The Beauty and Power of “Primitive” C++

Bjarne Stroustrup

Morgan Stanley, Columbia University

www.Stroustrup.com

This year is different

No grand philosophy, no standards lawyering, just playing with some code

Overview

- The problem
  - Mapping typed objects to/from bytes for messages
- Overall design
  - Flats and access types
  - System structure
- Parts of an implementation
  - Interfaces
  - Messages
  - Vectors and strings
  - Error handling
- Observations
  - C++ is quite good at this
  - But not perfect, we'll continue to improve it

"Programming in the small"

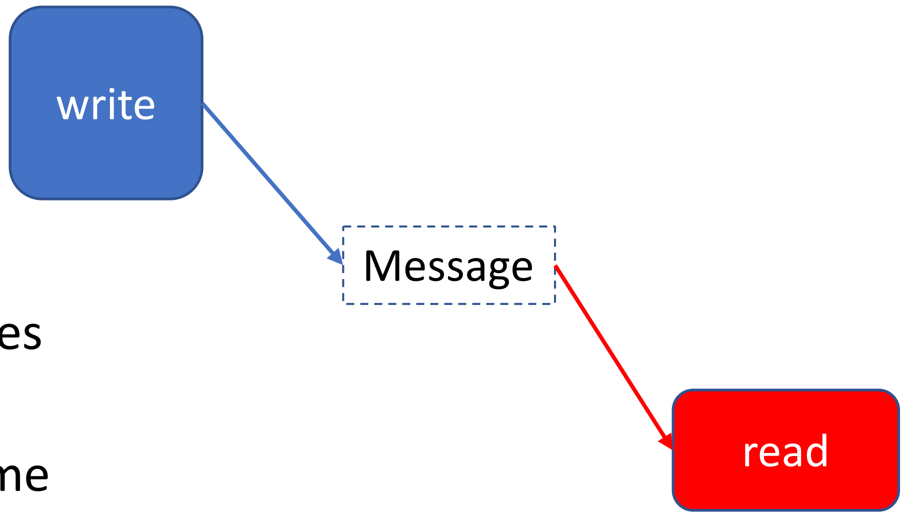
Alternative talk title:

"The fun and frustrations of writing low-level code"

My chosen problem

First decide what you want
aka
Requirements analysis

- Get relatively simple, structured information from A to B.
  - Everybody does that all of the time
- "Special constraints" (There are always many)
  - Maybe we need to store a message for 5 years
  - May require low latency
  - May require minimal space overhead
  - Read and write from different programming languages
    - C++, Java, Python, …
  - Maybe programming errors could have serious consequences
  - May require random access to fields
  - Maybe the size of some fields cannot be known until run time
  - Maybe read at B, modify, and pass along to C



My needs and constraints may not be yours

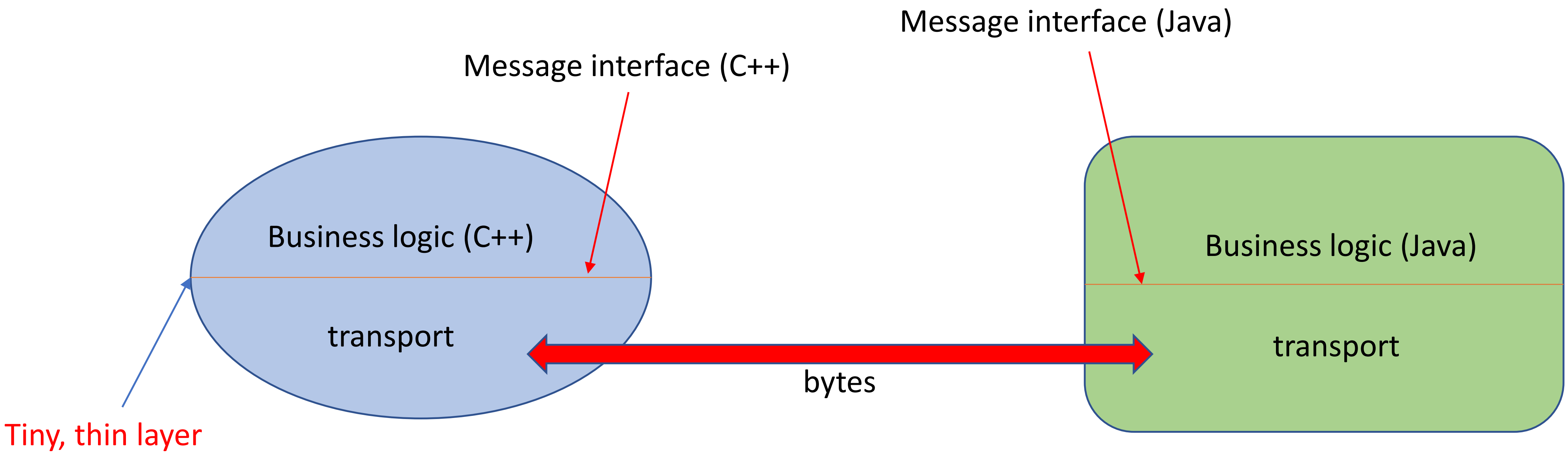
“Everybody does something like this”

- But
  - Different transport mechanisms/libraries have different interfaces
    - Google protocol buffers, Flatbuffers, SBE, JSON, XML, and many, many more
  - The transport interface bleeds into the business logic
    - Of many separately developed and maintained applications
    - We can’t easily change the transport mechanism
    - We can’t easily experiment with alternatives
- Separate reading and writing of a message from the transport mechanism
  - I describe a library offering an interface between typed objects and bytes in a message

Interface design

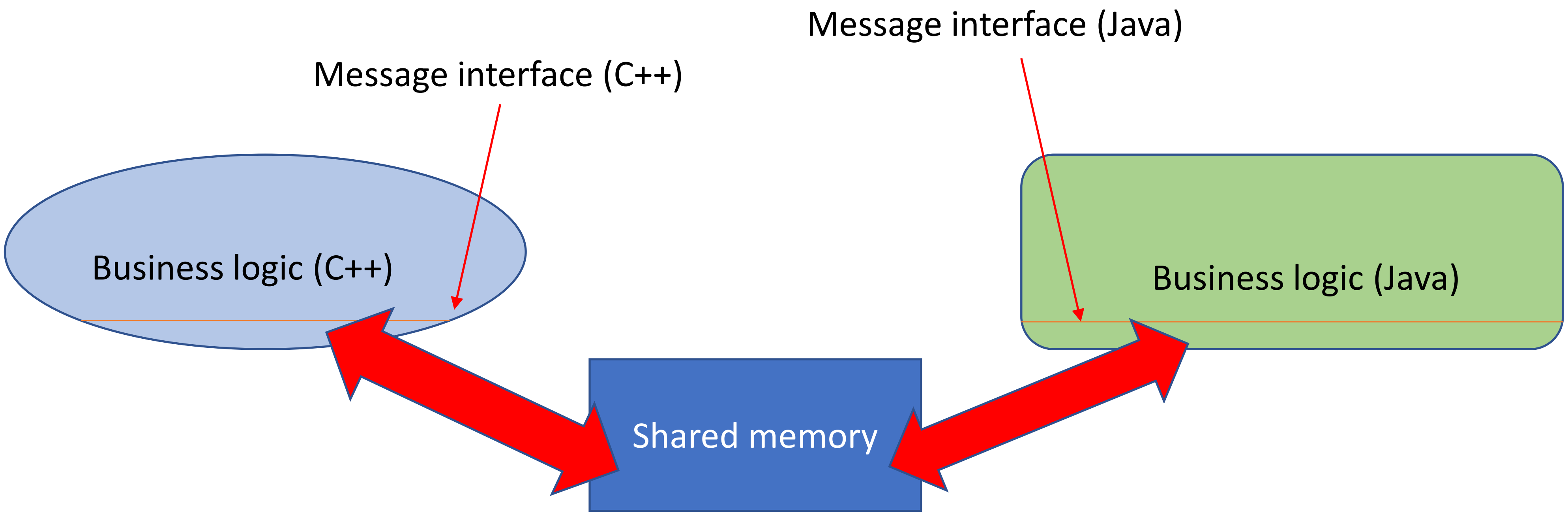
Application structure

- Make the "business logic" independent of the transport mechanism
  - here, I focus on "the message interface"



