

# Introducing a memory-safe successor language in large codebases

John McCall  
Apple

# Agenda

Predecessor languages in the Apple ecosystem

Why use a successor language?

How Swift meets our goals

Migrating incrementally to Swift

Bona fides

commit 02dee0a46a390ad3bbad16672bd44c62628ea1ce

Author: **John McCall** <rjmccall@apple.com>

Date: Sat Jul 25 04:36:53 2009 +0000

Semantic checking for main().

Fix some invalid main() methods in the test suite that were nicely exposed by the new checks.

llvm-svn: 77047



commit d14a86427f9cd8e5bc1a114b703f21cf21eefa9c

Author: John McCall <rjmccall@apple.com>

Date: Sat Nov 21 08:51:07 2009 +0000

"Incremental" progress on using expressions, by which I mean totally ripping into pretty much everything about overload resolution in order to wean BuildDeclarationNameExpr off LookupResult::getAsSingleDecl(). Replace UnresolvedFunctionNameExpr with UnresolvedLookupExpr, which generalizes the idea of a non-member lookup that we haven't totally resolved yet, whether by overloading, argument-dependent lookup, or (eventually) the presence of a function template in the lookup results.

Incidentally fixes a problem with argument-dependent lookup where we were still performing ADL even when the lookup results contained something from a block scope.

Incidentally improves a diagnostic when using an ObjC ivar from a class method. This just fell out from rewriting BuildDeclarationNameExpr's interaction with lookup, and I'm too apathetic to break it out.

The only remaining uses of OverloadedFunctionDecl that I know of are in TemplateName and MemberExpr.

llvm-svn: 89544



commit 8b3f5ffb0aa202e77d61bfb5abf70f730a109038

Author: John McCall <rjmccall@apple.com>

Date: Tue May 15 02:01:59 2012 +0000

Change the mangling of a ref-qualifier on a function type so that it is placed in a position which is never ambiguous with a reference-to-function type. This follows some recent discussion and ensuing proposal on cxx-abi-dev. It is not necessary to change the mangling of CV-qualifiers because you cannot apply CV-qualification in the normal sense to a function type. It is not necessary to change the mangling of ref-qualifiers on method declarations because they appear in an unambiguous location.

In addition, mangle CV-qualifiers and ref-qualifiers on function types when they occur in positions other than member pointers (that is, when they appear as template arguments).

This is a minor ABI break with previous releases of clang. It is not considered critical because (1) ref-qualifiers are relatively rare, since AFAIK we're the only implementing compiler, and (2) they're particularly likely to come up in contexts that do not rely on the ODR for correctness. We apologize for any inconvenience; this is the right thing to do.

llvm-svn: 156794

# Itanium C++ ABI

---

## Contents

- [Acknowledgements](#)
- [Chapter 1: Introduction](#)
  - [1.1 Definitions](#)
  - [1.2 Limits](#)
  - [1.3 Namespace and Header](#)
  - [1.4 Scope of This ABI](#)
  - [1.5 Base Documents](#)
- [Chapter 2: Data Layout](#)
  - [2.1 General](#)
  - [2.2 POD Data Types](#)
  - [2.3 Member Pointers](#)
  - [2.4 Non-POD Class Types](#)
  - [2.5 Virtual Table Layout](#)
  - [2.6 Virtual Tables During Object Construction](#)
  - [2.7 Array Operator `new` Cookies](#)
  - [2.8 Initialization Guard Variables](#)
  - [2.9 Run-Time Type Information \(RTTI\)](#)



# Objective-C Automatic Reference Counting (ARC)

---

## **1 About this document**

### **1.1 Purpose**

### **1.2 Background**

### **1.3 Evolution**

## **2 General**

## **3 Retainable object pointers**

### **3.1 Retain count semantics**

### **3.2 Retainable object pointers as operands and arguments**

#### **3.2.1 Consumed parameters**

#### **3.2.2 Retained return values**

#### **3.2.3 Unretained return values**

#### **3.2.4 Bridged casts**

### **3.3 Restrictions**

#### **3.3.1 Conversion of retainable object pointers**

#### **3.3.2 Conversion to retainable object pointer type of expressions with known semantics**

#### **3.3.3 Conversion from retainable object pointer type in certain contexts**

## **4 Ownership qualification**

### **4.1 Spelling**

#### **4.1.1 Property declarations**

### **4.2 Semantics**

### **4.3 Restrictions**



# Pointer Authentication

- Introduction
- Basic Concepts
  - Discriminators
  - Signing schemas
- Language Features
  - Language implementation
    - C data pointers
    - C function pointers
    - Null pointers
    - Return addresses and frame pointers
    - C++ virtual functions
  - Language extensions
    - Feature testing
    - `__ptrauth` qualifier
      - Non-triviality from address diversity

```
commit 86f437147862c0e091e42840e72df464f4da16c6
```

```
Author: John McCall <rjmccall@apple.com>
```

```
Date: Sat Aug 20 01:06:52 2011 +0000
```

```
Basic IR generation of tuple and oneof types.
```

```
Swift SVN r573
```

# Language Workgroup

The Swift Language Workgroup guides the development of the Swift language and standard libraries through the [Swift evolution process](#).

## Charter

The Swift Language Workgroup:

- works with the [Swift Core Team](#) to define a roadmap for the focus areas of language and library development in the upcoming releases of Swift;



# Summary

- I've been working on predecessor languages for a long time, including C++
- I've done a lot to fix and mitigate the problems of those languages
- I've also worked a lot to help make Swift a good successor language

