The Awkward World of Python and C++

Manasvi Goyal[1], Ianna Osborne[2], Jim Pivarski[2]

[1] Delhi Technological University, Delhi, India
[2] Princeton University, Princeton, NJ 08544, USA

E-mail: mg.manasvi@gmail.com

Abstract. There are undeniable benefits of binding Python and C++ to take advantage of the best features of both languages. This is especially relevant to the HEP and other scientific communities that have invested heavily in the C++ frameworks and are rapidly moving their data analyses to Python. Version 2 of Awkward Array, a Scikit-HEP Python library, introduces a set of header-only C++ libraries that do not depend on any application binary interface. Users can directly include these libraries in their compilation instead of linking against platform-specific libraries. This new development makes the integration of Awkward Arrays into other projects easier and more portable, as the implementation is easily separable from the rest of the Awkward Array codebase. The code is minimal; it does not include all of the code needed to use Awkward Arrays in Python, nor does it include references to Python or pybind11. The C++ users can use it to make arrays and then copy them to Python without any specialized data types - only raw buffers, strings, and integers. This C++ code also simplifies the process of just-in-time (JIT) compilation in ROOT. This implementation approach solves some of the drawbacks, like packaging projects where native dependencies can be challenging. In this paper, we demonstrate the technique to integrate C++ and Python using a header-only approach. We also describe the implementation of a new LayoutBuilder and a GrowableBuffer. Furthermore, examples of wrapping the C++ data into Awkward Arrays and exposing Awkward Arrays to C++ without copying them are discussed.

1. Introduction
Awkward Array [1] is an important tool for physics analysis in Python for High Energy Physics (HEP) community. It is part of the `Scikit-HEP` [2] ecosystem. Nested, variable-length lists ("ragged" or "jagged" arrays), records with differently typed fields, missing data, and other heterogeneous data (union/variant types) can be defined as a set of primitives using NumPy-like [3] phrases in Python [4]. In Awkward arrays, a single user-facing `ak.Array` consists of one small tree with large contiguous data buffers attached to each node [5], as shown in Figure 1. Compiled operations are performed on these data buffers, not on the objects they represent.

In this work, we present new tools for creating Awkward Arrays in C++. Previously, the main codebase was written in C++ [6] with the idea that the downstream code would link to `libawkward.so`, but that route is full of hidden issues [7]. The method of a small, header-only library that only fills array buffers for downstream code to pass from C++ to Python using only `ctypes` has considerably more promise.

2. Python-C++ Integration
Nowadays, more front-end users use Python [8], but large-scale processing still needs to have high performance of C++ [7]. That is why we combine Python and C++ to take advantage
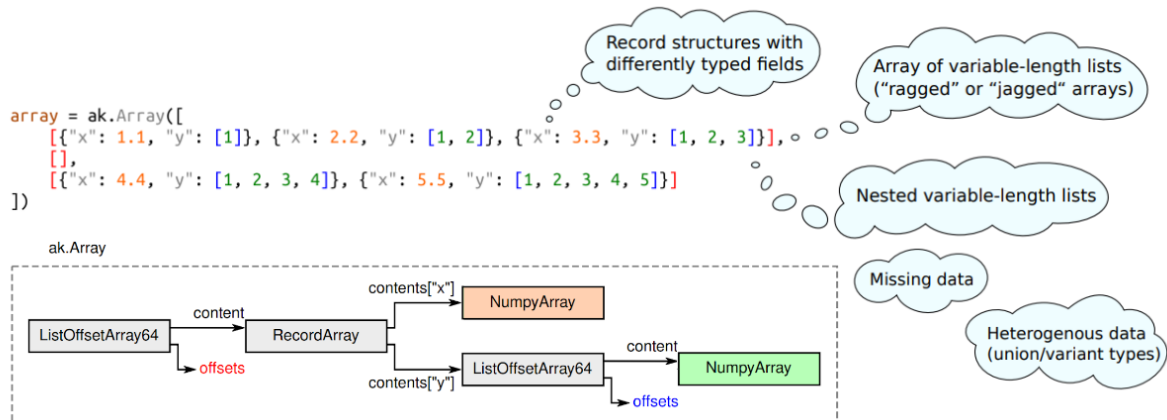
Figure 1. Structure of an Awkward Array with nested variable-length lists and records, color-coded with an array example.

of the best features of both languages so that we can have a Python user interface and, at the same time, take advantage of the performance and memory management of C++. HEP and other scientific communities have extensively invested in the C++ frameworks and are swiftly migrating their data analyses to Python. These communities are particularly interested in bridging the gap between the two languages [8]. This raises an important question: *'How to do Python-C++ integration the right way?'*, which is addressed in the following sections.

