A Tour of ADT

The LLVM Developer's Toolbox

Jakub Kuderski <jakubk@openxla.org>
Nod.ai

October 11 2023
US LLVM Devs' Meeting
Santa Clara, CA

Contributed to a Number of LLVM-based Projects

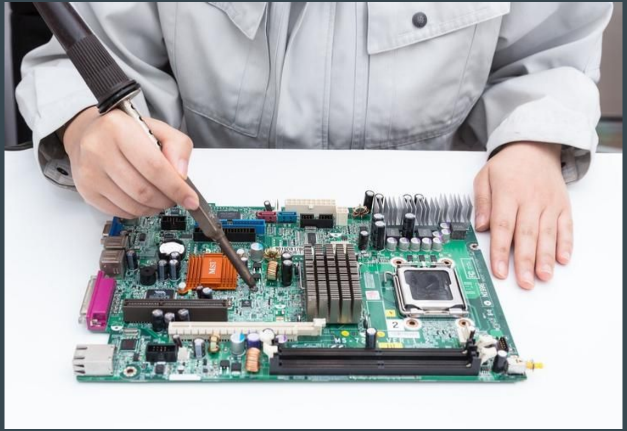
- LLVM
- Clang
- MLIR

- IREE (ML Compiler and Runtime)
- LLPC (Shader Compiler)
- SeaDsa (Pointer Analysis)
- SeaHorn (Program Verification)

Coding Interviews

With LLVM libraries

Plain C++



Distinct ‘Feeling’

- Opinionated coding style
- Opinionated testing style (LIT, FileCheck)

- Common utilities
  - ADT
  - Support
  - Command Line
  - TableGen

Agenda

1. Background and Motivation
   - Recap: STL in C++

2. LLVM's ADT
   - Better ergonomics
   - Backports from new C++ revisions

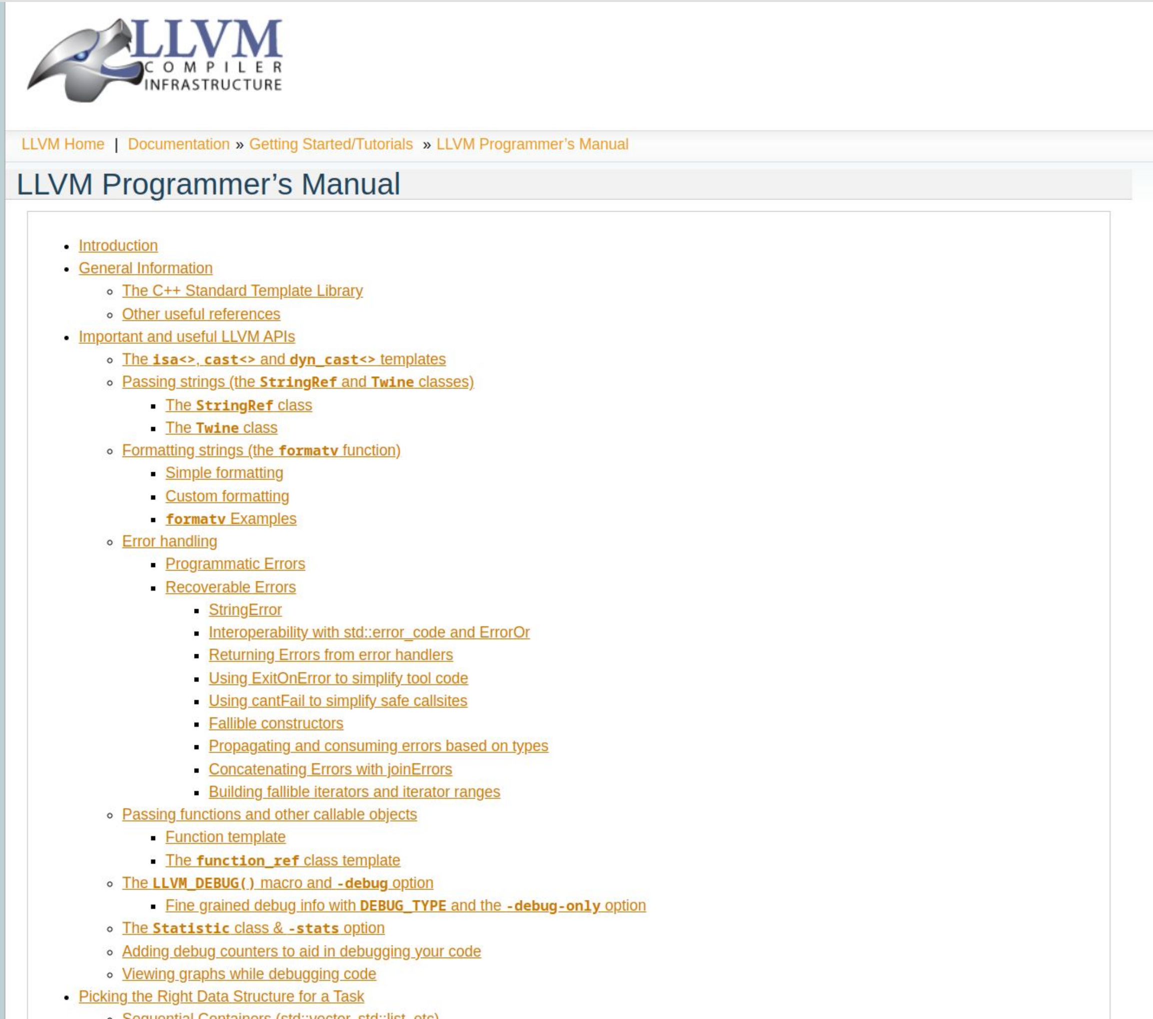
3. Top 10 Utilities You Won't Find in STL
   - Number 7 will surprise you

