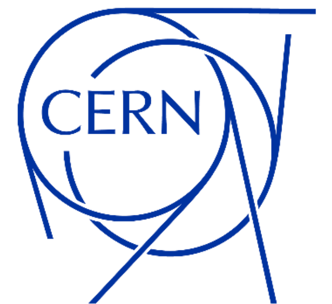
Marionette

Data Structure Description and Management for Heterogeneous Computing

WLCG/HSF Workshop
07/05/2025

Nuno dos Santos Fernandes
(nuno.dos.santos.fernandes@cern.ch)



LABORATÓRIO DE INSTRUMENTAÇÃO
E FÍSICA EXPERIMENTAL DE PARTÍCULAS

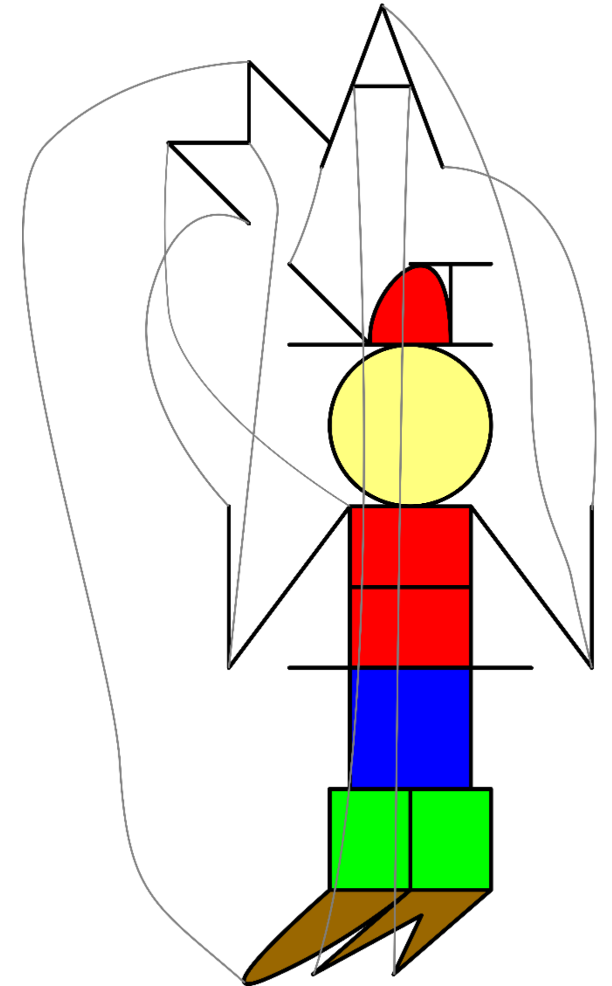
- **Marionette:** **M**emory **A**bstracted **R**epresentation with **I**nterfaces in **O**bjects **N**ecessitating **E**xtensively **T**emplated **T**ypes **E**DM

