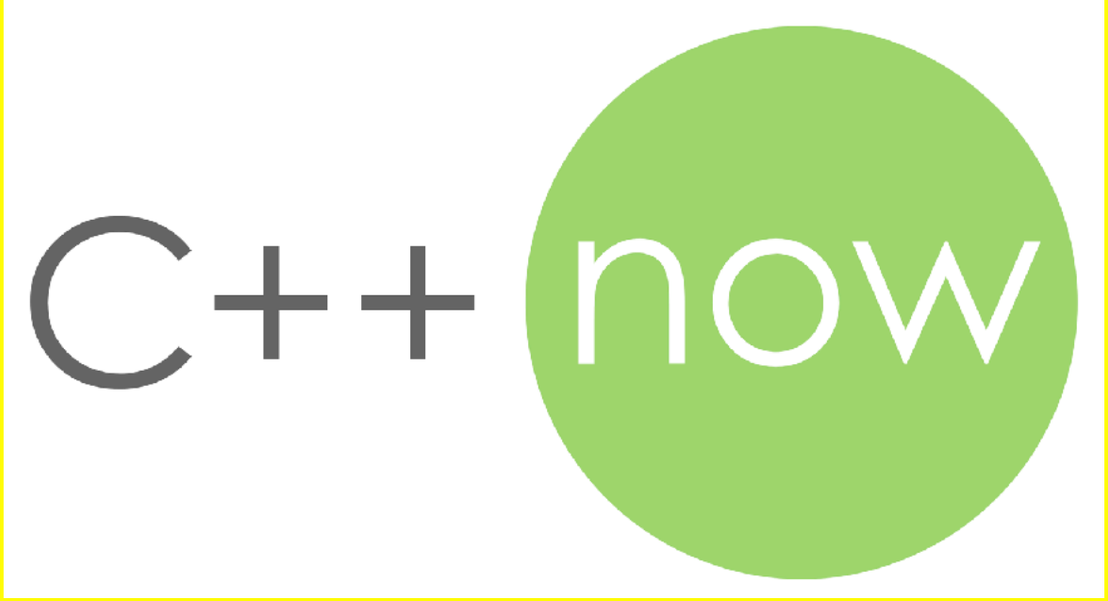
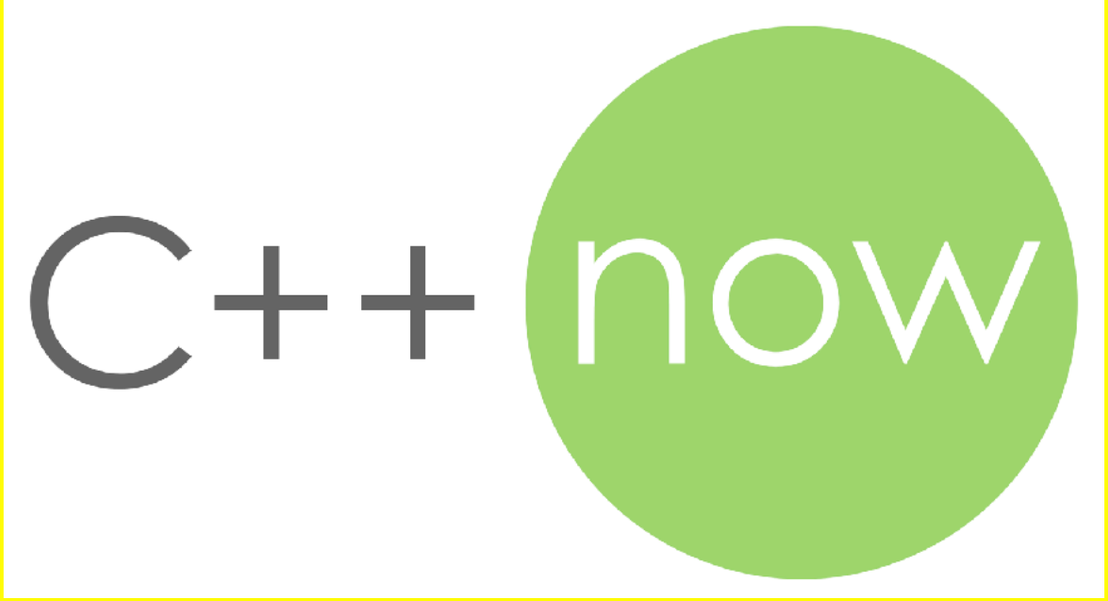