Application structure

- Make the “business logic” independent of the transport mechanism
  - Don’t forget shared memory
  - Here, the message interface cost isn’t drowned out by the transport cost



Communication model

Simplify!

- Exchange information at the start of a session
  - Individual messages are ***not*** self describing
  - Version and (possibly) layout information exchanged once per session
  - Where necessary, place a version identifier in each message
- Fill buffer, then send
  - Wait until buffer is full, then read
  - Any transport mechanism/library can be used for sending/receiving
    - Incl. a sequence of N bytes
    - Exact size know by sender, maximum size known by receiver
- Zero-copy: Read/write directly into byte buffer
  - minimize copying
  - No marshalling

Don't try to solve
Every problem

See what your likely users need

Data

Simplify!

- Relatively simple structures (“Flat” structures) called “flats”
  - No pointers
    - use offsets
  - No mutual references
  - Field types
    - Char, various integers, various reals/floats
    - Optional, variant, array, vector, string (nesting)
    - Simple user-defined types
    - Flat structures (nesting)
- For anything more complicated
  - Use a higher level of composition or control
    - E.g., sending a large message as a stream of small ones
    - E.g., sending a general graph

Don’t try to solve
Every problem

See what your likely users need

Message

Flats
A flat is a simple structure

- Optional header
  - Version
  - Allocator (for managing space in the tail)
- Optional tail
  - For data for which we don’t know the size until run time
  - Needed for vectors and variants only

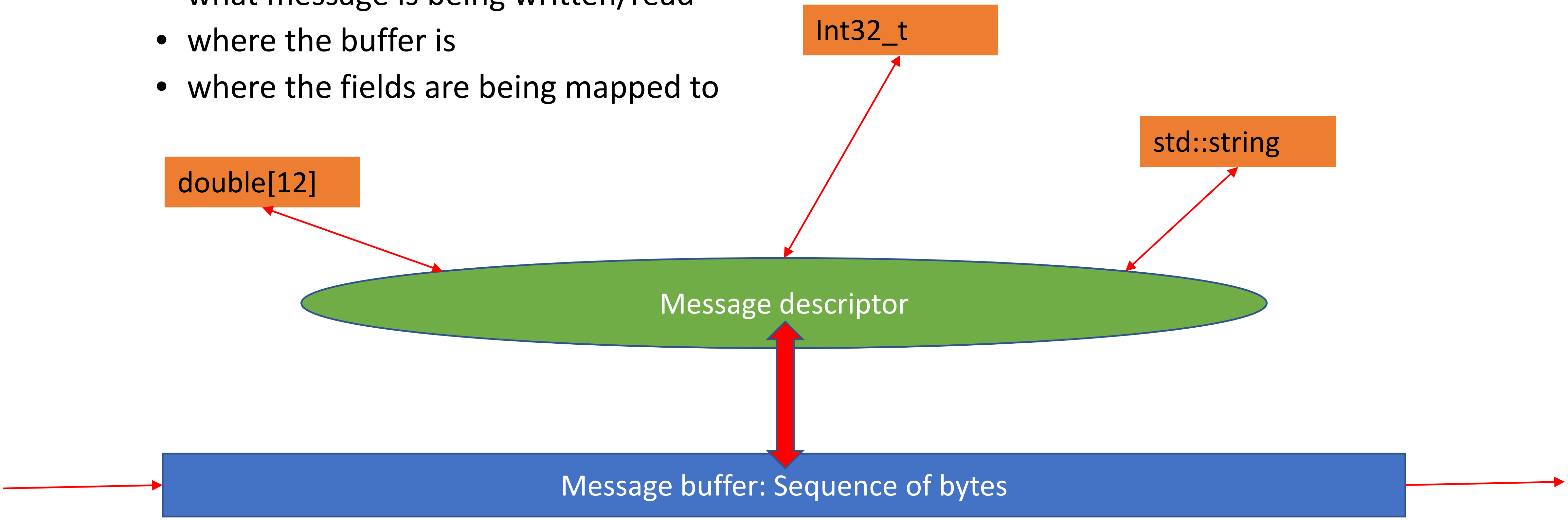
Flats in the message buffer

A flat is a simple structure

- char
- int
- Array<T,N>
- Vector<T>
- Optional<T>
- Optional<Array<T,N>>
- Variant<A,B,C>

Message descriptors map typed objects to/from bytes

- A message descriptor knows
  - what message is being written/read
  - where the buffer is
  - where the fields are being mapped to



Type mapping

- Keeping track of type mappings is not fit for humans
  - High -> low -> high
  - Fiddly, error-prone
  - So we generate such interface code

initializer type

```
// initializer for an Vector<String> called vs:
void vs(std::initializer_list<std::string> arg)
     { new(&mbuf->vs) Vector<String>(allo,arg); }
```