- Header-only library

- Only requires C++17

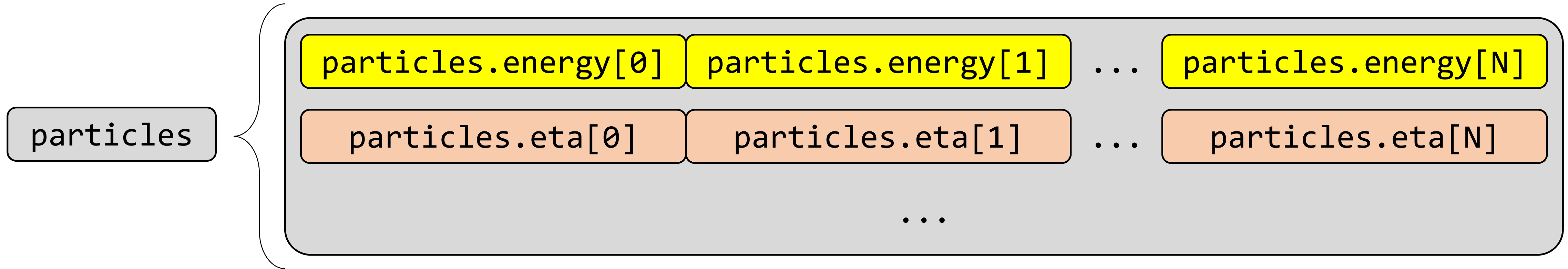
- Tested with GCC, Clang, MSVC and NVCC

- Repository here

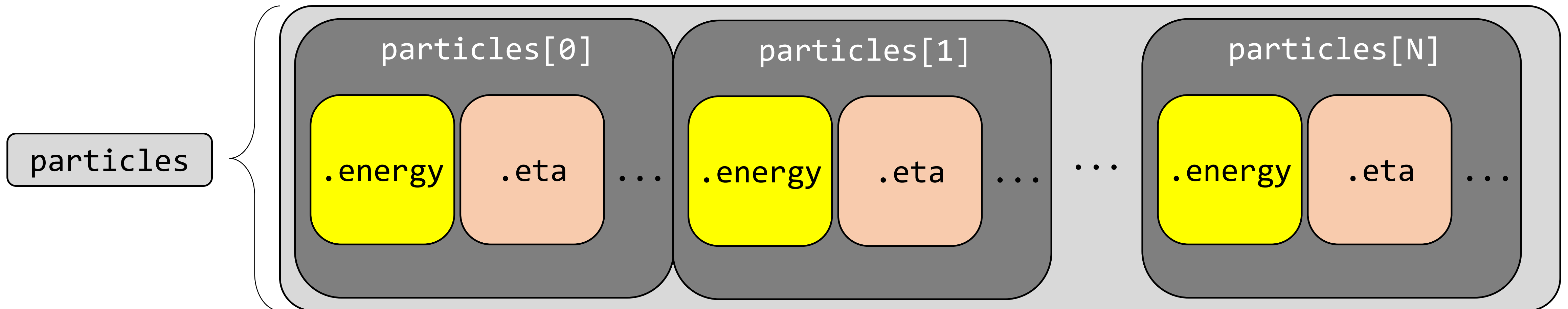


A Minor Inconvenience About Data Layout

- **Performance** may be best when **memory accesses** are **sequential: struct of arrays** (SoA)



- **Human minds** tend to work best if **all the properties of an object** are **grouped: array of structs** (AoS)

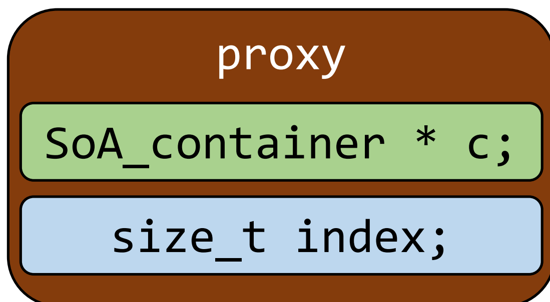


Struct of Arrays *versus* Array of Structs

- Code that uses **AoS** is not *too* hard to **adapt** to use **SoA** and vice-versa if it **deals with basic arrays**
  - This **still has to be done "by hand"**
- In reality, we tend to use **more complicated data structures**, with **pointers**, **member functions**, etc.
  - **Complexity/annoyance** of the conversion process **increases significantly**
- Even worse if one uses certain **constructions that implicitly assume AoS**, e. g. **range-based for**:

```
for (auto & particle : particles) {
  total_energy += particle.energy()
}
```

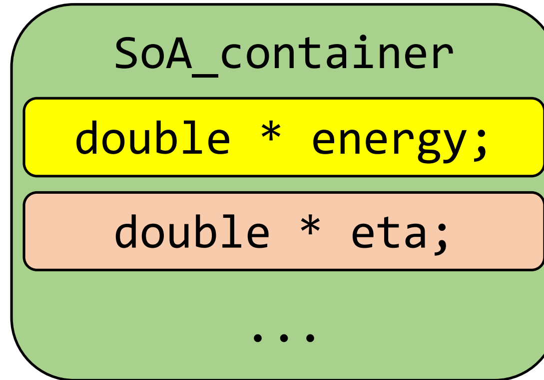
- The solution for this is to write some sort of **proxy object** that **gets the right thing from the SoA**:




```
proxy.energy()  →  (proxy.c)->energy[proxy.index]
   proxy.eta()     (proxy.c)->eta[proxy.index]
       ...                  ...
```

Further Complexity of Struct of Arrays

- Obvious downside of using proxies: functions we get "for free" with structs need to be written
  - Getters, setters, assignment operators...
- Since, in general, containers will have dynamic size, **every array is allocated separately**:



- That means writing functions that perform **allocation, deallocation, resizing**, ..., **for each property**
- Similarly for **host ↔ device transfers**: we must **write down the copies explicitly**, e. g. (CUDA):

```
cudaMemcpy(GPU_c.energy, CPU_c.energy, N * sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(GPU_c.eta, CPU_c.eta, N * sizeof(double), cudaMemcpyDeviceToHost);
...
```

Motivation for Marionette

- **What do we need to write** when implementing a data structure?
  - The **container** itself: members, allocation, deallocation, resizing…
  - The **proxies**: getters, setters, assignment…
  - The **CPU ↔ accelerator transfers**
- What if we want to deal with **more complex properties** than just a single value?
  - Natural case: each object holds a **vector of values**
  - Each object has **sub-objects** that are also rather complex
  - …
- What if we want to **experiment with/benchmark/validate** data structures with different ways of **laying out the data** in memory, while **keeping the interface** of the "old" data structures for compatibility?

Marionette was developed to answer all of these problems, and more!