# Predecessor languages in the Apple ecosystem

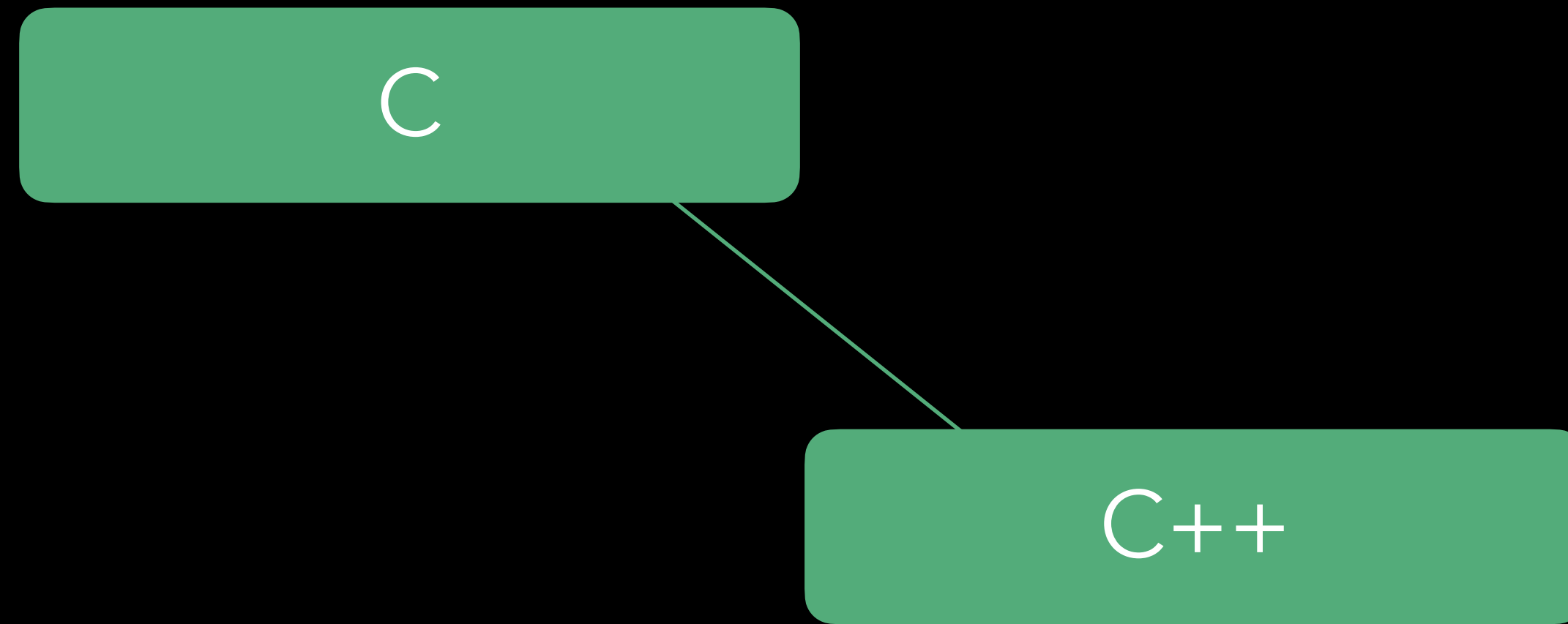
# Four predecessor languages



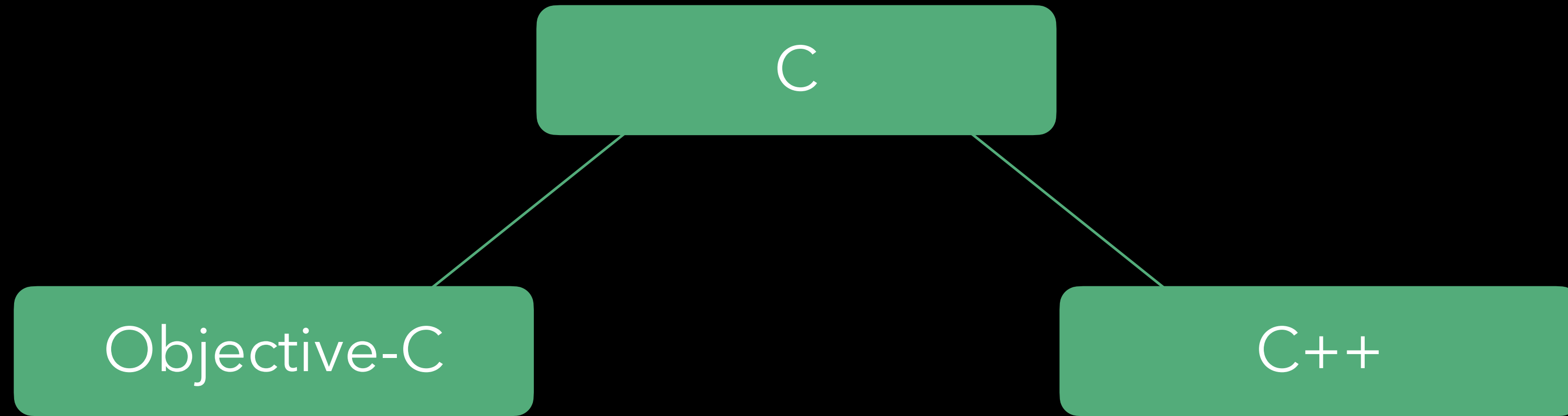
C