3. The 'Header-Only' Approach
A set of header-only C++ libraries has been introduced to address the issues in the Python-C++ integration of Awkward Arrays [7]. These templated C++ libraries are not dependent on any application binary interface. They can be directly included in a project's compilation without the need to link against platform-specific libraries. This 'header-only' approach not only simplifies the production of Awkward Arrays in a project but also enhances the portability of the Awkward Arrays. The code is minimal and does not constitute all of the code required to use Awkward Arrays in Python. It contains no references to Python or Python bindings. The header files can be used by C++ users to create Awkward Arrays, which can then be copied into Python without any specialized data types - only raw buffers, strings, and integers. This approach addresses the issue of packaging projects with native dependencies.

4. LayoutBuilder
A 'layout' consists of composable elements that determine how an array is structured. It can only build a specific view determined by the layout Form. `LayoutBuilder` [9] is a set of compile-time, templated static C++ classes implemented entirely in a header-only library. It uses a header-only `GrowableBuffer` (Figure 2), which is implemented as a linked list with smart pointers. `awkward::LayoutBuilder` specializes an Awkward data structure using C++ templates, which can be filled and converted to a Python Awkward Array through `ak.from_buffers`. The data comes out from LayoutBuilder as a set of named buffers and a JSON [10] Form. The `Form` is a unique description of an Awkward Array and returns a `std::string` that tells Awkward Array how to put everything together. LayoutBuilder is part of an `awkward-cpp` package that is separate from the `awkward` package. Both packages are individually pip-installable. The code does not have helper methods to pass the data to Python, so different projects can use different binding generators. The code relies on generalized lambda expressions to deduce parameter type

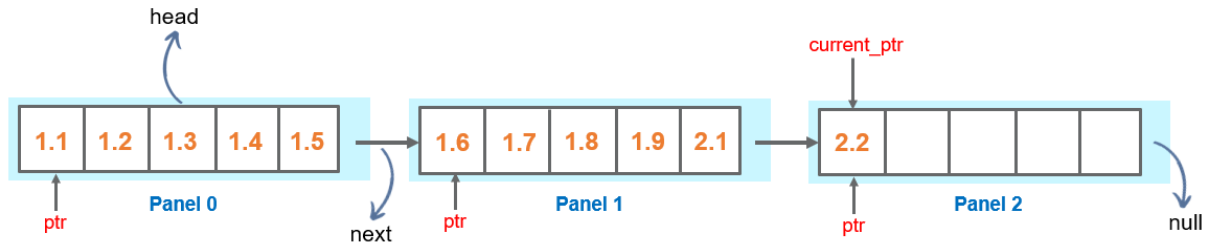during compile time, which is available from the C++14 standard.

Figure 2. Awkward Array GrowableBuffer implemented as a linked list with multiple panels, each of size = 5, that are allocated as needed, i.e., when the GrowableBuffer runs out of space.

ArrayBuilder [11] and LayoutBuilder are both used to create Awkward Arrays. The main difference between a LayoutBuilder and an ArrayBuilder is that the data types that can be appended to the LayoutBuilder are defined in advance, while any data types can be appended to an ArrayBuilder. LayoutBuilder is designed to build Awkward Arrays faster. The flexibility of ArrayBuilder comes with performance limitations since it needs to discover the data type, while LayoutBuilder knows it in advance.

5. User Interface of LayoutBuilder
This section explains the user interface of LayoutBuilder with the help of an example of an Awkward Array with nested records and variable-length lists.

5.1. Phases of LayoutBuilder
There are three phases of using LayoutBuilder:

(i) **Constructing a LayoutBuilder:** from variadic templates (It is an implicit template instantiation).
(ii) **Filling the LayoutBuilder:** while repeatedly walking over the raw pointers.
(iii) **Taking the data to user-allocated buffers:** the user can pass them to Python if needed.