Foundations of the Design of Marionette

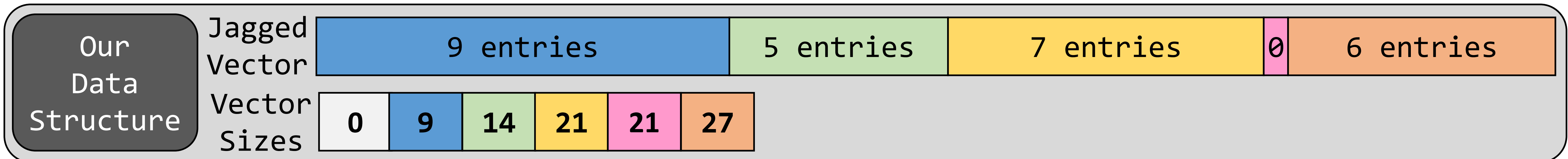
- **Main problem** with data layout: "**multiple sources of truth**" for the set of properties we want to store
  - Solution: simply **provide an explicit list of properties** to be stored, then have the infrastructure to derive the necessary data structures and functions from that
    - **Macros** are **not ideal** for this generation, given impacts on code extensibility and maintainability
    - **C++26 reflection** is **not enough** (and compiler support is not instantaneous…)
    - We already have a way to add member variables or functions to a class: inheritance
      - No runtime polymorphism, use **Curiously Recurring Template Pattern** (CRTP)
- We will **express each property as a compile-time class** which will **uniquely identify it**:

```
proxy.energy()   →   proxy.template get<Energy>()
   proxy.eta()   →   proxy.template get<Eta>()
      ...                     ...
```

- This class will contain some **relevant information** to describe what the property is (e. g. the type)
- We also add two member types to represent **`ObjectFunctions`** and **`CollectionFunctions`**
  - Accessors (`energy()`, `eta()`) can be generated through macros with user-provided property names

Types of Properties

- Currently supported types of properties were driven by the needs of our initial use case (calorimeter reconstruction in ATLAS); **extension is possible** and very much welcome based on other use cases!
  - `NoProperty`: no value stored, useful to share common interface functionality
  - `PerItemProperty`: a single (trivially copyable) value per entry
  - `JaggedVector`: a set of dynamically resizable arrays per entry, stored contiguously
  - `SubGroup`: combination of any properties, arbitrarily nestable
  - `ArrayProperty`: an array of a set of any properties, stored separately
- Orthogonally to the property type, **properties** may also be marked as **global** to associate them with the **whole container**, not to the individual objects
- **Properties are arbitrarily nestable** (with potential impact on compilation times…)
- For jagged vectors, a **prefix sum** of the number of entries per vector is stored as a global property:



Simple Data Structure Declaration Example

```
struct Particle {
  float energy, eta;
  std::array<float, 3> position;
};
using ParticleArrayOfStructs = std::vector<Particle>;
struct ParticleStructOfArrays {
  std::vector<float> energy;
  std::vector<float> eta;
  std::vector<std::array<float, 3>> position;
  //Plus any and all boilerplate to allow
  //something with a particle-like interface,
  //and push_back, and so on...
};
struct ParticleProxy {
  ParticleArrayOfStructs * ptr;
  std::size_t index;
  const float & energy() const { return ptr->energy[index]; }
  float & energy() { return ptr->energy[index]; }
  //Similarly for other accessors, also assignment to or from particles
};
//And also a constant proxy
//--------------------------------------------------------------------------------
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(energy, Energy, float);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(eta, Eta, float);
MARIONETTE_DECLARE_SIMPLE_ARRAY_PROPERTY(position, Position, 3, float);
using OurBasicParticle = Marionette::InterfaceDescription::PropertyList<Energy, Eta, Position>;
```

Complicated Data Structure Declaration Example: Normal

```
struct Particle {
  float m_energy;
  const float & energy() const { return m_energy; }
  float & energy() { return m_energy; }
  void setEnergy(float e) { m_energy = e; }
  struct FourMomentum {
    float p0, px, py, pz;
  };
  FourMomentum m_p;
  const FourMomentum & momentum() const { return m_p; }
  FourMomentum & momentum() { return m_p; }
  float m_position[3];
  const float & position(int i) const { return m_position[i]; }
  float & position(int i) { return m_position[i]; }
  const float & x() const { return m_position[0]; }
  const float & y() const { return m_position[1]; }
  const float & z() const { return m_position[2]; }
  float & x() { return m_position[0]; }
  float & y() { return m_position[1]; }
  float & z() { return m_position[2]; }
  float distanceFromCenter() { return sqrt(x() * x() + y() * y() + z() * z()); }
};
```

