Awkward Array

Accelerating Awkward Array Builders

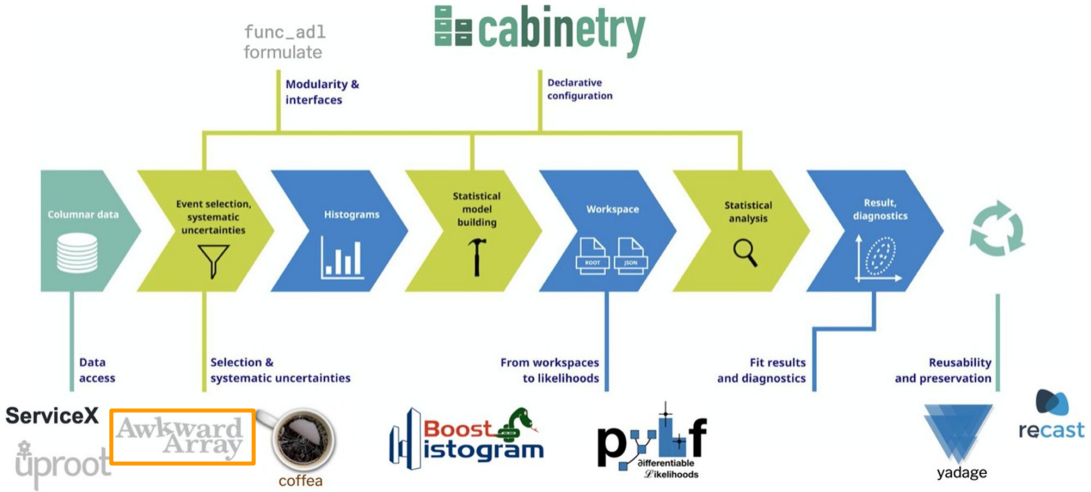
Manasvi Goyal
IRIS-HEP Fellow
Delhi Technological University

Ianna Osborne
IRIS-HEP Mentor
Princeton University

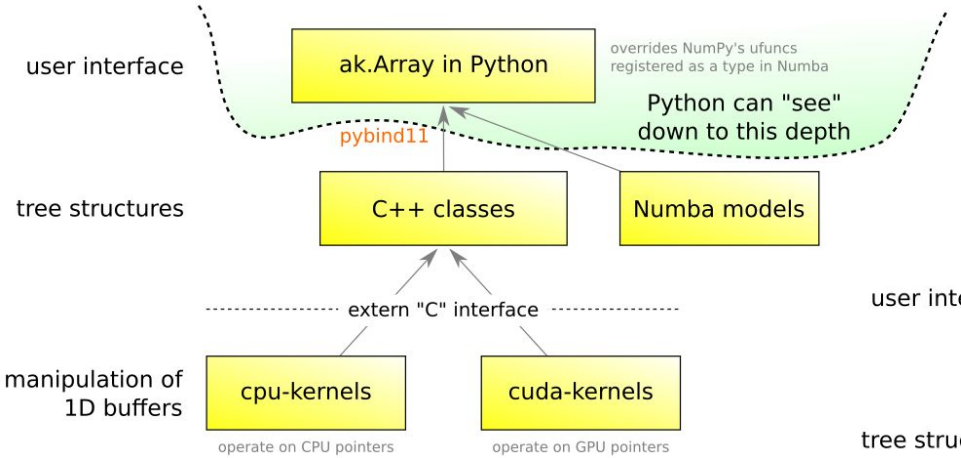
Jim Pivarski
IRIS-HEP Mentor
Princeton University

Awkward Arrays

- Awkward Array is a library for nested, variable-sized data, including arbitrary-length lists, records, mixed types, and missing data, to manipulate JSON-like data using *NumPy-like idioms*.