5.2. Illustrative Example
An example of `RecordBuilder` is illustrated in Listing 1. The first step is to include the LayoutBuilder header file (see [9] for the installation instructions). Next, the RecordBuilder is constructed with variadic templates. The contents of a RecordBuilder are heterogeneous type containers (`std::tuple`) that take the other Builders as the template parameters. The field names are non-type template parameters defined by the user. Currently, it is not possible to template on strings as this functionality comes only from C++20 and onwards. Therefore, for passing the field names as template parameters to the RecordBuilder, a user-defined `field_map`, with enumerated type field ID as keys and the field names as value, has to be provided. In the case of multiple RecordBuilder, a user-defined map has to be specified for each of the RecordBuilder used.

After that, the LayoutBuilder buffers are filled with the required data as shown in Listing 1. To make sure there are no errors while filling these buffers, the user can check their validity by using the `is_valid()` method, which can be called on every entry if they want to trade safety for speed. The example translates into the following Awkward Array in Python:

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

```cpp
#include "awkward/LayoutBuilder.h"

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

// Constructing a LayoutBuilder from variadic templates!
RecordBuilder<
    RecordField<Field::x, NumpyBuilder<double>>,
    RecordField<Field::y, ListOffsetBuilder<int64_t, NumpyBuilder<int32_t>>>
> builder(fields_map);

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

// Filling the LayoutBuilder
x_builder.append(1.1);
auto& y_subbuilder = y_builder.begin_list();
y_subbuilder.append(1);
y_builder.end_list();

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

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

Listing 1: Example of a LayoutBuilder with nested records and variable-length lists.

We want NumPy to own the array buffers so that they get deleted when the Awkward Array goes out of Python scope, not when the LayoutBuilder goes out of C++ scope. The hand-off, therefore, needs a few steps:

(i) Retrieve the set of buffer names and their sizes (as a number of bytes).

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

(ii) Allocate memory for these buffers in Python with `np.empty(nbytes, type = np.uint8)` and get `void*` pointers to these buffers by casting the output of `numpy_array.ctypes.data`.

(iii) Let the LayoutBuilder fill these buffers.

```cpp
std::map<std::string, void*> buffers;
builder.to_buffers(buffers);
```

(iv) Finally, JSON Form is generated with:

```cpp
std::string form = builder.form();
```

The Form generated for the example in Listing 1 is shown in Listing 2. Now, everything can be passed over the border from C++ to Python using the `py::buffer_protocol` of `pybind11` [12] for the buffers, as well as an integer for the length and a string for the Form. If the user ever needs to make a change in the format of the records (add, remove, rename, or change the field type), there is no need to change anything in the Python-C++ interface. All of that is contained in the specialization of the C++ template and the filling procedure, which are both in the C++ code.