Complicated Data Structure Declaration Example: Marionette

```
MARIONETTE_DECLARE_PER_ITEM_PROPERTY_SIZED(energy, Energy, NMaxParticles, float);

MARIONETTE_DECLARE_PER_ITEM_PROPERTY_SIZED(p0, P0, NMaxParticles, float);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY_SIZED(px, Px, NMaxParticles, float);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY_SIZED(py, Py, NMaxParticles, float);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY_SIZED(pz, Pz, NMaxParticles, float);
MARIONETTE_DECLARE_SUBGROUP_PROPERTY_SIZED(momentum, Momentum, NMaxParticles, P0, Px, Py, Pz);

MARIONETTE_DECLARE_SIMPLE_ARRAY_PROPERTY_SIZED(position, Position, NMaxParticles, 3, float);
struct FinalPos : Position {
  template <class F, class Layout> ObjectFunctions : Position::ObjectFunctions<F, Layout> {
    decltype(auto) x() const { return static_cast<const F *>(this)->position()[0]; }
    decltype(auto) x()       { return static_cast<F *>(this)->position()[0]; }
    decltype(auto) y() const { return static_cast<const F *>(this)->position()[1]; }
    decltype(auto) y()       { return static_cast<F *>(this)->position()[1]; }
    decltype(auto) z() const { return static_cast<const F *>(this)->position()[2]; }
    decltype(auto) z()       { return static_cast<F *>(this)->position()[2]; }
    float distanceFromCenter() const {
      const F * f = static_cast<const F *>(this);
      return sqrt(f->x() * f->x() + f->y() * f->y() + f->z() * f->z());
    }
  };
};

using OurParticleProperties = Marionette::InterfaceDescription::PropertyList<Energy, Momentum, FinalPos>;
```

Layouts

- In general, we may wish to have **different strategies of laying out the data**, different allocators, etc.
- Thus we must consider the "**layout type**" as yet another **input parameter** to our data structures
- Essentially encodes **how to allocate, deallocate and resize** the data structures
  - **Fully general**: only fundamental requirement is **access to each per-item property** at a given index
- The `layout_holder` member type receives **the list of properties** as template argument and will have the **member variables** to **hold the actual data** and the **member functions** to **access, resize, etc**.
- Mostly relevant for implementers (not necessarily end users) – **talk to me if you want to know more!**
- Two default layout types: **dynamic per-array allocation** and **maximum fixed size** allocated all at once
- The layout will also impact how to perform **data transfers** between the data structures
  - A **`TransferSpecification` class** describes how to **copy or move between data structures** (including copying or moving from data structures outside of Marionette), with sane defaults being provided
  - A **`TransferPriority` enumerate** allows selecting between potentially overlapping template specializations so that more general implementations (with lower priority) can be used as a fall back

Support for Heterogeneous Computing

- **Different layout types** may refer to **memory allocated in different memory spaces** (e. g. CPU *vs.* GPU)
  - Some memory spaces **may not be able to be accessed** from where the code is being executed
  - Copying memory between memory spaces is more fundamental than between different layouts: for example, every layout on the CPU could just `memcpy`
- Thus, we introduce the concept of a **`MemoryContext`**: yet another compile-time class that will provide information on the different memory contexts we support and how to **interact with it**
- Each memory context defines a **`ContextInfo`**, to hold **relevant runtime state** (e. g. device ID)
- **Copying between memory contexts** can then also be specified to **support any relevant combination**
- The memory context also informs us of its **`AccessProperties`**, so we can (at compile time) **prevent code from accessing memory it shouldn't** by erroring out when calling functions that attempt that
  - A natural generalization of this is to consider **`{Access, Resize, Mutability}Properties`** as **`InterfaceProperties`** to completely **specify what we can or cannot do with the data**
- Each **layout type** will specify a `MemoryContext` and `InterfaceProperties` associated with it

Final Design of Marionette

- Two main classes: `Collection` and `Object`

```
template <class LayoutType, class Properties, class MetaInfo> struct Collection;
template <class ObjectType, class Properties, class MetaInfo> struct Object;
```

- The **`Collection`** implements an **`std::vector`-like interface**, with **iterators** and **range-based `for`**
- **Accessing a collection** results in an **`Object`**, independent **(owning) objects** may also be instantiated
  - **Objects** representing a **single per-item property** have **all the operators** of the corresponding type
- `MetaInfo` is meant to be **handled internally** by Marionette (instantiated with default parameter)
- `Properties` holds the **description of the data structures** as a compile-time list of classes
- `LayoutType` and `ObjectType` describe **the way the data will actually be stored**
  - `ObjectType` is **handled internally**: `ProxyObject` or `OwningObject` (but in theory extensible)
  - Some layouts for internal use implement views into collections, subspans, etc.
- Many **other customisation points and functionality – talk to me if you want to know more!**

Simple Marionette Usage

```cpp
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(a, A, float);      // Equivalent to:
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(b, B, float);      // struct Example {
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(z, Z, int);        //   float a, b;
using Marionette::InterfaceDescription::PropertyList;   //   int z;
using Properties = PropertyList<A, B, Z>;               // };

float avg1(const Marionette::Collection<ExampleLayout, Properties> & c) {
  float ret = 0;
  for (ExampleLayout::size_type i = 0; i < c.size(); ++i) {
    ret += c[i].a();
  }
  return ret / c.size();
}

float avg2(const Marionette::Collection<ExampleLayout, Properties> & c) {
  float ret = 0;
  for (auto it = c.a().begin(); it != c.a().end(); ++it) {
    ret += *it;
  }
  return ret / c.size();
}

float avg3(const Marionette::Collection<ExampleLayout, Properties> & c) {
  float ret = 0;
  for (auto && obj : c) {
    ret += obj.a();
  }
  return ret / c.size();
}
```

