A true heterogeneous container in C++

may 3, 2017september 4, 2018 • andy g • c++, c++14, c++17, heterogeneous container, variable templates, visitor pattern

```
heterogeneous_container c;
c.push_back('a'); // char
c.push_back(1); // int
c.push_back(2.0); // double
c.push_back(3); // another int
c.push_back(std::string{"foo"}); // a string

// print it all
c.visit(print_visitor{}); // prints "a 1 2 3 foo"
```

(https://gieseanw.wordpress.com/2017/05/03/a-true-heterogeneous-container-in-c/)

Oftentimes I see questions StackOverflow asking something to the effect of

“Can I have a std::vector that holds more than one type?”

The canonical, final, never-going-to-change answer to this question is a thorough

“No”

C++ is a statically-typed (https://en.wikipedia.org/wiki/Type_system#Static_type_checking) language. A vector will hold an object of a single type, and only a single type.

“But…”

Of course there are ways to work around this. You can hide types within types! In this post I will discuss the existing popular workarounds to the problem, as well as describe my own radical new heterogeneous container that has a much simpler interface from a client’s perspective.

boost::variant (http://www.boost.org/doc/libs/1_61_0/doc/html/variant.html), for one,(and now std::variant (http://en.cppreference.com/w/cpp/utility/variant)) has allowed us to specify a type-safe union (http://en.cppreference.com/w/cpp/language/union) that holds a flag to indicate which type is “active”.

The downside to variant data structures is that you’re forced to specify a list of allowed types ahead of time, e.g.,

```
std::variant<int, double, MyType> myVariant;
```

To handle a type that could hold “any” type, Boost then created boost::any (http://www.boost.org/doc/libs/1_61_0/doc/html/any.html)(and with C++17 we’ll get std::any (http://en.cppreference.com/w/cpp/utility/any))

This construct allows us to really, actually hold any type. The downside here is that now the client generally has to track which type is held inside. You also pay a bit of overhead in the underlying polymorphism that comes with it, as well as the cost of an “any_cast (http://en.cppreference.com/w/cpp/utility/any/any_cast)” to get your typed object back out.

A visitor pattern

The real “but…” portion of the answer to these StackOverflow questions will state that you *could* use polymorphism to create a common base class for any type that your vector will hold, but that is ridiculous, especially if you want to hold primitive types like int and double. Assuming your design isn’t totally off-base, what you really wanted to do was use a std::variant or std::any and apply the Visitor Pattern (https://en.wikipedia.org/wiki/Visitor_pattern) to process it.

The visitor pattern implementation for this scenario basically works like this:

- Create a "callable" (a class with an overloaded or templated function call operator (http://en.cppreference.com/w/cpp/language/operators), or a polymorphic lambda (https://isocpp.org/wiki/faq/cpp14-language#generic-lambdas)) that can be called for any of the types.
- "Visit" the collection, invoking your callable for each element within.

I'll demonstrate using std::variant.

First we start by creating our variant:

```
std::variant<int, double, std::string> myVariant;
myVariant = 1; // initially it's an integer
```

Then we write our callable (our visitor):

```
struct MyVisitor
{
   void operator()(int& _in){_in += _in;}
   void operator()(double& _in){_in += _in;}
   void operator()(std::string& _in){_in += _in;}
};
```

This visitor will double the element that it visits. That is, "1" shall become "2", or "MyString" shall become "MyStringMyString".