4. Contributing to ADT
   - Practical advice

My Goals

and Your Expectations

- Beginner-friendly
  - Don’t need to be an LLVM or C++ Expert
- Overview of the most useful APIs
  - Won’t go very deep into implementation details
- Also focus on the *why*
- How to contribute
  - Explain conventions
  - How to prototype and test



LLVM Home | Documentation » Getting Started/Tutorials » LLVM Programmer’s Manual

LLVM Programmer’s Manual

llvm.org/docs/ProgrammersManual.html

Recap: STL in C++

- Part of the Standard Library
- Traditionally divided into 3 components
  - Algorithms (e.g., `std::find`, `std::size`, `std::remove_if`)
  - Containers (e.g., `std::vector`, `std::unordered_map`, `std::valarray`)
  - Iterators
- Also type traits, e.g., `std::enable_if`, `std::is_same`

Recap: STL in C++

- Generic over types
- Composable
- Generally well-tested, robust implementation

```cpp
std::vector<my::Decimal> numbers = foo();

std::sort(numbers.begin(), numbers.end());

auto it = std::lower_bound(numbers.begin(), numbers.end(), x);
```

Motivation: ADT in LLVM

- Abstract Data Types
- Collection of custom containers, utility functions, algorithms

- Most of the ADT code is general-purpose, some LLVM-specific
  - Provides extra data structures, iterators, algorithms, type traits missing from C++
    - Backports of C++ features from future standards (e.g., C++20)
  - Used across most of llvm-project
  - Attempts to make the implementation simpler, more concise, safer, faster
  - … while relying on LLVM-specific assumptions

ADT – Teaser

```
SmallVector<llvm::APInt> numbers = bar();

llvm::sort(numbers);

for (auto [idx, number] : enumerate(numbers)) {
  // ...
}

if (is_contained(numbers, APInt::getZero(64))) {
  // ...
}
```

LLVM-specific Assumptions

- No exceptions
- No allocators as container template parameters
- No API or ABI stability guarantees
- Less defensive implementation (e.g., no underscores)
- Cater for uses inside llvm-project only

Efficient Containers

- `Small*`
- `Dense*`
- `Sparse*`
- `*String*`
- `*Ref`
- `*BitVector`

ADT 101 – SmallVector

`std::vector<T>`

`std::vector<T, Allocator>`

Internally:

- `pointer begin`
- `pointer end`
- `pointer end_capacity`

`llvm::SmallVector<T>`

`llvm::SmallVector<T, N>`

Internally:

