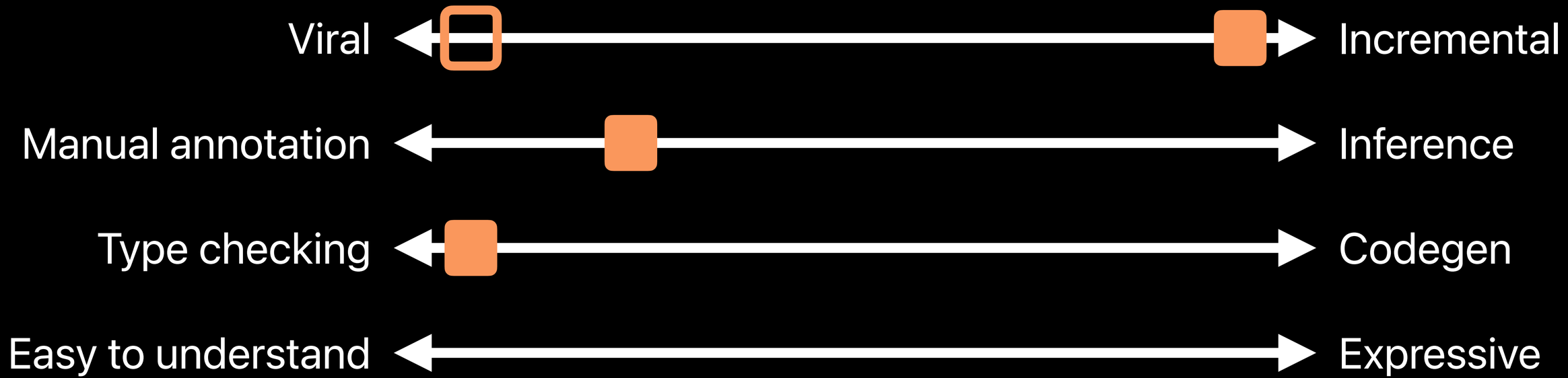
Different tradeoffs for the ease of adoption



C and C++ APIs are missing lifetime information

Adding the missing information has a huge design space

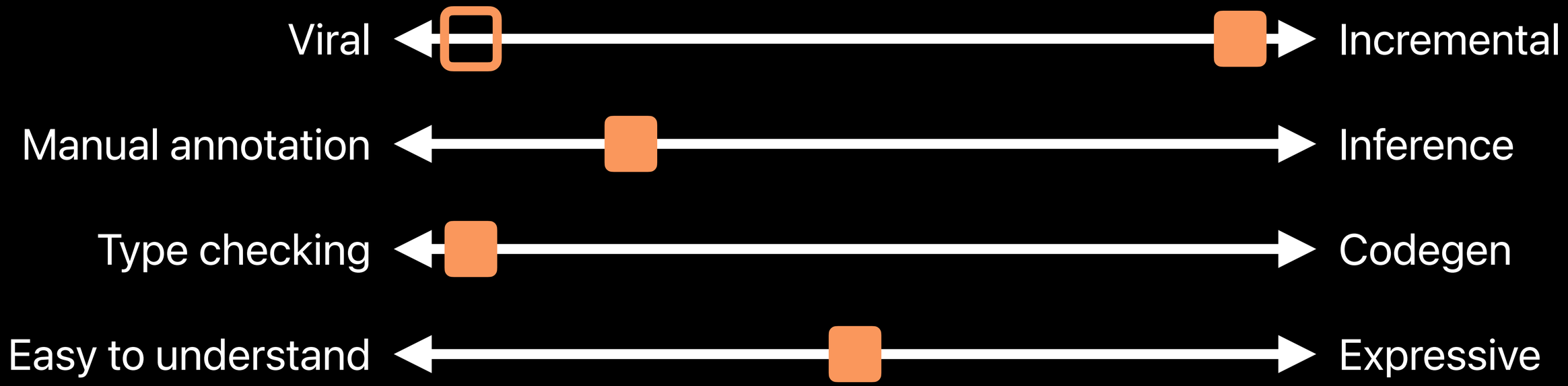
Different tradeoffs for the ease of adoption