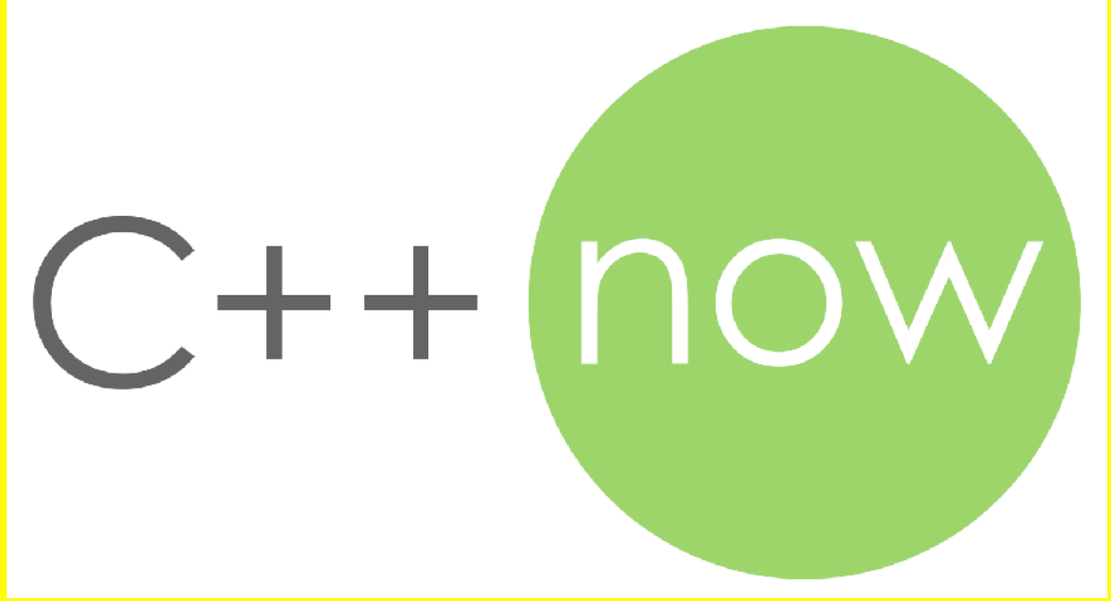