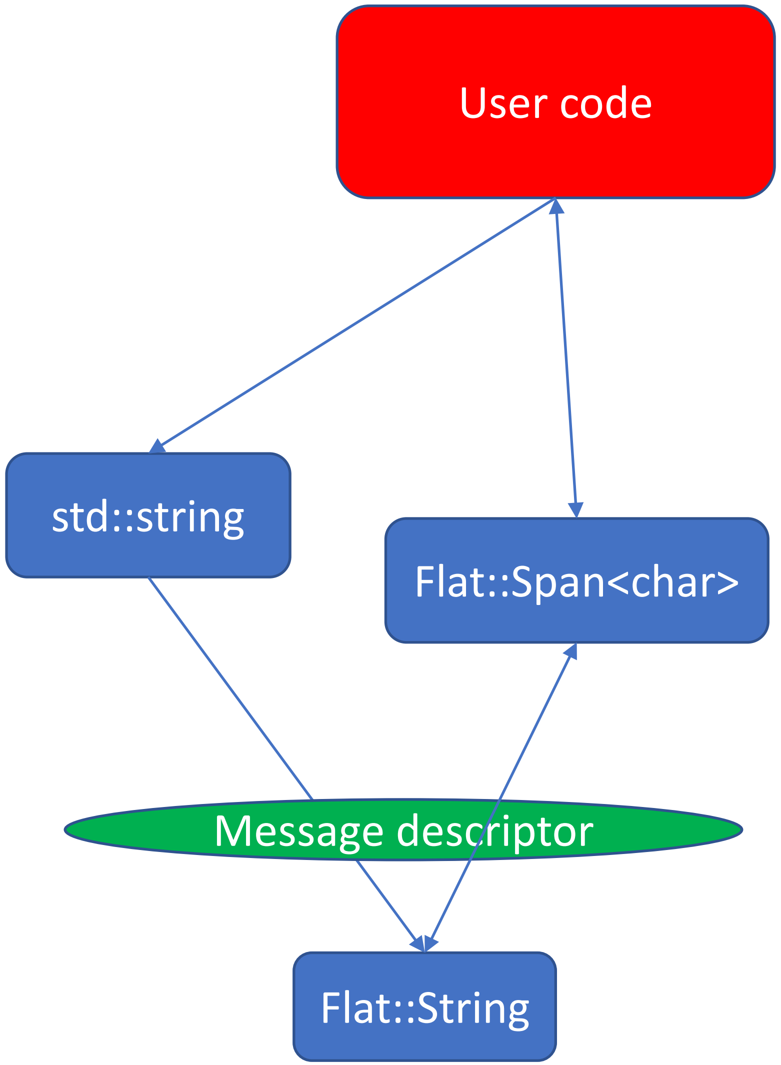
placement new

flat type

```
// accessor for an Vector<String> called vs:
Span<String> vs() { return mbuf->vs; }
```