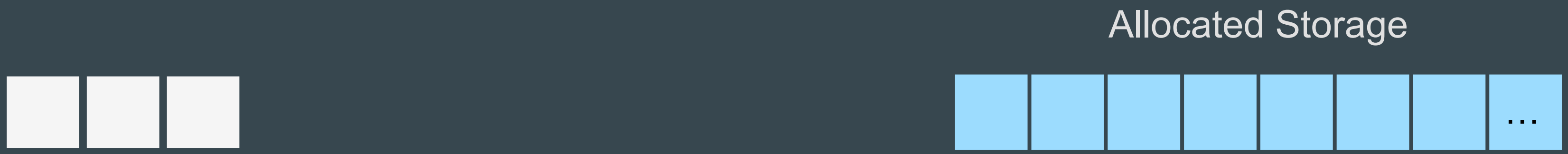
- `pointer begin`
- `SizeType size`
- `SizeType capacity`
- `char storage[SmallSize]`

ADT 101 – SmallVector

```
SmallVector<int> foos; // SmallVector<int, 12>
static_assert(sizeof(foos) == 64);

for (int i = 0; i < 12; ++i)
  foos.push_back(i);

foos.push_back(99); // First allocation.
```



ADT 101 – SmallVector

```
static_assert(sizeof(SmallVector<int, 0>) < sizeof(std::vector<int>));
static_assert(sizeof(SmallVector<int, 2>) == sizeof(std::vector<int>));
static_assert(sizeof(SmallVector<int, 4>) > sizeof(std::vector<int>));

static_assert(sizeof(SmallVector<char, 0>) == sizeof(std::vector<char>));
```

ADT 101 – SmallVector

```
auto getNums() { return seq(99); }

TEST(ADTTour, ToVector) {
  auto numVec = llvm::to_vector(getNums());
  auto numVecSz = llvm::to_vector_of<size_t>(getNums());
}
```

Other Small Containers

- `SmallSet<T, N>`, `SmallDenseMap<K, V, N>`
- `SmallPtrSet<T, N>`
- `TinyPtrVector<T>`
- `SmallString<N>`
- `SmallBitVector`

ADT 101 – DenseMap

`std::unordered_map<T>`

Internally:

- Array of buckets
- Buckets with linked list of entries
- Separate chaining

`llvm::DenseMap<T>`

Internally:

- Flat, open addressing
- Quadratically probed

ADT 101 – StringRef

`const char *`

`std::string`

`std::string_view`

`llvm::StringRef`

`llvm::StringLiteral`

Internally:

- `pointer begin`
- `SizeType size`

ADT 101 – ArrayRef

`T *`

`std::vector<T>`

`std::span<T>`
`std::span<const T>`

`llvm::ArrayRef<T>`

`llvm::MutableArrayRef<T>`

Internally:

- `pointer begin`
- `SizeType size`

Ranges

and Iterators

- Range-oriented APIs
- `zip*` and `enumerate`
- `reverse`
- `drop_*` and `take_*`
- `map_range`
- `make_filter_range`

Range Wrapper Functions

```
std::sort(x.begin(), x.end())             llvm::sort(x)

std::find(x.begin(), x.end(), v)          llvm::find(x, v)

std::all_of(x.begin(), x.end(), P)        llvm::all_of(x, P)
```

- More concise
- Safer:
    - Cannot confuse objects, e.g., `find(x.begin(), y.end(), v)`
    - Expensive checks, e.g., shuffle before unordered sorting

Custom Range Functions

`llvm::reverse(range)`

`llvm::is_contained(range, v)`

- Is v an element of `range`?

`llvm::all_equal(range)`

- `All elements equal?`

`llvm::append_range(container, newValues)`

Simplified Logic

```cpp
enum class Kind { A, B, C, D, E, F };

void bar(Kind k) {
  if (k == Kind::A || k == Kind::B || k == Kind::C)
    foo();
}

void baz(Kind p, Kind q, Kind r, Kind s) {
  if (p == q && p == r && p == s)
    foo();
}
```

Simplified Logic

```
enum class Kind { A, B, C, D, E, F };

void bar(Kind k) {
  if (is_contained({Kind::A, Kind::B, Kind::C}, k))
    foo();
}

void baz(Kind p, Kind q, Kind r, Kind s) {
  if (all_equal({p, q, r, s}))
    foo();
}
```