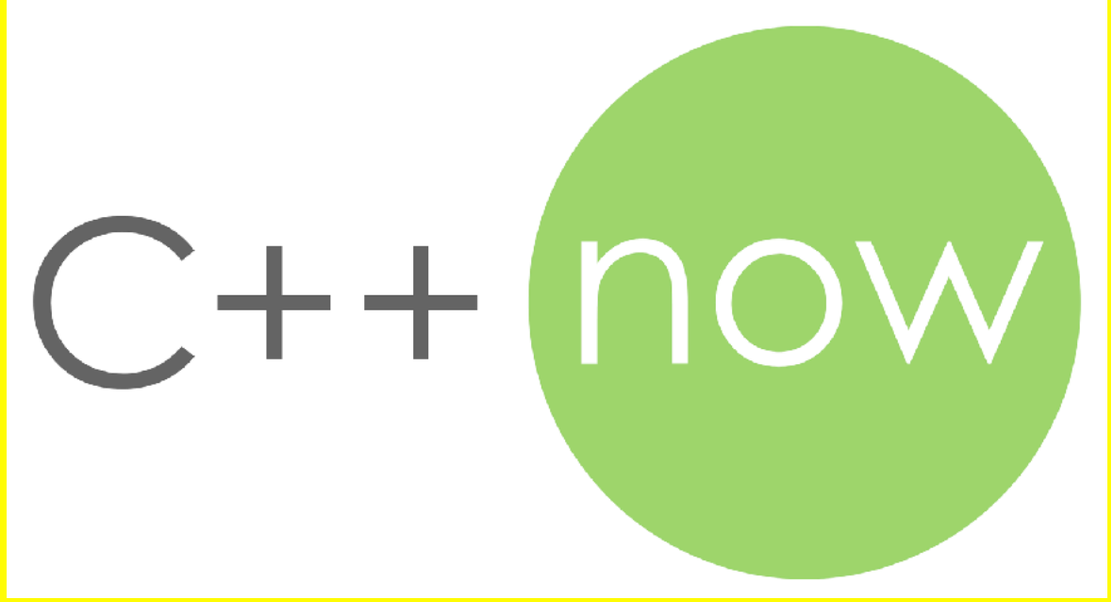
How Generic Programming



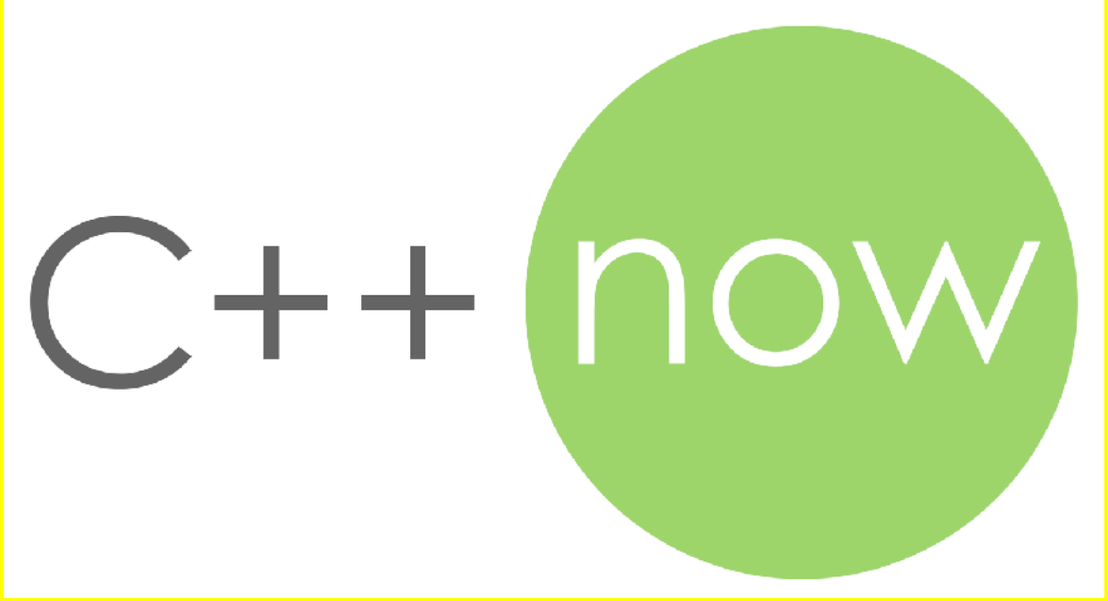
And C++ Portability



Give Great Performance

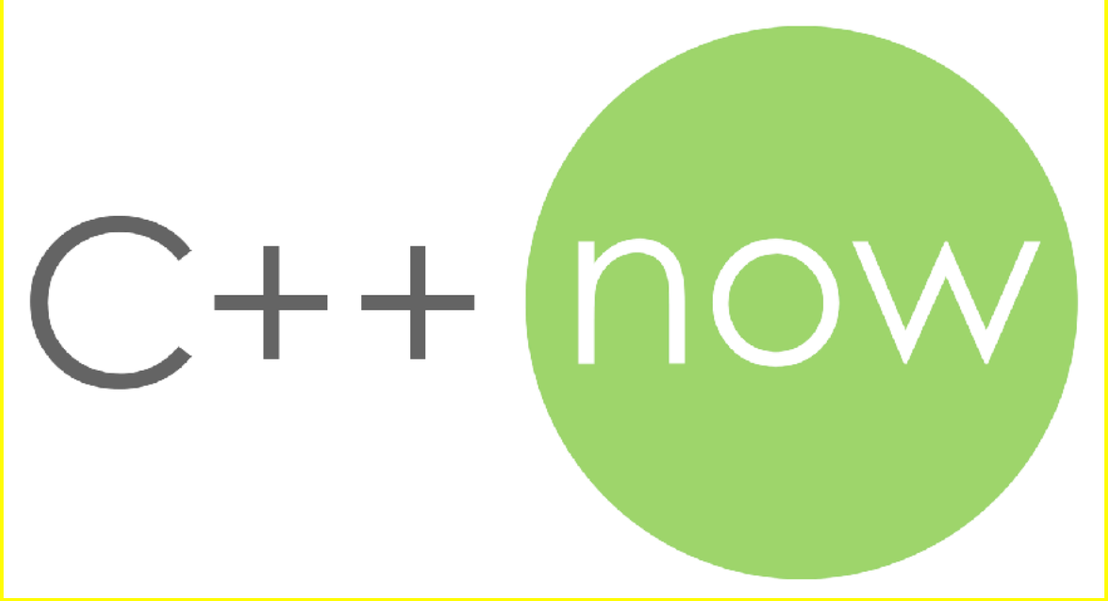


And Knowledge of What Performs Well



How Generic Programming and C++ Portability Give Great Performance and Knowledge of What Performs Well