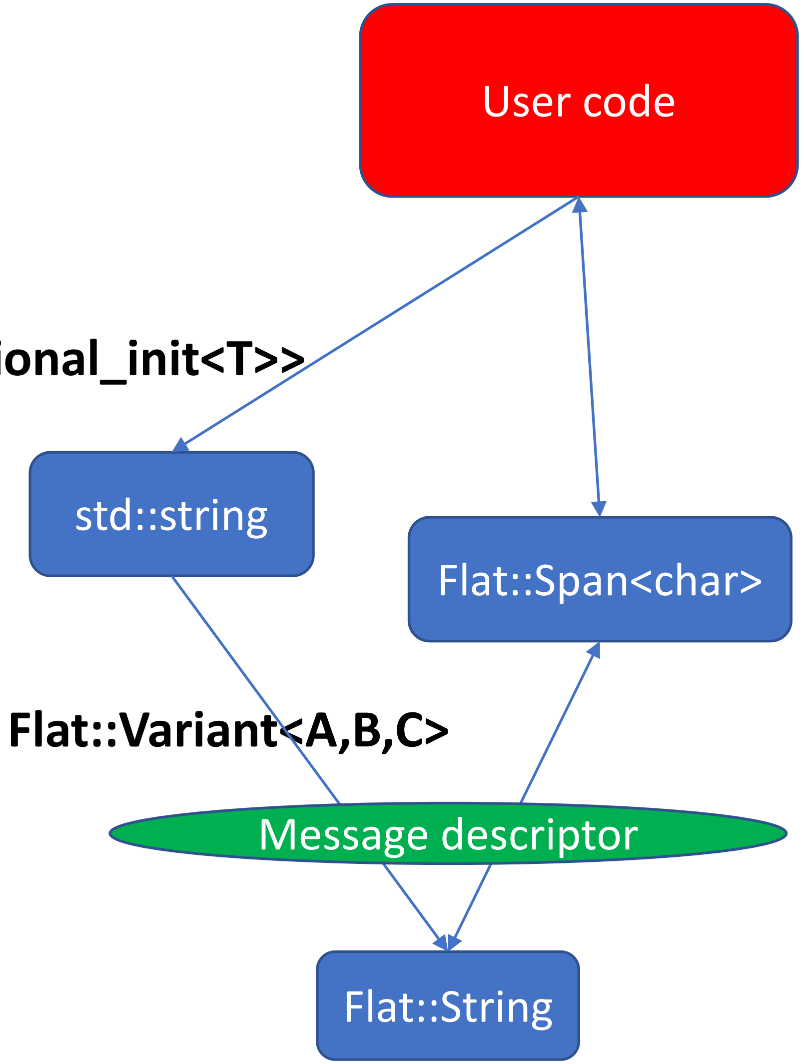
access type

a Span is a range of elements

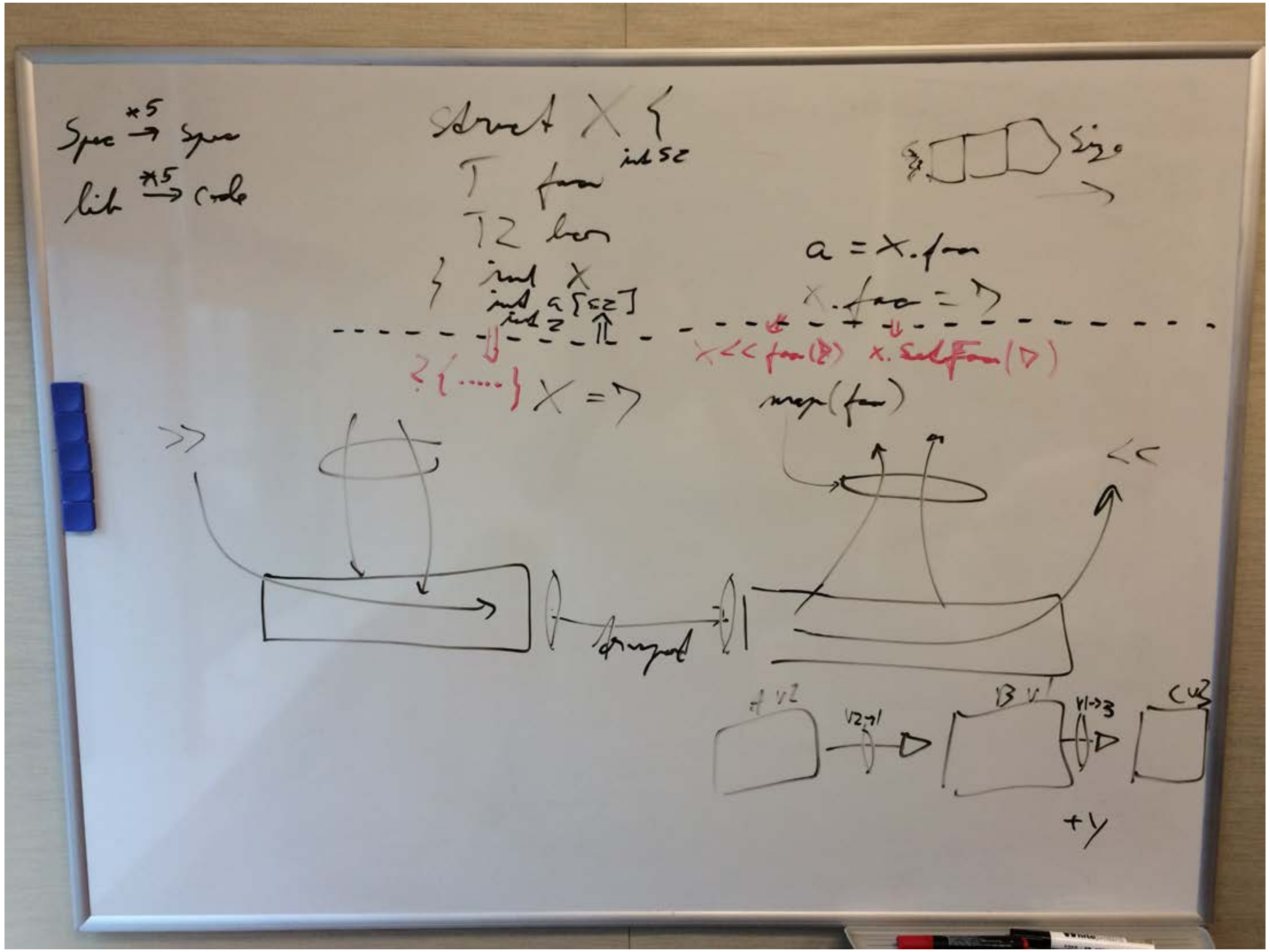


Type mapping

- Initializer types: ordinary types to initialize objects in the buffer
  - **char**, **int32_t**, **uint32_t**, …
  - Simple user-defined types; e.g., **TimeStamp**
  - **char***, **std::string**, **std::initializer_list<T>**, **std::initializer_list<Optional_init<T>>**
- In-buffer types: for objects in the buffer
  - **char**, **int32_t**, **uint32_t**, …
  - Simple user-defined types; e.g., **TimeStamp**
  - Special communication types, e.g. **Flat::Uint24**
  - **Flat::Array<T,N>**, **Flat::Vector<T>**, **Flat::String**, **Flat::Optional<T>**, **Flat::Variant<A,B,C>**
- Accessor types: to access to objects in the buffer
  - Simple user-defined types; e.g., **TimeStamp**
  - **char**, **int32_t**, **uint32_t**, …
  - **Flat::Span<T>, Flat::Span_ref<T>**



The first draft design looked like this



My favorite tool for initial design:
A whiteboard

Interface Definition Language

First decide what you want

- To support several programming languages, we need an IDL
- That's an opportunity
  - The IDL can be simpler than what's found in the various languages
  - The IDL can accommodate specialized application needs
- Start simple and add features only when needed

```
Pair : flat {              // a flat is a type/structure that we can map to the byte buffer
    name : string          // vector of characters; length determined at initialization
    value :  int32         // plain old int: int32_t
}
```

Generality

Variable-sized sequence of characters

- Types and nesting

Built-in type | flat | optional

```
Pair : flat { name : string;  value : int32 }

Transaction : flat { id : int32; time : TimeStamp; values : Pair[12]; extra : optional<float64> }
```

User-defined type | Fixed-sized sequence (array)

- ***Type design***: generality and recursion is good
- ***Applications***: nobody needs (all of) that generality
  - Can lead to bloat
- ***Code design***: generality and elegance gives hope of correctness
- ***Current state***: quite general, but not completely general nesting
  - Don’t add complexity without a use case

Given an IDL

Now see what it takes to get what we want
aka
Design and implementation

- We need a parser/generator
  - Generate “ideal” interfaces and implementations
  - Have a tools infrastructure based on generation

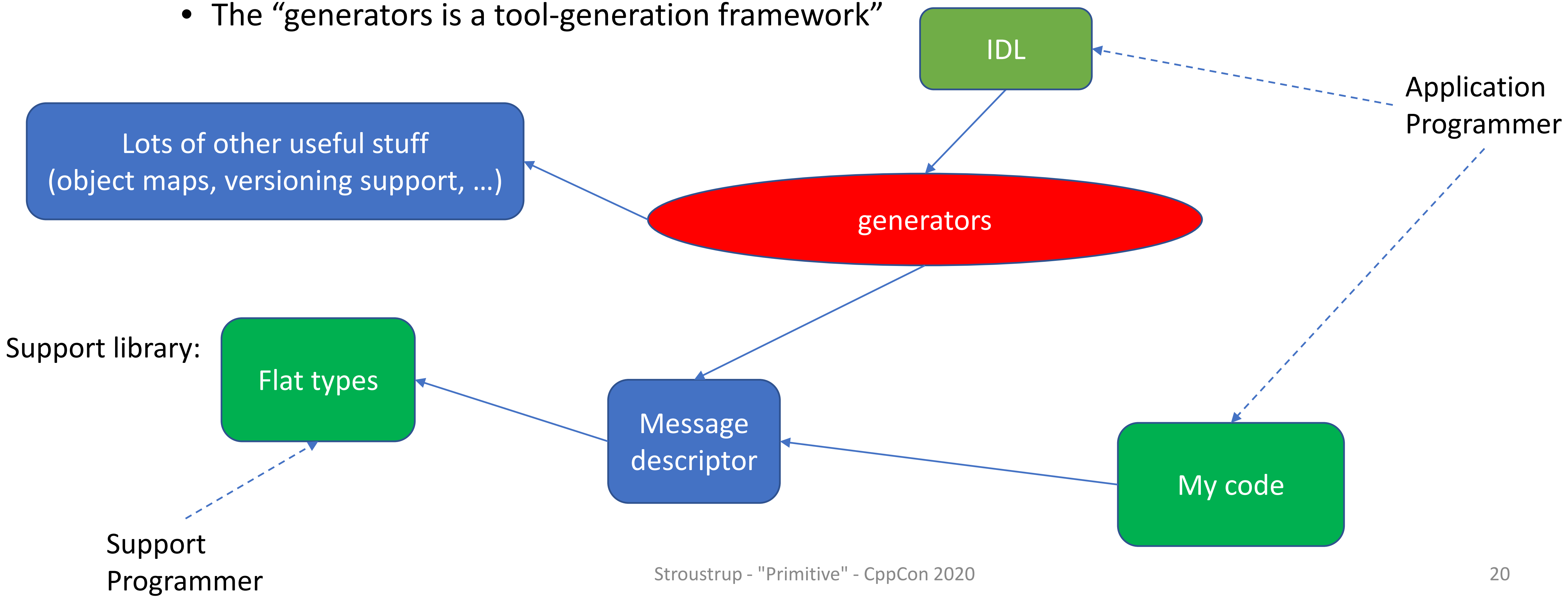
Given that, we can do anything!

“fiddly, unsafe” code can be left to a generator

- Here, I’ll concentrate on C++
  - We can also generate Java and Python
  - Interfaces must be “culturally appropriate” for each language

Flats library and tools framework

- IDL, generation, support library, my code, (transport), your code
  - The “generators is a tool-generation framework”



Implementation – "small is good"

To be fast and correct,
code has to be small and simple

- IDL Parser/generator
  - IDL -> simple, general internal representation -> anything I want
    - Message descriptors with access functions
    - Object maps
    - Version control
    - ~1,000 lines of simple C++
- Flats support library
  - Vector, String, Optional, Variant, Array, Span, Span_ref, Uint32
    - Minimal in-buffer representations
      - No pointers
    - ~500 lines of simple C++

"Programming in the small"

Why write 2,000 lines of C++ when a 500,000-line application framework will do?