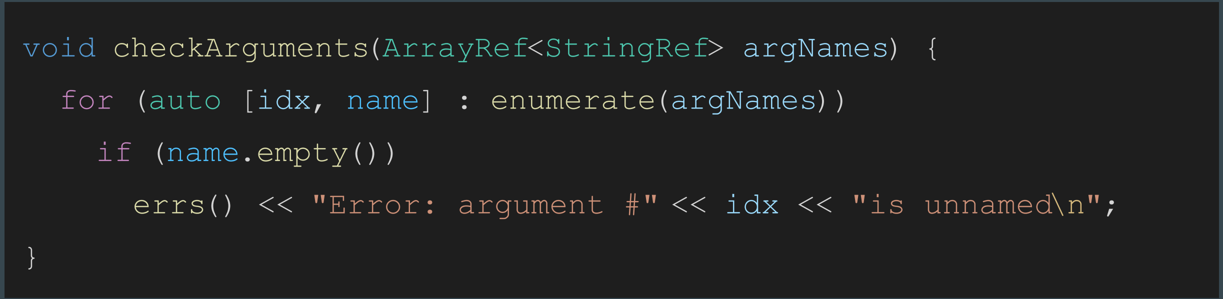
Simplified Appends

```
void processFeatures() {
    SmallVector<Kind> supported;
    // ...
    if (isTargetA()) {
        supported.push_back(Kind::A);
        supported.push_back(Kind::B);
        supported.push_back(Kind::C);
    }
    // ...
}
```

Simplified Appends

```
void procesFeatures() {
  SmallVector<Kind> supported;
  // ...
  if (isTargetA()) {
    Kind ks[] = {Kind::A, Kind::C, Kind::E};
    llvm::append_range(supported, ks);
  }
  // ...
}
```

Iteration Functions – enumerate

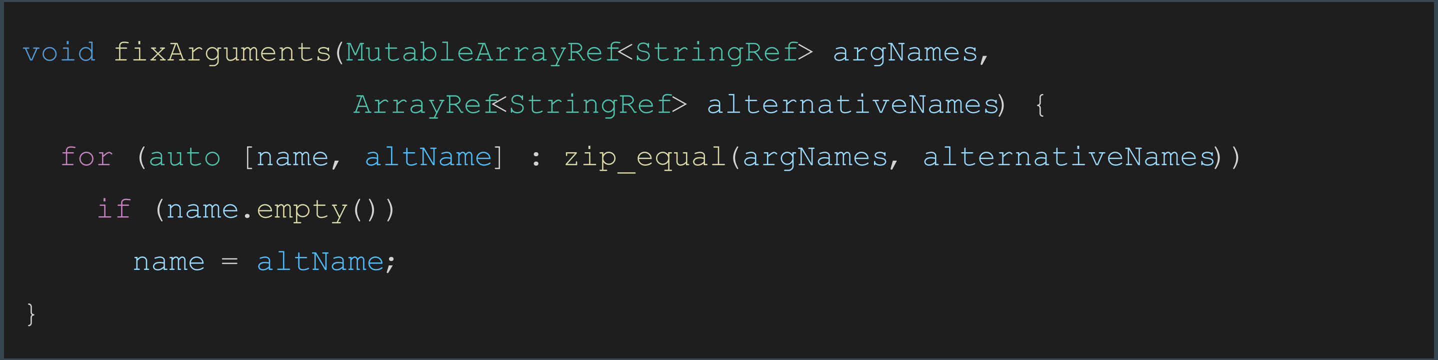


```
void checkArguments(ArrayRef<StringRef> argNames) {
  for (auto [idx, name] : enumerate(argNames))
    if (name.empty())
      errs() << "Error: argument #" << idx << "is unnamed\n";
}
```

Iteration Functions – enumerate

```
void fixupArguments(MutableArrayRef<StringRef> argNames,
                    ArrayRef<StringRef> alternativeNames) {
  for (auto [idx, name, altName] : enumerate(argNames, alternativeNames)) {
    if (name.empty()) {
      errs() << "Warning: argument #" << idx << "is unnamed\n";
      name = altName;
    }
  }
}
```

Iteration Functions – zip*



```
void fixArguments(MutableArrayRef<StringRef> argNames,
                  ArrayRef<StringRef> alternativeNames) {
  for (auto [name, altName] : zip_equal(argNames, alternativeNames))
    if (name.empty())
      name = altName;
}
```

- `zip_equal` – requires all input ranges have the same length
- `zip` – iteration stops when the end of the shortest range is reached
- `zip_first` – requires the first range is the shortest one
- `zip_longest` – iteration continues until the end of the longest range is reached

Sequences

- `seq`
- `enum_seq`
- `EnumeratedArray`

seq

```
auto someNumbers() {
  return llvm::map_to_vector(seq(1, 100),
                             [](int v) { return std::to_string(v); });
}
```