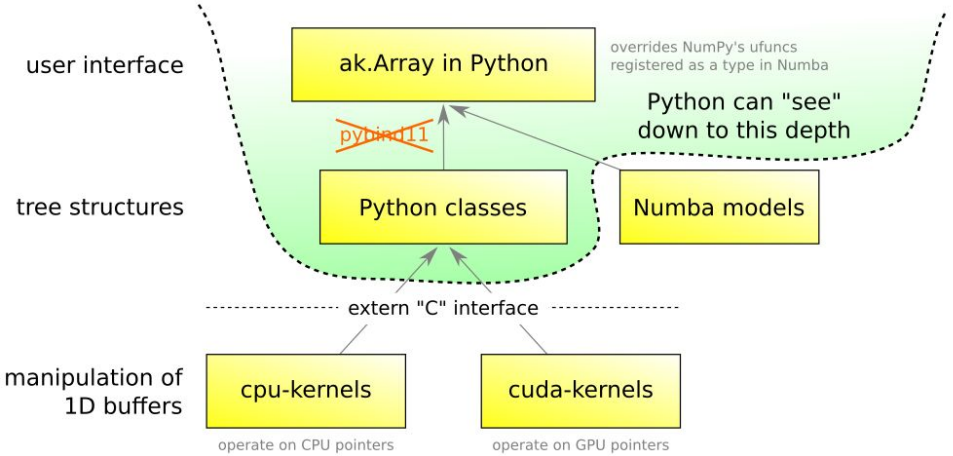


Evolution of Architecture



Architecture of Awkward 1.x

Architecture of Awkward 2.x



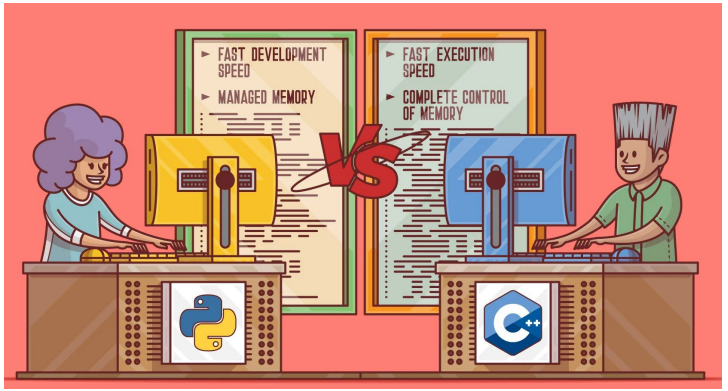
Lessons learned in Python-C++ integration
Jim Pivarski, Princeton University
ACAT 2021

About the Project

- My project concentrates on accelerating the performance of the Awkward Array Layout Builders by preventing unnecessary memory copies, optimised allocation of memory and improving the speeds.
- Modifying Growable Buffer to use multi-panels approach.
- To develop compile-time, templated, header-only C++ libraries which can be dropped into any external project.
- Writing unit tests in C++, documentation and a user guide.

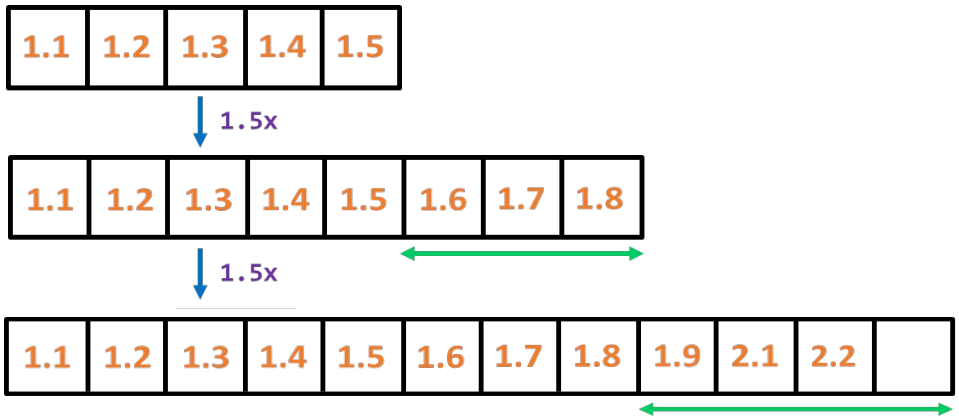
Python-C++ Integration

- Binding Python and C++ to get advantage of best features of both languages.
- The header-only implementation allows using Awkward Arrays in an external project without linking to the awkward libraries.
- Minimal code, no specialised data types.
- This facilitates dynamically generating Layout Builder from strings in Python and then compiling it with Cling.



