The Ntuple Library

Richard Peschke

University of Hawaii

E-mail: peschke@hawaii.edu

Abstract: This paper introduces **ntuples**, a novel library designed for storing heterogeneous data types within a container object, akin to traditional tuples. Unlike standard tuples, which are limited to element access by index or, if unique, by element type, **ntuples** extend this functionality by enabling access to elements through their designated names. The methodology employed in this library not only enhances data handling but also lays the groundwork for constructing dataframes reminiscent of those found in the Pandas library, with the added feature of accessing each axis by name. This document elaborates on the implementation and capabilities of the **ntuple** library.

Contents

1 Overview

This chapter introduces the fundamental concepts, focusing on the objective of "**ntuple**s" to establish a generic data container for heterogeneous data types, which uniquely enables user access to data elements by name. The code snippet presented in listing 1 illustrates the utilization of the library and the corresponding console output.

In the initial line, the constructor of the **ntuple** class is invoked, followed by the specification of arguments to the constructor in the subsequent lines. Here, property1 and property2 are identifiers established via the "nt_new_field_name" macro, while property3 is "on-the-fly" defined using the "nt_field" macro. Once the **ntuple** is instantiated, its members can be accessed using the names of the properties, as demonstrated in the example below.

As shown in listing 2, each element of an **ntuple** retains its name since it is stored within the element itself; thus, when an element from one **ntuple** is utilized to construct a new **ntuple**, it preserves its original name. However. Additionally, direct access to the underlying element is achievable by invoking the "**v**" member. The conditions under which automatic decay to the underlying type should occur remain a subject of discussion.

Listing 1. Your caption here

```
nt_new_field_name(property1);
nt_new_field_name(property2);

int main(){
    auto nt = ntuple{
        property1 = 123,
        property2("hello_World"),
        nt_field(property3) = 42.123
    };

    std::cout
        << "nt.property1:" << nt.property1 << "\n"
        << "nt.property2:" << nt.property2 << "\n"
        << "nt.property3:" << nt.property3 << "\n";
}

// output
// nt.property1: 123
// nt.property2: hello_World
// nt.property3: 42.123
```

Listing 2. Your caption here

```
  auto nt2 = nt::ntuple(
    nt.property1,
    ax_maker(property4) = nt3.property3.v
  );

  std::cout
  << "nt2.property1:" <<  nt2.property1 << "\n"
  << "nt2.property4:" <<  nt2.property4 << "\n";

  // output
  // nt2.property1: 123
  // nt2.property4: 42.123
```

2 Implementation

2.1 Core Concept

In C++, class extension can be achieved through either composition or inheritance. With composition, the class author assigns a name to the member, and all members of the subclass are accessible solely via this designated name, necessitating an additional layer of indirection. Conversely, in the case of inheritance, members of the base class are directly accessible through the container class, eliminating the need for extra indirection, as the container class effectively extends the subclass. Hence, it becomes evident that the addition of members with a name unknown to the library author can only be accomplished through inheritance.

Listing 3. Your caption here

```

struct property1_ {
 int property1;
}

struct property2_ {
 string property2;
}

struct property3_ {
 double property3;
}

  template <typename... T>
struct proto_ntuple : T...{
    template<typename... Ts>
    proto_ntuple(Ts&&... t1) : Ts(t1)... {}
}

int main(){
    auto nt = proto_ntuple{
        property1_{123},
        property2_{"hello world"},
        property3_{123.45}

    };

    std::cout
    << "nt.property1: " <<  nt.property1 << "\n";

    // output
    // nt.property1: 123
}
```

The listing 3 and figure 1 presents a simplified overview of the **ntuple** library, specifically focusing on the **proto_ntuples** to demonstrate its functionality. The **proto_ntuple** class simply inherits from the classes provided to it through its constructor. Therefore, when objects of types **property1_**, **property2_**, and **property3_** are passed to the constructor, **proto_ntuple** consequently inherits from these classes. By copying the arguments to the respective base classes, the generic container class is instantiated.

While the **proto_ntuple** class is functional and could be deployed in a production environment, there are several aspects where it could be enhanced. A notable limitation is that the individual elements do not retain their name/type association, resulting in the properties being perceived merely as their underlying types to anyone accessing them.