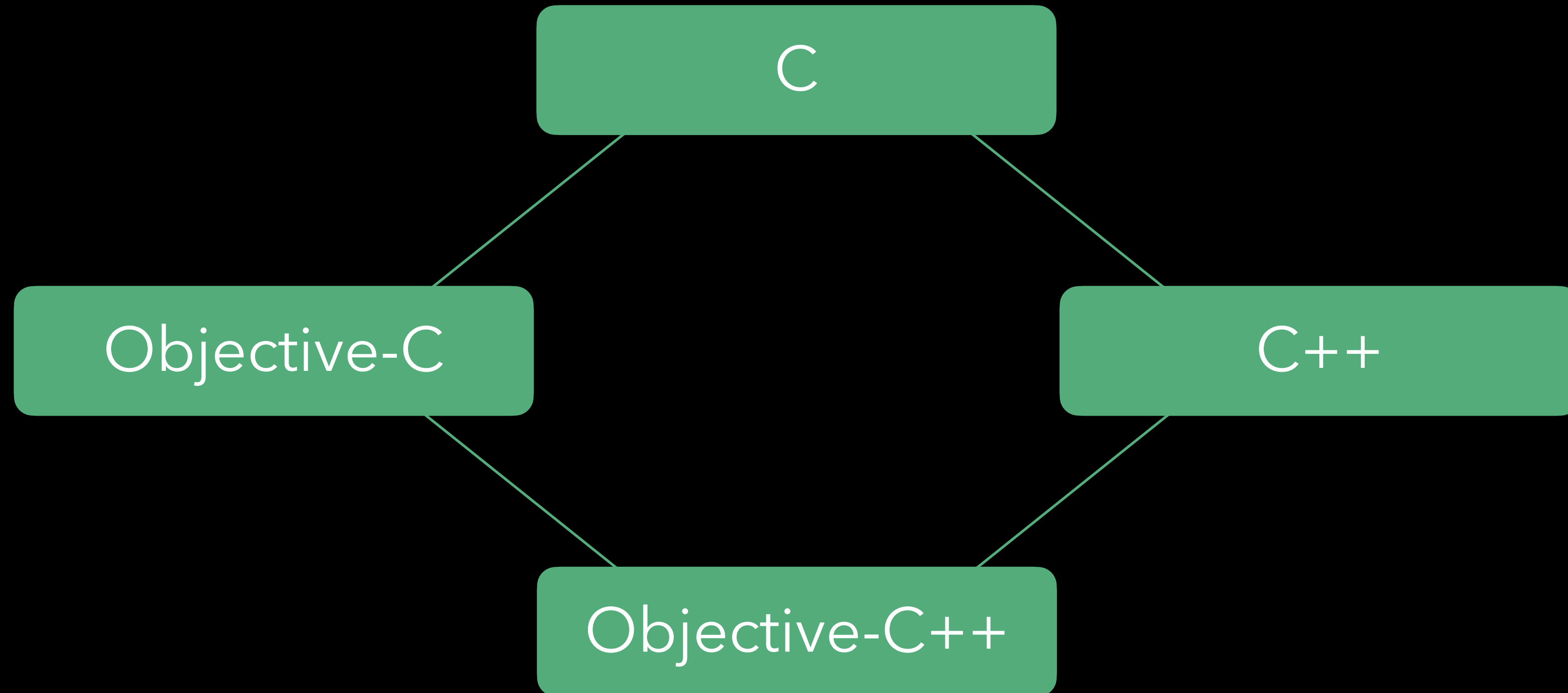
# Four predecessor languages



# Four predecessor languages

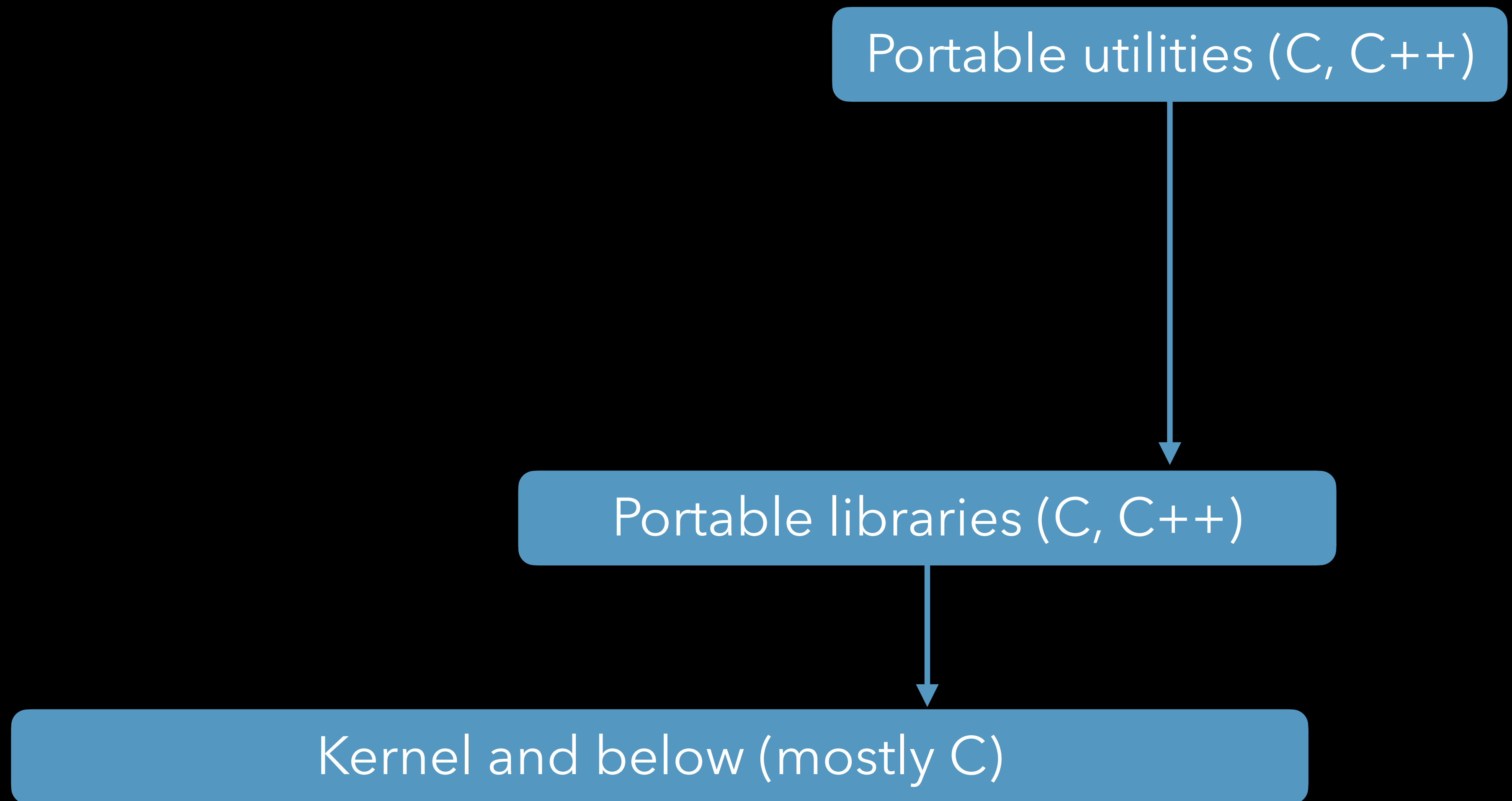