- `seq(100)` – generates 0..99 inclusive
- `seq(1, 100)` – generates 1..99 inclusive
- `seq_inclusive(1, 100)` – generates 1..100 inclusive

enum_seq

- Opt-in – not all enums are contiguous
- Enabled through a trait specialization
- Works with enums defined outside of llvm

```
enum class Release : int {
  r1 = 1,
  r2 = 2,
  r3 = 3,
};

template <> struct enum_iteration_traits<Release> {
  static constexpr bool is_iterable = true;
};

void foo() {
  for (Release r : enum_seq_inclusive(Release::r1, Release::r3))
    llvm::outs() << "Release " << static_cast<int>(r) << "\n";
}
```

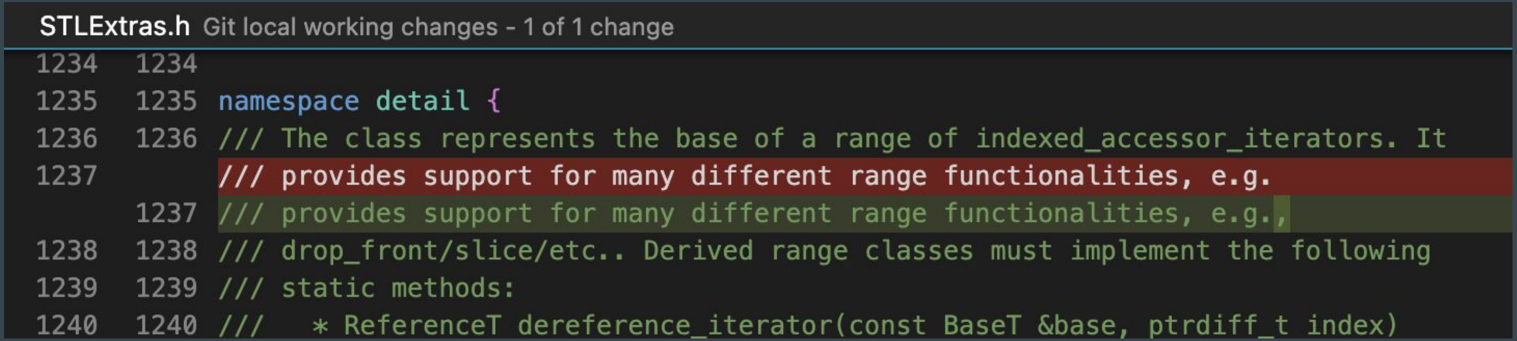
Graphs

- `depth_first`
- `post_order`
- `scc_iterator`
- `GraphTraits`

Misc

- `PtrIntPair`
- `scope_exit`
- `is_detected`

How To Contribute



STLExtras.h Git local working changes - 1 of 1 change

```

namespace detail {
/// The class represents the base of a range of indexed_accessor_iterators. It
/// provides support for many different range functionalities, e.g.
/// provides support for many different range functionalities, e.g.,
/// drop_front/slice/etc.. Derived range classes must implement the following
/// static methods:
///   * ReferenceT dereference_iterator(const BaseT &base, ptrdiff_t index)
```

```
➜ ninja check-all
[0/4873] Building CXX object lib/Support/CMakeFiles/LLVMSupport.dir/...
```

Overcoming Long Compilation and Test Times

Problem: ADT code is included by almost every `.cpp` file.

1. Iterate inside unit tests when possible
   - ○ Isolate and dump reproducer data

2. Compile and run the ADT tests **only**
   - ○ Filter the test cases executed

3. Run the full test suite before submitting for review / landing
   - ○ Use ccache and a fast linker (mold, lld)

```
➜ ninja unittests/ADT/ADTTests

➜ unittests/ADT/ADTTests

➜ unittests/ADT/ADTTests \
    --gtest_filer='MyTest.*'

➜ ninja check-all
```

Testing with gtest

```
TEST(ADTTour, Basic) {
  SmallVector<StringRef> Names;
  EXPECT_TRUE(Names.empty());
  EXPECT_EQ(Names.size(), 0u);

  Names.push_back("Alice");
  EXPECT_FALSE(Names.empty());
  EXPECT_EQ(Names.size(), 1u);

  EXPECT_TRUE(is_contained(Names, "Alice"));
}
```