Jagged Vector Usage

```cpp
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(left, Left, int);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(right, Right, int);
MARIONETTE_DECLARE_JAGGED_VECTOR_PROPERTY(jag_vec, JagVec, int, Left, Right);
using JVProps = Marionette::InterfaceDescription::PropertyList<JagVec>;

void fill_vectors(Marionette::Collection<Layout, JVProps> & c) {
  for (int i = 0; i < c.size(); ++i) {
    c[i].clear();
    for (int j = 0; j < i; ++j) {
      c[i].push_back(j);
    }
  }
  print(c);                                     //[] [0] [0, 1] [0, 1, 2] ...

  c[0] = c[1];
  print(c);                                     //[0] [0] [0, 1] [0, 1, 2] ...

  c[0][0] -= 1;
  print(c);                                     //[-1] [0] [0, 1] [0, 1, 2] ...

  c[1].insert(c[1].begin(), 99);
  print(c);                                     //[-1] [99, 0] [0, 1] [0, 1, 2] ...

  c[2].push_back(100);
  for (const auto & v : c[3]) { c[2].back() += v; }
  print(c);                                     //[-1] [99, 0] [0, 1, 103] [0, 1, 2] ...
}
```

Unspecified Accesses into Array Properties

```
constexpr int arr_size = 3;
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(entry, Entry, int);
MARIONETTE_DECLARE_ARRAY_PROPERTY(inner, Inner, arr_size, Entry);
MARIONETTE_DECLARE_ARRAY_PROPERTY(middle, Middle, arr_size, Inner);
MARIONETTE_DECLARE_ARRAY_PROPERTY(outer, Outer, arr_size, Middle);
using Props = Marionette::InterfaceDescription::PropertyList<Outer>;

void test() {
  Marionette::Collection<Layout, Props> c(3);
  for (int i = 0; i < arr_size; ++i) {
    for (int j = 0; j < arr_size; ++j) {
      for (int k = 0; k < arr_size; ++k) {
        c[i][j][k][0] = i;
        c[i][j][k][1] = j;
        c[i][j][k][2] = k;
      }
    }                                                //|(0,0,0)(0,0,1)(0,0,2)||(1,0,0)(1,0,1)(1,0,2)||(2,0,0)(2,0,1)(2,0,2)|
  }                                                  //|(0,1,0)(0,1,1)(0,1,2)||(1,1,0)(1,1,1)(1,1,2)||(2,1,0)(2,1,1)(2,1,2)|
  print(c); // <------------------------------------ //|(0,2,0)(0,2,1)(0,2,2)||(1,2,0)(1,2,1)(1,2,2)||(2,2,0)(2,2,1)(2,2,2)|

  c.outer().middle().inner() = std::vector<int>{0,0,0};
  c[2][1].inner()  = std::vector<int>{7,7,7};
  c[1].middle()[2] = std::vector<int>{9,9,9};        //|(4,5,6)(0,0,0)(0,0,0)||(4,5,6)(0,0,0)(9,9,9)||(4,5,6)(0,0,0)(0,0,0)|
  c.outer()[0][0]  = std::vector<int>{4,5,6};        //|(0,0,0)(0,0,0)(0,0,0)||(0,0,0)(0,0,0)(9,9,9)||(7,7,7)(7,7,7)(7,7,7)|
  print(c); // <------------------------------------ //|(0,0,0)(0,0,0)(0,0,0)||(0,0,0)(0,0,0)(9,9,9)||(0,0,0)(0,0,0)(0,0,0)|

                                                     //|(4,5,6)(0,0,0)(0,0,0)||(4,5,6)(4,5,6)(4,5,6)||(4,5,6)(0,0,0)(0,0,0)|
  c[1].middle().inner() = c[0][0][0];                //|(0,0,0)(0,0,0)(0,0,0)||(4,5,6)(4,5,6)(4,5,6)||(7,7,7)(7,7,7)(7,7,7)|
  print(c); // <------------------------------------ //|(0,0,0)(0,0,0)(0,0,0)||(4,5,6)(4,5,6)(4,5,6)||(0,0,0)(0,0,0)(0,0,0)|
}
```