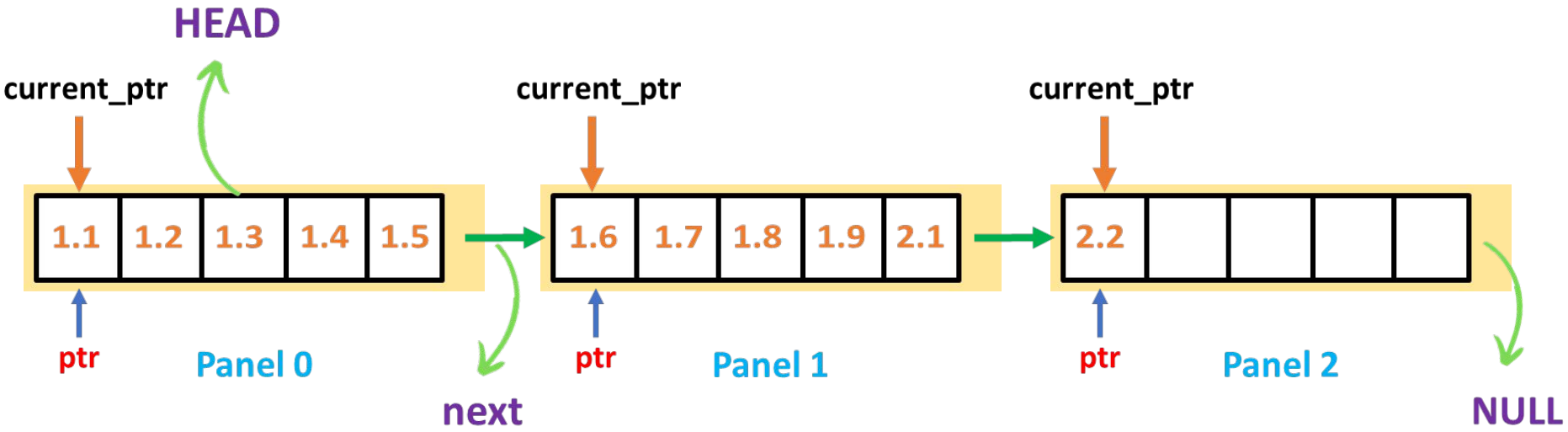
Previous Growable Buffer

```
size_t reserved_ = 5, resize = 1.5;

double data[11] = {1.1, 1.2, 1.3, 1.4, 1.5, 1.6,
                   1.7, 1.8, 1.9, 2.1, 2.2};
```




```
template <typename T>
void GrowableBuffer<T>::append(T datum) {
  if (length_ == reserved_)
    set_reserved((size_t)ceil(reserved_ * resize));
  ptr_.get()[length_] = datum;
  length_++;
}

template <typename T>
void GrowableBuffer<T>::set_reserved(size_t minreserved) {
  if (minreserved > reserved_) {
    UniquePtr ptr(reinterpret_cast<T*>(awkward_malloc(
    (int64_t)(minreserved * sizeof(T)))));
    memcpy(ptr.get(), ptr_.get(), length_* sizeof(T));
    ptr_ = std::move(ptr);
    reserved_ = minreserved;
  }
}
```