Presented by Eduardo Madrid,
Continuing work with Scott Bruce, Rockset
May 2023



How Generic Programming and C++ Portability Give Great Performance and Knowledge of What Performs Well

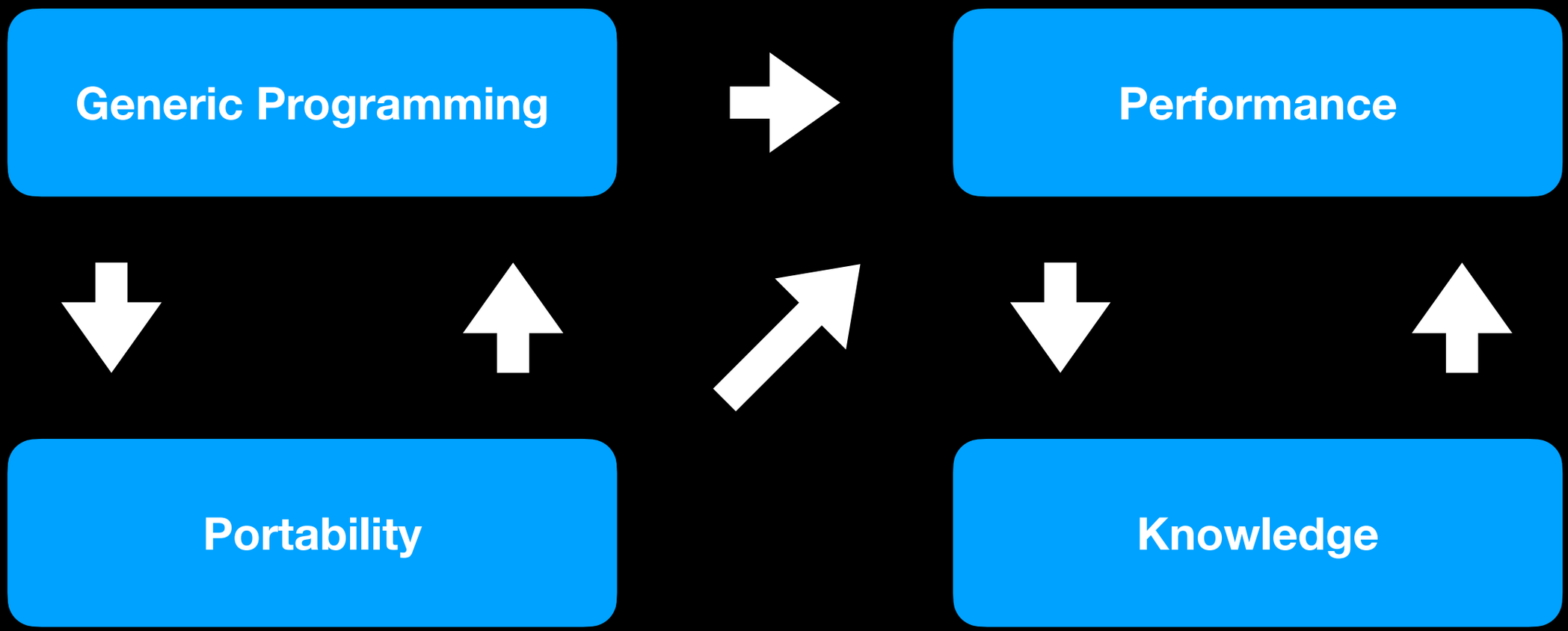
Presented by Eduardo Madrid,
Continuing work with Scott Bruce, Rockset
May 2023