Subspans

```
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(entry, Entry, int);
using Props = Marionette::InterfaceDescription::PropertyList<Entry>;

void subspanning() {

  Marionette::Collection<Layout, Props> c(10, 1); // 0   1   2   3   4   5   6   7   8   9
  print(c);                                       //|1| |1| |1| |1| |1| |1| |1| |1| |1| |1|

  auto first_three = c.front_range(3);
  for (auto && v : first_three) : { v = 2; }      // 0   1   2   3   4   5   6   7   8   9
  print(c);                                       //|2| |2| |2| |1| |1| |1| |1| |1| |1| |1|

  c.insert(c.begin(), 0);                         // 0   1   2   3   4   5   6   7   8   9   10
  print(c);                                       //|0| |2| |2| |2| |1| |1| |1| |1| |1| |1| |1|

  for (auto && v : first_three) : { v += 3; }     // 0   1   2   3   4   5   6   7   8   9  10
  print(c);                                       //|3| |5| |5| |2| |1| |1| |1| |1| |1| |1| |1|

  auto back_part = c.back_range(2);
  back_part.resize(4, 7);                          // 0   1   2   3   4   5   6   7   8   9   10  11  12
  print(c);                                        //|3| |5| |5| |2| |1| |1| |1| |1| |1| |1| |1| |7| |7|

  c.erase(c.begin() + 9);                          // 0   1   2   3   4   5   6   7   8   9   10  11
  print(c);                                        //|3| |5| |5| |2| |1| |1| |1| |1| |1| |1| |7| |7|

  for (auto && v : back_part) : { v += 2; }        // 0   1   2   3   4   5   6   7   8   9   10  11
  print(c);                                        //|3| |5| |5| |2| |1| |1| |1| |1| |3| |3| |9| |9|

  c.range(5, 7) = std::vector<int>{7,2,0,2,1,0,5}; // 0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
  print(c);                                        //|3| |5| |5| |2| |1| |7| |2| |0| |2| |1| |0| |5| |1| |3| |3| |9| |9|
}
```

Fast Size Changing Operations

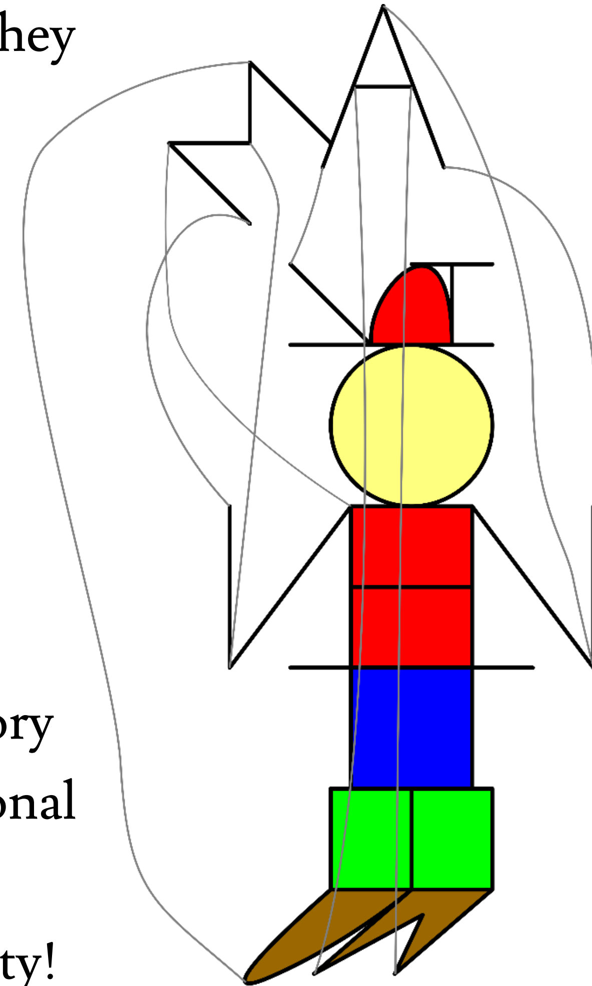
```
using CollT = Marionette::Collection<Layout, Properties>;
//Properties has LargestProperty as a per-item property corresponding to the largest type.

void fast_size_changes(CollT & c) {
  CollT temp(Marionette::fast_resize, 50);  //No need to initialize as we will overwrite.

  for (int i = 0; i < 50; ++i){ temp[i] = some_operation(c[2 * i], c[2 * i + 1]); }

  CollT other(Layout::memory_context().clone_info(c.memory_context_info())); //New collection with equivalent context info.

  for (const auto & v : temp){ if (some_filter(v)) { other.push_back(v); } }

  c.fast_insert(50, other.size()); //Copy in batches of 50, do not initialize new elements.

  for (int i = 0; i < other.size(); ++i) { c[50 + i] = other[i]; }

  const auto buf_size = temp.size() * sizeof(LargestProperty::Type);        /*+----------------------------------------------+*/
  other.fast_insert(temp.largest_property().data(), //buffer                 | In this case, this means copying all the     |
                    Layout::memory_context(),       //buffer context         | elements of other to the buffer,             |
                    temp.memory_context_info(),     //buffer context info    | as the total number of bytes to be copied    |
                    buf_size,                       //buffer size in bytes   | is never larger than the size of the buffer, |
                    0,                              //index at which to insert | and then back to the array in other        |
                    other.size());                  //size of the insertion   +----------------------------------------------+

  c.fast_erase(temp.largest_property().data(),      //                        +----------------------------------------------+
               Layout::memory_context(),            //                        | Since the buffer will be smaller than the    |
               temp.memory_context_info(),          //                        | total number of bytes, we copy in batches    |
               buf_size,                            //                        | of that maximum size to the buffer, and then |
               0,                                   //                        | from there to the destination array.         |
               other.size()                         //                        | We pass an extra argument of immediate       |
               Marionette::immediate);              //                        | so the operation finishes before returning.  |
}                                                   //                        +----------------------------------------------+
```