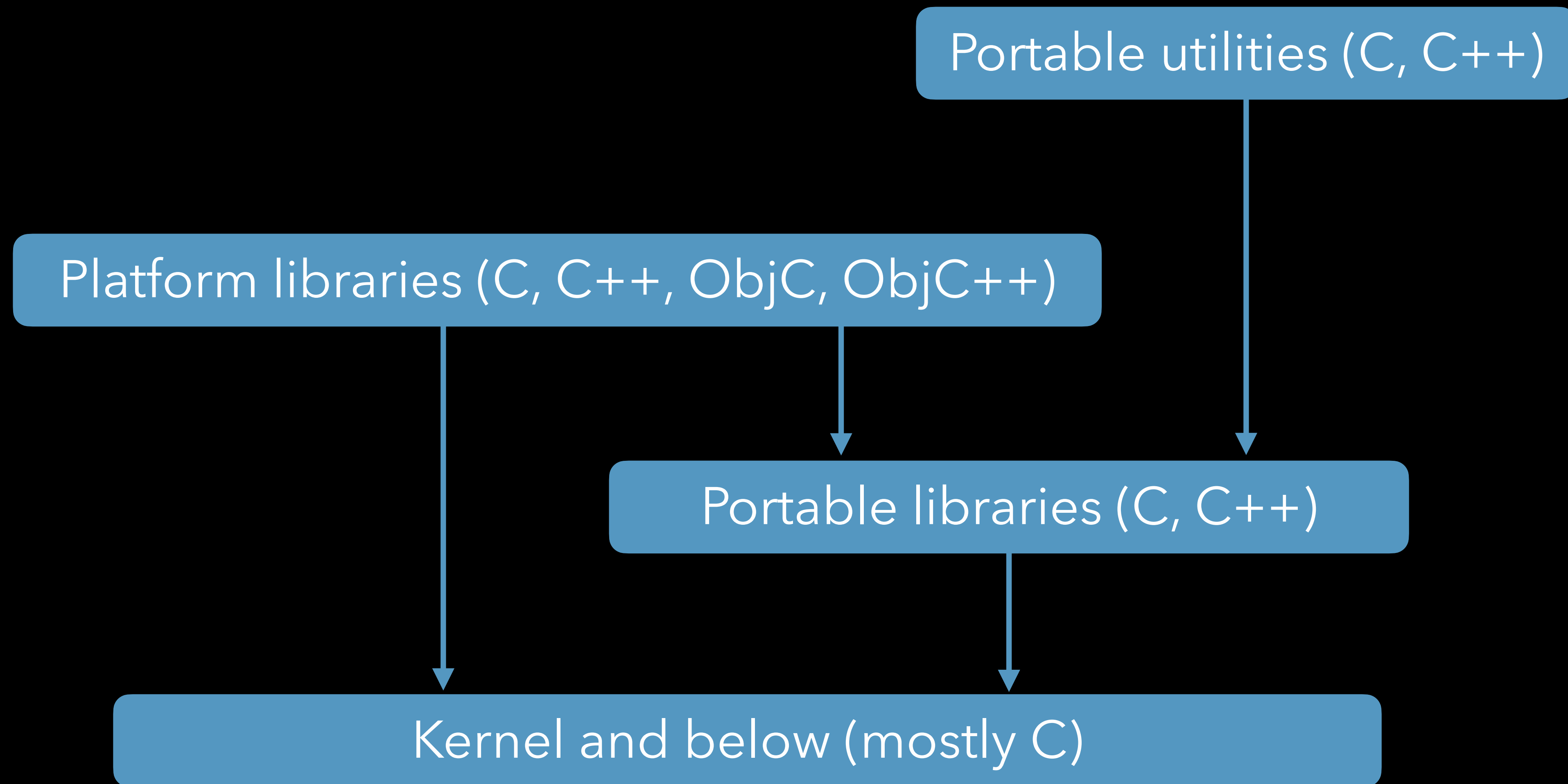
# Four predecessor languages



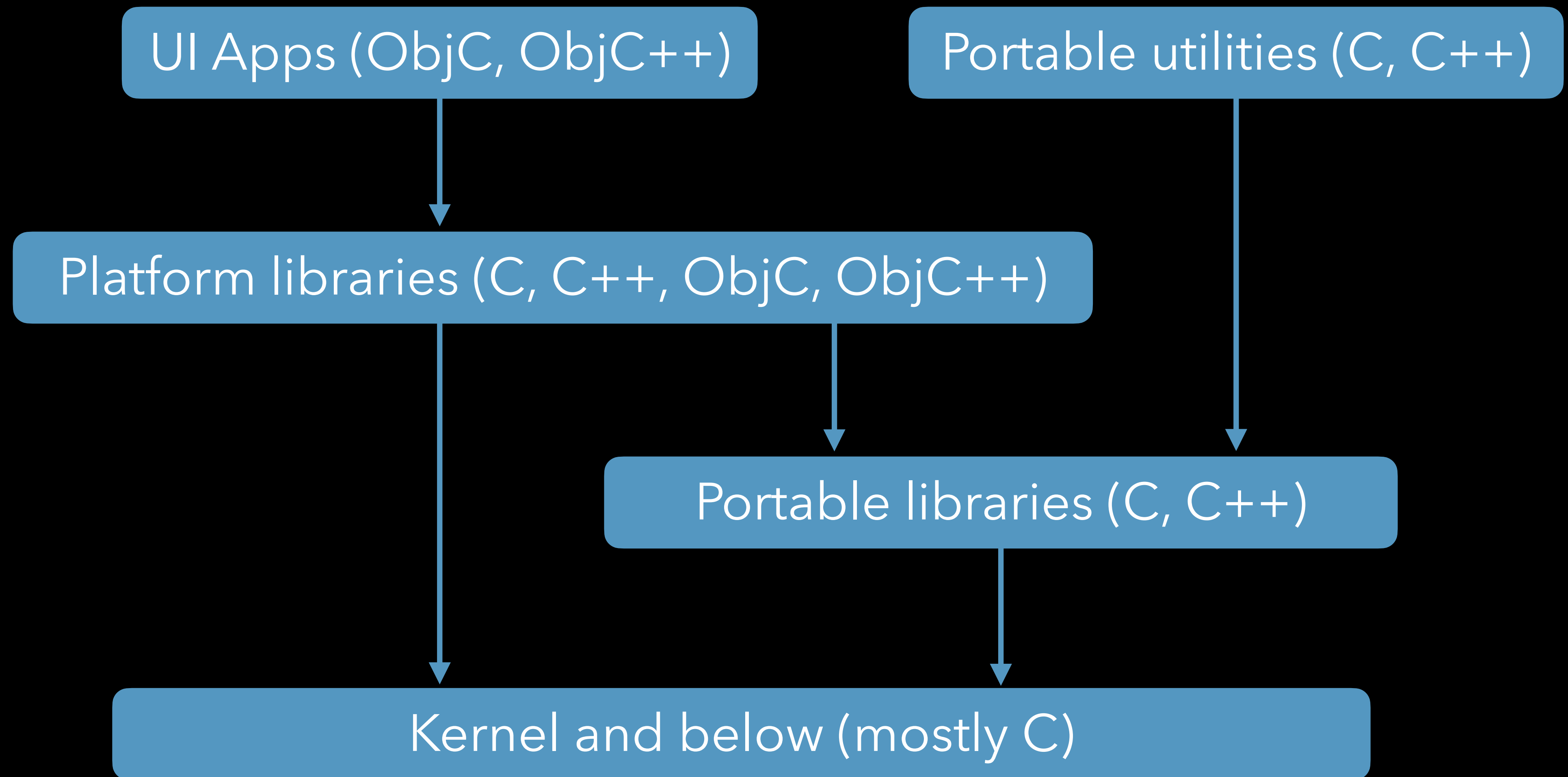


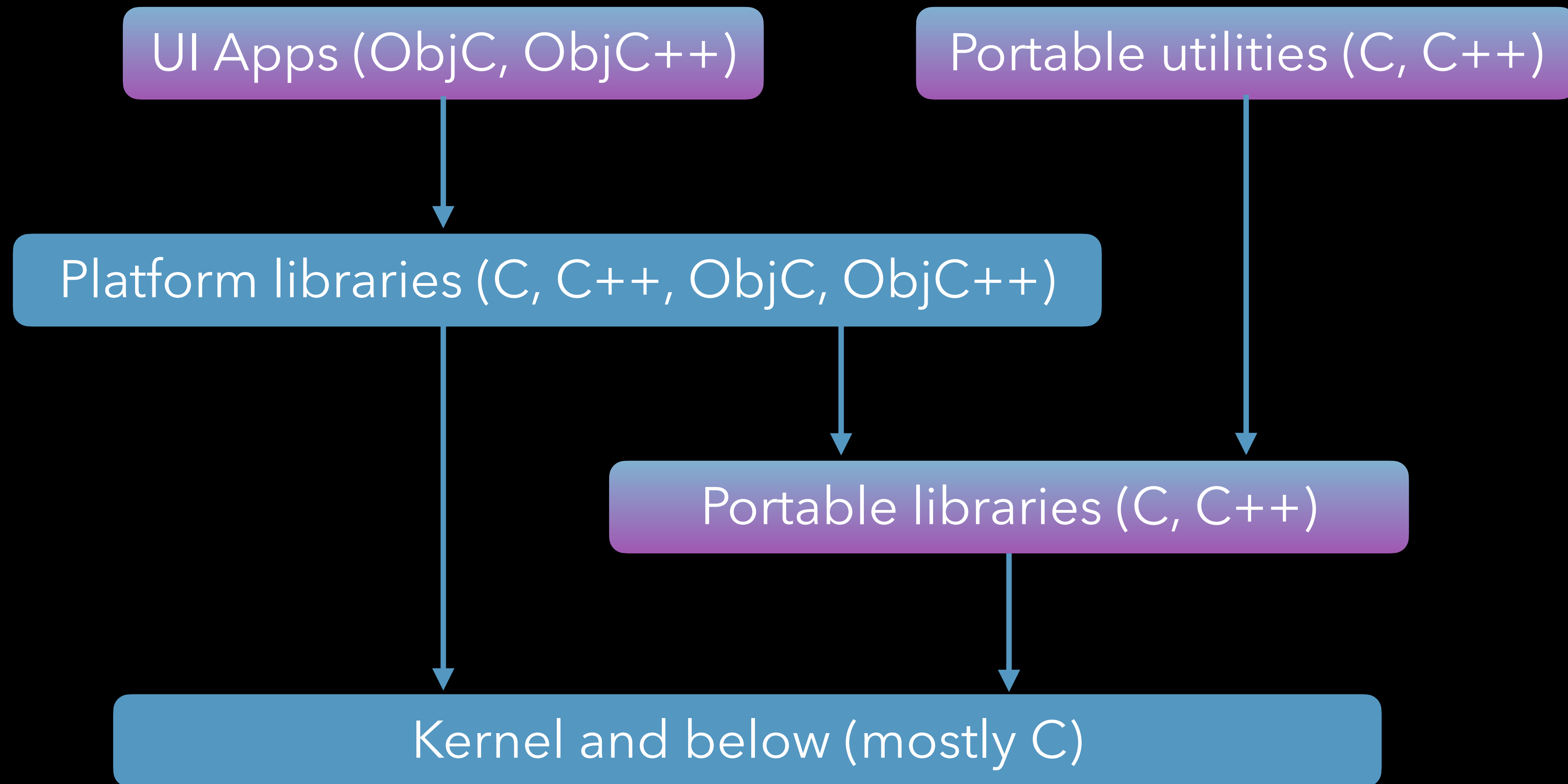
Kernel and below (mostly C)