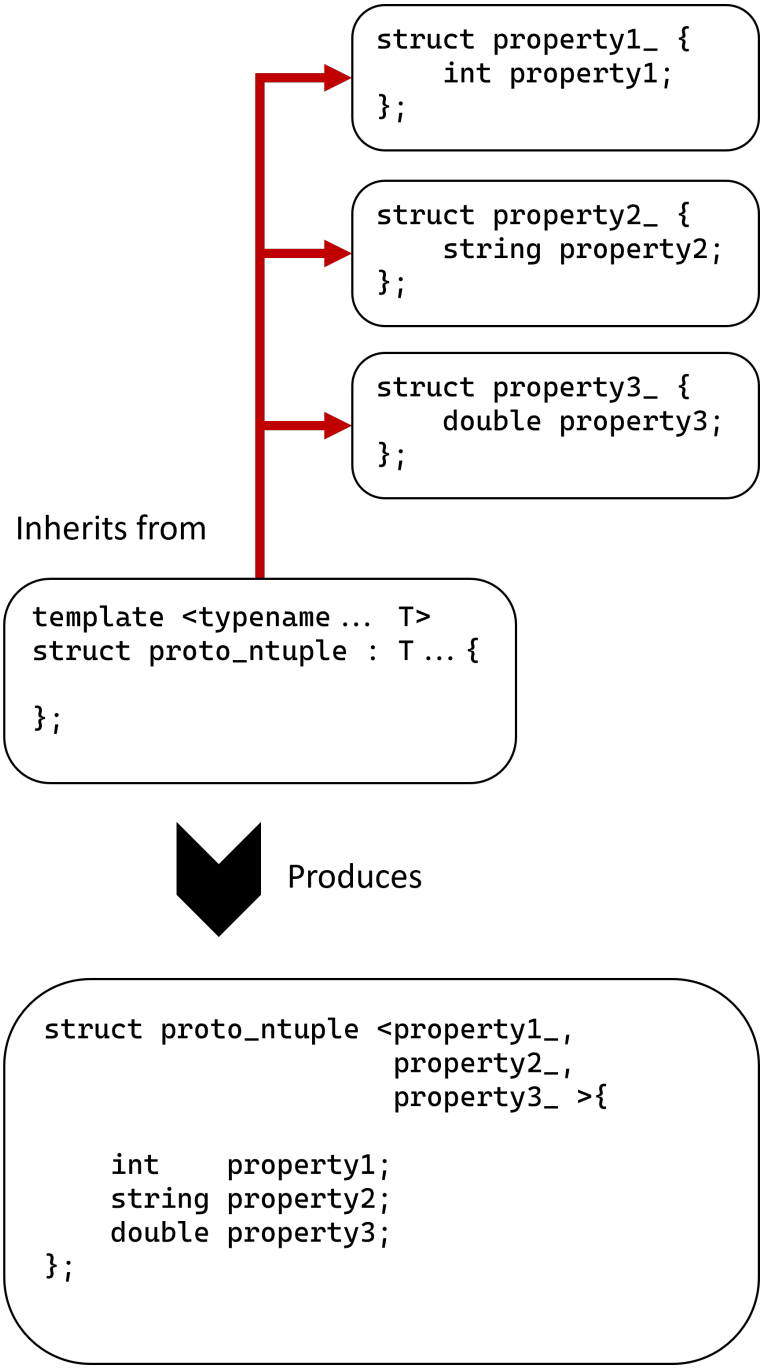

Figure 1. Proto Ntuple

2.2 Creating a new Name

The initial step necessitates the generation of these property classes, which are tasked with storing the name information as previously described. Consequently, their creation must be facilitated by a macro, as only macros possess the capability to utilize the names supplied as arguments.

To facilitate the availability of the name within the program, a macro is required, which the user

Listing 4. Your caption here

```
#define __nt_new_field_core(name_)                    \
struct zt##name_{                                     \
    template <typename T>                             \
    struct base_t {                                   \
        base_t() {}                                   \
        template <typename T1>                        \
        base_t(  T1 &&  e_):                         \
        name_( std::forward<T1>(e_) ) {}              \
        T name_;                                      \
    };                                                \
    static auto get_name() {                          \
      return #name_;                                  \
    }                                                 \
    template <typename T>                             \
    static constexpr auto& get(T& t) {                \
      return  t.name_;                                \
    }                                                 \
  }


#define nt_new_field(name_)                           \
namespace __nt{                                       \
    __nt_new_field_core(name_);                       \
    }                                                 \
static  constexpr inline auto                         \
  name_ = nt::field_name_container<                   \
            __nt::zt##name_                           \
            >{}
```

invokes under the name **nt_new_field**. As demonstrated in listing 4, this macro generates a new type within a concealed **namespace** and subsequently instantiates a compile-time constant object of **field_name_container** with the newly created type as its template argument.

The newly formulated struct encompasses several components, among which is a templated struct named "type_wrap". This struct possesses a single member of type T, designated with a name specified by the macro argument. For instance, if the macro argument is "property1", then this member within the "type_wrap" struct will be titled "property1". This struct represents the locus where the names of the **ntuple** elements are stored.