Marionette GPU Usage Example (CUDA)

```
MARIONETTE_DECLARE_PER_ITEM_PROPERTY_SIZED(energy, Energy, 256, float);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY_SIZED(time, Time, 256, float);
struct Foo : Marionette::InterfaceDescription::NoProperty{
  template <class Final, class Layout> struct ObjectFunctions {
    __host__ __device__ int foo() const { return 42; }
  };
};
struct Bar : Marionette::InterfaceDescription::NoProperty {
  template <class Final, class Layout> struct CollectionFunctions {
    __host__ __device__ void bar() { static_cast<Final *>(this)->energy()[0] *= 1.21e9f;
  };
};
using Props = Marionette::InterfaceDescription::PropertyList<Energy, Time, Foo, Bar>;
template <class Context>
using OurCollection = Marionette::Collection<Marionette::LayoutTypes::DynamicStructInContext<Context, int>, Props>;
using CPUCollection = OurCollection<Marionette::MemoryContexts::CUDAHostPinned>;  //Pinned memory for faster transfers
using GPUCollection = OurCollection<Marionette::MemoryContexts::CUDAStandardGPU>; //Normally allocated GPU memory
float main {
  CPUCollection coll(42);                                                 //Instantiate a collection of 42 elements
  std::vector<float> desired_times(50, 10.f);                             //Instantiate a vector for initialization
  coll.time() = desired_times;                                            //coll.time() behaves as a vector
  GPUCollection gpu_coll = coll;                                          //Copy-construct a collection on the GPU
  some_kernel<<<4, 64>>>(Marionette::Collections::pass_by_value(gpu_coll)); //Pass that collection to a GPU kernel
  coll = gpu_coll;                                                        //Copy-assign back to the CPU collection
}
```

Key Messages

- Marionette allows describing general data structures independently of the way they will be stored in memory
  - In particular, for heterogeneous computing platforms
- Intuitive object-oriented interface, extensible with arbitrary functions
  - The behaviour of pre-existing structures can be replicated entirely
    - Porting pre-existing code to use Marionette requires minimal changes
- Focus on configurability and extensibility to cater to many possible use cases
- Minimal to no impact on performance compared to writing everything by hand
  - No boilerplate required to support multiple ways of laying out data in memory
  - Optimised data transfers/conversion made possible without any additional effort from the end user
- Many more features than described here, and always eager to extend functionality!



Get in touch if you are interested in trying out Marionette!

Thank you for your attention!

Backup Slides

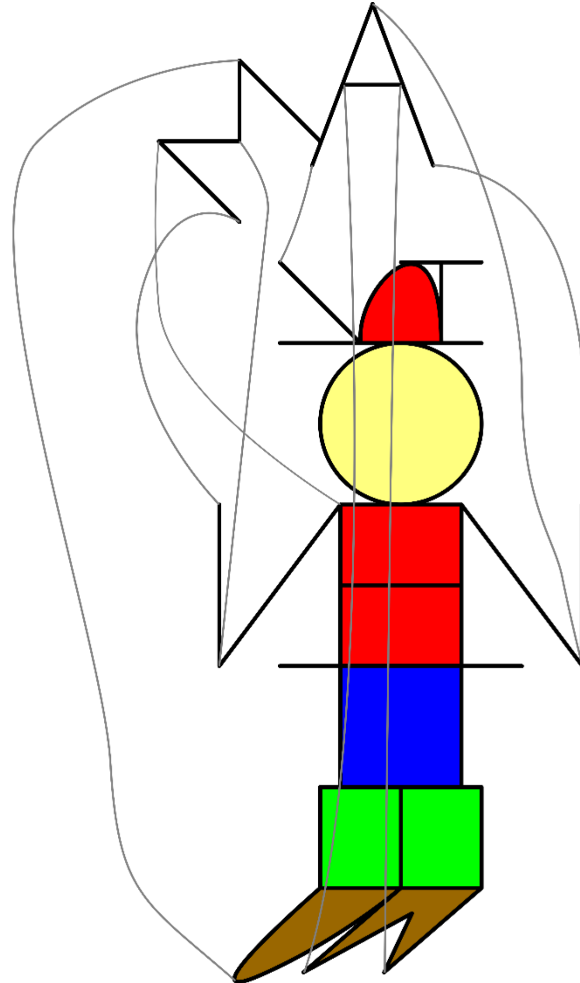
What is Marionette?

Memory **A**bstracted **R**epresentation with **I**nterfaces in **O**bjects
Necessitating **E**xtensively **T**emplated **T**ypes **E**DM

Memory Abstracted:
Final interface does not depend on where data is laid out in memory

Representation (I):
We describe what we want to store, not (necessarily) how it is stored

Representation (II):
Non-owning “views” are handled in the same way as owning containers, with equivalent functionality



Interfaces in Objects:
The interface of the final classes can be expanded with arbitrary functions

Objects:
Regardless of how the memory is stored, we can represent an element of a collection

Necessitating Extensively Templated Types:
Why I’ve been working on this for over a year now

EDM:
Event Data Model, and I needed the E to complete the word somehow…

Property Description

```
struct PropertyName: <PropertyType> (, GlobalProperty)
{
 (static constexpr MaximumSizeType max_entries = ...;)

  template <class Final, class FinalLayout>
  struct CollectionFunctions;

  template <class Final, class FinalLayout>
  struct ObjectFunctions;

 (using Type = ...;)/(using Properties = ...;)
};
```

For layouts that allocate a maximum size, 0 if this should be unlimited

Add arbitrary functions to the interface of the final object via CRTP, potentially constraining them depending on the FinalLayout.