Next we invoke our visitor using std::visit (http://en.cppreference.com/w/cpp/utility/variant/visit):

```
std::visit(MyVisitor{}, myVariant);
```

That's it! You'll notice that all the code for each type in MyVisitor is identical, so we could replace it all with a template. Let's go with that template idea and create a visitor that will print the active element:

```
struct PrintVisitor
{
   template <class T>
   void operator()(T&& _in){std::cout << _in;}
};
```

Or we could similarly have created an identical polymorphic lambda like so:

```
auto lambdaPrintVisitor = [](auto&& _in){std::cout << _in;};
```

Applying our PrintVisitor or lambdaPrintVisitor will have the same effect:

```
std::visit(PrintVisitor{}, myVariant); // will print "2"
std::visit(lambdaPrintVisitor, myVariant); // will also print "2"
```

And for a string:

```
myVariant = "foo";
std::visit(MyVisitor{}, myVariant); // doubles to "foofoo"
std::visit(lambdaPrintVisitor, myVariant); // prints "foofoo"
```

Here's a working demo on Wandbox (https://wandbox.org/permlink/J5Omu5nGxaZ1Gb3z) that demonstrates the above code snippets.

Now a "heterogeneous" container with a variant

Moving from a single variant to a collection of variants is pretty easy — you shove the variant into a std::vector:

```
std::vector<std::variant<int, double, std::string>> variantCollection;
variantCollection.emplace_back(1);
variantCollection.emplace_back(2.2);
variantCollection.emplace_back("foo");
```

And now to visit the collection, we apply our visitors to each element in the collection:

```
// print them
for (const auto& nextVariant : variantCollection)
{
    std::visit(lambdaPrintVisitor, nextVariant);
    std::cout << " ";
}
std::cout << std::endl;

// double them
for(auto& nextVariant : variantCollection)
{
    std::visit(MyVisitor{}, nextVariant);
}

// print again
for (const auto& nextVariant : variantCollection)
{
    std::visit(lambdaPrintVisitor, nextVariant);
    std::cout << " ";
}
    std::cout << std::endl;
```

Here's a live demo of that code. (https://wandbox.org/permlink/Zap1iKCpVIwzzIIp)

Now we begin to see how tedious it is to write those loops manually, so we encapsulate the vector into a class with a visit mechanism. We make it a template so that the client can specify the underlying types that go into the variant:

```
template <class T>
struct VariantContainer
{
    template <class V>
    void visit(V&& visitor)
    {
        for (auto& object : objects)
        {
            std::visit(visitor, object);
        }
    }
    using value_type = std::variant;
    std::vector<value_type> objects;
};
```

And then our code from above to visit it is nicely shortened to:

```
VariantContainer<int, double, std::string> variantCollection;
variantCollection.objects.emplace_back(1);
variantCollection.objects.emplace_back(2.2);
variantCollection.objects.emplace_back("foo");

// print them
variantCollection.visit(lambdaPrintVisitor);
std::cout << std::endl;

// double them
variantCollection.visit(MyVisitor{});

// print again
variantCollection.visit(lambdaPrintVisitor);
std::cout << std::endl;
```

Live code demonstration (https://wandbox.org/permlink/Byaes0CvdRpoWsGa)

At this point we're thinking we are pretty smart. The client can simply direct us on how heterogeneous they want that container to be! And then we start wrapping up the vector a little nicer, hiding the underlying storage, and replicating the rest of std::vector's interface so someone can call "emplace_back" directly on our container etc, and we're rolling!

But what if we could do this instead

```
heterogeneous_container c;
c.push_back('a'); // char
c.push_back(1); // int
c.push_back(2.0); // double
c.push_back(3); // another int
c.push_back(std::string{"foo"}); // a string

// print it all
c.visit(print_visitor{}); // prints "a 1 2 3 foo"
```

“Impossible”

A C++ programmer would tell you. “Unless you’re doing a bunch of very expensive cast-and checks somewhere” (e.g. with std::any)

But I will now demonstrate how such an interface is possible (without RTTI (https://en.wikipedia.org/wiki/Run-time_type_information)) using C++14 and C++17 features (the C++17 features are not necessary, just nice to haves).

(***__Authors note:__ The following is intended as a toy, not to be used in any real implementation. It has a gaping security hole in it. Think of this more as an exercise in what we can do with C++14 and C++17 ***)

A Heterogeneous Container in C++

Now, admittedly. We can never be as flexible as a duck-typed language such as Python. We cannot create new types at runtime and add them to our container, and we can’t easily iterate over the container; we must still use a **visitor pattern**.