Eduardo Madrid:

- Author of Open Source
- Formerly Team/Tech lead at
  - Snap, Inc. maker of Snapchat
  - Systematic Trading Hedge Funds
- Frequent presenter

This presentation:

- Is related to last year's "Rehashing Hash Tables and Associative Containers", they mutually complement
- Does not use benchmarking to discuss performance
- Uses very little code, makes an emphasis on the principles.
- Some examples are obvious, the intention is to pick subtleties that are important later.
- Does not have the magic that Scott Bruce might have brought to it.


Generic Programming
Performance
Portability
Knowledge

We know that knowledge and performance is a virtuous loop, but how does Generic Programming helps performance?

Part 1:

Generic Programming helps performance

Obvious:
Micro Optimizations

- Using templates gives the compiler much more information than using other techniques to achieve generality.

`std::sort` **versus** `qsort`

- `std::sort`:
  - Much better performing
  - Much more general: it supports any “random access” (“asterisk”) container of basically any sortable element type, `qsort` only supports arrays of elements with what we call trivial copyability
- Example of just better, because it relies on concepts such as:
  - Comparability
  - Copyability
  - Movability
  - Iterator
- that require or would require lots of explicit parameters without the concepts

`std::sort` **versus** `qsort`

- `qsort` requires an explicit comparison function because it has no concept of array-element comparability, `std::sort` allows us to have an explicit comparison or to rely on the container element type comparability property. Our choice.

- `qsort` needs to commit to a particular data structure, the array, because it knows nothing of containers, iterators, very sophisticated concepts of Stepanov's Generic Programming library that became the STL.

- `qsort` only supports trivially copyable types because the C language itself has no concept of non-trivial copyability.

- As stated, before, with `qsort`, the compiler is blind!

Quick Digression

- Complicated templates do not “confuse” the compiler and lead to inefficient object code.

- See this example, from my CPPCon 2019 about SWAR:

  - The technique of “divide and conquer” frequently leads to recursive solutions, one might think that the compiler would have a problem with that, but it does not.

  - An operation implemented using a platform intrinsic, when invoked within the context of a loop, got automatically converted by the compiler into a logical equivalent that was optimized quite dramatically by the compiler.