- Benefits
  - Transparency
  - Few dependencies (just bits of the C++ standard library)
  - Simplified tool chain
  - Direct solution to “our needs”
  - Tunability
  - Smaller executables
- Counterarguments
  - The huge framework is
    - Familiar to many
    - Maintained by “others”
    - Third-party tools
    - No low-level fiddling (maybe)
  - You have to maintain those 2,000 lines “forever”
  - Anything new will be unfamiliar

Flats library and tools framework

- Parser
  - Simple recursive descent
- Generator
  - Object maps, Message descriptors, Message setup
  - Accessors
  - Initialization
- Run-time support library
  - Flat types: Vector, String, Array, Variant, Optional, Span, Span_ref
  - Error handling

Red: mentioned here

Interfaces

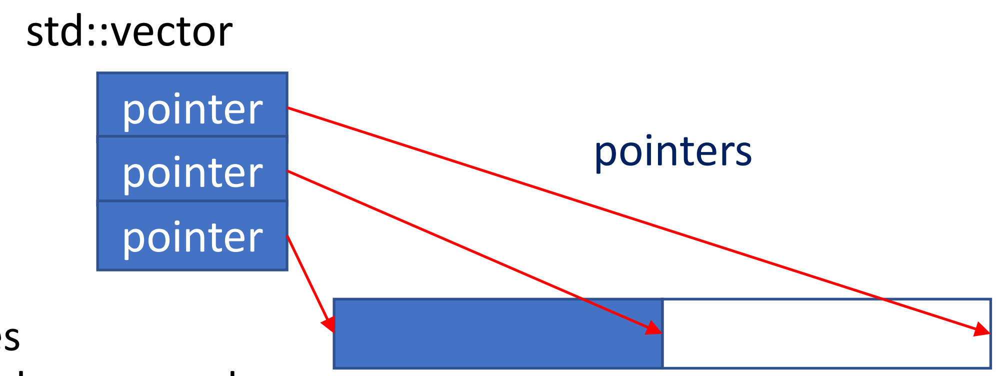
- Good interfaces are always essential
  - And not always easy to design
  - Tastes differ

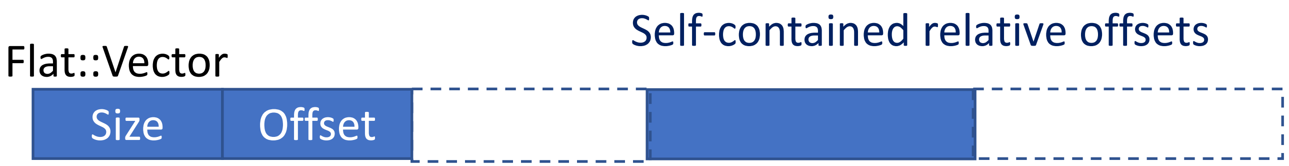
```
event.vvi({ {1,2 }, {3,4,6},{}, {2} });         // vvi is a vector<vector<int32>> member of event
std::cout << "vvi: " << event.vvi() << '\n';   // print vvi – does nested range-for loops

event.voi({ 1, 99, Empty{}, 0, 77+x });    // voi is a vector<optional<int32>>
auto rr = event.voi();
if (rr[1].present())
      std::cout << "voi[1]: " <<  rr[1] << '\n';
```

One interface – many implementations

- We can't just
  - Read/write structs
    - Interfaces to different transports differ
    - Some want zero-copy
    - Version skew
    - Some want guarantees that some fields are always written
    - Some necessary types are not native in all needed languages
    - Some want static guarantees that only a subset of fields can be accessed
  - Use standard-library types
    - They are not represented as simple byte sequences
  - Use our favorite vendors interface
    - Lock-in
    - Inability to experiment/compare at scale
- From a good interface
  - We can map to any implementation (transport mechanism)

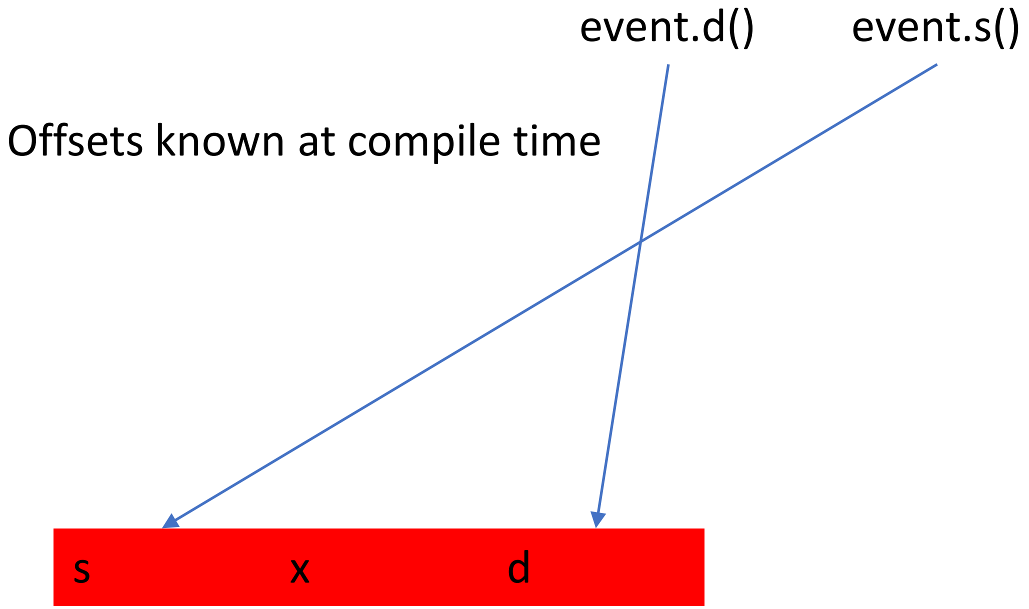




Direct access

- When you know exactly how a received message is defined

```
F : flat {
    s : string
    x : int32
    d : float32
}
```

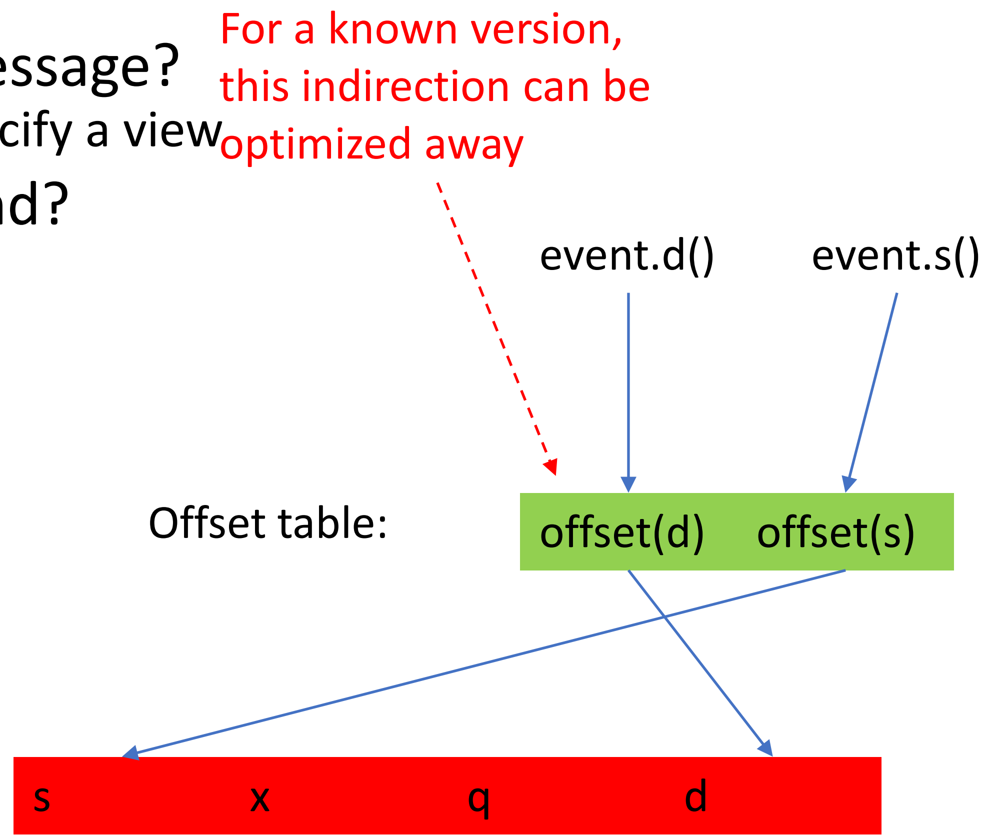


Views – ignore fields that you are not interested in

- What if you are just interested in part of a message?
  - Specify which members you are interested in: specify a view
- What if later version have moved fields around?
  - Access through an table of offsets
- A view can be used to read many versions

```
F : flat {        // version 2
      s : string
      x : int32
      q : vector<string>
      d : float32
}

V : view of F {
      d : float32
      s : string
}
```



How to read and write?

I need data
to guide design

- Loop through writing, copying, and then copying elsewhere
  - 1,000,000 messages, 11 fields, 68 bytes
  - Simple, brittle measurements to get a first impression
    - Average of best N-2 of N runs
    - Cache effects, Inlining effects
    - Looking for ratios, not absolute figures
    - There is no real substitute for use in a demanding application
  - Repeated many times in many variations
    - Multiple compilers
- Bytes, Structs, Accessors, out-of-order access: about 4 ns
- Getters and setters about 9 ns

Argument and result copying

Pair message – definition and layout

- IDL *(user written)*

```
Pair : flat {
    name : string      // vector of characters; length determined at initialization
    value :  int32     // plain old int
}

Pair_mess : message of Pair    // make a message descriptor to hold the Pair
```

- Generated C++ struct *(memory layout)*

```
struct Pair {
    Pair() {}
    String name;
    std::int32_t value;
};
```

Pair message – Write and write

Message descriptor:
maps from types to bytes in buffer

```
void test(Pair_direct md)
{
    md.name("Badger");                                           // initialize: name now has 6 characters
    md.value(78);
    std::cout << md.name() << ' ' << md.value() << '\n';        // read

    md.name() = "Bear  ";                                        // write
    md.value() = 1234;
    std::cout << md.name() << ' ' << md.value() << '\n';

    md.name() = "Elephant";                                      // trying to overflow
    std::cout << md.name() << ' ' << md.value() << '\n';
}
```

initialization

access

What happens here?
Truncation + error

Pair message – generated message descriptor

Typed buffer for the Flat

Allocator for the message tail

```
struct Pair_direct {
  Pair* mbuf;
  Allocator* allo;
  Pair_direct(Pair* pp, Allocator* a) :mbuf{pp}, allo{a} {}

  Span<char> name() { return mbuf->name; }

  void name(const char* arg) {  new(&mbuf->name) String(allo,arg); }
  void name(const std::string&  arg) {  new(&mbuf->name) String(allo,arg); }
  void name(Extent arg) { new(&mbuf->name) String(allo,arg); }

  std::int32_t& value() { return mbuf->value; }
  void value(std::int32_t arg) {  new(&mbuf->value) std::int32_t(arg); }
};
```

Place in buffer

Read/write access

initializers

Field names

Why that **()**?

- It’s a call to the message descriptor’s access function
  - Necessary for mapping actions
  - It provides a common interface to a variety of implementations
  - For example, **std::cout << md.name() << ' ' << md.value() << '\n’;**
- That **()** is ubiquitous and sometimes confusing
  - Easy to forget
- I really miss **operator.()**
  - We could have had **std::cout << md.name << ' ' << md.value << '\n’;**

Variable size?

Elements allocated
at construction time

- Array
  - Number of elements known at compile time
- Vector
  - Number of elements known at construction/initialization time

Fixed-sized fields

- Change size after construction?
  - Not currently supported

Don’t generalize
Without use cases

Strings – common and often important

- Fixed size: **Flat::Array<char,20>**
  - Perfect if you really know the size
  - Often wastes space (over-allocate for most cases)
  - Often adds complexity (store actual size or terminating zero)
  - **md.s3(); … md.s3() = "Ritchie";** **//** *14 characters unused*

- Size unknown until construction: **Flat::Vector<char>**
  - Place characters in tail
  - Store size and offset in “descriptor” (one word)
  - Descriptor must be initialized, or disaster if accessed
  - **md.s2("Dahl");** **//** *4 characters plus 4 bytes vector size*

  - **using String = Flat::Vector<char>;** **//** *so easy that it feels like cheating*

Vector definition


```
template<typename T>
struct Vector {     // Refers to the tail of a message starting at sizeof(this message);
        Size sz = 0;            // number of Ts
        Offset pos = 0;         // pos is relative to the position of this; Is 16 bits sufficient for every object?

        Vector() = default;
        Vector(Allocator* a, Extent sz);            // sz uninitialized elements
        template<class X> Vector(Allocator* a, std::initializer_list<X> lst);          // allocate list in the tail
        Vector(Allocator* a, const std::string& s);         // allocate characters s in the tail
        Vector(Allocator* a, const char* s) ;               // allocate characters s in the tail

        operator Span<T>() { return { begin(),end() }; }   // range checked by default
        // ....
};
```

Default initialize

access

Vector definition – vector needs an allocator

- An allocator allocates memory in the tail
  - so far no needs for delete or reallocation


```
template<typename T>
struct Vector {
    // …

    Offset alloc(Allocator* a)    // allocate sz elements in the tail
    {
        pos = a->allocate(sz * sizeof(T));                       // position in Flat
        pos -= reinterpret_cast<Byte*>(this) - a->flat();       // position relative to this
        return pos;
    }

    Tail_ref place(const char* str);        // place C-style string in tail
                                            // returns {Size,Offset} pair

    // …
};
```

Messy mapping code
Not in application code

Vector definition – place string in tail

- Placing a string in the tail

{Offset,Size}

Messy mapping code
Not in application code

```
Tail_ref Allocator::place(Allocator* a, const char* str)
        // allocate the C-style string s in the tail
{
    Offset pos = a->next;
    pos -= reinterpret_cast<Byte*>(flat()) + pos;        // position relative to this
    Size sz = cstring_copy(p, str, a->max-a->next);      // range-checking copy
    next += sz;
    return {pos,sz};
}
```

Vector definition – characters are special

Could I ban those?
My guess: no

- **std::string** is good when you know the number of characters
- **char*** (both literal and non-literal) C-strings are common
- Both have to be mapped to sequences of characters in the buffer (in the tail)

```
template<typename T>
struct Vector {
    Vector(Allocator* a, const std::string& s)          // here we know the number of characters
        :sz{ narrow(s.size()) }, pos{ alloc(a) } { std::copy(s.data(),s.data()+s.size(), begin()); }

    Vector(Allocator* a, const char* s)                 // here we must count characters
    {   auto r = a->place(s);                                   // uses cstring_copy()
        pos = r.pos - (reinterpret_cast<Byte*>(this)-a->flat());   // convert to relative offset
        sz = r.sz;
    }
    // ...
};
```

I lack a good way to return a pair of values
to use as a pair of member initializers

Vector definition – prevent overflow

```
constexpr Error_handling check_cstring = Error_handling_action;

inline Size cstring_copy(char* to, const char* from, int max)
        // copy at most max characters
        // optimizable?
{
        const char* first = to;
        while (*from) {
                expect<check_cstring>([max] { return 0<max; }, Error_code::cstring_overflow);
                --max;
                *to++ = *from++;
        }
        return to-first;
}
```

Conditionally enabled

Kind of error

A bit like assert()

Condition expected to be true

Error handling – Litter the code with checks

- Compile-time selection
- Detailed control

Fine grain control

```
template<Error_handling action = Error_handling_action, class C>
constexpr void expect(C cond, Error_code x)                    // C++17; a bit like assert()
{
    if constexpr (action == Error_handling::logging)
        if (!cond()) std::cerr << "Flats error: " << int(x) << ' ' << error_code_name[int(x)] << '\n';
    if constexpr (action == Error_handling::throwing)
        if (!cond()) throw x;
    if constexpr (action == Error_handling::terminating)
        if (!cond()) terminate();
    // or no action
}
```

Minimal

Morgan Stanley

Error handling – in-code control

- Litter the code with checks
  - Can be suppressed
    - Don’t suppress
  - Define “clusters” of related tests
    - Great for testing/experimentation

The default

```
constexpr Error_handling Error_handling_action= Error_handling::throwing;

constexpr Error_handling check_cstring = Error_handling::throwing;
constexpr Error_handling check_truncation = Error_handling::logging;
```

Cost of strings (usual caveats)

- Allocation in tail vs. fixed sized arrays
  - The user has a choice: Both alternatives carry costs
  - Initialization vs. assignment
  - Std::string vs. C-style strings

- Alternatives (averages over multiple runs, 12 strings, 189 chars, rounded)
  - **md.s4("abcdefgh");** // 130 ns
  - **md.s4("abcdefgh");** // *no overflow check* // 120 ns
  - **md.s4("abcdefgh"s);** // 510 ns
  - **md.s4(str4);** // 110 ns
  - **md.s4(Extent{8}); md.s4() = "abcdefgh";** // 255 ns
  - **md.s4(Extent{8});** // *just to compare* // 55 ns
  - **md.s4(Extent{8});** … // *no narrowing check* // 150 ns
  - **Array<char> assign** // 135 ns
  - **Array<char> init** // *no truncation check* // 90 ns
  - **Array<char> assign** // *no truncation check* // 100 ns

One check per character

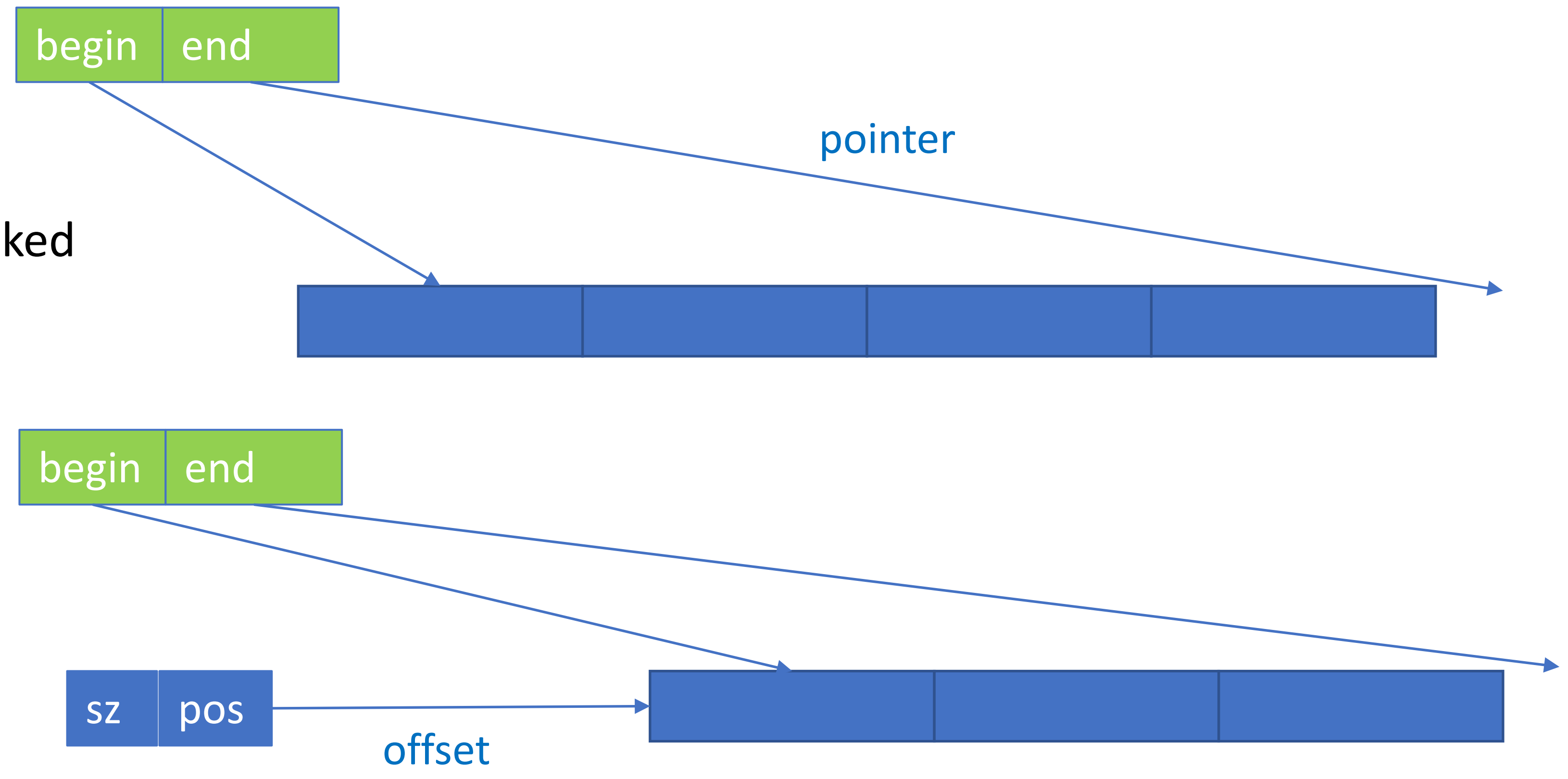
Unsafe

Build string
then copy the characters

Known size
Just copy the characters

Access to ranges

- **Array** and **Vector** accessors return **Spans**
  - Efficient
  - Uniform access
  - Subscripting
    - By default range checked
  - Range-for



```
template<typename T>
struct Span {          // a pure accessor read/write access to an array of T
	T* first;
	T* last;

	T* begin() { return first; }
	T* end() { return last; }
	const T* begin() const { return first; }
	const T* end() const { return last; }
	int size() const { return last-first; }

	T& operator[](int i) { expect([&] {return i < size(); }, Error_code::bad_span_index); return first[i]; }
	const T& operator[](int i) const { expect([&] {return i < size(); }, Error_code::bad_span_index); return first[i]; }

	void operator=(const char* p)   // convenience and optimization
	{
		for (T& t : *this) t = *p++;
		expect<check_truncation>([p] { return *p == 0; }, Error_code::truncation);
	}
	// …
};
```

Almost fits on a slide

Span_ref – a Span that constructs accessors

- Accessing a Flat object in the buffer requires a message descriptor:
  - **Vector<int32_t>** ---> **Span<int32_t>** **//** *simple field type*
  - **Pair** ---> **Pair_direct**
- So what about a **Vector<Pair>**?
  - Must return a **Pair_direct** for each element
  - **Vector<Pair>** ---> **Span_ref<Pair>** **//** *a Flat as field type*

```
template<class T, class TD>
struct Span_ref {        // Span over an array of flats T
    // when returning an element of type T, it constructs an accessor TD for the element value
    // …
    TD operator[](int i)
    {
            expect([&] {return i < size(); }, Error_code::bad_span_index);
            return TD{ first + i, allo };
    }
}
```

```
template<class T, class TD>
struct Span_ref {     // an ordinary span except for constructing accessors upon access
    T* first;
    T* last;
    Allocator* allo;
    Span_ref(T* p, T* q, Allocator* a) : first{ p }, last{ q }, allo{ a } { }

    struct Ptr_ref {
        Ptr_ref(T* pp, Allocator* a) : p{ pp }, allo{ a } {}
        T* p;
        Allocator* allo;
        Ptr_ref& operator++() { ++p; return *this; }
        // ...
        TD operator*() { return { p,allo }; }            // return accessor
    };

    Ptr_ref begin() { return { first,allo }; }
    Ptr_ref end() { return { last,allo }; }
    // …
};
```

Carry the allocator along
with the pointer to element

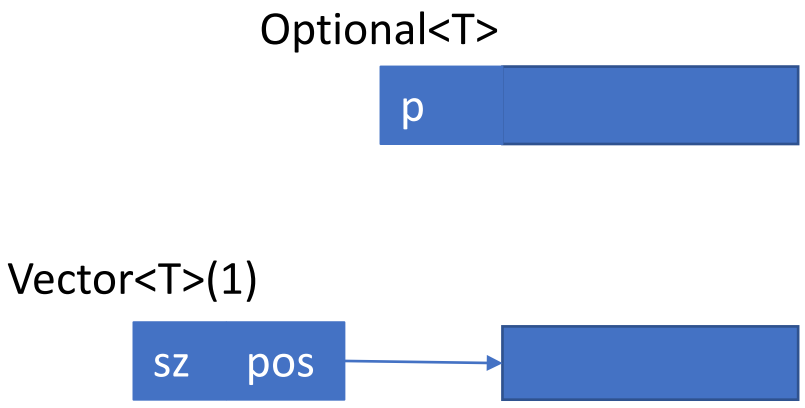
Initialization -Time vs. space vs. safety

- Vectors, Optionals, and Variants need to be initialized before use
  - Should they be default initialized?
    - Yes, users can't consistently initialize without support (experience)
    - Default: "empty"
  - Should it be possible to change their sizes after initialization?
    - No, no use cases so far
  - Should repeated initialization be allowed?
    - Yes, needed to change from default, hard to prevent
- Should every field be default initialized?
  - No, to which value? costly?
- Should the programmer be able to specify defaults for fields
  - No, the reader or the writer? Use cases?

Learn from experience
Listen to users

Allow change of size after construction?

- Cases
  - Size of Flat::Vector
  - Type of object in Flat::Variant
- Currently not supported
  - In principle easy
  - No solid use case (so far)
  - Instead, just re-initialize, there can be no leak
- Object in Flat::Optional
  - An optional always takes up space for an object
  - If you don’t like that
    - use a **Flat::Vector** of 0 or 1 elements
    - Change a **Flat::Vector(0)** to **Flat::Vector(1)** by initialization
    - Trivial to do



Morgan Stanley

Not covered for lack of time

Even in <2,000 lines of code

- There is always a lot of “nagging details”
  - The parser and generators
    - object maps, Java, views
  - Initialization design issues
    - Guarantees, Defaults, std::initializer_list or tuples
  - Optional
  - Variant
  - Details of Arrays
  - Flats of flats
  - How to provide an allocator
  - Message implementation details
  - Concepts
- Code on Github “any day now”

Language observations

- Modern C++ is great for low-level code
  - And for high-level “ordinary code”
  - And for mapping between higher-level “ordinary code” to messy low-level code
- C++ is flexible and tunable
  - Great when you know what you are doing
- Simple “value types” are key
  - No wrapper classes
  - No use of general free store
  - No spurious indirections
  - Compact and fast (close to zero overhead)
- Generic programming is essential
  - Simplify generic programming
- Compile-time evaluation and inlining are very powerful tools
  - Minimize run-time indirections
- I miss operator dot

Debugging observations

The bug is always where you aren't looking

- Sources of annoying bugs and problems
  - Signed/unsigned mess: e.g, **sizeof** and offset calculations
  - Overflow protection: rare bugs or verbosity-and-run-time-cost
    - Systematic error detection is relatively cheap
    - Error reporting needs to be flexible
    - Some application may not terminate

I use exceptions for such

  - Order of argument evaluation: when it bites, it's really nasty
  - Using both pointers and references can be confusing
  - It can be hard to figure out the root cause of a compile-time error in generic code
    - I failed to use concepts

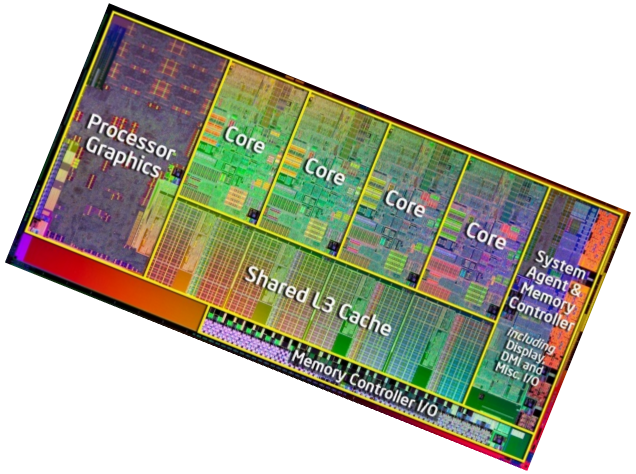
A BIG mistake

General observations

- A type-safe mapping to/from raw memory
- Great time-and-space performance
  - No spurious memory use
  - No spurious indirections
  - Minimal number of function calls (inlining)
- No compiler hacking
  - No macros
  - Only necessary casts
- Fiddling with byte and addresses
  - Is necessary
  - Is slow, unpleasant, and error-prone
  - Leave it to specialized support libraries, compilers, and generators
- Many alternatives can be supported through code generation
  - Generated code is more likely to be correct than hand-written code
- Good design rests on balancing concerns and making tradeoffs

Why isn't "Flats" a standards proposal?

- Not every useful idea belongs in the standard
  - Look at WG21 documents to see what that takes: www.isocpp.org/std
  - Also §3.4 of my HOPL-4 paper: Thriving in a crowded and changing world: C++ 2006-2020.
- It's a design exercise to stimulate thought
  - No analysis of general utility
  - Not enough user experience
    - What mistakes will people make?
  - The implementation is not general enough
    - There may be hidden design flaws
  - No thorough design document
    - Design decisions should be explicit
  - No tutorial
    - Are simple cases simple enough?
  - No thorough performance analysis
- low-level "primitive" code is an important part of the C++ world
  - Mapping from high-level type-safe code to the machine's level



Morgan Stanley

I'll take questions at the AMA Wednesday morning