Let’s begin with a feature that was added in C++14: **Variable templates**

Variable templates

If you’ve done any templating in C++ before, you’re familiar with the typical syntax to template a function so that it can operate on many types:

```
template <class T>
T Add(const T& _left, const T& _right)
{
    return _left + _right;
}
```

But variable templates (http://en.cppreference.com/w/cpp/language/variable_template) allow us to interpret a *variable* differently depending on a type. Here’s an example where we can interpret the mathematical constant π (pi) differently:

```
template<class T>
constexpr T pi = T(3.1415926535897932385);
```

And now we can explicitly refer to pi<double> or pi<float> to more easily express the amount of precision we need.

Let's abuse variable templates

Recall that when you instantiate a template, you are really telling the compiler to copy-paste the template code, substituting for each type. Variable templates are the same way. That is, "pi<double>" and "pi<float>" are two separate variables.

What happens when we move a variable template into a class?

Well, the rules of C++ dictate that such a variable template becomes static (http://en.cppreference.com/w/cpp/language/static), so any instantiation of the template will create a new member across all class instances. But with our heterogeneous container we want instances to only know or care about the types that have been used *for that specific instance*! So we abuse the language and create a mapping of container pointers to vectors:

```
namespace andyg{
struct heterogeneous_container{
private:
    template<class T>
    static std::unordered_map<const heterogeneous_container*, std::vector<T>> iter
public:
    template <class T>
    void push_back(const T& _t)
    {
        items<T>[this].push_back(_t);
    }
};

// storage for our static members
template<class T>
std::unordered_map<const heterogeneous_container*, std::vector<T>> heterogeneous_c
} // andyg namespace
```

And now suddenly we have a class which we can add members to *after* creating an instance of! We can even declare a struct later and add that in, too:

```
andyg::heterogeneous_container c;
c.push_back(1);
c.push_back(2.f);
c.push_back('c');
struct LocalStruct{};
c.push_back(LocalStruct{});
```

Destruction

There are quite a few shortcomings we still need to address first before our container is really useful in any way. One of which is the fact that when an instance of andyg::heterogeneous_container goes out of scope, all of its data still remains within the static map.

To address this, we will need to somehow track which types we received, and delete the appropriate vectors. Fortunately we can write a lambda to do this and store it in a std::function. Let's augment our push_back function:

```
template<class T>
void push_back(const T& _t)
{
    // don't have it yet, so create functions for destroying
    if (items<T>.find(this) == std::end(items<T>))
    {
        clear_functions.emplace_back(
            [](heterogeneous_container& _c){items<T>.erase(&_c);});
    }
    items<T>[this].push_back(_t);
}
```

Where "clear_functions" becomes a local member of the class that looks like this:

```
std::vector<std::function<void(heterogeneous_container&)>> clear_functions;
```

Whenever we want to destroy all elements of a given andyg::heterogeneous_container, we can call all of its clear_functions. So now our class can look like this:

```
struct heterogeneous_container
{
    public:
    heterogeneous_container() = default;

    template<class T>
    void push_back(const T& _t)
    {
        // don't have it yet, so create functions for printing, copying, moving, a
        if (items<T>.find(this) == std::end(items<T>))
        {
            clear_functions.emplace_back([](heterogeneous_container& _c){items<T>
        }
        items<T>[this].push_back(_t);
    }

    void clear()
    {
        for (auto&& clear_func : clear_functions)
        {
            clear_func(*this);
        }
    }

    ~heterogeneous_container()
    {
        clear();
    }

    private:
    template<class T>
    static std::unordered_map<const heterogeneous_container*, std::vector<T>> iter
    std::vector<std::function<void(heterogeneous_container&)>> clear_functions;
};

template<class T>
std::unordered_map<const heterogeneous_container*, std::vector<T>> heterogeneous_c
} // andyg namespace
```