C and C++ APIs are missing lifetime information

Adding the missing information has a huge design space

Different tradeoffs for the ease of adoption



Annotating lifetimes in Clang

```
const char* p = std::string("Hello").data();
const char* q = std::string_view("Hello").data();
```

Annotating lifetimes in Clang

```
const char* p = std::string("Hello").data();
const char* q = std::string_view("Hello").data();

warning: object backing the pointer will be destroyed at the end of the full-expression
    5 |     const char* p = std::string("Hello").data();
      |                     ^~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Annotating lifetimes in Clang

```
const char* p = std::string("Hello").data();
const char* q = std::string_view("Hello").data();

warning: object backing the pointer will be destroyed at the end of the full-expression
    5 |     const char* p = std::string("Hello").data();
      |                     ^~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
struct string {
   const char* data() const [[clang::lifetimebound]];
};
```

Annotating lifetimes in Clang

```
const char* p = std::string("Hello").data();
const char* q = std::string_view("Hello").data();

warning: object backing the pointer will be destroyed at the end of the full-expression
    5 |     const char* p = std::string("Hello").data();
      |                     ^~~~~~~~~~~~~~~~~~~~~~~~~~~
```

clang-7

```
struct string {
   const char* data() const [[clang::lifetimebound]];
};
```

Annotating lifetimes in Clang

Annotating lifetimes in Clang

```
std::set<std::string_view> s;
addToSet(std::string(), s);
```

Annotating lifetimes in Clang

```
std::set<std::string_view> s;
addToSet(std::string(), s);
```

```
warning: object whose reference is captured by 's' will be destroyed at the end
of the full-expression
   10 | addToSet(std::string(), s);
      |          ^~~~~~~~~~~~~
```

Annotating lifetimes in Clang

```
std::set<std::string_view> s;
addToSet(std::string(), s);
```

```
warning: object whose reference is captured by 's' will be destroyed at the end
of the full-expression
   10 | addToSet(std::string(), s);
      |          ^~~~~~~~~~~~~
```

```
void addToSet(std::string_view a [[clang::lifetime_capture_by(s)]],
              std::set<std::string_view>& s);
```

Annotating lifetimes in Clang

```
std::set<std::string_view> s;
addToSet(std::string(), s);
```

```
warning: object whose reference is captured by 's' will be destroyed at the end
of the full-expression
   10 | addToSet(std::string(), s);
      |          ^~~~~~~~~~~~~
```

clang-20

```
void addToSet(std::string_view a [[clang::lifetime_capture_by(s)]],
              std::set<std::string_view>& s);
```

Attribute for lifetime constraints on types

```
class                       StringRef {
public:
    StringRef() : ptr(nullptr), len(0) {}

    std::string toString() const;
private:
    const char* ptr;
    size_t len;
};

std::string normalize(const std::string& path);

// The path needs to be normalized.
StringRef fileName(const std::string& path);
```

Attribute for lifetime constraints on types

```
class                       StringRef {
public:
    StringRef() : ptr(nullptr), len(0) {}

    std::string toString() const;
private:
    const char* ptr;
    size_t len;
};

std::string normalize(const std::string& path);

// The path needs to be normalized.
StringRef fileName(const std::string& path);
```

Attribute for lifetime constraints on types

```
class                       StringRef {
public:
    StringRef() : ptr(nullptr), len(0) {}

    std::string toString() const;
private:
    const char* ptr;
    size_t len;
};

std::string normalize(const std::string& path);

// The path needs to be normalized.
StringRef fileName(const std::string& path);
```

```
warning: the returned type 'StringRef' is annotated as a reference
 type; its lifetime dependencies must be annotated
```

Attribute for lifetime constraints on types

```
class SWIFT_NONESCAPABLE StringRef {
public:
    StringRef() : ptr(nullptr), len(0) {}

    std::string toString() const;
private:
    const char* ptr;
    size_t len;
};

std::string normalize(const std::string& path);

// The path needs to be normalized.
StringRef fileName(const std::string& path);
```

Attribute for lifetime constraints on types

```
class SWIFT_NONESCAPABLE StringRef {
public:
    StringRef() : ptr(nullptr), len(0) {}

    std::string toString() const;
private:
    const char* ptr;
    size_t len;
};

std::string normalize(const std::string& path);

// The path needs to be normalized.
StringRef fileName(const std::string& path);
```

Attribute for lifetime constraints on types

New

```
class SWIFT_NONESCAPABLE StringRef {
public:
    StringRef() : ptr(nullptr), len(0) {}

    std::string toString() const;
private:
    const char* ptr;
    size_t len;
};

std::string normalize(const std::string& path);

// The path needs to be normalized.
StringRef fileName(const std::string& path);
```

Consuming annotations from Swift

```
class                       StringRef {
public:
    StringRef() : ptr(nullptr), len(0) {}

    std::string toString() const;
private:
    const char* ptr;
    size_t len;
};

std::string normalize(const std::string& path);

// The path needs to be normalized.
StringRef fileName(const std::string& path);
```

Consuming annotations from Swift

```
class                       StringRef {
public:
    StringRef() : ptr(nullptr), len(0) {}

    std::string toString() const;
private:
    const char* ptr;
    size_t len;
};

std::string normalize(const std::string& path);

// The path needs to be normalized.
StringRef fileName(const std::string& path);
```

```
func getFileName(_ path: borrowing std.string) -> StringRef {
    let normalizedPath = normalize(path)
    return fileName(normalizedPath)
}
```

Consuming annotations from Swift

```
class                       StringRef {
public:
    StringRef() : ptr(nullptr), len(0) {}

    std::string toString() const;
private:
    const char* ptr;
    size_t len;
};
```

```
warning: expression uses unsafe constructs but is not marked with 'unsafe'
    return fileName(normalizedPath)
           ^~~~~~~~~~~~~~~~~~~~~~~~
           unsafe
note: reference to global function 'fileName' involves unsafe type 'StringRef'
```

```
func getFileName(_ path: borrowing std.string) -> StringRef {
    let normalizedPath = normalize(path)
    return fileName(normalizedPath)
}
```

Consuming annotations from Swift

```
class SWIFT_NONESCAPABLE StringRef {
public:
    StringRef() : ptr(nullptr), len(0) {}

    std::string toString() const;
private:
    const char* ptr;
    size_t len;
};

std::string normalize(const std::string& path);

// The path needs to be normalized.
StringRef fileName(const std::string& path);
```

```
func getFileName(_ path: borrowing std.string) -> StringRef {
    let normalizedPath = normalize(path)
    return fileName(normalizedPath)
}
```

Consuming annotations from Swift

```
class SWIFT_NONESCAPABLE StringRef {
public:
    StringRef() : ptr(nullptr), len(0) {}

    std::string toString() const;
private:
    const char* ptr;
    size_t len;
};

std::string normalize(const std::string& path);

// The path needs to be normalized.
StringRef fileName(const std::string& path);
```

```
warning: the returned type 'StringRef' is annotated as non-escapable;
 its lifetime dependencies must be annotated
    :     return fileName(normalizedPath)
    : }
```

Consuming annotations from Swift

```
std::string normalize(const std::string& path);

// The path needs to be normalized.
StringRef fileName(const std::string& path [[clang::lifetimebound]]);
```

Consuming annotations from Swift

```
std::string normalize(const std::string& path);

// The path needs to be normalized.
StringRef fileName(const std::string& path [[clang::lifetimebound]] );
```

Consuming annotations from Swift

```
std::string normalize(const std::string& path);

// The path needs to be normalized.
StringRef fileName(const std::string& path [[clang::lifetimebound]] );
```

```
func getFileName(_ path: borrowing std.string) -> StringRef {
    let normalizedPath = normalize(path)
    return fileName(normalizedPath)
}
```

Consuming annotations from Swift

```
std::string normalize(const std::string& path);

// The path needs to be normalized.
StringRef fileName(const std::string& path [[clang::lifetimebound]] );
```

```
func getFileName(_ path: borrowing std.string) -> StringRef {
    let normalizedPath = normalize(path)
    return fileName(normalizedPath)
}
```

```
error: lifetime-dependent value escapes its scope
    return fileName(normalizedPath)
           ^
note: it depends on the lifetime of variable 'normalizedPath'
    let normalizedPath = normalize(path)
        ^
note: this use causes the lifetime-dependent value to escape
    return fileName(normalizedPath)
    ^
```

Annotating templated types

```
std::vector<StringRef>   g(StringRef);
std::vector<std::string> h(StringRef);
```

Annotating templated types

```
std::vector<StringRef>   g(StringRef);
std::vector<std::string> h(StringRef);
```

```
SWIFT_ESCAPABLE_IF(T)
template<class T, ...>
struct vector;
```

Annotating templated types

```
std::vector<StringRef>   g(StringRef);
std::vector<std::string> h(StringRef);
```

New

```
SWIFT_ESCAPABLE_IF(T)
template<class T, ...>
struct vector;
```

Annotating independence

```
StringRef capital(StringRef country                    ) {
    if (country == "Germany") {
        return "Berlin";
    } else if (...) {
        // ...
    }
    return "Unknown";
}
```

Annotating independence

```
StringRef capital(StringRef country                    ) {
    if (country == "Germany") {
        return "Berlin";
    } else if (...) {
        // ...
    }
    return "Unknown";
}
```

```
warning: the returned type 'StringRef' is annotated as non-escapable;
 its lifetime dependencies must be annotated
```

Annotating independence

```
[[clang::lifetime_immortal]]
StringRef capital(StringRef country [[clang::noescape]]) {
    if (country == "Germany") {
        return "Berlin";
    } else if (...) {
        // ...
    }
    return "Unknown";
}
```

Annotating independence

```
[[clang::lifetime_immortal]]
StringRef capital(StringRef country [[clang::noescape]] {
    if (country == "Germany") {
        return "Berlin";
    } else if (...) {
        // ...
    }
    return "Unknown";
}
```

Annotating independence

```
[[clang::lifetime_immortal]]
StringRef capital(StringRef country [[clang::noescape]] {
    if (country == "Germany") {
        return "Berlin";
    } else if (...) {
        // ...
    }
    return "Unknown";
}
```

Annotating independence

New

```
[[clang::lifetime_immortal]]
StringRef capital(StringRef country [[clang::noescape]]) {
    if (country == "Germany") {
        return "Berlin";
    } else if (...) {
        // ...
    }
    return "Unknown";
}
```

Annotating independence

New

New

```
[[clang::lifetime_immortal]]
StringRef capital(StringRef country [[clang::noescape]]) {
    if (country == "Germany") {
        return "Berlin";
    } else if (...) {
        // ...
    }
    return "Unknown";
}
```

Incremental annotations

Incremental annotations

Supports incremental adoption

```
int* id(int* p) { return p; }

int* f(int* p [[lifetimebound]]) {
  return id(p);
}
```

Incremental annotations

Supports incremental adoption

Not exhaustive

```
int* id(int* p) { return p; }

int* f(int* p [[lifetimebound]]) {
  return id(p);
}

const int& min(const int& lhs [[lifetimebound]],
               const int& rhs);
```

Incremental annotations

Supports incremental adoption

Not exhaustive

Limited expressivity,
lowest common denominator

```
int* id(int* p) { return p; }

int* f(int* p [[lifetimebound]]) {
  return id(p);
}

const int& min(const int& lhs [[lifetimebound]],
               const int& rhs);

int* first(std::pair<int*, int*> p);
```

Incremental annotations

Supports incremental adoption

Not exhaustive

Limited expressivity,
lowest common denominator

Best effort on the C++ side, fully enforced in the safe language

```
int* id(int* p) { return p; }

int* f(int* p [[lifetimebound]]) {
  return id(p);
}

const int& min(const int& lhs [[lifetimebound]],
               const int& rhs);

int* first(std::pair<int*, int*> p);

int* id(int* p [[lifetimebound]]) { return p; }
int* p = nullptr;
{
    int a;
    p = id(&a);
}
```

Summary

Express lifetime contracts not available in the type system

Existing Clang features with minimal extensions make interop possible

C++ and memory-safe language both benefit from the annotations

Easy adoption is crucial

- Start growing the islands of safe code as soon as possible
- No friction/push back from C++ code owners
- When full contract checking is a must, use a safe language