Depending on the type of property

- Properties that derive from `GlobalProperty` have their sizes **tracked independently** from the main collection and are not part of the objects; this is **non-transitive and nestable**
  - In general, then, we have to consider different "**size tags**" to describe these independent sizes
- Macros are provided to make it easier to write the most common case of a property with accessors

Specifying Properties With Macros

- We currently offer a set of macros to specify a property of the supported types with getters and setters:

```
MARIONETTE_DECLARE_PER_ITEM_PROPERTY[_SIZED](NAME, CAPNAME, [MAXSIZE,] TYPE)
MARIONETTE_DECLARE_JAGGED_VECTOR_PROPERTY[_SIZED](NAME, CAPNAME, [MAXSIZE,] ...)
MARIONETTE_DECLARE_SUBGROUP_PROPERTY[_SIZED](NAME, CAPNAME, [MAXSZ,] ...)
MARIONETTE_DECLARE_PROPERTY_ARRAY_PROPERTY[_SIZED](NAME, CAPNAME, [MAXSZ,] ARRSZ,...)
MARIONETTE_DECLARE_SIMPLE_PROPERTY_ARRAY_PROPERTY[_SIZED](NAME, CAPNAME, [MAXSZ,] ARRSZ, TYPE)
```

- `NAME` is the name of the property starting with a lower case letter
- `CAPNAME` is the name starting with upper case
- `MAXSZ` (for `_SIZED` versions of the macros) is the maximum size
- The last parameters specify either the underlying type of a simple property (or an array thereof) or the set of properties (referred to by their `CAPNAME`s) which this property will contain
- `ARRSZ` (for property arrays) is the number of entries of the array

Transfer Specification

- Transfers between different `ObjectTypes` or collections of different `LayoutTypes` are handled by:

```
template <class Destination, class Source, TransferPriority priority, class SFINAE_Extra>
struct TransferSpecification;
```

- `TransferPriority` is an enum that more easily allows us to establish an **order of priority** (with three levels for both default and user-provided transfer specifications)
- `TransferSpecification` provides `can_copy`, `can_move`, `copy`, `move`
- These classes are then used within the object and collection definition to automatically provide the **appropriate `operator=` with valid types** and also **`copy` and `move`** that receive **additional parameters**
- Basic use cases already provided, falling back to a per-array copy if no other option is available
- Users can and should **specialize for particular layouts** to have **optimal transfers** (e. g. `memcopy` everything if the properties are the same)
- TransferSpecifications that copy from a non-Marionette class to Marionette can also be written
  - Futurely, `copy` (and possibly `move`) free functions may be provided to allow the reverse direction

More Examples of Marionette Usage

```
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(energy, Energy, float);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(time, Time, float);
struct Foo : Marionette::InterfaceDescription::NoObject {
  template <class Final, class Layout> struct ObjectFunctions {
    int foo() const { return 42; }
  };
};
struct Bar : Marionette::InterfaceDescription::NoObject {
  template <class Final, class Layout> struct ObjectFunctions {
    float bar() const { return static_cast<const Final *>(this)->energy(); }
  };
};
using ExampleType = Marionette::InterfaceDescription::PropertyList<Energy, Time, Foo, Bar>;
using OurCollection = Marionette::Collections::Collection<Marionette::LayoutTypes::StandardVectorPerItem, ExampleType>;

OurCollection coll(42);
std::vector<float> desired_times(50, 10.f);
coll.time() = desired_times;
float incrementor = 1.f;
for (auto && e : coll.energy()) {
  e = incrementor;
  incrementor += 1.f;
  std::cout << e << std::endl;
}
for (auto && obj : coll) {
  obj.setStuff(obj.energy() * obj.time());
}
std::cout << coll[9].foo() + coll[10].bar();
coll[11] = coll[33];
```

Additional Considerations Regarding Properties

- In some cases, we want to have properties that are **indexed by integers** and still **refer to the arrays** corresponding to **each index individually** (`std::vector<T[N]>` interface, `std::vector<T>[N]` storage)
  - To handle this **without undefined behaviour**, we associate an **extent** with every property
    - Instead of `T * array`, we have `T * array[extent]` so we can **index it at run time**
- Some properties can be only meaningfully assigned to the **whole container**, not to the individual objects: **global properties**
- In some cases, each **entry** may hold a **variable number of values** (a **vector**)
  - We want to be able to **store this in an efficient way**: all values stored contiguously as **jagged vector**
  - We store a **prefix sum** of the number of entries per object in the collection to know the bounds: