Battle.net infrastructure

- About 325,000 lines of C++
  - Servers + client libraries
- "Battle.net Game Service"
  - Authenticate players
  - Social: friends, presence
  - Matchmaking (cooperative/competitive)
  - Achievements/profiles

- Explain what battle.net does.
- Game service vs Battle.net umbrella.
- Define terms.

Battle.net is highly…

- Distributed
- Asynchronous
- Configured
- Fault-prone
- Architecture-varied
  - inheritance
  - composition
  - value-oriented

- Many machines connected.
- Almost everything asynchronous, callback-driven.
- Lots of configuration read at startup time from git repo.
- Code is pretty good, but size => faults occur.

└A familiar situation

A familiar situation

- No practice at unit testing
- Large project with many moving parts
- Mature lower level libraries
- New code (features) added at an alarming rate

- A familiar situation for me and my colleagues.
- Game industry is not accustomed to unit testing.

└What’s typically well-tested?

What’s typically well-tested?

- UTF-8 string conversion
- String interpolation
- URL parsing/decomposition
- Stats/math code

These things are "easy mode" for tests.

- Usually well-tested.
- Not worth thinking about edge cases - can use off-the-shelf tests (eg UTF-8).

Not-so-well tested?

- Matchmaking algorithms
- Queueing/Load balancing algorithms
- Other high-dependency, asynchronous, "large" code

These things are harder to test. Where to start?

- This is what I started to think about.
- I read Kent Beck and Bob Martin. I watched Misko Hevery.
- Conclusion: we aren't practised at testing.
- Need to practise - use TDD
- Extend unit testing framework as I go

No magic bullet

- I wrote a lot of mocks
- Set up a lot of data structures for test
- A lot of testing code to keep bug-free
- But along the way I found
  - better code structure
  - useful techniques

- My journey.
- Things were messy for a while. (They even shipped messy.)
- But I found some useful things to share.

2015-04-08

Testing Battle.net
└Testing legacy code
└Monolithic classes

Monolithic classes

Problem 1: Getting started testing huge legacy classes.

(What idiot wrote this code? Oh, it was me, 3 months ago...)

- in a codebase this size, there are some classes that get large
- and they do complex things
- and we need to test them

2015-04-08

Testing Battle.net
└Testing legacy code
└Exhibit A: hard to test

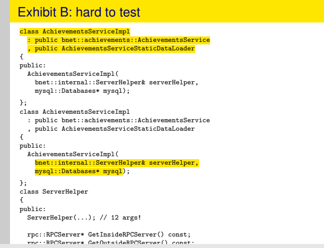
Exhibit A: hard to test

```
class ChannelBase : public rpc::Implementor<protocol::channel::Channel>;
class ChannelImpl : public ChannelBase;

class PresenceChannelImpl : public ChannelImpl
{
public:
  PresenceChannelImpl(
    Process* process,
    rpc::RPCDispatcher* insideDispatcher,
    const EntityId& entityId,
    ChannelDelegate* channelDelegate,
    ChannelOwner* owner,
    const PresenceFieldConfigMap& fieldMap);
};
class ChannelBase : public rpc::Implementor<protocol::channel::Channel>;
class ChannelImpl : public ChannelBase;

class PresenceChannelImpl : public ChannelImpl
{
public:
  PresenceChannelImpl(
    Process* process,
    rpc::RPCDispatcher* insideDispatcher,
    const EntityId& entityId,
    ChannelDelegate* channelDelegate,
```

- Explain types.
- Deep inheritance that mixes concerns.
  1. What is RPC doing in there?
  2. And protocol dependency.
  3. "Traditional" interface-impl hierarchy.
- Constructor takes 6 args.
  1. Some constructor args have a wide interface.
  2. Again RPC.
  3. Lots of configuration.
  4. These things are onerous to mock.