Testing with gmock

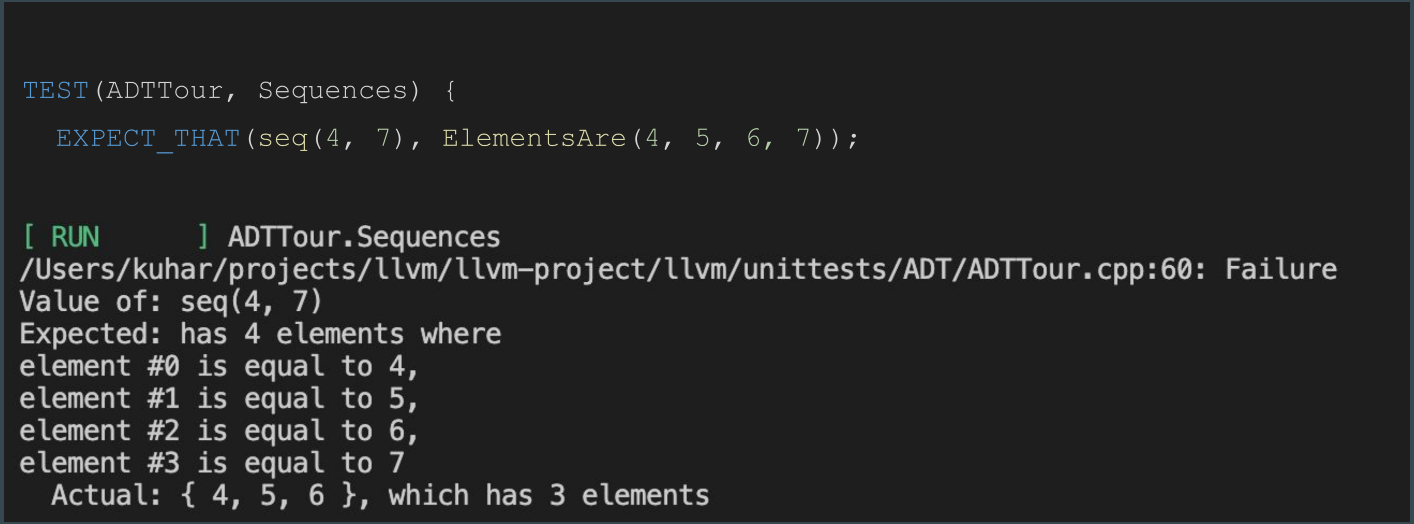
```
#include "gmock/gmock.h"

namespace {
using namespace llvm;
using ::testing::ElementsAre;
using ::testing::UnorderedElementsAre;
using ::testing::Pair;

TEST(ADTTour, Sequences) {
  EXPECT_THAT(seq(4, 7), ElementsAre(4, 5, 6));

  SmallDenseMap<StringRef, int> StrToNum = {{"1", 1}, {"2", 2}, {"3", 3}};
  EXPECT_THAT(StrToNum,
              UnorderedElementsAre(Pair("3", 3), Pair("1", 1), Pair("2", 2)));
}
```

Testing with gmock



```
TEST(ADTTour, Sequences) {
  EXPECT_THAT(seq(4, 7), ElementsAre(4, 5, 6, 7));
```

```
[ RUN      ] ADTTour.Sequences
/Users/kuhar/projects/llvm/llvm-project/llvm/unittests/ADT/ADTTour.cpp:60: Failure
Value of: seq(4, 7)
Expected: has 4 elements where
element #0 is equal to 4,
element #1 is equal to 5,
element #2 is equal to 6,
element #3 is equal to 7
  Actual: { 4, 5, 6 }, which has 3 elements
```

‘Death’ Tests

```
TEST(ADTTour, Death) {
  SmallVector<StringRef> Names;
  EXPECT_TRUE(Names.empty());
  EXPECT_EQ(Names.size(), 0u);

#if defined(GTEST_HAS_DEATH_TEST) && !defined(NDEBUG)
  EXPECT_DEBUG_DEATH(Names[1], "idx < size()");
#endif
}
```

Note: In excess, death tests can be very slow

- Especially with dynamic libraries

Do not Land on Friday Evening

- Plethora of build configurations and toolchains
- Expect some build bots to take >hours to pick up changes
- You may need to work around compiler bugs

Recap

Key Points

- Data locality for general performance and quick fast-path code
- Range-based interfaces: ergonomics, less bug-prone
- Debug checks with `assert`, fast release code
- Customization and opt-in via traits
- Specific build and test targets for faster prototyping

Thank you.

Questions?