Copying

Our class is starting to become pretty useful, but we still have issues with copying. We'd have some pretty disastrous results if we tried this:

```
andyg::heterogeneous_container c;
c.push_back(1);
// more push_back
{
   andyg::heterogeneous_container c2 = c;
}
```

The solution is fairly straightforward; we follow the pattern as when we implemented "clear", with some additional work to be done for a copy constructor and copy assignment operator. On push_back, we'll create another function that can copy a vector<T> from one heterogeneous_container to another, and in copy construction/assignment, we'll call each of our copy functions:

```
struct heterogeneous_container
{
    public:
    heterogeneous_container() = default;
    heterogeneous_container(const heterogeneous_container& _other)
    {
       *this = _other;
    }

    heterogeneous_container& operator=(const heterogeneous_container& _other)
    {
        clear();
        clear_functions = _other.clear_functions;
        copy_functions = _other.copy_functions;

        for (auto&& copy_function : copy_functions)
        {
            copy_function(_other, *this);
        }
        return *this;
    }

    template<class T>
    void push_back(const T& _t)
    {
        // don't have it yet, so create functions for copying and destroying
        if (items<T>.find(this) == std::end(items<T>))
        {
            clear_functions.emplace_back([](heterogeneous_container& _c){items<T>

            // if someone copies me, they need to call each copy_function and pass
            copy_functions.emplace_back([](const heterogeneous_container& _from, h
            {
                items<T>[&_to] = items<T>[&_from];
            });
        }
        items<T>[this].push_back(_t);
    }

    void clear()
    {
        for (auto&& clear_func : clear_functions)
        {
            clear_func(*this);
        }
    }
```

```
    ~heterogeneous_container()
    {
        clear();
    }

    private:
    template<class T>
    static std::unordered_map<const heterogeneous_container*, std::vector<T>> iter

    std::vector<std::function<void(heterogeneous_container&)>> clear_functions;
    std::vector<std::function<void(const heterogeneous_container&, heterogeneous_c

};

template<class T>
std::unordered_map<const heterogeneous_container*, std::vector<T>> heterogeneous_c
} // andyg namespace
```

And now our container is starting to become useful. What other function do you think you could implement to follow the pattern we established with “clear” and “copy”? Perhaps a “size” function? A “number_of<T>” function? A “gather<T>” function to gather all elements of type T for that class?

Visiting

We can really do anything with our andyg::heterogeneous_container yet, because we cannot iterate over it. We need a way to *visit* the container so that useful computation can be performed.

Unfortunately we cannot do it as easy as calling std::visit on a std::variant. Why? Because a std::variant implicitly advertises the types it holds within. Our andyg::heterogeneous_container does not (and cannot). So we need to push this advertisement onto the visitor. That is, the *visitor* will advertise the types it is capable of visiting. Of course this would be a pain point to using an andyg::heterogeneous_container, but alas, tradeoffs must be made.

A visitor that advertises the types it may visit

So how do we make it easy for the client to write a visitor? We can use some lightweight inheritance to automatically provide client visitors a way to publish the types they can visit.

First we’ll start with a generic “type list” kind of class that takes advantage of a the variadic template (https://en.wikipedia.org/wiki/Variadic_template#C.2B.2B) feature that arrived with C++11:

```
template<class...>
struct type_list{};
```

And then we'll create a templated base class for visitors that defines a type_list:

```
template<class... TYPES>
struct visitor_base
{
    using types = andy::type_list<TYPES...>;
};
```

How can we use this base class? Let's demonstrate with an example.

Say I have some heterogeneous container:

```
andyg::heterogeneous_container c;
c.push_back(1); // int
c.push_back(2.2); // double
```

and now I want to visit this class so that I can double the members. Similar to how we wrote a visitor class for std::variant, we'll write a structure that overloads the function call operator for each type. This structure should inherit from our "visitor_base"

```
struct my_visitor : andyg::visitor_base<int, double>
{
    void operator()(int& _i)
    {
        _i+=_i;
    }

    void operator()(double& _d)
    {
       _d+=2.0;
    }
};
```

And now our visitor implicitly defines a type called "types" that is templated on "int" and "double".