In addition to the templated struct, the generated struct incorporates three additional members. Firstly, there is the "using base_t" template, which facilitates referencing "type_wrap". Following this is the static "get_name" function, responsible for storing the property's name as a string. Lastly, the struct includes the get function, providing a generic method to access the data member of a container of this type.

The templated struct "type_wrap" serves as the property class from which the **ntuple** class will inherit; being templated, it is capable of containing an arbitrary type. Figure 2 shows the

inheritance diagram of **field_type**. The diagram illustrates how **field_type** first inherits from **field_name_container**, which in turn inherits from the struct zt##name_ created by the nt_new_name macro.

2.3 Storing Names: field_name_container and field_type

Listing 5. Your caption here

```
template <typename Data_T, typename field_name_container_T_>
struct field_type : field_name_container_T_ {
  ...
  Data_T v = {};
  constexpr field_type(Data_T t1) :
  v(std::move(t1)) {}

  operator Data_T() {
    return v;
  }
};

template <typename TBase>
struct field_name_container  : TBase{

  ...

  template <typename T>
  constexpr auto operator=(T t) const {

    return field_type<
    std::remove_cvref_t<T>,
    field_name_container> {
      std::move(t)
    };

  }

};
```

The compile-time constant object created is of type **field_name_container**, with a hidden struct serving as its template argument. By inheriting from this template argument, the **field_name_container** struct extends the functionality of the template. A notable addition provided by this inheritance is the operator= function. This enhancement enables instances of **field_name_container** to be used as named function parameters, as illustrated in Listing 1. In this context, the function captures the input argument and encapsulates it within a newly formed class named **field_type**. The template struct **field_type** requires two parameters: the first is

the type of the object to be stored, and the second is the **field_name_container**. The definitions of **field_name_container** and **field_type** are provided in Listing 5.

The struct **field_type** functions as the container for the individual elements of the **ntuple**, effectively acting as a wrapper around the underlying type T, and it inherits from a struct that holds the name information (**field_name_container**).

2.4 The ntuple Class

The final step in establishing the core functionality is the creation of the **ntuple** struct itself. As shown in Listing 6, the class is relatively concise, consisting primarily of a variadic template that inherits from an objects generated from its template argument. The core of its functionality resides in the **base_maker_t** template, which takes two parameters: the first is a type derived from **field_name_container**, and the second is the data type. This combines the struct holding the name information with the struct holding the data type. As a result, the **ntuple** ultimately inherits from:

```
zt##name_::type_wrap<field_type<T, field_name_container<zt##name_>>>
```

where **T** represents the underlying data type. For example, in the case of **property1** from Listing 1, **T** would be **int**. The identifier **zt##name_** refers to the struct created by the **nt_new_field** macro. For example, calling **nt_new_field(property1)** would create a struct named **ztproperty1**.

The variadic template **ntuple** can be instantiated with any type that provides a nested template named **base_t**. The **base_t** template is the actual base class from which **ntuple** inherits. The helper alias **base_maker_t** simplifies the process of inheriting from **base_t** by extracting and instantiating it. Although **base_maker_t** takes two template parameters, this is more general than strictly required for **ntuple**, and its broader applicability will be illustrated in a later example.

3 In place Name generation

In previous chapters, we demonstrated the process of creating name properties globally using macros. This chapter delves into the creation of names in-place. Unfortunately, C++ lacks the functionality for named function arguments, a feature readily available in languages like Python. Consequently, to make the name accessible, it must be implemented via a macro. Listing 1 exemplifies the on-the-fly creation of a property/field using the **nt_field** macro, which encompasses all necessary components to generate a fully functional property.

3.1 Core Concept

As highlighted in the previous chapter, it is essential to create a class containing a member with the corresponding name, enabling the **ntuple** class to inherit from it. The current objective involves constructing these classes within another function. Utilizing a rarely employed feature in C++, classes can indeed be defined inside functions. However, this approach comes with certain limitations, one of which is particularly relevant for this project: a class defined within a function cannot be a templated class (see Listing 7).