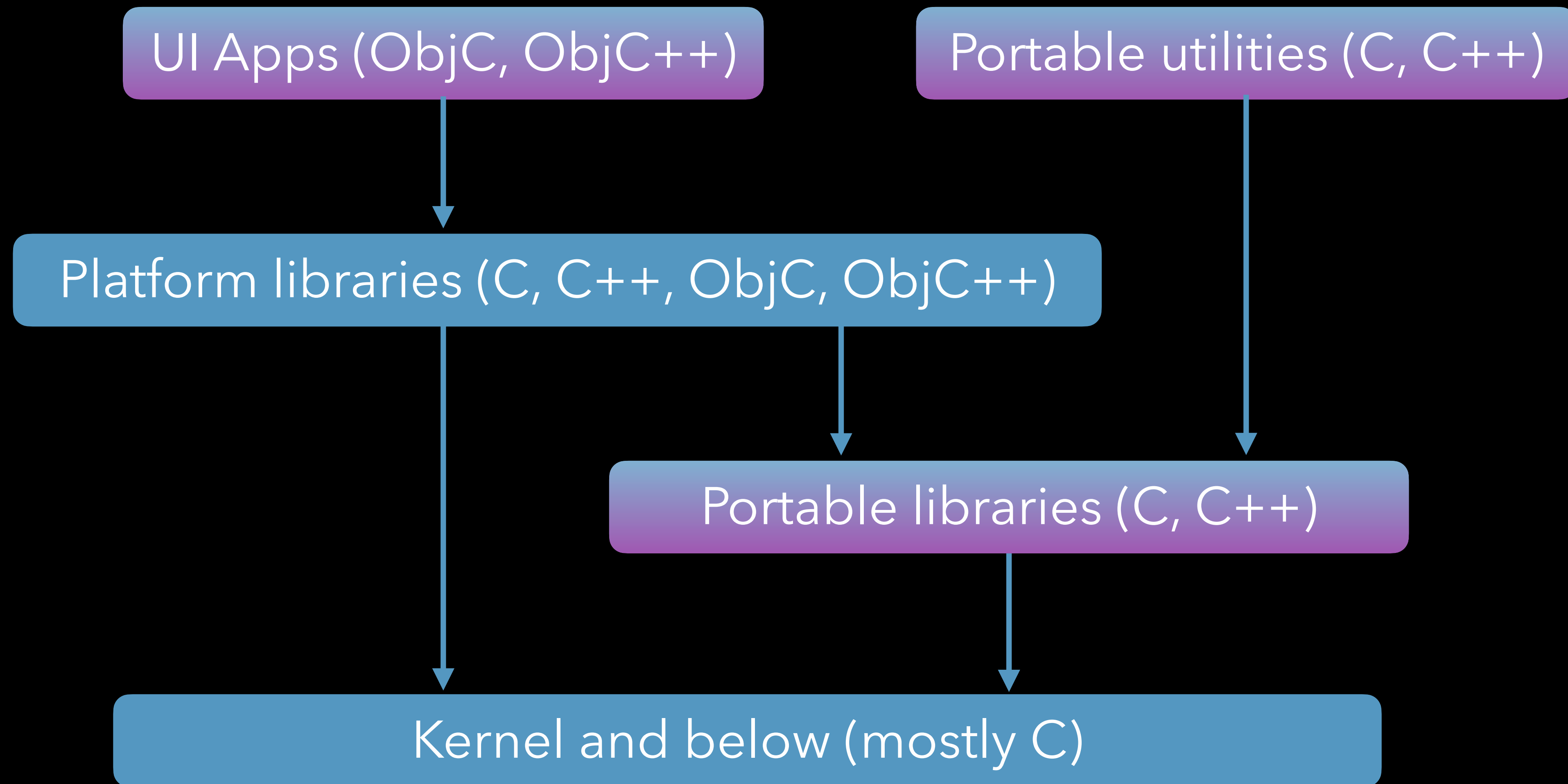


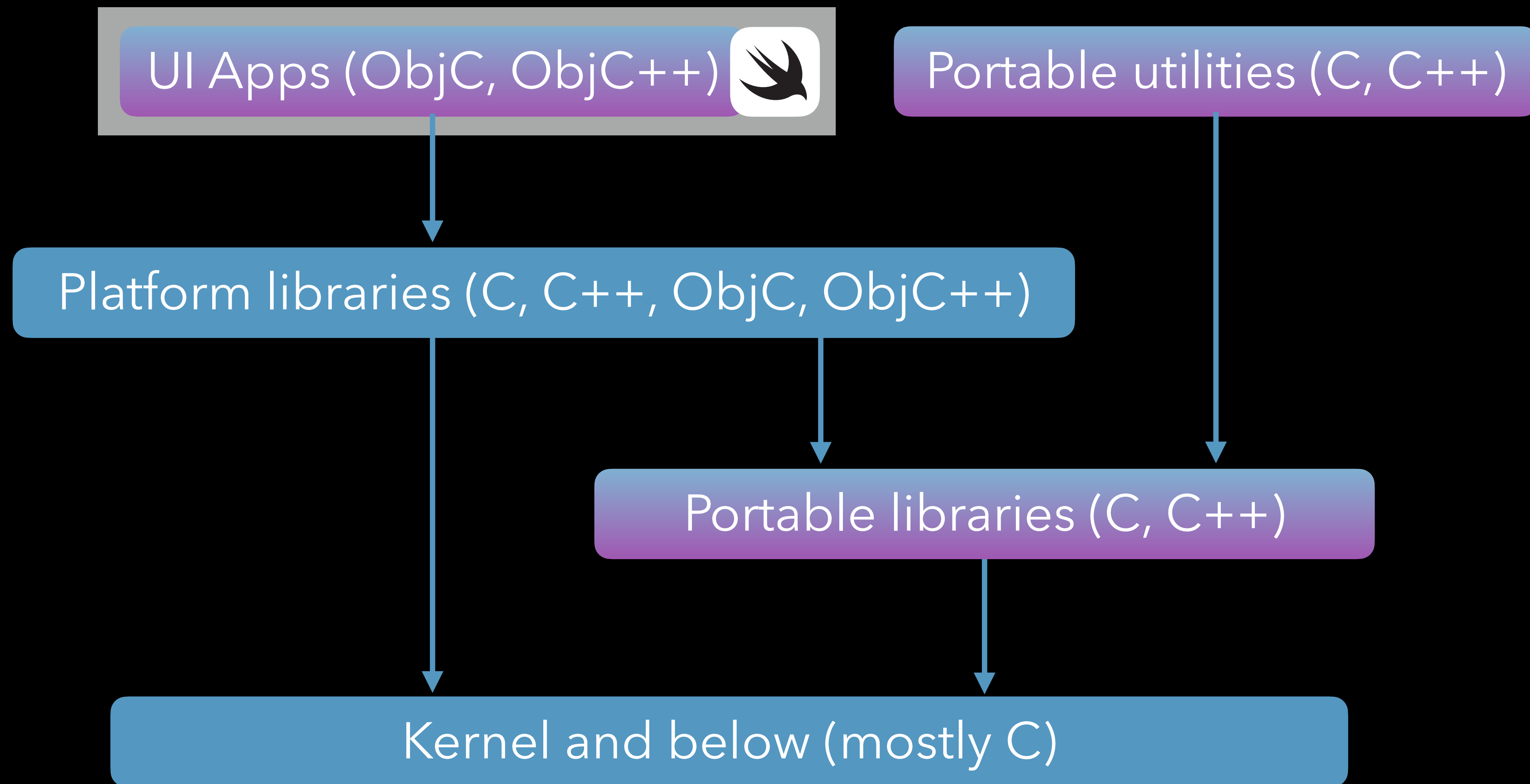













UI Apps (ObjC, ObjC++) 

Portable utilities (C, C++) 

Platform libraries (C, C++, ObjC, ObjC++) 

Portable libraries (C, C++) 

Kernel and below (mostly C) 



# Basic expectations

# Basic expectations

- One language

# Basic expectations

- One language
- Be accessible enough to serve as a first programming language

# Basic expectations

- One language
- Be accessible enough to serve as a first programming language
- Provide enough control to work in constrained environments

# Basic expectations

- One language
- Be accessible enough to serve as a first programming language
- Provide enough control to work in constrained environments
- Scale down to a minimal runtime



# Basic expectations

- One language
- Be accessible enough to serve as a first programming language
- Provide enough control to work in constrained environments
- Scale down to a minimal runtime
- Achieve high performance without a JIT

# Why use a successor language?

# Problems with C

# Problems with C

- Function pointer syntax is kind of weird

# Problems with C

- Function pointer syntax is kind of weird

cdecl

C gibberish ↔ English

```
char ** const * const x
```

declare x as const pointer to const pointer to pointer to char

[permalink](#)

# Problems with C (and C++ and Objective-C)

- Function pointer syntax is kind of weird



# Problems with C (and C++ and Objective-C)

- Function pointer syntax is kind of weird
- The preprocessor is pervasive and yet annoyingly limited
- C++ and Objective-C use completely different syntax
- Many simple template errors aren't caught until instantiation
- Headers are susceptible to cascading failures
- There are still some things you can't do in constexpr
- Build times could be better
- Different translation units are optimized like they're in different universes
- Member pointers don't really pull their own weight

# Correctness, safety, and security

- A program's first priority is to be correct
- Languages can do a lot to support correctness
- Programming errors are still inevitable
- Languages should not escalate those errors into security problems

# Correctness, safety, and security

## Supporting correctness

- Clarity of code is itself a security mitigation

# Correctness, safety, and security

## Supporting correctness

- Clarity of code is itself a security mitigation
- Important decisions should be explicit

# Correctness, safety, and security

## Supporting correctness

- Clarity of code is itself a security mitigation
- Important decisions should be explicit
- It should take work to ignore things

# Correctness, safety, and security

## Supporting correctness

- Clarity of code is itself a security mitigation
- Important decisions should be explicit
- It should take work to ignore things
- Don't be so verbose it's hard to focus



# Correctness, safety, and security

## Avoiding security problems

- Protect the abstract machine

# Correctness, safety, and security

## Avoiding security problems

- Protect the abstract machine
- Critical preconditions should always be enforced
  - Statically if possible, dynamically if necessary

# Correctness, safety, and security

## Avoiding security problems

- Protect the abstract machine
- Critical preconditions should always be enforced
  - Statically if possible, dynamically if necessary
- It's better to halt than to corrupt
  - Emergency recovery can be managed by a supervisor process

# Correctness, safety, and security

## Avoiding security problems

- Protect the abstract machine
- Critical preconditions should always be enforced
  - Statically if possible, dynamically if necessary
- It's better to halt than to corrupt
  - Emergency recovery can be managed by a supervisor process
- Also helpful in development and debugging

# Correctness, safety, and security defects in C and C++

- Uninitialized memory
- Dangling pointers and references
- No null checks on pointers and references
- No bounds checks on pointers, references, and standard data structures
- Unchecked type casts of pointers and references
- Unrestricted aliasing of pointers and references
- Integer overflow just wraps (or worse)
- No guardrails on concurrency

Can these defects be fixed?