Growable Buffer with Panels



Growable Buffer Unit Tests

```
void test_complex() {
  int data_size = 9;
  std::complex<double> data[9] = {{1.1, 0.1}, {2.2, 0.2}, {3.3, 0.3},
                                  {4.4, 0.4}, {5.5, 0.5}, {6.6, 0.6},
                                  {7.7, 0.7}, {8.8, 0.8}, {9.9, 0.9}};

  awkward::BuilderOptions options { 3, 1 };
  auto buffer = GrowableBuffer<std::complex<double>>::empty(options);

  for (int64_t i = 0; i < data_size; i++)
    buffer.append(data[i]);

  std::complex<double>* ptr = new std::complex<double>[data_size];
  buffer.concatenate(ptr);

  for (int64_t at = 0; at < buffer.length(); at++) {
    assert(ptr[at] == data[at]);
  }
}
```

```
void test_extend() {
  size_t data_size = 15;
  double data[15] = {1.1, 1.2, 1.3, 1.4, 1.5,
                     1.6, 1.7, 1.8, 1.9, 2.1,
                     2.2, 2.3, 2.4, 2.5, 2.6};

  awkward::BuilderOptions options { 5, 1 };
  auto buffer = awkward::GrowableBuffer<double>
                ::empty(options);

  buffer.extend(data, data_size);

  double* ptr = new double[buffer.length()];
  buffer.concatenate(ptr);

  for (size_t i = 0; i < buffer.length(); i++) {
    assert(ptr[i] == data[i]);
  }
}
```

Layout Builders

- Layout Builder is a templated static C++ code, implemented entirely in header files, and easily separable from the rest of the Awkward C++ codebase.
- It uses header-only GrowableBuffer.
- Three phases:
  - *Constructing a Layout Builder:* from variadic templates!
  - *Filling the Layout Builder:* while repeatedly walking over the raw pointers within the LayoutBuilder
  - *Taking the data out to user allocated buffers:* Then user can pass them to Python if they want.

User Interface

- A Layout Builder provides information about :
  - What's the Awkward Form and its form keys?
  - What are the names, size (in bytes) and number of the buffers?
    - Map the node names to the numbers of bytes on the buffer nodes .
  - What data is filled in the buffers?
    - For a form_key and a user-given pointer, fill data into this pointer.

Record Builder Example

```
enum Field : std::size_t {x, y};

UserDefinedMap fields_map({
  {Field::x, "x"},
  {Field::y, "y"}});

  RecordBuilder<
      RecordField<Field::x, NumpyBuilder<double>>,
      RecordField<Field::y, ListOffsetBuilder<int64_t,
          NumpyBuilder<int32_t>>>
  > builder;

builder.set_field_names(fields_map);

auto& x_builder = builder.field<Field::x>();
auto& y_builder = builder.field<Field::y>();
```

```
x_builder.append(1.1);
auto& y_subbuilder =
  y_builder.begin_list();
y_subbuilder.append(1);
y_builder.end_list();
```

Record 1

```
x_builder.append(2.2);
y_builder.begin_list();
y_builder.end_list();
```

Record 2

```
x_builder.append(3.3);
y_builder.begin_list();
y_subbuilder.append(1);
y_subbuilder.append(2);
y_builder.end_list();
```

Record 3

```
[
  {"x": 1.1, "y": [1]},
  {"x": 2.2, "y": []},
  {"x": 3.3, "y": [1, 2]},
]
```

Record Builder User Interface

➔ Check the validity of the buffer

```
std::string error;
assert (builder.is_valid(error) == true);
```

➔ Retrieve the names and the size of the buffers in bytes

```
std::map<std::string, size_t> names_nbytes = {};
builder.buffer_nbytes(names_nbytes);
assert (names_nbytes.size() == 3);
```

➔ Allocate the buffers, map using the same names/sizes as above, and fill them.

```
auto buffers = empty_buffers(names_nbytes);
builder.to_buffers(buffers);
```

Layout Builder Form

```
{
    "class": "RecordArray",
    "contents": {
        "x": {
            "class": "NumpyArray",
            "primitive": "float64",
            "form_key": "node1"
        },
        "y": {
             class": "ListOffsetArray",
             "offsets": "i64",
             "content": {
                 "class": "NumpyArray",
                 "primitive": "int32",
                 "form_key": "node3"
              },
              "form_key": "node2"
        } "
    },
    "form_key": "node0"
}
```

Builders for Records with No Fields

Empty Record Layout Builder

If is_tuple = false

Record

```
{
    "class": "RecordArray",
    "contents": [],
    ""form_key": "node0"
}
// [{}, {}, {}]
```