2015-04-08

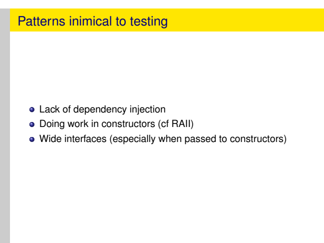
Testing Battle.net
└─Testing legacy code
  └─Exhibit B: hard to test



- Achievements actually quite well-tested
- Again the pattern of deriving from protocol
- Static data loader => IO going on in constructor?
- Some DI going on (database interface)
- Constructor args have wide interfaces
- ServerHelper legitimized the pattern of coupling IO/RPC and functionality

2015-04-08

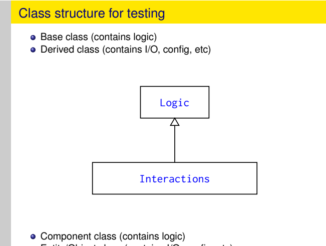
Testing Battle.net
└─Testing legacy code
  └─Patterns inimical to testing



- Everyone tells us that dependency injection is required for testing
- But it’s not enough
- RAII is bad: testable things shouldn’t own resources
- Wide interfaces to construction are bad

2015-04-08

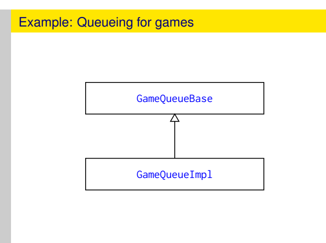
Testing Battle.net
└─Testing legacy code
└─Class structure for testing



- Instead of "traditional" interface-impl split
- Use the split of logic vs interactions
  - Logic in base
  - Interactions in derived
  - Derived has as few dependencies as possible
  - Ruthlessly inject dependencies
- Good news: this is quite easy to apply

2015-04-08

Testing Battle.net
└─Testing legacy code
└─Example: Queueing for games



- Explain queueing for games.
- Manage multiple queues.
- Server capacity, link capacity. KR/TW problem.
- Rate limiting even in the presence of adequate server capacity.

2015-04-08

Testing Battle.net
└─Testing legacy code
└─Queueing for games

Queueing for games

`GameQueueBase` contains the queueing logic

```
class GameQueueBase
{
public:
  GameQueueBase(
    shared_ptr<ServerPoolInterface> interface,
    const PopCallback& popCb,
    const UpdateCallback& updateCb,
    const PollTimerCallback& pollTimerCb,
    const NotificationTimerCallback& notificationTimerCb);

  bool   Push(...);
  size_t Pop(...);
  void   Remove(...);
  size_t PollQueue(...);

  ...
};
class GameQueueBase
{
public:
  GameQueueBase(
```

- Moderately complex queueing logic all in the base.
- Logic in standalone class: no RPC inheritance.
- Constructor args have narrow interfaces.
  - callbacks (1-function interface)
  - server pool: a couple of functions for server capacity information
- Interface not cluttered with other concerns: just queueing stuff.

2015-04-08

Testing Battle.net
└─Testing legacy code
└─Queueing for games

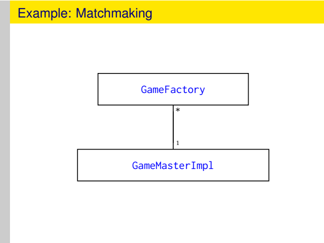
Queueing for games

`GameQueueImpl` deals with protocols

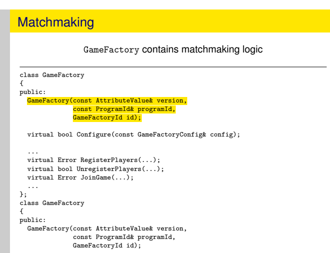
```
class GameQueueImpl
  : public GameQueueBase
  , public protocol::game_queue::GameQueue
{
public:
  // protocol handler functions
  virtual void AddToQueue(...);
  virtual void RemoveFromQueue(...);
  ...

  // system events
  bool OnInit(...);
  bool OnFlush(...);
  void OnShutdown(...);
  void OnPeerDisconnected(...);
  ...
};
```

`GameQueueImpl` deals with protocols

- Derive impl from base, using the logic-interaction divide
- Derived class implements
  - rpc calls
  - config
  - interaction with system
- Some of this stays at the level of the impl
- Some is dependency-injected to control the logic
  - keep base testable with as little setup as poss

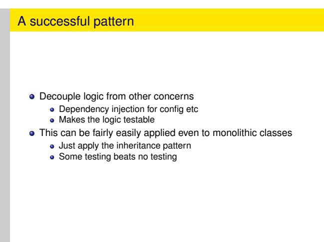


- Explain matchmaking
  - composition-based
  - game factory segments player base by difficulty, act, hardcore/non
  - deals with arbitrary groups of players
  - lots of telemetry
  - matching by attributes
- game factory implements matchmaking strategy



- Small constructor interface
- Configuration required, but deferred => default config will be testable
  - Constructor leaves object initialised properly
- Just the MM logic in factory

- system events
- config injection
- rpc interface



- Side effect: not bad for optimization
  - layout: logic members at start of class
- If you have monolithic classes, you can start splitting logic out as a base class
  - you get <u>something</u> testable
  - once you have something testable, you can build on it
  - tested code is easier to refactor even if it starts out ugly

Testable classes

Dependency injection is probably the biggest factor affecting whether or not code *is testable at all.*

Even with DI, classes are *onerous to test* unless constructors take few arguments, using narrow interfaces.

- A practical guideline

Testing for scalability

Problem 2: Confidence in my code's ability to scale.

(I don't want a 3am call from devops.)

- The code has to work when a million players come along

Testing Performance/Efficiency

- Different solutions for
  - thousands (performance)
  - millions (performance + algorithms)
  - billions (algorithms by construction)
- Battle.net's working sets are in the millions
  - e.g. matchmaking

- data set in the thousands =>
  - performance is king (cache effects etc)
  - algorithms not really important
- billions =>
  - usually highly distributed (can't run on dev machine)
  - use abstract algebra to achieve correct-by-construction algorithms
- millions =>
  - can run on a single machine
  - performance is important (caching)
  - but algorithms are also important
  - can run on dev machines but without scalable data sets

Problems in million-land

- Computations can run on a single machine
- Data structures are important to performance
  - Caching concerns, optimizations can get you 100x
  - But they can't get you 100,000x
- Algorithms are important to efficiency

- Perf only gets you so far
- You need algorithms to avoid blowup at scale

└Testing for performance

Testing for performance

- Timed tests are easy, not so useful
- My machine is a Windows desktop
- Production machine is a CentOS blade
- Timed tests
  - compare times when optimizing
  - can't tell me if code is fast enough in an absolute sense

- Of course I can time tests
- This can give me some gross idea of optimizations
- It's still hard to do things properly (my desktop isn't the production hardware)

└Efficiency: easy to lose

Efficiency: easy to lose

- Team of engineers hacking away on features
- $O(\log n)$ or less is required
- Easy to accidentally turn it into $O(n)$ (or worse)
- I need a way to test for algorithmic efficiency

- I work with good engineers, but we're all human
- I was concerned about this
- I want the computer to help enforce this

Testing for efficiency

- Run the same test with different sized inputs

$$T_1 = (\textit{time for run on data of size } N)$$
$$T_2 = (\textit{time for run on data of size } kN)$$

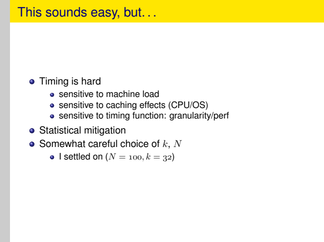
$$\begin{aligned} T &\propto N \\ T_1 = T(N) &= aN \\ T_2 = T(kN) &= akN \\ \frac{T_2}{T_1} &= k \end{aligned}$$

- empirical method
- explain

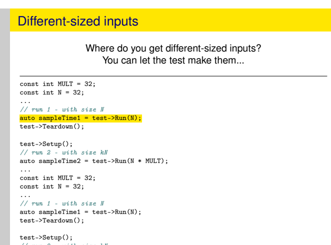
Common cases

$$O(1) \Rightarrow \frac{T_2}{T_1} = 1$$
$$O(log\, n) \Rightarrow \frac{T_2}{T_1} = 1 + \frac{log(k)}{log(N)}$$
$$O(n) \Rightarrow \frac{T_2}{T_1} = k$$
$$O(n\, log\, n) \Rightarrow \frac{T_2}{T_1} = k\,(1 + \frac{log(k)}{log(N)})$$
$$O(n^2) \Rightarrow \frac{T_2}{T_1} = k^2$$

- simple math to get figures for each bucket I care about



- Statistical mitigation = run multiple times, discard outliers, average
  - be clear: this is for machine effects
  - multiple runs occur on the same data
- constants need to be big enough to elicit the required effect
- but small enough not to make the test slow
- fast, high frequency timing function is desirable
- The nice thing is that you don't need to run this optimized
  - optimization tends only to make things better



- Affects the timing if done naively (i.e. wrongly)
  - Adds an $O(n)$ component to the test
  - So move the timing code inside the test also
- Boilerplate in test code
- It's not ideal. . .

2015-04-08

Testing Battle.net
└─Testing scalability (I)
└─Let the test make them?

Let the test make them?

Result: a typical test

- ~40 lines setup
- ~10 lines timing
- ~5 lines actual logic
- ~5 lines test macros

Yuck.

- I was working with objects that needed some setup
- monolithic classes, remember?

2015-04-08

Testing Battle.net
└─Testing scalability (I)
└─Let the test make them?

Let the test make them?

- It works well enough to give me confidence
  - Matchmaking won't blow up with a million players
- So I lived with this for a while. . .
- But I'm lazy, I don't want to maintain all this code

- Shipped with this because sometimes good enough is good enough

Autogenerating test inputs

Problem 3: Generating test input automatically.

(Laziness, Impatience, Hubris. Check.)

- I'm a student of Haskell (Quickcheck)
- The idea of property-Based testing
- Usually established in languages with reflection
- Or sufficiently powerful type systems
- Explain property-based testing

Wish-driven development

What I have

```
DEF_TEST(TestName, Suite)
{
  ...
  return test_result;
}
```

What I want

```
DEF_PROPERTY(TestName, Suite, const string& s)
{
  // do something with s
  // that should be true for any input
  ...
  return property_holds;
}
```

- I need a way to generate values of "any type"
- There are lots of things we already do for any type (eg print)

How to generate TYPE?

Use a template, naturally

```
template <typename T>
struct Arbitrary
{
  static T generate(size_t /*generation*/, unsigned long int /*seed*/)
  {
    return T();
  }
};
```

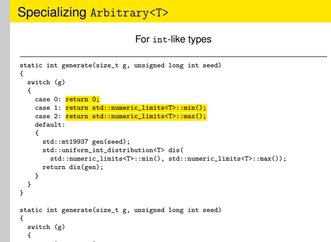
And specialize...

- The basic form
- generation is some idea of how complex the generated thing is
- and plumb through a random seed for reproducibility

Specializing Arbitrary<T>

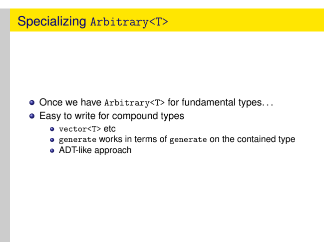
- Easy to write Arbitrary<T> for arithmetic types
- Front-load likely edge cases
  - 0
  - numeric_limits<T>::min()
  - numeric_limits<T>::max()
- Otherwise use uniform distribution over range

- Explain
- Generating arithmetic types is easy

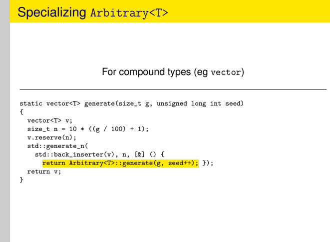


- (Code formatted for slide: in reality, I don't create a mersenne twister on the stack every call)
- For bools, it's trivial
- For chars, generate printable values

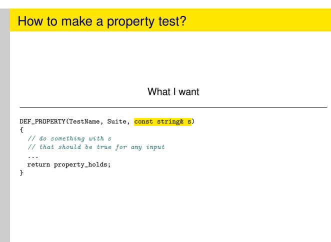
2015-04-08
Testing Battle.net
└Property-based testing
└Specializing Arbitrary<T>



- Compound types are made of other types of course
- Can be built up recursively



- Explain
- The idea of a "generation" deals with things like how long to make vectors, strings etc
- Generate for compound type works recursively by generating the contained types



- So far, I know how to generate the type
- Now I needed to figure out how to deal with the test
- Normally, tests don’t have arguments

Test macros expand into functions

Macro...

```
DEF_PROPERTY(TestName, Suite, const string& s)
{
  ...
}
```

Expands to...

```
struct NonceStruct
{
  ...
  bool operator()(const string& s);
};
bool NonceStruct::operator()(const string& s)
{
  ...
}
```

- the macro instantiates a function object
- I can discover the type of the operator() argument

Discover the type of the function argument

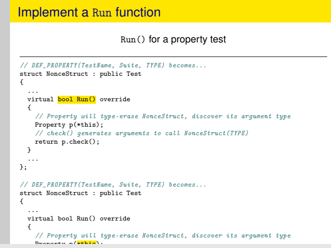
Simple `function_traits` template

```
template <typename T>
struct function_traits
  : public function_traits<decltype(&T::operator())>
{};

template <typename R, typename A>
struct function_traits<R(A)>
{
  using argType = A;
};

template <typename C, typename R, typename A>
struct function_traits<R(C::*)(A)>
  : public function_traits<R(A)>
{};

...

template <typename T>
struct function_traits
  : public function_traits<decltype(&T::operator())>
{};
```

- googling function traits turns up something very like this
- explain (slowly)
- omitted further specializations dealing with various const & ref qualifiers
- now I know
  - The argument type to generate
  - How to generate it
- All I need to do is figure out how to write Run() for a property test
- I need to take the operator() function, whose type varies for each test
- And make it callable in a uniform way
- Single-function interface on a varying-type object
- tailor-made for type erasure

2015-04-08

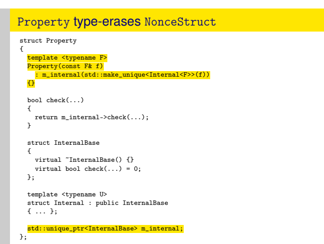
Testing Battle.net
└─Property-based testing
└─Implement a `Run` function



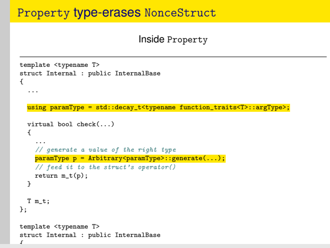
- Run() function is inherited from Test: this is quite standard
- "this" is the struct whose operator() varies
  - gets type-erased by Property
- Property exposes check() which calls the type-erased operator()

2015-04-08

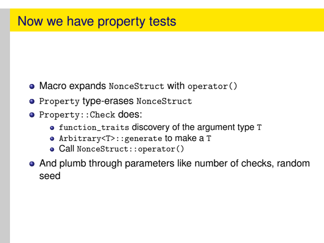
Testing Battle.net
└─Property-based testing
└─`Property` type-erases `NonceStruct`



- formatted for slide
- standard type-erasure pattern
- here’s the constructor that’s a template and captures the passed-in type
- here’s the stored type-erased thing
- here’s the exposed interface: the check function
- the omitted args are the generation and random seed params we saw earlier that will be used with the call to Arbitrary::generate
- let’s look inside Internal



- check generates a value using Arbitrary::generate
- passes it to the operator() of the NonceStruct



- recap
- now we can use this ability to generate to power algorithmic tests
- but before we get to that, shrink

Better checks for compound types

When a check fails, find a minimal failure case

```
template <typename T>
struct Arbitrary
{
  static std::vector<T> shrink(const T& /*t*/)
  {
    return std::vector<T>();
  }
};
```

`shrink` returns a `vector` of "reduced" `T`'s

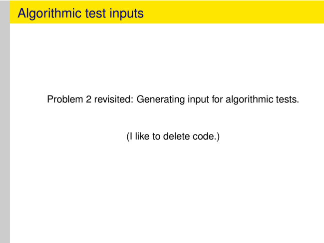
- borrowed from Quickcheck
- we can do more than just generate
- shrink returns a vector of T's

Better checks for compound types

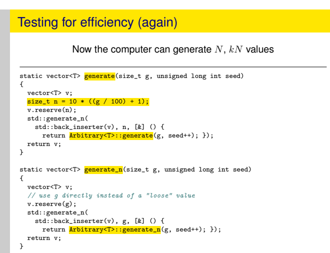
A simple binary search

```
static std::vector<std::basic_string<T>> shrink(
  const std::basic_string<T>& t)
{
  std::vector<std::basic_string<T>> v;
  if (t.size() < 2)
    return v;
  auto l = t.size() / 2;
  v.push_back(t.substr(0, l));
  v.push_back(t.substr(l));
  return v;
}

static std::vector<std::basic_string<T>> shrink(
  const std::basic_string<T>& t)
{
  std::vector<std::basic_string<T>> v;
  if (t.size() < 2)
    return v;
  auto l = t.size() / 2;
  v.push_back(t.substr(0, l));
  v.push_back(t.substr(l));
```

- base case: return empty vector
- recurse, making the returned vector elements smaller
- for the containers, just use a binary search strategy
- explain how the calling code will follow failing cases



- Now I can take my property test code and apply it to the algorithmic complexity tests



- For algorithmic tests, we need to lock down a specific size
- Otherwise generate_n works exactly the same as generate
- the calling code doesn't need to follow failures
- these tests are just for timing

2015-04-08

Testing Battle.net
└Testing scalability (II)
└Now I can write

Now I can write

A sample complexity test

```
DEF_COMPLEXITY_PROPERTY(TestName, Suite, ORDER_N, const string& s)
{
  // something that's supposed to be order N...
  ...
  std::max_element(s.begin(), s.end());
  ...
}
```

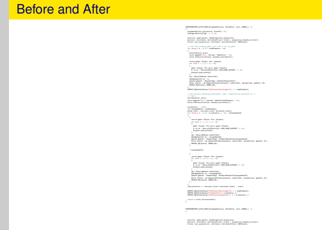
And specialize `Arbitrary` for my own types as necessary
Much less boilerplate to maintain

- can use O(1), O(log n), O(n), O(n log n), O(nˆ2)
- if the test comes in at or under the specified order, that's a pass
- specialize my own type generation:
  - random for average case data
  - bastard mode for worst case data
  - for ranges
  - unfortunately c++ has no newtype

2015-04-08

Testing Battle.net
└Testing scalability (II)
└Before and After

Before and After



- Get rid of
  - generation code
  - timing code
- refactor code made unnecessary by the new framework
- ~80 lines -> ~20 lines

2015-04-08

Testing Battle.net
└─Future thoughts
└─The reward for good work is more work

The reward for good work is more work

Status quo/future possibilities.

(People are never satisfied.)

- Arbitrary opens up new possibilities
- Next slide is a roundup

2015-04-08

Testing Battle.net
└─Future thoughts
└─Where I am now

Where I am now

- Dependency injection (little work in constructors)
- Separate logic from interaction (even in monolithic classes)
- Regular tests for "normal, identified" cases
- Timed tests when I'm optimizing
- Property-based tests for invariants
- Algorithmic complexity tests for scalability confidence

- regular tests are still good
- property tests make you think harder
- in practice, the efficiency bar for Battle.net efficiency is < O(n)

The future?

- `Arbitrary` opens the door for fuzz testing?
- Alternative walk strategies through the input space
  - Hilbert
  - Morton
  - etc
- Using `Arbitrary` to find poorly-performing data (P99)
- I'm still lazy; the computer isn't doing enough for me yet

- Fuzz testing is possible, but I didn't need it so much at the protocol level
  – Protobufs have sum and product types now
  – Illegal states can be unrepresentable
- Exercise poor performance in a couple of ways
  – Make tests do bad things
  – Make Arbitrary generation give bad data

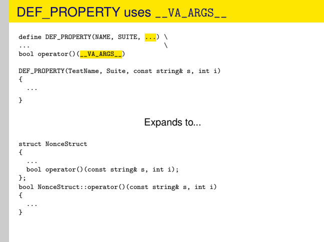
Epilogue: more property-based testing

Cool, can you do multiple arguments?

```
DEF_PROPERTY(TestName, Suite, const string& s)
{
  // do something with s
  // that should be true for any input
  ...
  return property_holds;
}

DEF_PROPERTY(TestName, Suite, const string& s, int i)
{
  // do something with s, i
  // that should be true for any input
  ...
  return property_holds;
}
```

- when people see me riding a unicycle, they ask if I can juggle at the same time

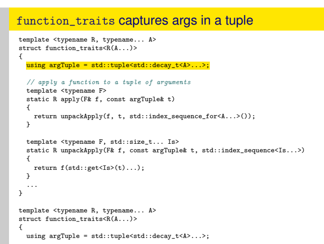
└─DEF_PROPERTY uses `__VA_ARGS__`



- C++ has variadic macros in the standard now

2015-04-08
Testing Battle.net

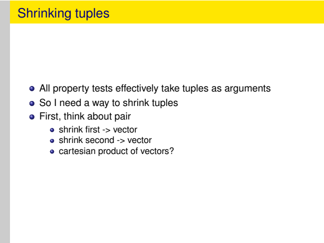
└─`function_traits` captures args in a tuple



- I was surprised how easy it was to apply a type transformation
- This is basically apply from the library fundamentals TS
  - But without the forwarding references

Testing Battle.net

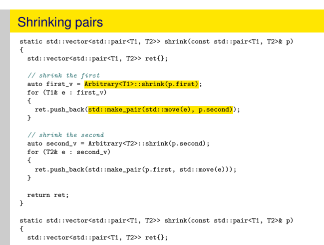
└─Shrinking tuples



- tuples were a relatively late addition
- at first I didn't implement shrink very well
- I went back to pair and I had a comment there
- cartesian product not necessary because of machinery
- N+M solution

Testing Battle.net

└─Shrinking pairs



- I thought about doing the cartesian product
- like applicative on lists in haskell

```
ghci> [(\(x,y) -> (1,y)), (\(x,y) -> (2,y))] <*> [(0,4),(0,5),(0,6)]
[(1,4),(1,5),(1,6),(2,4),(2,5),(2,6)]
```

- but the N+M solution works just fine
- when you're done following the first shrink path, use the second

From pairs to tuples

- So I go to cppreference.com
  - `make_tuple`
  - `tie`
  - `forward_as_tuple`
  - `std::get`
  - `tuple_cat`

- I see these things (go through them)
- tuple_cat? hmmm. . .

From pairs to tuples

- `first` is `std::get<0>()`
  - or `tuple_head()`?
- `second` is `tuple_tail()`
- `make_pair` is `tuple_cons`
  - put a head together with a tail

(Pretend these functions exist so we can write `shrink` for tuples)

- I've done some functional programming

Shrinking tuples

```
static std::vector<std::tuple<Ts...>> shrink(const std::tuple<Ts...>& t)
{
  std::vector<std::tuple<Ts...>> ret{};

  // shrink the head
  using H = std::decay_t<decltype(std::get<0>(t))>;
  auto head_v = Arbitrary<H>::shrink(std::get<0>(t));
  for (H& e : head_v)
  {
    ret.push_back(tuple_cons(std::move(e), tuple_tail(t)));
  }

  ...

  return ret;
}

static std::vector<std::tuple<Ts...>> shrink(const std::tuple<Ts...>& t)
{
  std::vector<std::tuple<Ts...>> ret{};

  ...

  // shrink the tail recursively
  using T = std::decay_t<decltype(tuple_tail(t))>;
```

- pretend these exist
- as with pairs, so with tuples
- first shrink the head, cons them onto the tail
- then shrink the tail (it will work recursively)
- cons normal heads on to the shrunk tails

tuple_cons and tuple_tail

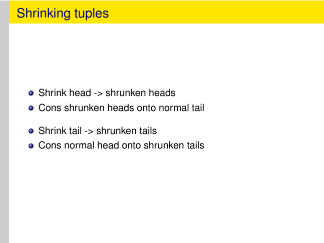
```
template <typename U, typename T>
auto tuple_cons(U&& u, T&& t)
{
  using Tuple = std::decay_t<T>;
  return tuple_cons(std::forward<U>(u), std::forward<T>(t),
                    std::make_index_sequence<std::tuple_size<Tuple>::value>());
}

template <typename U, typename T, std::size_t ...Is>
auto tuple_cons(U&& u, T&& t, std::index_sequence<Is...>)
{
  return std::make_tuple(std::forward<U>(u),
                         std::get<Is>(std::forward<T>(t))...);
}

template <typename T>
auto tuple_tail(T&& t)
{
  using Tuple = std::decay_t<T>;
  return tuple_tail(std::forward<T>(t),
                    std::make_index_sequence<
                      std::tuple_size<Tuple>::value - 1>());
}

template <typename T, std::size_t ...Is>
```

- tuple_cons is easy (explain)
- tuple_tail (explain)
- the power of tuple, variadic templates and index_sequence is great

2015-04-08

Testing Battle.net

└─Shrinking tuples



- easy
- and that's really it

2015-04-08

Testing Battle.net

└─Thanks for listening (again)

Notes

- Introductory (short)
- Brief overview of Battle.net server topology
- The problem: moving beyond "easy-mode" unit testing of base libraries to testing real components with real interactions, IO, configuration, etc
- Designing for testability
- Separating and injecting dependencies
- Test-friendly class hierarchy design
- Identifying invariants, structuring logic for tests
- Testing strategies (and the C++ that powers them)
- Regular edge cases
- Planning for and testing failure in a distributed system
- Gaining confidence in scalability without incurring the cost of running a full environment*