# Can these defects be fixed?

- Common pattern:



# Can these defects be fixed?

- Common pattern:
  - We can make the current situation better

# Can these defects be fixed?

- Common pattern:
  - We can make the current situation better
  - It's only a partial fix

# Can these defects be fixed?

- Common pattern:
  - We can make the current situation better
  - It's only a partial fix
  - It often has performance trade-offs

# Can these defects be fixed?

- Common pattern:
  - We can make the current situation better
  - It's only a partial fix
  - It often has performance trade-offs
  - It does not promote correctness

# Uninitialized memory

```
size_t len;
```

```
auto objects = getAvailableObjects();  
if (objects) {  
    len = objects->size();  
}
```

```
void *buffer = malloc(headerSize + len * sizeof(object));
```

# Uninitialized memory

- There's been work to initialize all locals
  - Eliminates U.B., low cost, real progress

# Uninitialized memory

- There's been work to initialize all locals
  - Eliminates U.B., low cost, real progress
- JF Bastien has proposed just guaranteeing zero-initialization in C++
  - <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2723r1.html>
  - Would also be real progress

# Uninitialized memory

```
size_t len;
```

```
auto objects = getAvailableObjects();
```

```
if (objects) {
```

```
    len = objects->size();
```

✗ Error: 'len' is used here without always having been initialized

```
}
```

not initialized on the 'else' path here

```
void *buffer = malloc(headerSize + len * sizeof(object));
```



# Integer overflow

# Integer overflow

- Lots of discussion over the last few years about defining signed overflow
  - Would be real progress

# Integer overflow

- Lots of discussion over the last few years about defining signed overflow
  - Would be real progress
- Overflow is usually a logic error, signed or unsigned

# Integer overflow

- Lots of discussion over the last few years about defining signed overflow
  - Would be real progress
- Overflow is usually a logic error, signed or unsigned
- Wrapping is better than U.B. but still not actually the right default

# Integer overflow

- Lots of discussion over the last few years about defining signed overflow
  - Would be real progress
- Overflow is usually a logic error, signed or unsigned
- Wrapping is better than U.B. but still not actually the right default
- No reasonable path to make all overflow trap by default retroactively

# Bounds checks

# Bounds checks

- We could make std containers do bounds checks
  - Debug feature of many implementations
  - Rarely used in production

# Bounds checks

- We could make std containers do bounds checks
  - Debug feature of many implementations
  - Rarely used in production
- High overheads that are difficult to eliminate in C++



# Bounds checks

```
template <class C>
void commaSeparate(const C &c, std::ostream &out) {
    size_t index = 0;
    for (auto &v : c) {
        if (index++) out << ",";
        out << v;
    }
}
```

# Bounds checks

```
template <class C>
void commaSeparate(const C &c, std::ostream &out) {
    size_t index = 0;
    for (auto i = c.begin(), e = c.end(); i != e; ++i) {
        auto &v = *i;
        if (index++) out << ",";
        out << v;
    }
}
```

# Bounds checks

```
template <class C>
void commaSeparate(const C &c, std::ostream &out) {
    size_t index = 0;
    for (auto i = c.begin(), e = c.end(); i != e; ++i) {
        auto &v = *i;
        if (index++) out << ",";
        out << v;
    }
}
```

# Bounds checks

```
template <class C>
void commaSeparate(const C &c, std::ostream &out) {
    size_t index = 0;
    for (auto i = c.begin(), e = c.end(); i != e; ++i) {
        auto &v = *i;
        if (index++) out << ",";
        out << v;
    }
}
```

# Pointers

- We cannot reason locally about pointers
  - have no bounds
  - have no ownership
  - have no lifetime/escape restrictions
  - have no aliasing/mutability restrictions
- C++ references interconvert and so inherit all of this

# Fixing pointers

- We need pointers that follow stricter rules

# Fixing pointers

- We need pointers that follow stricter rules
- Programmers will need to use different pointers in different situations:
  - `commaSeparate` wants to take a “borrowed” immutable reference
  - Other contexts may need to carry bounds or ownership

# Fixing pointers

- We need pointers that follow stricter rules
- Programmers will need to use different pointers in different situations:
  - `commaSeparate` wants to take a “borrowed” immutable reference
  - Other contexts may need to carry bounds or ownership
- Common iterator idioms are often hard to reason about locally



# Fixing pointers

- We need pointers that follow stricter rules
- Programmers will need to use different pointers in different situations:
  - `commaSeparate` wants to take a “borrowed” immutable reference
  - Other contexts may need to carry bounds or ownership
- Common iterator idioms are often hard to reason about locally
- Requires a lot of source changes
  - or serious compromises to safety and performance

# Fixing these problems without a new language

# Fixing these problems without a new language

- Still a lot of work
  - A lot of mandatory code changes
  - A lot of interoperation problems with existing code

# Fixing these problems without a new language

- Often a challenging project
  - Some kinds of changes are difficult, like silent changes in behavior
  - Some structural changes will be necessary
  - Loose boundaries between old and updated code
  - Pressure to do a lot of non-incremental rewriting

# Fixing these problems without a new language

- Changing the default rules means breaking a lot of code
  - Much harder to do incrementally
  - Real danger of incompatible interpretations, e.g. between files

# Fixing these problems without a new language

- Changing the default rules means breaking a lot of code
  - Much harder to do incrementally
  - Real danger of incompatible interpretations, e.g. between files
- Not changing the defaults means the default behavior is unsafe
  - Bigger, uglier source changes that undermine clarity
  - Easier to accidentally use unsafe features
  - It will take expertise to write safe code

# The need for a successor language

- A “successor” version of C or C++ would essentially be a new language
  - Migrating code to it would be a major project
  - But the language would still be deeply compromised by compatibility