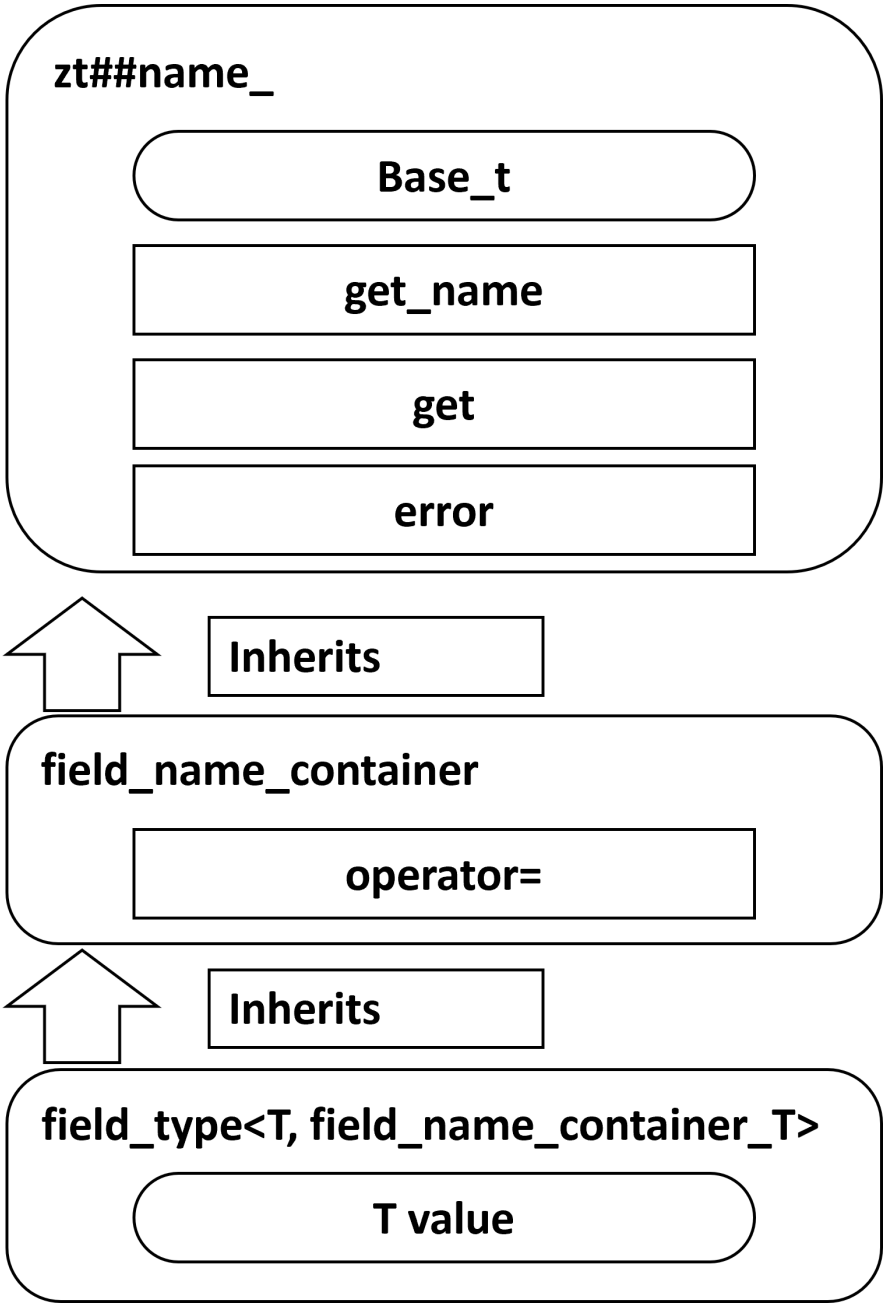

Figure 2. The diagram illustrates the inheritance hierarchy of the field_type wrapper. At the pinnacle is the class produced by the **`nt_new_name`** macro, encapsulating all name-specific code, including the **`Base_t`** class, which facilitates the addition of named members to the **`ntuple`** class. Subsequently, the **`field_name_container`** class inherits from the macro-generated class, augmenting it with additional functionality that is not name-dependent. This class solely holds the name information, and a compile-time constant of this type is utilized to create a new variable embodying this name information. Finally, the field_type class merges the element's name with its actual data type, culminating the hierarchy.

As illustrated in Listing 8, the macro **`proto_field_make`** establishes a lambda function wherein the property class is defined. While the property class itself is not templated, the lambda