Quick Digression

- Links:
  - https://godbolt.org/z/oqpDB1
  - https://youtu.be/H-bm71KmYTE?t=2444
    - (At around 40 mins, 44 seconds in)

Less Obvious:
Avoids us to commit to suboptimal “one size fits all”

- We saw `qsort` commitment to array and trivial copiability, let’s learn from another failure, `std::function`: for the very many ways in which we want “trampolines”, it commits to only one set of choices.
    - Do you want move-only trampoline targets? Hence Facebook’s Folly `Function`.
    - The actual internal buffer size (and alignment!)
- When the one-size given does not fit, then you incur massive wrapping or conversion penalties. Or your code is polluted with unclear design choices trying to avoid needing one of the unsupported “sizes”

Less Obvious:
Digression: interminable “bike shedding”

- A consequence to the forced commitment to a few choices when many have valid uses is that they induce in your developer’s team an interminable argument about exactly which cases to support natively.

- This argument can escalate to the scale of the standard library itself: To palliate the fixed choices in `std::function` there are multitudes of proposals that address one or two aspects. I don’t think they are solutions because zoo’s type erasure shows one way to conceptualize type erasure so there is no commitment to any one choice, but all flavors are user-specified/selected.

Most important: Systematic Exploration of Performance Sensitive Choices

- The performance of critical libraries tends to be complex phenomena, what I mean is that the combination of what seems to be the best performing choices when measured individually, may even be far from the best performance when integrated into a complete solution.

- Why? Because of interactions between choices.

Hypothetical Example

- Imagine some performance-critical function is implemented in two ways:
  - Using a lookup table to not have to calculate some value,
  - Or just using computation.
- Benchmarking reveals that the lookup table implementation is 20% faster.
- Then you put that implementation in your application, and discover that it is 5% slower than using the other choice. Why?

Hypothetical Example

- It could be something in the application itself is memory-demanding in a way that interferes with the table-lookup implementation.
- Performance boosting resources in processors such as
  - Caches
  - Branch Predictor Slots
  - Decoder ports, pipelines, execution units
- Are obviously finite, yet, most micro-benchmarking is oblivious to this reality: things are measured in isolation, this represents the case in which the performance boosting resources are infinite, misleading

A Real Example

- At CPPCon 2020, when discussing the performance of implementations of type erasure, I showed how the *predictability* of the trampoline targets had an impact of performance of an order of magnitude, and made qualitative differences:

- When the target is “highly unpredictable”, most of the performance inefficiency of `std::function` is hidden because of the miss-prediction execution pipeline stall.

- When this stall is expected, then there is the “execution budget” to run otherwise counterproductive techniques such as reducing parameter count by encoding and decoding into a single one.

Systematic Exploration

- I submit the “hunch” that the way Generic Programming helps performance the most is because it is the only practical way to explore systematically the complexity of the interaction between performance sensitive choices.

Part 1 Recap

- Generic Programming helps performance because:

    1. Clearly, it allows the compiler a more active role resulting in significant micro-optimizations

    2. Makes it possible to make performance sensitive choices

    3. I claim, most importantly, it allows the systematic exploration of the choices.

Part 2: How portability helps performance

- IMO, Portability has the same dichotomy as templates: “same but different”. The idea of portability is that the code has the same meaning in a different execution platform; but from the differences in the execution platforms, subtle differences in the meaning arises; just like it tends to happen for the specific characteristics of template parameters.

How portability helps performance

- Most of the code does not depend on the details of the execution platform, then portability unlocks the value of the good code.

- But then, good portability is the key to determine where the execution platform differences emerge. This improves the concepts devised to attain portatibility.

Back to Generic Programming

- We saw that Generic Programming helps through good concepts.

- I think we do Generic Programming only when we devise new concepts, not when we use them.

- How should we call the act of devising concepts? “Conceptualizing”? A set of concepts a “conceptualization”?