# The need for a successor language

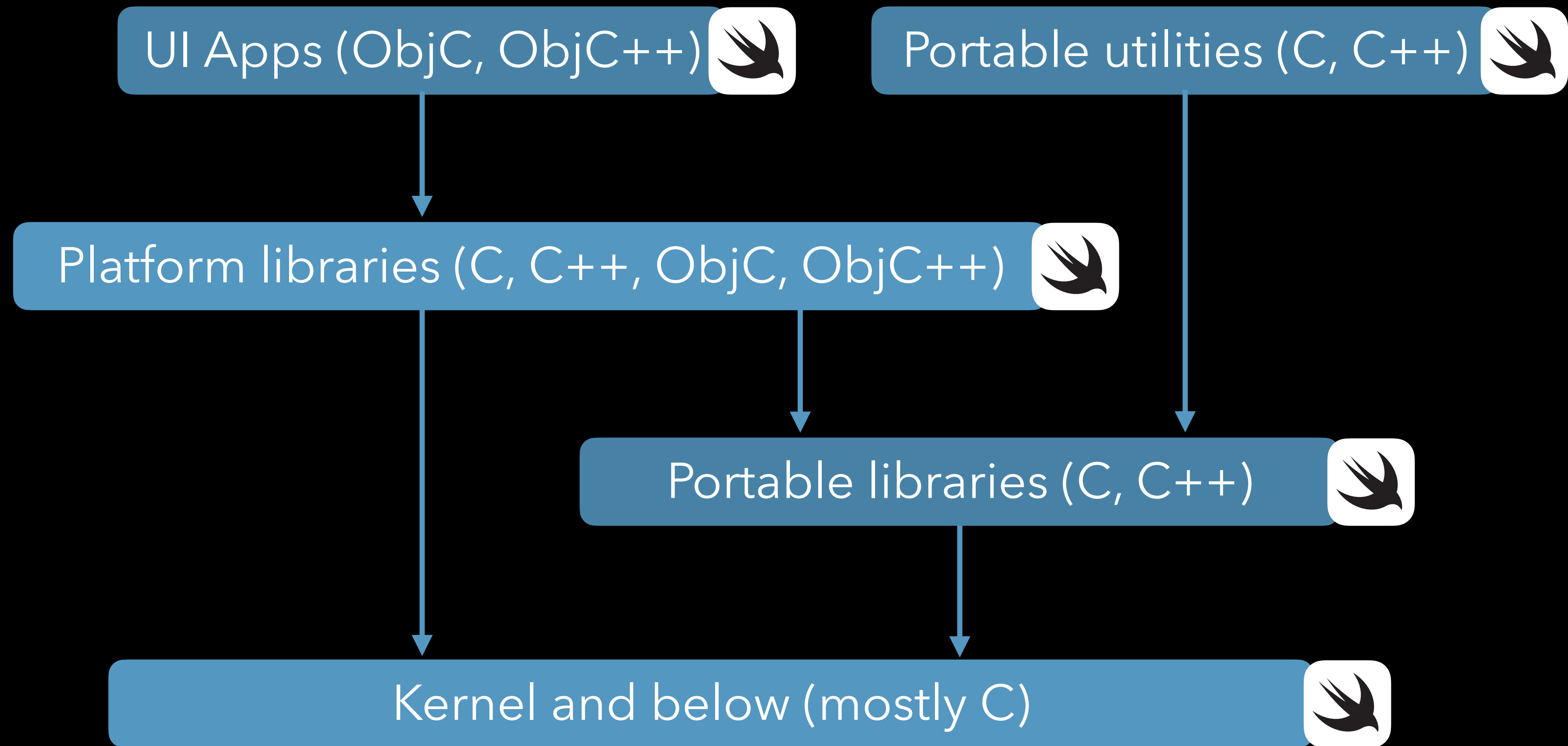
- A “successor” version of C or C++ would essentially be a new language
  - Migrating code to it would be a major project
  - But the language would still be deeply compromised by compatibility
- We can’t give up on trying to improve C and C++
  - But we need to acknowledge that it’s incremental and incomplete



# The need for a successor language

- A “successor” version of C or C++ would essentially be a new language
  - Migrating code to it would be a major project
  - But the language would still be deeply compromised by compatibility
- We can’t give up on trying to improve C and C++
  - But we need to acknowledge that it’s incremental and incomplete
- We need a new language that gets the defaults right

# An overview of Swift



# Local variables

```
let x = 4
```

```
x = 5
```

✗ Error: 'x' is immutable and cannot be reassigned

```
var y: Int
```

```
print(y)
```

```
y = 4
```

```
print(y)
```

✗ Error: 'y' is used here without always having been initialized

# Structs

```
struct Temperature {  
    var celsius: Double  
}
```

```
var t1 = Temperature(celsius: 10)  
var t2 = t1  
t1.celsius = 20  
t2 // still 10°C
```

# Structs

```
struct Temperature {  
    var celsius: Double  
  
    init(celsius: Double) {  
        self.celsius = celsius  
    }  
    init(fahrenheit: Double) {  
        self = Temperature(celsius: (fahrenheit - 32) * 4 / 9)  
    }  
}
```

# Non-mutating methods

```
extension Temperature {  
  func isBoiling() -> Bool {  
    return celsius >= 100  
  }  
}
```

# Mutating methods

```
extension Temperature {  
  mutating func increase(celsius change: Double) {  
    celsius += change  
  }  
}
```



# Collections

```
var array1 = [1, 2, 3, 4] // has type Array<Int>
```

```
var array2 = array1
```

```
array1.append(5)
```

```
array2 // still [1,2,3,4]
```

# Collections

```
extension Array {  
    func totalling<T: Numeric>(value: (Element) -> T) -> T {  
        var total: T = 0  
        for elt in self {  
            total += value(elt)  
        }  
        return total  
    }  
}
```

# Collections

```
extension Array {  
    func totalling<T: Numeric>(value: (Element) -> T) -> T {  
        var total: T = 0  
        for elt in self {  
            total += value(elt)  
        }  
        return total  
    }  
}
```

# Collections

```
extension Array {  
    func totalling<T: Numeric>(value: (Element) -> T) -> T {  
        var total: T = 0  
        for elt in self {  
            total += value(elt)  
        }  
        return total  
    }  
}
```

# Enums

```
enum MajorWeatherEvent {  
    case thunderstorm  
    case hurricane  
    case heatWave  
}
```

# Enums

```
enum MajorWeatherEvent {  
    case thunderstorm  
    case hurricane  
    case heatWave(Temperature)  
}
```

# Enums

```
extension MajorWeatherEvent {  
    func isHeatWave(over target: Temperature) -> Bool {  
        switch self {  
        case .thunderstorm: return false  
        case .hurricane: return false  
        case .heatWave(let temp): return temp > target  
        }  
    }  
}
```

# Classes

```
class WeatherStation {  
    let identifier: String  
    var lastRecordedTemperature: Temperature  
}
```



# Classes and exclusivity

```
extension WeatherStation {  
    func recordTemperature(_ value: Temperature) {  
        lastRecordedTemperature = value  
    }  
}
```

# Optionals

```
enum Optional<Unwrapped> {  
    case none  
    case some(Unwrapped)  
}
```

// Can also be written as `T?`

# Optional references

```
let station =      // has type Optional<WeatherStation>  
  allStations.first { s in s.identifier == "ABC" }
```

```
s.lastRecordedTemperature // type error
```

```
s?.lastRecordedTemperature // has type Optional<Temperature>
```

# Error handling

```
extension WeatherStation {  
    func update(from channel: ByteStream) throws {  
        recordTemperature(try channel.readDouble())  
    }  
}
```

# Error handling

```
extension WeatherStation {  
    func update(from channel: ByteStream) throws {  
        recordTemperature(try channel.readDouble())  
    }  
}
```

# Error handling

```
extension WeatherStation {  
    func update(from channel: ByteStream) throws {  
        recordTemperature(try channel.readDouble())  
    }  
}
```

# Concurrency

```
extension Connection {  
    func readString(callback: (String) -> ())  
    func readDouble(callback: (Double) -> ())  
}  
  
c.readString { identifier in  
    c.readString { temp in  
        allStations[identifier]?.updateTemperature(temp)  
    }  
}
```

# Concurrency

```
extension Connection {  
  func readString() async -> String  
  func readDouble() async -> Double  
}
```

```
let identifier = await c.readString()  
let temp = await c.readDouble()  
allStations[identifier]?.updateTemperature(temp)
```



# Data isolation

```
let station = allStations[identifier]

// Not allowed by default: WeatherStation is not thread-safe
// because it's a class reference with mutable properties
Task {
    station?.updateTemperature(temp)
}
```

# Swift as a successor language

# Adopting a new language

- Be incremental
- Keep moving forward
- Let the whole team be involved

# Adopting a new language

## Be incremental

- Massive, all-at-once rewrites are treacherous

# Adopting a new language

## Be incremental

- Massive, all-at-once rewrites are treacherous
  - Need to keep shipping the old version

# Adopting a new language

## Be incremental

- Massive, all-at-once rewrites are treacherous
  - Need to keep shipping the old version
  - The second-system effect adds risk and delays

# Adopting a new language

## Be incremental

- Massive, all-at-once rewrites are treacherous
  - Need to keep shipping the old version
  - The second-system effect adds risk and delays
  - Starts to look like a waste of time

# Adopting a new language

## Be incremental

- Much better to rewrite code incrementally



# Adopting a new language

## Be incremental

- Much better to rewrite code incrementally
  - Maintain one codebase

# Adopting a new language

## Be incremental

- Much better to rewrite code incrementally
  - Maintain one codebase
  - Keep the whole system working and shippable

# Adopting a new language

## Be incremental

- Much better to rewrite code incrementally
  - Maintain one codebase
  - Keep the whole system working and shippable
  - Immediately test and integrate any second-system rewrites

# Adopting a new language

## Keep moving forward

- Try to write new code in the new language first

# Adopting a new language

## Keep moving forward

- Try to write new code in the new language first
- Build up an understanding of the blockers

# Adopting a new language

## Keep moving forward

- Try to write new code in the new language first
- Build up an understanding of the blockers
- Prioritize rewrites that will enable more code to migrate

# Adopting a new language

Let the whole team be involved

# Adopting a new language

## Let the whole team be involved

- It's natural for people to be worried or ambivalent



# Adopting a new language

Let the whole team be involved

- It's natural for people to be worried or ambivalent
- Incremental progress is key:

# Adopting a new language

Let the whole team be involved

- It's natural for people to be worried or ambivalent
- Incremental progress is key:
  - No artificial boundaries between the rewriters and the maintainers

# Adopting a new language

Let the whole team be involved

- It's natural for people to be worried or ambivalent
- Incremental progress is key:
  - No artificial boundaries between the rewriters and the maintainers
  - Gets people weighing changes technically instead of reacting in the abstract

# Adopting a new language

Let the whole team be involved

- It's natural for people to be worried or ambivalent
- Incremental progress is key:
  - No artificial boundaries between the rewriters and the maintainers
  - Gets people weighing changes technically instead of reacting in the abstract
  - Creates opportunities for people to learn and come along at their own pace

# Enabling incremental adoption

- Incremental adoption requires small-scale interoperation
- Must generate linkable object files to fit into build systems
- Must be able to use interfaces from predecessor languages
- Must be able to provide interfaces to predecessor languages

# Interoperation with C and Objective-C

- Swift has supported file-by-file interoperation with C and Objective-C from 1.0
  - Can import headers and interpret them as Swift declarations
  - Can export a header describing Swift declarations
- No need for things like process separation or a complex FFI

# POSIX in C

```
int dup(int);
```

```
int dup2(int, int);
```

```
int pipe(int [2]);
```

```
ssize_t read(int, void *, size_t);
```

# POSIX in Swift

```
func dup(Int32) -> Int32
```

```
func dup2(Int32, Int32) -> Int32
```

```
func pipe(UnsafeMutablePointer<Int32>) -> Int32
```

```
func read(Int32, UnsafeMutableRawPointer?, Int) -> Int
```



# CoreGraphics in C

```
typedef struct CGColorSpace *CGColorSpaceRef;  
  
CGColorSpaceRef CGColorSpaceCreateWithName(CFStringRef name);  
  
CFStringRef CGColorSpaceCopyName(CGColorSpaceRef space);  
  
bool CGColorSpaceSupportsOutput(CGColorSpaceRef space);
```

# Example

```
class CGColorSpace {  
    init?(name: CFString)  
  
    var name: CFString? { get }  
  
    var supportsOutput: Bool { get }  
}
```

# Interoperation with C++

- Swift has been adding the same interop capabilities for C++
- Same basic design as for C and Objective-C:
  - Can import headers into Swift and interpret them as Swift declarations
  - Can export headers that expose Swift declarations

# Adding Swift to your build system

- cmake directly supports building `.swift` files into targets

```
cmake_minimum_required(VERSION 3.26)
```

```
project(hello LANGUAGES CXX Swift)
```

```
add_executable(hello
```

```
    MyLib.cpp
```

```
    Hello.swift)
```

```
target_compile_options(hello PUBLIC
```

```
    "$<$<COMPILE_LANGUAGE:Swift>:-cxx-interopability-mode=default>")
```

# Importing Swift into C++

- Actually quite a bit more powerful than importing into C and Objective-C
- C++ headers are very expressive
- Can work retroactively with any Swift library
- Can use almost any Swift declaration, even generic functions

# Importing Swift into C++

```
public struct Temperature {  
    public var celsius: Double  
    public init(celsius: Double)  
}
```

# Importing Swift into C++

```
class Temperature final {  
    double getCelsius() const;  
    void setCelsius(double value);  
};
```

# Importing C++ into Swift

- C has very simple default import rules
- Objective-C has strong idioms that allow very idiomatic import
- C++ is a very rich language without many consistent idioms



# Importing safely into Swift

- A lot of C++ interfaces assume they're used in certain patterns
  - Methods return references into this (or dependent on it)
  - Constructors capture references
- We don't want C++ import to completely compromise Swift safety

# Importing safely into Swift

- Swift makes reasonable default assumptions about C++ APIs:
  - Reference parameters will honor their const-ness, won't escape
  - Reference return values depend on arguments, most likely `this`
  - Similar rules apply to "view types" like `std::span`

# Importing safely into Swift

- Swift makes reasonable default assumptions about C++ APIs:
  - Reference parameters will honor their const-ness, won't escape
  - Reference return values depend on arguments, most likely `this`
  - Similar rules apply to "view types" like `std::span`
- Can override defaults with attributes

# Importing safely into Swift

- Some amount of unsafety is inevitable given interoperation
- Goal is to build up more code using safe features
  - Swift using C++ shouldn't be less safe than C++ using C++
  - Swift can also enforce stronger rules, e.g. around exclusivity
  - Swift can encourage more reliable patterns

# C++ functions

```
std::vector<std::string> getWeatherStationNames();
```

# C++ value types

```
func getWeatherStationNames() -> std::vector<std::string>
```

# C++ value types

```
struct std.vector<std.string>: Collection {  
    func size() -> Int  
    subscript(index: int) -> std.string { get set }}
```

# Containers

- Importer can recognize types that look like containers
- Automatically imported with a Collection conformance
  - Swift algorithms can be automatically used with imported containers
  - Iterator use can be abstracted within functions that provably use them right



# Non-copyable types

- Not all C++ types are copyable
- Swift has advanced ownership features that can express this
- Swift can import a type like `std::unique_ptr` as a non-copyable struct
- Swift understands data flow directly and doesn't need `std::move`

# Non-value types

- Not all C++ classes are meant to be used as value types
- Like the CoreGraphics example, common to only pass some classes indirectly
  - Idiomatically, often wrapped in smart pointers
- Can annotate these classes to import as Swift classes
- Plenty of work still to improve the ergonomics

# FoundationDB

- Open source distributed key-value database created in 2013
- ~500k lines of C and C++ code
- Swift adoption: <https://github.com/apple/foundationdb/pull/10156>

# FoundationDB

```
struct SWIFT_CXX_REF MasterData : NonCopyable,  
                                   ReferenceCounted<MasterData> {  
    ...  
    Version minKnownCommittedVersion;  
    ...  
};
```

# FoundationDB

```
public func updateLiveCommittedVersion(myself: MasterData,  
                                       req: ReportRawCommittedVersionRequest) {  
    myself.minKnownCommittedVersion =  
        max(myself.minKnownCommittedVersion,  
            req.minKnownCommittedVersion)  
  
    ...  
}
```

# FoundationDB

```
public func updateLiveCommittedVersion(myself: MasterData,  
                                       req: ReportRawCommittedVersionRequest) {  
    myself.minKnownCommittedVersion =  
        max(myself.minKnownCommittedVersion,  
            req.minKnownCommittedVersion)  
  
    ...  
}
```

# FoundationDB

```
public func updateLiveCommittedVersion(myself: MasterData,  
                                       req: ReportRawCommittedVersionRequest) {  
    myself.minKnownCommittedVersion =  
        max(myself.minKnownCommittedVersion,  
            req.minKnownCommittedVersion)  
  
    ...  
}
```

# Work in progress

- Still lots of room for improvement
- Some hard questions left around ownership and safety
- Nothing we think we can't solve



# In summary

- Predecessor languages will stay with us for a long time
- We can and should work to improve those languages
- We also need to start moving on

# Questions?