Relocation:

Blazing Fast Save And Restore, Then More!

EDUARDO MADRID

Cppcon
The C++ Conference

2024
September 15 - 20

Let’s do the opposite of pointer chasing

Alternative title

What’s what we mean to improve?

- Our systems need objects to refer to each other, and we need runtime polymorphism
- Nothing bad per se, just that it craters performance
- What if we are willing to do a lot of effort to improve performance a lot?
- One option is to apply Data Orientation techniques in general.
- Today, we give extra attention to “relocatability”

Relocatability:

- Does NOT refer to relocatability as what it means talking about “Position Independent Code”, relocation tables for dynamically linked libraries, …
- It is about the application of Data Orientation techniques with an emphasis of being able to move the data.
  - Why would we want to do that?
    - To keep the data where it leads to the best performance

Data Orientation

- In 2021 I explained, **in a very C++ centric way**, about Data Orientation techniques, the benefits to be obtained and how to apply them.
    - I think it has aged well.
- Today, we focus on how to do Data Orientation in a way in which we also have freedoms to move “things around”, for extra benefits
- Data Orientation gives primacy to the data so that computation over it has better performance than the *normal way* to design and architect systems

The “normal way”

- Classes/structs of members of heterogeneous types: won’t ever map efficiently to the computing resources
- The use of pointers for objects to refer to each other: many problems
- The use of inheritance and virtual overrides for runtime polymorphism:
  - Extra indirection, lots of allocations, cognitive load (lifetimes, sharing, unique_ptr, shared_ptr, …)

The “abnormal” way

Recap Begins

Columnar Representation (Scattering)

- Transforming collections of structs of members of heterogeneous types into one structure with arrays of homogeneous data types (going from what in databases is called a row-representation to a columnar representation) also called “scattering” or achieving structures of arrays
- …word salad?

Row-wise representation

Heterogeneous (different) data types

```
struct Item {
    int id;
    double price;
    Specials *specials;
    Category *category;
    bool is_homemade;
};
```

Pointer. Can it be null? Optional?

Necessary but performance-hostile, memory layout inefficient

Column-wise representation

```
template <std::size_t N>
struct State {
    std::array<int, N> ids;
    std::array<double, N> prices;
    EfficientMap<std::size_t, Specials> specials;
    std::array<Category_handle, N> categories;
    EfficientSet<std::size_t> homemade;
};
```

Entity-Component Systems

- Feedback from 2021:
    - After scattering, we don't have "objects" anymore.
        - What were objects will now be called "**entities**"
        - Members ==> "**components**"
        - The "system" binds components to entities:
            - Very simple binding of components to entities: The index into the global structure of arrays that contains the components

Handles

- Underneath they are an integer (the index of the components)
- Structs/Classes ==> handle

Handles

```
auto state = State<N>;

class ItemHandle {
    std::size index;
public:
    int id() { return state.ids[index]; }
    double price() { return state.prices[index]; }
    Specials *specials() { return state.specials[id()]; }
    Category_handle category() { return state.categories[id()]; }
    bool &is_homemade() { return state.homemade.contains(id()); }
};
```

What about runtime polymorphism?

- Programmers make the virtual tables
    - Zoo type-erasure already generates virtual tables
- The handles have the accessor to the virtual table ***INDEX*** as a member
- Each subtype has its own “state” struct of arrays or memory arena for their specific members (components)
- Subtype handles inherit from the supertype handle

What about resource managers?

- For example a type that creates an std::ostream to a file:
  - Managed as special cases
  - That’s not something that can be helped either
  - Fortunately, true resource management objects are very few.

We pick our poison

- Libraries, user code will be in the “normal” way, we would have to convert to the columnar representation and apply Data Orientation techniques somehow.
  - C++ still does not have reflection (introspection): we have to “pick our poison”:
    - Doing the conversion manually (good luck, you’ll need it)
    - Automating:
      - Preprocessing (my poison of choice, twice already)
      - Code generators
    - …

Why?

- Because computing with homogeneous data can extract full performance from the execution resources (SIMD, CUDA, …)
- No allocations (or much cheaper slot available management)
- Dis-incentives to communicate via state changes, others
- In general: put data where it will be used for the best performance

Recap Ends

One thing more

- Because of pointers, persisting the data of state of execution can’t be just a memory dump:
    - Because of fragmentation, and
    - when you next run the same app, the allocator will give you different addresses

Also…
Allocators don’t care about you.
Much less for the efficient locations for your data!

Then you must be free to move your data!

State changes

- As the app continues to run, lots of objects are created and destroyed
  - Then the maintenance of slots in arenas, while easier and much more performing than allocating, begin to resemble allocation/deallocation

As usual, let’s introduce a level of indirection

We need stable indices

think-cell

Cache Friendly Data + Functional + Ranges = ❤️

Björn Fahller

2024

Björn Fahller

Björn Fahller @C++ On Sea 2024

- Recapping:
    - He calls stable indices “stable ids”, and the technique is to use an extra indirection that pays for itself in terms of performance.
    - Furthermore, you can have both the forward mapping and the backward mapping, depending on your needs.
- This indirection is what provides the freedom to relocate!
- Björn Fahller presented benchmarks that dove into details of wall-clock performance, cache misses, IMO his results are representative of what would be real world use.

Advantages of Relocatability

- From Data Orientation:
  - The predictability and control of where the data will reside: where it can be used most efficiently
  - Saving the state externally is just a memory dump because the external layout will be the same as the internal, live layout.
    - Even memory mapping!
- If things (components and entities) are “shaken”, so they are not efficient anymore, you can relocate to make them efficient again.

What do we mean by “Shaken”?

- Entities get removed: they leave “holes”, empty slots in the arrays of their components: Slow down the performance of the remaining entities.
  - Relocatability allows you to “defrag” the arrays.
- New entities are added to the place of old, removed entities: they lessen time, space localities, worse memory performance.
  - Or introduce “false sharing”

What’s essential for better memory performance?

- Improving time and space locality!
- If your design a-priori has knowledge of how to increase time or space locality, relocatability is probably the set of techniques to capitalize on this knowledge

What other things could help?

Value Managers

- The focus until now has been in what to do, to accomplish relocatability.
- Could new Generic Programming Concepts help this work?
  - Last time, I introduced the concept of handle of scattered components.
  - I showed a preprocessing library to simulate reflection (introspection) to generate both the “normal” classes/structs and the handles
  - Now, there are many needs that arise in “production”:

Value Managers

- Interfacing with non-scatter aware functions that want to use the “normal” objects:
  - There is the need to create temporaries to interface
  - Lots of needs to interoperate between “normal” objects and “scattered” entities
  - Even subtler concerns

Value Managers

- From conversation with Nina Ranns:
  - The situation is that std::optional<T> does not need allocators, then it does not (currently) have a template argument for custom allocators. But std::optional can be of a std::vector type that would have a template argument for custom allocator, then, std::optional obstructs the custom allocator argument of std::vector.
  - The problem here is severe:
    - What is going on is that std::optional is what I call a *value manager* for values of other types, but the standard library has not yet become aware of this concept, much less articulated it.
    - Custom allocation is just one of very many concerns of managed values that are obstructed by their value managers!

Value Management

- The concern of allocator is clearly not the only plausible concern that concerns management of values that needs to be communicated from types that handle values to the values they manage. Vice versa: the managed values may need to communicate concerns to the manager.

- Then, we need to articulate a new concept: A Value Manager

Clear & Complete Value Management scenario

- In Type Erasure: `std::any`, especially `std::function`:
  - The **type** of the managed value is “forgotten” at compilation time: brings up the puzzle of how to manage a value you don’t know its type
  - There are different types of value managers (for type erasure):
    - Local: concerns such as buffer size and alignment
    - Non-local:
      - Allocation?
      - How to hide the indirection
      - How to take advantage of the indirection (the `ByValue` manager with the zoo::Move

Clear & Complete Value Management scenario

- What would be the value management concerns for
  - Managers that scatter data
  - Managers that manage collections of values of heterogeneous types (as in the C++ for C# Delegate we worked on at Snap)
  - Managers that do relocation!
    - Concerns about the forward and backward maps

Zoo Type Erasure

Key Successes

- Breaking type erasure into two separate concerns, of:
  - Value Management (and thus inventing the concept)
  - Affordances.
- The other things that are not relevant (top performance, infinite refinement, support for poly and paraphyletic sub typing relations)

Value Managers

- They need a template programming interface in the style of Alexandrescu’s Policy Design Pattern

Conclusion

- The language needs to improve (reflection (introspection)), libraries need to improve support for scattering and handles, programmers need to avail themselves of Data Orientation and relocatability techniques more often because they let go of massive performance gains

- Value Management concerns that emerge within the context of Data Orientation and Relocatability improve the understanding new concept of Value Manager