```
{"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"}
```

Listing 2: Awkward Array Form for the example in Listing 1.

6. Applications

The header-only approach allows for multiple applications in both static and dynamic projects. Awkward RDataFrame [13] uses the C++ header-only libraries to simplify the process of just-in-time (JIT) compilation in `ROOT` [14]. The `ak.from_rdataframe` [15] function converts the selected ROOT RDataFrame [16] columns as native Awkward Arrays. The templated header-only implementation constructs the Form from the primitive data types [17]. The generation of all the types via templates makes it easier to dynamically generate LayoutBuilder from strings in Python and then compile it with `cling` [18].

Another application of header-only LayoutBuilder could be in the `ctapipe` [19] project, which is currently in the planning stage. `ctapipe` is a framework for prototyping the low-level data processing algorithms for the Cherenkov Telescope Array [20]. It does some processing on structured ("awkward") event data, and the developers want to refactor their implementation to use Awkward Arrays. They already have C++ code that iterates over the custom file format, which has array types that are known at compile-time. The easiest way to Awkward Arrays in this project is to use a LayoutBuilder to fill the buffers and then send them to Python through pybind11.

7. Conclusion

The header-only approach presented in this paper facilitates the Python-C++ integration of Awkward Arrays and enhances their portability. A set of templated header-only libraries use only C-types (integers, strings, and raw buffers) to build Awkward Arrays and send them to Python by generating a JSON Form. A standalone awkward header-only C++ package opens up the doors for users to analyze their data in Python. Awkward Arrays can be seamlessly integrated with external projects without linking against platform-specific libraries or worrying

about native dependencies. This new development also allows the extension of the use cases of Awkward Arrays to scientific communities beyond HEP.

8. Acknowledgment

References

[1] Pivarski J, Osborne I, Ifrim I, Schreiner H, Hollands A, Biswas A, Das P, Roy Choudhury S, Smith N and Goyal M 2018 Awkward Array [Computer software] *Zenodo* https://doi.org/10.5281/zenodo.4341376

[2] Rodrigues E 2019 The Scikit-HEP Project *EPJ Web Conf.* **214** 06005 DOI:10.1051/epjconf/201921406005

[3] Harris C R, Millman K J, van der Walt S J et al. 2020 Array programming with NumPy *Nature* **585** 357–362. DOI: 10.1038/s41586-020-2649-2

[4] Pivarski J, Nandi J, Lange D and Elmer P, 2019 Columnar data processing for HEP analysis *EPJ Web Conf.* **214** 06026 DOI: 10.1051/epjconf/201921406026

[5] Pivarski J, Elmer P and Lange D 2020 Awkward Arrays in Python, C++, and Numba *EPJ Web Conf.* **245** 05023 DOI: 10.1051/epjconf/202024505023

[6] Pivarski J, Osborne I, Das P, Biswas A and Elmer P 2020 Awkward Array: JSON-like data, NumPy-like idioms *Proc. of the 19th Python in Science Conf. (SCIPY 2020)* 78-84

[7] Pivarski J 2021 Lessons learned in Python-C++ integration *20th International Workshop on Advanced Computing and Analysis Techniques in Physics Research* https://indi.to/N69ds

[8] Pivarski J, Rodrigues E, Pedro K, Shadura O, Krikler B and Stewart G A 2022 HL-LHC Computing Review Stage 2, Common Software Projects: Data Science Tools for Analysis *[arXiv 2202.02194]* DOI: 10.48550/arXiv.2202.02194

[9] User Guide: How to Use Header-Only LayoutBuilder in C++ *Awkward Array Documentation* https://awkward-array.org/doc/main/user-guide/how-to-use-header-only-layoutbuilder.html

[10] JSON https://www.json.org/

[11] ArrayBuilder https://awkward-array.org/doc/main/reference/generated/ak.ArrayBuilder.html

[12] Jakob W, Rhinelander J and Moldovan D 2016 pybind11 - Seamless operability between C++11 and Python. https://github.com/pybind/pybind11

[13] Osborne I and Pivarski J 2022 Awkward RDataFrame Tutorial *PyHEP 2022 (virtual) Workshop* https://doi.org/10.5281/zenodo.7081586

[14] Brun R, Rademakers F, Canal P, Naumann A, Couet O, Moneta L, Vassilev V, Linev S, Piparo D, GANIS G, Bellenot B, Guiraud E, Amadio G, wverkerke, Mato P, TimurP, Tadel M, wlav, Tejedor E, Blomer J, Gheata A, Hageboeck S, Roiser S, marsupial, Wunsch S, Shadura O, Bose A, CristinaCristescu, Valls X and Isemann R 2019 root-project/root: v6.18/02 (v6-18-02) *Zenodo* https://doi.org/10.5281/zenodo.3895860

[15] from_rdataframe https://awkward-array.org/doc/main/reference/generated/ak.from_rdataframe.html

[16] Piparo D, Canal P, Guiraud E, Pla X V, Ganis G, Amadio G, Naumann A and Tejedor E 2018 RDataFrame: Easy Parallel ROOT Analysis at 100 Threads *EPJ Web Conf.* **214** 06029 DOI: 10.1051/epjconf/201921406029

[17] Osborne I and Pivarski J 2022 Awkward to RDataFrame and back *[arXiv 2302.09860]* DOI: 10.48550/arXiv.2302.09860

[18] Cling [Online]. https://root.cern.ch/cling

[19] ctapipe documentation https://ctapipe.readthedocs.io/en/latest/

[20] ctapipe observatory https://www.cta-observatory.org/