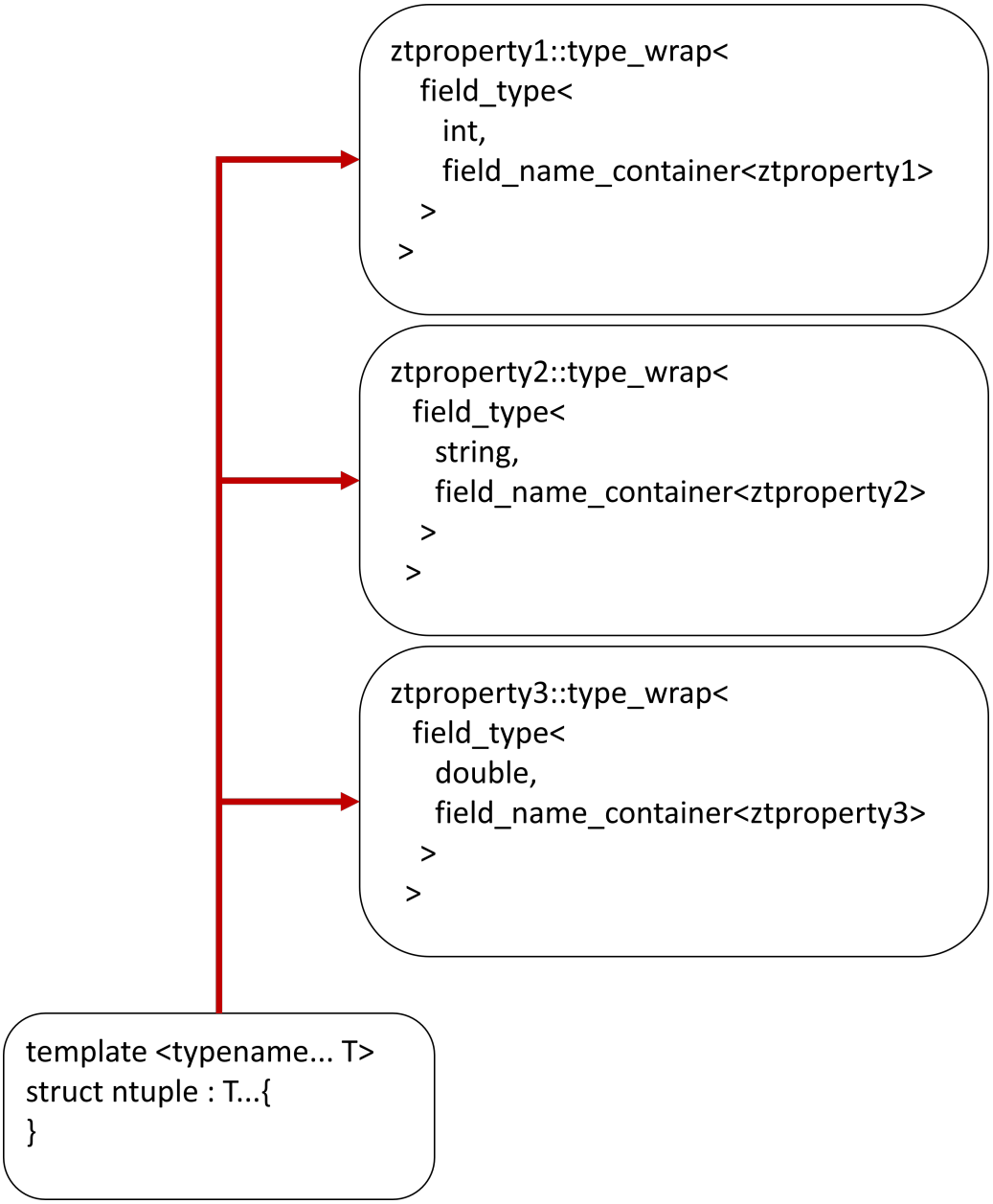

Figure 3.

function is, allowing the property class to adopt various types based on the type used to invoke the lambda. This method enables the creation of properties on the fly for the simplified **`proto_ntuple`** class. However, for the more complex **`ntuple`** class, this approach is overly simplistic, given that the **`lambda_type_wrap`** class is not templated.

3.2 Creating Templates inside of Functions

To adapt the aforementioned approach for use with the **`ntuple`** class, it is necessary to navigate around the restriction of not being able to create templated classes inside functions. As demonstrated

Listing 6. Your caption here

```
template <typename T,
          typename data_T>
struct base_maker
{
  using type =
      typename  T::template
              base_t<data_T >;
};

template <typename T, typename data_T>
using base_maker_t =
  typename base_maker<T, data_T>::type;


template <typename... T>
struct ntuple :
  base_maker_t<
      std::remove_cvref_t<T>, T>... {

  template<typename... Ts>
  ntuple(Ts&&... t1) :
      base_maker_t<
          std::remove_cvref_t<T>, T>(
          std::forward<Ts>(t1)) ...  {}

};
```

Listing 7. Your caption here

```

auto fun() {
    struct fun_struct{
        double fun_struct_member;
    };

    return fun_struct{};
}
```

in the previous chapter, it is feasible to design a templated function that defines a class based on the template parameter. This technique forms the basis of the method to be employed in this chapter. Listing 9 presents a simplified version of the **nt_field** macro. This macro constructs a lambda function, which is executed immediately. Within this lambda, another lambda function is defined, which in turn defines the **Zt##field_name** class and returns it.

The second lambda function is encapsulated within the **field_name_container_base** class, where only the type of the lambda is utilized, and the lambda itself is never actually invoked. This process is depicted in a simplified form in Listing 10 . The **field_name_container_base** class

Listing 8. Your caption here

```

#define proto_field_make(name, value) [](auto x) {\
    struct lambda_member_name_base{\
        decltype(x) name;\
    };\
    return lambda_member_name_base{x};\
}(value)

int main(){
 auto nt = proto_ntuples{
    proto_field_make(property5, 123),
    proto_field_make(property6, 123.12)
 };
}
```

Listing 9. Your caption here

```
  template <typename T1>
  struct type_container {
    using type = T1;
  };


#define nt_field(field_name)  [] () { \
    auto field_name_template_lambda = [](auto e) { \
        struct  Zt##field_name{\
            Zt##field_name(decltype(e)  e_): field_name(e_) {} \
            decltype(e) field_name; \
        }; \
        return Zt##field_name{e};\
    };\
    return  nt::field_name_container_base<\
                nt::field_name_container_base<\
                    decltype(\
                        field_name_template_lambda\
                        )\
                    > \
            >{}; \
    }()
```

includes a using statement:
template <typename Data_T>
using base_t = ...
which serves as the gateway for the **base_maker** previously discussed. This statement functions by extracting the type from the function call of **struct_maker** nested within a **decltype** statement. Consequently, the function call is not executed but is instead employed solely for type determination during compile time.

The function **struct_maker** uses the **field_name_template_lambda** lambda and the type **T** to produce a new **Zt##field_name** class. This is done inside a **decltype** expression, so the lambda is never executed; it is used purely at compile time to determine the type of its return value. The **field_name_container_base** class does not access the lambda itself — it only holds its type, stored in **field_name_template_lambda**. Since the lambda is never instantiated, the function employs **std::declval** to simulate calling it within the **decltype** expression. The simulated call uses a **std::declval** of the template argument, allowing **decltype** to extract the resulting **Zt##field_name** type. An instance of this type is then created. In practice, this function is only ever used inside **decltype** expressions, so none of its contents affect runtime performance.

Listing 10. Your caption here

```
1
2   template <typename field_name_template_lambda_>
3   struct field_name_container_base {
4     using field_name_template_lambda = field_name_template_lambda_;
5
6
7     template <typename T>
8     static constexpr  auto struct_maker() {
9       return    decltype(
10                    std::declval<field_name_template_lambda>()(
11                        std::declval<T>() )
12                ){};
13    }
14
15    template <typename Data_T>
16    using base_t =  decltype(
17                        struct_maker<Data_T>()
18                        );
19
20  };
```

3.3 Detailed Description

The previous chapter introduced the idea of creating class templates from within functions by using lambda expressions. This section explains the exact implementation used in this library.

Here, the name of a field is stored in a **field_name_container<T>** class, where **T** is the base class from which **field_name_container** inherits. For this mechanism to work, **field_name_container** imposes four requirements on its template argument:

1. It must define a templated type **base_t** that can serve as a base class for the **ntuple**.
2. It must provide a **get** function that, given an arbitrary **ntuple**, can retrieve its own type from that **ntuple**.
3. It must provide a **get_name** function that returns its own name as a string.
4. It must define a function to handle errors.

When the **nt_new_field** macro is used, these requirements are met automatically by generating a **zt##name_** class with the necessary members. However, when using the in-place **nt_field** macro, as discussed previously, it is not possible to define a templated class inside a function to satisfy all of these requirements directly. To address this, an intermediate class called **field_name_container_base** is introduced. This class adapts the lambda function provided by **nt_field** into a form compatible with what **field_name_container** expects.

One drawback of using lambda expressions in heavily templated code is that compiler error messages can quickly become overwhelming. With this in mind, consider the most straightforward solution: simply create four separate lambda functions and pass them to the **field_name_container_base** class. This approach works and was, in fact, the initial implementation used in this library. However, the resulting compiler diagnostics were so verbose as to be unreadable.

To address this, a different strategy was chosen: pass only a single lambda function to **field_name_container_base** and let that lambda handle all four operations. As shown in Listing 11, the lambda is divided into four sections using **if constexpr**, with each section implementing one of the required behaviors.

The argument passed to the lambda function is an instance of the **type_wrap** as defined in Listing 11. Here, **N_value** acts as an enumerator identifying which part of the lambda function should be executed.

Listing 11. Your caption here

```
auto field_name_template_lambda = [](auto e) constexpr {

    if constexpr (e.N_value == c_struct_maker)
    {
        ...
    }
    else if constexpr (e.N_value == c_getter)
    {
      ...
    }
    else if constexpr (e.N_value == c_get_name)
    {
      ...
    }
    else if constexpr (e.N_value == c_error)
    {
      ...
    }

};
```

3.3.1 Struct maker

When `N_value` is set to `c_struct_maker`, the lambda returns a `ztb##field_name` class, where the type of its `field_name` member is taken from the type of `val`.

The code that invokes the lambda to extract the `ztb##field_name` type is shown Listing 12. Here, `struct_maker` uses `std::declval` to form a call expression to the stored lambda at compile time, without ever instantiating it. The `decltype` operator then extracts the type returned by the lambda, which becomes the `base_t` type.

3.3.2 getter

The getter function, whose purpose is to retrieve an element of a given type from an `ntuple`, is implemented as shown in Listing 13. It begins by setting the `N_value` of the `type_wrap` argument to `c_getter`. This `type_wrap` instance is then passed to the `field_name_template_lambda`. As described earlier, the lambda body is divided into four `if constexpr` blocks, each corresponding to one of the required operations. With `N_value` set to `c_getter`, only the getter block is instantiated.

Within this block, a class named `getter_t` is defined. This class contains a single static member function, `get`, which takes one argument of the type specified by `e.val` and returns the value of the associated field name. As in previous examples, the lambda itself is never executed; instead, the `getter_t` class is extracted purely at compile time by invoking the lambda inside a `decltype` expression. Once the `getter_t` class has been obtained, its static `get` function is called with the `ntuple` as the argument.

3.3.3 get Name

The `get_name` functionality is implemented in the same way as the getter function, with one key difference: the static member function of the class returned by the lambda takes no arguments. Instead, it simply returns the name of the field. See Listing 14.

3.3.4 Error

Lastly, the purpose of the `error` function is to provide clearer compile-time diagnostics in situations where, for example, a user attempts to extract a field from an `ntuple` that does not contain it. Such cases naturally produce a compile-time error, but the default compiler message can be difficult to interpret. To improve readability, the library wraps this situation in a dedicated check: if the `get` function of a `field_name_container` is called for a field that is not present in the given `ntuple`, a `static_assert` is triggered with a clearer diagnostic message.

Placing this `static_assert` directly in `field_name_container` would, however, lose information about which specific field caused the compilation to fail. To preserve this context, the macro generates a dedicated `error` function for each field type, ensuring that the field name remains available at the point where the `static_assert` is issued. See Listing 15.

4 Examples

4.1 Ranges

One area where the library particularly excels is when used together with the C++20 ranges library. Listing 17 shows an example demonstrating this integration.

Listing 12. Your caption here

```
 //inside nt_field macro
auto field_name_template_lambda = [](auto e) {
...
    if constexpr (e.N_value == c_struct_maker)
    {
        struct Ztb##field_name
        {
          constexpr Ztb##field_name() {}
          constexpr Ztb##field_name(const decltype(e.val) &e_) :
          field_name(e_) {
          }

          constexpr Ztb##field_name(decltype(e.val) &e_) :
          field_name(e_) {
          }
          decltype(e.val) field_name;
        };
        return Ztb##field_name{};
    }

}
    // inside field_name_container_base class
template <typename T, int N>
struct type_wrap
{
    static constexpr int N_value = N;
    T val;
    using type = T;
};

template <typename T>
static constexpr auto struct_maker()
{
    return decltype(
        std::declval<Lambda_T>()(
            std::declval<type_wrap<T, c_struct_maker>>()
        )
    ){};
}

template <typename Data_T>
using base_t = decltype(struct_maker<Data_T>());
```

Listing 17 uses the C++20 ranges library to generate and process a sequence of ntuples in a highly expressive, declarative style. The pipeline begins with **`iota(1, 20)`**, which generates the integers from 1 up to (but not including) 20. The sequence is then passed through a **`transform`** view, which converts each integer **`i`** into an **`ntuple`** containing three fields: **`index`** set to **`i`**, **`index_squared`** set to **`i * i`**, and an in-place **`nt_field(cubed)`** set to **`i * i * i`**.