G P

- I began researching into the practice of making good concepts.
  - There is “Elements of Programming” by Stepanov, which is a very formal treatment of programming, many concepts that today are very familiar to C++ programmers were introduced, but in such a formal way that it is perhaps impractical.
- The C++ standard defines a lot of concepts, but mostly as terminology to discuss them in the specification of the language and library

G P

- With “From Mathematics To Generic Programming” by Stepanov and Daniel Rose you begin to “get the hang” of how to *in practice* create new, useful concepts

- There are also presentations such as Dave Abrahams keynote last year, but rather than helping learn how to make concepts they focus on improving the understanding about already made concepts such as “value semantics”.

- Myself, I’ve only devised a few concepts, not qualified to give ample guidelines on the matter

Concepts Can Be Improved

- Last year I mentioned that `boost::flat_map` expresses a flawed concept: it is a container, with the property that the elements are ordered and then the lookup is of logarithmic complexity, the problem being that it is a container, while clearly, it ought to be a *container adapter*.

- Yesterday, in conversation with Nina Ranns, I discovered that there is a serious deficiency in the conceptualization of what I call “value managers” in the standard library, including the containers and the vocabulary types such as variant, optional that internally handle a value:

  - For very practical concerns, such as the allocation of the values, there is no way to inject this into the types they manage; that their concepts do not allow any form of this type of dependency injection.

Concepts Can Be Improved

- Heard the phrase “an interface that leaks implementation details”?
- I believe that G.P. concepts may leak specifics of their realizations, furthermore, even they may leak “abstractions” to their realizations, one example may be that `std::function` leaks the abstraction of copyability to the realizations of trampoline targets (they not only have to be able to be invoked with the given signature, but also be copyable).

One recommendation I'm able to give:
Practice improving concepts

Part 2 Recap

- Good portability helps identify where the platform details matter, improve those concepts, and cycle back into better portability.

Part 3:
Concepts for
Robin Hood Hash Tables

- Because of closed addressing, they “preallocate”, hence concepts related to that

- Because they are hash tables, the concepts related to that: Hash function, “transparent” types. Should we allow weak hash functions? We did. Then, should we offer access to the mechanisms that support weak hashing, like the “hoisted hash code”?

- More questions: Because our “implementation details” are not fixed, we give access to the programmers to this.

Our Concepts for Robin Hood Hash Tables

- Do we add a “tail” of elements to not have to check reaching the end of the search while finding? We call this the Skarupke tail.

- Maintaining the “Probe Sequence Length” is essential metadata. We committed to maintain the PSL as an heterogeneous collection of data together with the “hoisted hash code”

- We decided to split hashing into three parts:

    1. User provided hash function,

    2. from hash code to slot index,

    3. from hash code to hoisted hash code in the metadata

- We committed to encode this metadata (PSL and hoisted hash code bits) by number of bits to each, but the collection of “PSL/hoisted bits” is not necessarily an array, it can be spread into an architecture of metadata blocks with more properties.

- I’ve been working into structuring the data into cache lines (blocks of 64 bytes aligned to 64)

Our Concepts
Robin Hood Hash Tables

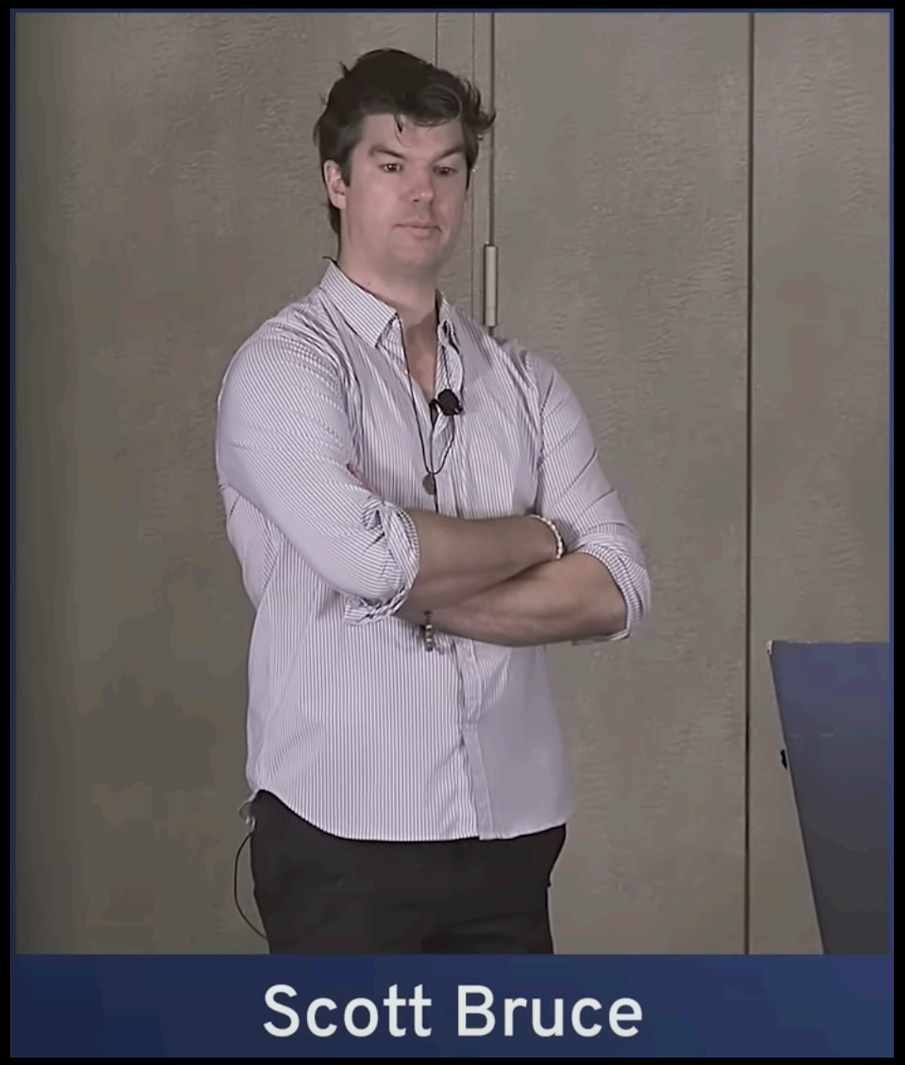
- We committed to SWAR-based finding, but, but our SWAR is extensible to physical SIMD registers. Then there are multiple concepts related to SWAR, such as how to support idiosyncrasies of SIMD registers.

Current Status

- I’ve not publicly released a complete implementation because my aim is to get the project to production strength, which means an implementation that rivals the best out there, but illustrating the Generic Programming principles unique to it.

- Early release incurs the risk of all sorts of misunderstandings.

Epilog



- I want to thank Scott Bruce publicly

Epilog

- Scott took an hiatus from the project that might be permanent.
- What we have been doing requires inspiration and mountains of work

Why Do We Come To Present?

- In my case, as many others, including Scott, this is not our “job”.
- But we are curious about ideas that look good and worthy on working on them, but that don’t fit easily into normal work.
- As an engineer I want to improve the intuition about what makes good ideas good:
  - There's no recourse but to make the work to turn them into reality
  - It is expected the majority of ideas are not better, but there's a lot of value in determining exactly why not.

Why Do We Come To Present?

- Sometimes, as in zoo's type erasure, the ideas are good enough that get sponsorship to be "productionized",
    - Great, a feedback loop of adoption and sponsorship for improvement
    - Then the work becomes bulkier, as in Edison's quote of "1% inspiration and 99% perspiration".
- This is when hard-limits hit
    - Also when cooperation becomes important

Why Do We Come To Present?

- An example: to get our RH to “production strength”, I need to learn ARM SIMD and become knowledgeable about how to control prefetching, both in x86 and ARM, but both are distractions from much higher priorities.

Thanks!