PXXXXR0: Runtime-Indexed Tuples

Date: 2026-04-17

Project: Programming Language C++

Audience: Library Evolution Working Group (LEWG)

Author: Abdul Muneem

Reply-to: itfllow123@gmail.com

1. Abstract

This proposal provides a new standard library type std::runtime_indexed_tuple that can be indexed at runtime unlike ordinary tuples. It also introduces necessary specializations for std::variant<T&...> and std::optional<T&> to ensure the interface of such a type remains efficient and adheres to the zero-overhead principle.

2. Motivation

Existing std::tuple implementations are optimized only for limiting space usage since they can only be indexed at compile time. The common implementation is a recursive structure that inherits from its previous elements.

To index this using a runtime index, developers currently must implement a switch statement that returns a std::variant containing all possible types:

```
template<typename... T>
std::variant<T...> Get_at_index(std::tuple<T...> list, std::size_t index) {
   switch (index) {
       case 0: return std::get<0>(list);
       case 1: return std::get<1>(list);
       // ... scales poorly for large packs
   }
}
```

2.1 The Need for a New Type

Existing tuples cannot be optimized for runtime indexing without breaking the Application Binary Interface (ABI). Furthermore, switch statements are not guaranteed to be the fastest option for tuples with many elements. This proposal allows implementations to use techniques specifically designed for runtime subscripting efficiency at scale.

3. Proposed Solution

3.1 Basic Design Blueprint

```
template<typename... T>
struct runtime_indexed_tuple<T...> {
   // Requirements: The same as ordinary tuples
   // Provides operator[] returning std::variant<T&...>
};
```

3.2 std::variant<T&...> Specialization

Currently, std::variant cannot hold references. This proposal introduces a specialization for std::variant<T&...> with the following properties:

- No Valueless State: The variant cannot be valueless by exception.
- Immutable Type Selection: Every std::variant<T&...> is constructed with a reference to a type T that it holds for its entire lifetime.
- Assignment Logic: Assignment modifies the underlying value referred to by the variant rather than changing the active type.

3.3 std::optional<T&> Conversion

To extract values safely, we propose a std::optional<T&> that can be explicitly constructed from std::variant<T&...>. The optional will contain a value only if the T& inside the variant matches the T& passed as the template argument.

4. Technical Specifications

4.1 Constexpr and Noexcept

The subscript operator for the runtime_indexed_tuple, as well as constructors for the proposed variant and optional specializations, shall be constexpr and noexcept.

4.2 Handling Duplicate Types

In cases where duplicate types exist, the return type of the subscript operator can be defined using a meta-function to ensure all unique types from the original pack are represented, preventing information loss.

4.2 Example of possible Optimizations

Reference implementation illustrating potential optimizations and O(1) dispatch using current C++ standards:

```
#include <iostream>

#include <type_traits>

#include <variant>

#include <functional>
```

```
#include <array>



template<template<std::size_t N, typename...> class F, std::size_t N, typename
... Args>

using Delay = F<N, Args...>;//credits to bjarne stroustrup's c++ book from 2013
for this one

template< std::size_t N, typename Head, typename ...Types>

struct get_variant_of_unique_ts :

    std::conditional_t<

    (std::is_same_v<Head, Types> || ...),

    Delay<get_variant_of_unique_ts, N - 1, Types... >,

    Delay<get_variant_of_unique_ts, N - 1, Head, Types... >

    >




{

    using intermediate =

        std::conditional_t< (std::is_same_v<Head, Types> || ...),

        Delay<get_variant_of_unique_ts, N - 1, Types... >,

        Delay<get_variant_of_unique_ts, N - 1, Head, Types... >

        >;

    using Variant_t = typename  intermediate::Variant_t;

};


template<typename Head, typename... Types>

struct get_variant_of_unique_ts<0, Head, Types...>

{

    using Variant_t = std::variant<Head*, Types*...>;

};

template<typename complex_elem, typename ... Types>
```

```
struct reference_holding_arrays {

    std::array<complex_elem, (sizeof...(Types))> book_keeper;

    template<typename ... Types_>

    constexpr reference_holding_arrays(Types_&... b) : book_keeper{
complex_elem { &b }... } {}

    constexpr complex_elem operator[](size_t index) {

        return book_keeper[index];

    }

};

template<typename ... Types>

struct tuple {



    std::tuple<Types...> data;

    using complex_elem = typename  get_variant_of_unique_ts<sizeof...(Types) -
1, Types...>::Variant_t;

    using Book_keeping_t = reference_holding_arrays<complex_elem, Types...>;

    Book_keeping_t book_keeper;

    template<typename ... Types_>

    constexpr tuple(Types_&&... b) :data{ std::forward<Types_>(b)... },

        book_keeper{

        std::apply([](auto&... args) {

        return Book_keeping_t{args...};

        }, data)



    }

    {

    }




    constexpr complex_elem operator[](size_t index) {

        return book_keeper[index];
```

```
    }
    ~tuple() = default;
};
template<typename Head>
struct tuple<Head> {
    Head head;
    constexpr tuple(Head&& a) : head{ a } {}
    constexpr tuple(Head& a) : head{ a } {}
    constexpr std::variant<Head*> operator[](size_t index) {
        return std::variant<Head*>{ &head };
    }
    ~tuple() = default;

};



int main() {
    tuple<int, double, float, int> list{ 1, 2.0, 3.0f,3 };
    std::cout << *std::get<2>(list[0]) << std::endl;
    std::cout << *std::get<0>(list[1]) << std::endl;
    std::cout << *std::get<1>(list[2]) << std::endl;
  //The reason the get<N> is shuffled in my usage is because while trying to
remove duplicates, the exact order is lost. This is not an issue,since, the
  //usage of runtime tuples should be done using std::visit
    return 0;
}
```

5. Summary

A standardized interface for runtime-indexed tuples prevents developers from reinventing inefficient wheels. By providing a specialized layout, implementations can optimize for runtime indexing without violating the zero-overhead principle or breaking ABI boundaries.

6. Acknowledgements

Special thanks to Simon Schröder for professional feedback and technical inquiries. Additional credits to Sebastian Wittmeier, Bjorn Reese, Thiago Macieira, Marcin Jaczewski, Andre Kostur, Weinrich Steve, Jason McKesson, Adrian Johnston, and David Brown.