If is_tuple = true

Tuple

```
{
    "class": "RecordArray",
    "contents": (),
    ""form_key": "node0"
}
// [(), (), ()]
```

Tuple Layout Builder

no fields!

```
{
    "class": "RecordArray",
    "contents": [
        {
            "class": "NumpyArray",
            "primitive": "float64",
            "form_key": "node1"
        },
        {
            class": "ListOffsetArray",
            "offsets": "i64",
            "content": {
                "class": "NumpyArray",
                "primitive": "int32",
                "form_key": "node3"
             },
           "form_key": "node2"
        } "
    },
    "form_key": "node0"
}
// [(1.1, [1]), (2.2, [1, 2]), (3.3, [1, 2, 3])]
```

Classes in Layout Builders

There are 14 Classes in Layout Builder namespace -

- ➔ Numpy Layout Builder
- ➔ ListOffset Layout Builder
- ➔ List Layout Builder
- ➔ Empty Layout Builder
- ➔ Record Layout Builder
- ➔ Empty Record Layout Builder
- ➔ Tuple Layout Builder
- ➔ Regular Layout Builder
- ➔ Indexed Layout Builder
- ➔ Indexed Option Layout Builder
- ➔ Unmasked Layout Builder
- ➔ ByteMasked Layout Builder
- ➔ BitMasked Layout Builder
- ➔ Union Layout Builder

to_buffers in Layout & Array Builders

Float64 ArrayBuilder

(concatenates data, typecast, returns form)

```
const std::string
Float64Builder::to_buffers(BuffersContainer& container, int64_t& form_key_id) const {
  std::stringstream form_key;
  form_key << "node" << (form_key_id++);

  buffer_.concatenate(
    reinterpret_cast<double*>(
      container.empty_buffer(form_key.str() + "-data",
      buffer_.length() * (int64_t)sizeof(double))));

  return "{\"class\": \"NumpyArray\", \"primitive\": \"float64\", \"form_key\": \""
         + form_key.str() + "\"}";
}
```

```
void
to_buffers(std::map<std::string, void*>& buffers) const noexcept {
   offsets_.concatenate(static_cast<PRIMITIVE*>
     (buffers["node" + std::to_string(id_) + "-offsets"]));
   content_.to_buffers(buffers);
}
```

ListOffset LayoutBuilder

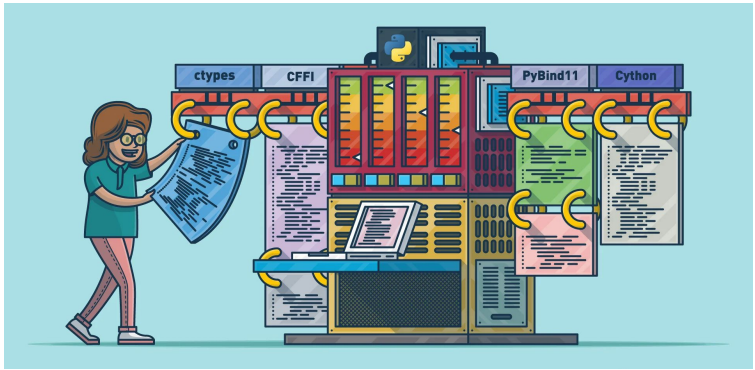
(concatenates data and fills it in a map)

Layout Builders in RDataFrame

```
NumpyBuilder = cppyy.gbl.awkward.LayoutBuilder.Numpy[data_type]
builder = NumpyBuilder()
form = ak._v2.forms.from_json(form_str)
builder_type = type(builder).__cpp_name__

cpp_buffers_self.fill_from[builder_type](builder)
names_nbytes = cpp_buffers_self.names_nbytes[builder_type](builder)

buffers = empty_buffers(cpp_buffers_self, names_nbytes)
cpp_buffers_self.to_char_buffers[builder_type, data_type](builder)

array = ak._v2.from_buffers(
    form,
    builder.length(),
    buffers,
)
return _wrap_as_record_array(array)
```