Listing 13. Your caption here

```
    //inside nt_field macro
auto field_name_template_lambda = [](auto e) {
    ...
        else if constexpr (e.N_value == c_getter)
        {
          struct getter_t
          {
            static constexpr decltype(auto) get(decltype(e.val) x)
            {
                return (x.field_name);
            }
          };
          return getter_t{};
        }
    ..
    }
    // inside field_name_container_base class
    template <typename T>
    static constexpr decltype(auto) get(T &t)
    {
      using getter1 = decltype(
                std::declval<Lambda_T>()(
                        std::declval<type_wrap<T &, c_getter>>()
                        )
                );

      return getter1::get(t);
    }
```

The resulting sequence of ntuples is then filtered using a **`filter`** view, which only passes through those elements whose **`cubed`** value is greater than or equal to 216. Finally, the filtered range is iterated over with a range-based **`for`** loop, and each ntuple is printed to standard output via **`std::cout`**.

4.2 Named Function Arguments

Named parameters improve readability by making the role of each argument explicit, allow arguments to be passed in any order, and simplify handling of optional and required parameters without creating numerous overloads.

The first three lines declare **`argument_1`**, **`argument_2`**, and **`argument_3`** as named parameters. In **`my_function`**, **`bind_args`** sets defaults for **`argument_1`** and **`argument_2`** and marks **`argument_3`** as required. When called, it merges these defaults with the arguments passed by the caller. If a required parameter is missing, a runtime error is thrown.

Listing 14. Your caption here

```
1
2       //inside nt_field macro
3   auto field_name_template_lambda =  [](auto e) {
4       ...
5            else if constexpr (e.N_value == c_get_name)
6            {
7            struct name_getter_t
8               {
9               static constexpr const char* get_name()
10              {
11                return #field_name;
12              }
13             };
14             return name_getter_t{};
15           }
16      }
17      // inside field_name_container_base class
18      static constexpr decltype(auto) get_name()
19      {
20        using name_getter = decltype(
21                               std::declval<Lambda_T>()(
22                                   std::declval<type_wrap<int, c_get_name>>()
23                                   )
24                               );
25
26        return name_getter::get_name();
27      }
```

4.3 The Dataframe Class

As a proof of concept, the library also includes a dataframe class. This raises a fundamental choice in data storage: whether to maintain a collection of vectors, each representing an axis, or to store an array of sets. While the straightforward method of housing **ntuple**s within an std::vector is feasible, it is not the primary focus of this paper. Instead, this paper delves into the development of a generic container that allocates a distinct std::vector for each element, presenting a more intriguing and complex approach to data organization.

Listing 15. Your caption here

```
    //inside nt_field macro
auto field_name_template_lambda =  [](auto e) {
    ...
         else if constexpr (e.N_value == c_error)
         {
           struct has_error    {
                  static constexpr void static_assert_fail(){
                      static_assert(
                            dependent_false<decltype(e)>::value,
                            "[NTUPLE ERROR] Field `"
                            #field_name
                            "` does not exist in this ntuple"
                            );
                  }
            };
          return has_error{};
        }
    }


    // inside field_name_container_base class
    static constexpr decltype(auto) static_assert_fail()
    {
      using has_error = decltype(
                            std::declval<Lambda_T>()(
                                std::declval<type_wrap<int, c_error>>()
                                )
                            );
      has_error::static_assert_fail();
    }
```

Listing 16. Your caption here

```
nt_new_field_name(index);
nt_new_field_name(index_squared);

int main(){
  for (auto &&e : iota(1, 20)
                  | transform([](auto i) {
                      return nt::ntuple(
                        index = i,
                        index_squared = i * i,
                        nt_field(cubed) = i * i * i
                        );
                      })
                  |  filter([](auto &&t) {
                       return t.cubed >= 216;
                     }))
  {
      std::cout << e << std::endl;
  }
}
```

Listing 17. Your caption here

```

nt_new_field_name(argument_1);
nt_new_field_name(argument_2);
nt_new_field_name(argument_3);

template <typename... ARGS>
void my_function(ARGS &&...args)
{
  auto t0 = bind_args(
        argument_1 = 0,
        argument_2 = 15,
        argument_3 = nt::requiered<int>()
      )(args...);

  std::cout << t0 << std::endl;
}

int main() {
  try
  {
    my_function();
  }
  catch (const std::runtime_error &e)
  {
    std::cerr << "Error: " << e.what() << std::endl;
  }

  my_function(4654, argument_2 = 234234, argument_3  = 150);
}
```

Listing 18. Your caption here

```
1
2   template <typename T1, typename T2>
3   struct ax_type2 : T1, T2 {
4     constexpr ax_type2() = default;
5     using struct_maker = T2;
6     using data_t = T1;
7   };
8
9   template <typename... Ts>
10  struct dataframe :
11    nt::base_maker_t<
12            nt::_Remove_cvref_t<Ts>,
13            nt::ax_type2<
14                std::vector<Ts>,
15                typename Ts::struct_maker>
16    >... {
17
18};
```