We could rewrite our visitor with a template instead, like before:

```
struct my_visitor : andyg::visitor_base<int, double>
{
    template<class T>
    void operator()(T& _in)
    {
        _in +=_in;
    }
};
```

Although you wouldn't be allowed to declare this struct locally within a function on account of the template.

Next up: writing the "visit" method of the andy::heterogeneous_container class

visit()

Like we did for our "VariantCollection" above, the visitor pattern basically amounts to calling std::visit for each element in the container. Problem is, std::visit won't work here so we have to implement it ourself.

Our strategy will be to use the types published by the visitor class and invoke the function call operator for each type. This is easy if we use some helper functions. But first, the main entry point, "visit()":

```
template<class T>
void visit(T&& visitor)
{
    visit_impl(visitor, typename std::decay_t<T>::types{});
}
```

Note that I don't really constraint the template at all. So long as T has a type named "types", I use it. In this way, one wouldn't be constrained to use an andyg::visitor_base.

Next you'll notice that the call to "visit_impl" not just passes on the "visitor" received in "visit", but it additionally tries to *construct an instance* of "T::types". Why? Well the reasoning is similar to the reasoning behind tag dispatching (http://www.boost.org/community/generic_programming.html#tag_dispatching), but not quite. We simply want an easy way to pass our typelist ("types") as a template parameter to "visit_impl".

I think it's better explained if we look at the declaration for "visit_impl":

```
template<class T, template<class...> class TLIST, class... TYPES>
void visit_impl(T&& visitor, TLIST<TYPES...>)
```

Like before, we receive our visitor as “T”, but then we use a template template argument to indicate that the incoming “types” object itself is templated, and that those types it is templated on can be referred to as “TYPES”. When we call “visit” with the “my_visitor” defined above, the resulting type substitution will appear as:

```
void visit_impl(my_visitor& visitor, andyg::type_list<int, double>)
```

The second argument is unnamed, indicating that it is unused and really only there to help us out in our metaprogramming. An optimizing compiler should be able to optimize out any real construction of the type, but even if it doesn’t we only lose 1 byte because the class itself is empty.

“Iterating” over the types

Essentially we want to say “for each type in TYPES, iterate over the associated vector<type> and visit each element”. However C++ doesn’t allow “iteration” over types. In the past we’ve been forced to use tail recursion — receive a parameter pack like “<class HEAD, class… TAIL>” process “HEAD”, then recurse for “TAIL”, eventually reaching a base case of a single type. Such expansion creates a lot of work for the compiler, and the separation of functions makes the code that much harder to read.

An alternative that was done in C++11 and C++14 was called “simple expansion” and involved abusing the comma operator (https://en.wikipedia.org/wiki/Comma_operator), and placing a function call into an array initializer list, for which the array was cast to void so that the compiler would not actually allocate. It sounds complicated because it is. It would have looked like this:

```
using swallow = int[];
(void)swallow{0, (void(visit_impl_help<T>(visitor), 0)...};
```

Fortunately for us, C++17 made such hackery redundant with the introduction of fold expressions (http://en.cppreference.com/w/cpp/language/fold), which, for simplicity’s sake you can imagine as calling a single function for each type in a parameter pack. The syntax does take a little getting used to, but here’s what the implementation of “visit_impl” looks like:

```
template<class T, template<class...> class TLIST, class... TYPES>
void visit_impl(T&& visitor, TLIST<TYPES...>)
{
   (..., visit_impl_help<std::decay_t<T>, TYPES>(visitor));
}
```

This is formally called a unary left fold and what happens is that the resulting expression for our "my_visitor" will cause it to be expanded as such:

```
visit_impl_help<T, int>(visitor), visit_impl_help<T, double>;
```

This approach also abuses the comma operator. You can see that the introduction of more types in "TYPES" would result in more commas and expressions between those. The result is a function call for each type.

You'll again notice that I'm calling *yet another helper* and I promise it's the last one! I wanted a clean looking function to iterate over a vector. Here's its definition:

```
template<class T, class U>
void visit_impl_help(T& visitor)
{
    for (auto&& element : items<U>[this])
    {
        visitor(element);
    }
}
```

By overloading the function call operator in our visitor class, we can treat "visitor" itself as a function here and simply call it for each element. That's pretty neat! In the final demo code, I additionally added a static_assert (http://en.cppreference.com/w/cpp/language/static_assert) (which itself uses some complicated metaprogramming to detect (http://en.cppreference.com/w/cpp/experimental/is_detected) a proper overloaded function call operator) to visit_impl_help so that clients don't get stuck in template error hell if they miswrote their visitor class.

Putting it all together

At this point we can basically do anything we want with the class, creation, copying, destroying, assignment, and visiting (everything else is gold plating (https://en.wikipedia.org/wiki/Gold_plating)). We can even declare a new class after creating our container instance, and then add an instance of that new class into the container. Whoa.

Here's a sample code run:

```
// my_visitor defined as above, and print_visitor pretty obvious
auto print_container = [](andyg::heterogeneous_container& _in){_in.visit(print_vis
andyg::heterogeneous_container c;
c.push_back('a');
c.push_back(1);
c.push_back(2.0);
c.push_back(3);
c.push_back(std::string{"foo"});
std::cout << "c: ";
print_container(c);
andyg::heterogeneous_container c2 = c;
std::cout << "c2: ";
print_container(c2);
c.clear();
std::cout << "c after clearing c: ";
c.visit(print_visitor{});
std::cout << std::endl;
std::cout << "c2 after clearing c: ";
print_container(c2);
c = c2;
std::cout << "c after assignment to c2: ";
print_container(c);
my_visitor v;
std::cout << "Visiting c (should double ints and doubles)\n";
c.visit(v);
std::cout << "c: ";
print_container(c);
struct SomeRandomNewStruct{};
c.push_back(SomeRandomNewStruct{});
```

And its output:

```
c: 1 3 2 a foo
c2: 1 3 2 a foo
c after clearing c:
c2 after clearing c: 1 3 2 a foo
c after assignment to c2: 1 3 2 a foo
Visiting c (should double ints and doubles)
c: 2 6 4 a foo
```

And there you have it. Here's a live running demo on WandBox (https://wandbox.org/permlink/oXJ1MG7vBV548GZ4). The live demo includes some additional tests and exercise of some "nice-to-have" features of a heterogeneous container. (For you experts, of course the templating is simplified and in a production environment there should be better forwarding semantics, static_asserts, and flexibility in the templating).

A final note on performance

The primary difference in storage between an andyg::heterogeneous_container and a std::vector<std::any> is that, in an andyg::heterogneous_container, all elements of the same type are stored contiguously. This allows extremely fast iteration as compared to a std::vector<std::any>, which gets bogged down by a number of try-catches during visitation. By "extremely fast", I mean that it's actually an order of magnitude faster. Run the timings yourself here (https://wandbox.org/permlink/GEXJuWpR5MYO1T0t).

Conclusions

C++14 and C++17 offer us some pretty powerful new metaprogramming tools in the form of template variables, variadic lambdas, standardized variants, and fold expressions, among others. By experimenting with these we can begin pushing the boundaries of what we thought was possible in C++. In this post we created a new kind of visitor pattern that has very nice syntax and won't be slowed down by any run-time type inference (all the visiting knows exactly which types it's iterating over already), but also has some drawbacks (the least of which is the gaping security hole where one andyg::heterogeneous_container can see the contents of *any other* andyg::heterogeneous_container).

Even though I consider the container I just wrote to be an incomplete plaything that is only interesting only for the concepts it demonstrates, depending on your use case you might actually want to "borrow" it to satisfy yet another impossible requirement your customer gives you.