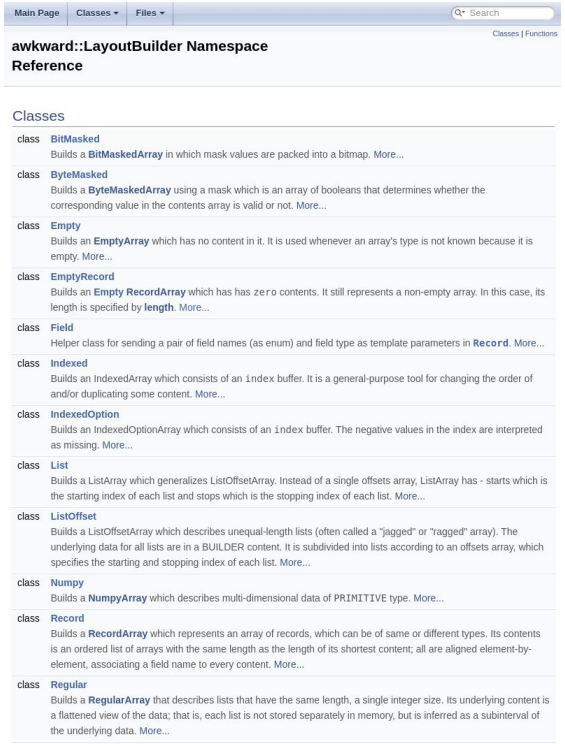
Key Insights

- Minimum Requirements of Layout Builders is C++14.
- The C++ tests are configured to be built by using CMakeLists.txt.
- Memory Management - ***std::unique_ptr*** owns and manages the memory of the multiple panels.
- ArrayBuilder is now using the GrowableBuffer with multi-panels, so now the only full copy of the data is allocated and owned by NumPy.



Closing Remarks

- Documentation of the Layout Builder and GrowableBuffer.
- User Guide - How to use header-only Layout Builder
- This project is selected for an Oral Presentation at ACAT 2022 - The Awkward World of Python and C++.



Main Page | Classes | Files | Search

Classes | Functions

awkward::LayoutBuilder Namespace Reference

Classes

class **BitMasked**
Builds a **BitMaskedArray** in which mask values are packed into a bitmap. More...

class **ByteMasked**
Builds a **ByteMaskedArray** using a mask which is an array of booleans that determines whether the corresponding value in the contents array is valid or not. More...

class **Empty**
Builds an **EmptyArray** which has no content in it. It is used whenever an array's type is not known because it is empty. More...

class **EmptyRecord**
Builds an **Empty RecordArray** which has has `zero` contents. It still represents a non-empty array. In this case, its length is specified by **length**. More...

class **Field**
Helper class for sending a pair of field names (as enum) and field type as template parameters in **Record**. More...

class **Indexed**
Builds an IndexedArray which consists of an `index` buffer. It is a general-purpose tool for changing the order of and/or duplicating some content. More...

class **IndexedOption**
Builds an IndexedOptionArray which consists of an `index` buffer. The negative values in the index are interpreted as missing. More...

class **List**
Builds a ListArray which generalizes ListOffsetArray. Instead of a single offsets array, ListArray has - starts which is the starting index of each list and stops which is the stopping index of each list. More...

class **ListOffset**
Builds a ListOffsetArray which describes unequal-length lists (often called a "jagged" or "ragged" array). The underlying data for all lists are in a BUILDER content. It is subdivided into lists according to an offsets array, which specifies the starting and stopping index of each list. More...

class **Numpy**
Builds a **NumpyArray** which describes multi-dimensional data of `PRIMITIVE` type. More...

class **Record**
Builds a **RecordArray** which represents an array of records, which can be of same or different types. Its contents is an ordered list of arrays with the same length as the length of its shortest content; all are aligned element-by-element, associating a field name to every content. More...

class **Regular**
Builds a **RegularArray** that describes lists that have the same length, a single integer size. Its underlying content is a flattened view of the data; that is, each list is not stored separately in memory, but is inferred as a subinterval of the underlying data. More...