1 Introduction

Note: This assignment is harder than what we can expect from an exam assignment. If you manage to solve this exercise without looking at the given solution, then your chances are *very good* for the final exam.

In this assignment we will create a (somewhat simplified) variant class using the techniques discussed during the seminar. Our implementation will be similar to `std::variant` but not exactly the same.

Recall what we discussed during the "Sum Type" seminar. Specifically:

- To store arbitrary objects we need to create an array of bytes (`char`) with the same size as the *largest type* that can be stored in the variant.
- We need to keep track (during runtime) of what type is currently stored in the variant. In the seminar we used an enum for this, but in the context of this exercise we cannot do that since the list of types is not fixed. A solution for this is proposed further down.
- Whenever we *change* what type is stored, we need to call the destructor of the old object and then use *placement new* to construct a new object inside our array (this means we have to include `<new>`).
- When we want to access our initialized object we have to use `reinterpret_cast` and `std::launder` (which will work *only* if we access the type currently stored in the variant).

This assignment will be divided into several parts. In each part you will be asked to solve a specific problem, and hopefully at the end you will have a working `Variant` class.

2 The assignment

Create a class template `Variant` that takes a variadic parameter pack `Types`. The variadic pack will contain all the types that should be possible to store in the variant. This means the following types should all be legal:

```
Variant<int>
Variant<float, bool>
Variant<std::string, Variant<int>>
Variant<int, bool, float>
```

and so on...

The first thing we need to realize is that this class must store an array of `char` which has the same size as the *largest* type in the `Types` pack. But in order to do this we need a way to calculate (during compile-time) the size of the largest type...

Part I - Find the largest type

In this part we will find the largest type inside a variadic pack. A suggestion for how to this is the following:

Create a struct template called `Variant_Helper` that takes a variadic template parameter called `Types`. The primary implementation of this struct is going to be empty. We will instead define the whole struct with specializations (this way we can utilize variadic recursion). For the variadic recursion to work we need two cases:

1. One specialization were we extract the first type into `Type` and the rest into `Types`.
2. One specialization for when `Types` is empty.

Inside these specializations we will create a static constexpr `std::size_t` variable called `size`. In the base case (i.e. the second specialization) this variable is set to `0`. In the recursive specialization define it according to this relationship:

```
size = max(sizeof(Type), largest size in Types)
```

We utilize variadic recursion to calculate the `largest size in Types`. **Hint:** `std::max`

Once you know your solution works, then use `Variant_Helper` to create a C-array called `memory` inside `Variant` which is as large as the largest type in the passed in types.

Part II - The type tag

Now that we have defined the memory inside `Variant`, the second step is to make sure that we can store a tag that helps us keep track of which type is currently being stored in the variant.

There are two main ways to create such a tag. These are:

- Use the index within the variadic pack of the currently stored types. To make this work we need to add functionality to: find wheter a type is present within a variadic pack, as well as finding a types index (see the exercise "Variadic templates" in seminar 5 for how this can be achieved).
- Use RTTI (**R**un**t**ime **T**ype **I**nformation), more specifically `typeid`. Each type has a `std::type_info` object associated with them which is retrieved by using `typeid`. For example: `typeid(int)`, `typeid(std::string)` etc.

  Two `std::type_info` objects can be compared with each other using `operator==`, and they are *guaranteed* to be equal if and only if the two types are the same. For example: `typeid(int) == typeid(int)`.

  Note that `std::type_info` objects *cannot* be copied, so we are unable to store them directly. There are two options: we can either store a pointer to the currently set `std::type_info` object, *OR* we can use `std::type_index` (this is the recommended way). If you use the pointer solution you have to compare the *objects* not the pointers.

Decide how you want to store what type is currently active and update your `Variant` class accordingly. After this we have all the data members we need in `Variant`.

Part III - Constructor

Now it is time to create an appropriate constructor for the variant. The constructor must be a function template that takes *one* parameter of arbitrary type `T`. This parameter must be taken as a forwarding reference as to make sure that it works in the general case (some constructors might expect references of different types). It also ensures that parameter passing is as efficient as possible.

Note that the due to the inner workings of forwarding references `T` will be deduced as a *lvalue reference type* if the passed in parameter happens to be an lvalue. This is important to note since `T&` and `T` counts as different types. Likewise it is important to note that `const` matters as well, meaning that the types `T const` and `T` are also different. Therefore we must *normalize* the type `T` by removing any references and any potential `const` qualifiers. This is most easily done with the type traits (found in the header `<type_traits>`) `std::remove_reference_t` and `std::remove_cv_t`. We give the normalized version of `T` the name `NormalizedType`. **Hint:** You can take an extra template parameter called `NormalizedType` that has default value:

```
std::remove_cv_t<std::remove_reference_t<T>>
```

The constructor must initialize the type tag to represent the `NormalizedType` type, and then it needs to use placement new to initialize an object of type `NormalizedType` inside `memory`. This should call the appropriate copy/move constructor of `NormalizedType` by forwarding the passed in parameter to the constructor call. Note that we must include `<new>` to get access to placement new.

Once the constructor is done you should start to think about writing some testcases to see that everything compiles for different types.

After this we are ready to start implementing the actual member functions of `Variant`.

Part IV - The getter

Arguably the most important function of `Variant` is the function that allows the user to retrieve the value currently stored in the variant. The so called `get()` function.

The idea is similar to `std::variant`, but our `get()` function will be a *member function* rather than a free function. This means that we want the function to be called like this:

```
Variant<int, double> v { 10 }; // int-constructor

// Retrieve the currently stored value as int
std::cout << v.get<int>() << std::endl;

// It should be possible to modify the stored value through
the get function as well, like this:
v.get<int>() = 7;
```

`get()` should therefore be a function template that takes one type `T` as a template type parameter and returns a `T` reference.

The first step in implementing the `get()` function is to somehow *interpret* the bytes stored in `memory` as a `T` object. This is done with `reinterpret_cast` and `std::launder`. But before we do that we first have to make sure that the currently stored type is equal to the requested type. If the requested type is not the same as the stored one, then the function must throw a `std::bad_cast` exception.

Once the `get()` function has been implemented you should be able to test it as well. But something important is still missing: the assignment operator.

There are two cases we need to consider when implementing the assignment operator:

1. If they assigned value *has the same type* as the currently stored one.
2. If the assigned value *has a different type* from the currently stored one.

The first case is quite simple: we simply use `get()` to retrieve the currently stored value, and then we assign the new value to it.

Implement the first case in your `Variant`. The passed in parameter should be passed as a forwarding references (same reason as with the constructor). This means that you must consider the `NormalizedType` here as well.

Recall from the seminar on sum types that we need to call the destructor of the old object before we assign a new value of a different type to the variant. How do we do that in the general case?

Part V - Calling the destructor of the previous type

It is not (necessarily) known during compile-time what type of object is currently stored in the variant whenever an assignment occur. Consider for example this case:

```
Variant<int, std::string> v { 0 };

int x { };
std::cin >> x;

// If the user enters an even number then v will be assigned
// a string, otherwise it will stay as an int.
if (x % 2 == 0)
  v = "string";

// We cannot know until the program runs what type is
// currently stored in v, so how do we know which
// destructor to call?
v = 3;
```

This means that we have to figure out which destructor to call based on the type tag. There are many ways to do this, but the simplest one is to use variadic recursion.

We can put our destroy functionality in `Variant_Helper` to avoid a lot of boilerplate code. Create a function called `destroy()` that takes two parameters:

- a `char*` which represents the memory address of the `memory` array inside our `Variant`
- and the currently stored type tag.

With this information we can recursively go through all types in `Types` and for each one check whether they match the current type tag. If they do, then we can call the destructor. Otherwise we will continue the recursion.

Note that we have to *reinterpret* the passed in pointer as the appropriate type in order to call the destructor on it. Therefore it might be useful to move the `reinterpret_cast` and `std::launder` calls from the `get()` function into a free function (i.e. *not* a member function) called `as()`. That way we can call the constructor as such:

```
if (/* the passed in tag represents the current type Type */)
{
  as<Type>(pointer).~Type();
}
else
{
  // continue recursing through the Types pack
}
```

Where `Type` represents the type from the variadic pack `Types` that we are currently examining in the variadic recursion. Put this functionality inside a function called `destroy()`. It will be easier if you add this as a static function inside `Variant_Helper`.

Part VI - The assignment operator

Now that we have the `destroy()` function, we should be able to fully implement the assignment operator (specifically the second case), by first calling the appropriate destructor (use `destroy()`) and then initialize the new object (use placement new + perfect forwarding). Finally we must remember to update the type tag.

Do this, and then test *thoroughly* that the assignment operator works. Try changing between different types, and make sure that all the destructors are called correctly.

Final part

Now you should have all the minimum functionality required to actually use the variant. This assignment description does not cover everything you need to do, there might be some things that have been left out. It is your job to figure out the exact details for everything.

There are some testcases given in `main.cc`, if you have done everything correctly then all those testcases should work and there should be no memory leaks.

Note that there are two solutions given in the `solutions/` directory. Please try to refrain from looking at them until you are done with your own implementation. The two solutions are very similar with the difference that one of them uses `typeid` (`Variant_typeid.h`) to store the tags while the other one uses the index (`Variant_index.h`) of each type in the variadic pack.