A fun example of polymorphism with std::function

Inspiration

- A StackOverflow question: "*Is it possible to declare a pointer to a function with unknown (at compile time) return type?*"
- The person asking wanted to support callbacks that may have different return types (`double` and `int` in their case).
- Generalizing this is where the fun starts :)

Polymorphism?

- The provision of a single interface to entities of different types.
- Our entities will be callable objects.
- The interface is provided by `std::function`.

std::function 101

- A class template from the C++ standard library.
- Can store any *copyable* entity that may be invoked as a function.
- Is itself a callable object that supports `operator()`.
- The supported signature needs to be specified up front, e.g.

```
std::function<void(double)>
std::function<int()>
```

The Actual Call Itself

- Is type safe.
  - The argument types must be convertible to the declared parameter types.
  - The return value is implicitly converted to the declared return type.
    - If the declared return type is `void`, the return value is properly discarded.
- Provides great flexibility.

Discard the Return Value – Declare it void

```
std::function<void(int)> f;
f = [](int x) { return x*2; };
f = [](int x) { return std::to_string(x); };
f = std::to_upper;
// Whatever gets returned, it's static_cast to void
```

Where We Started – Different Return Types

- We originally wanted to support a return value that is one of several types.
- Sounds like a union!
- We can compose std::function with std::variant.

Answering the Original Question

```
std::function<std::variant<double, int>(double)> f;
f = static_cast<...>(std::abs);
f = [](double x) { return static_cast<int>(x); }

// The variant supports conversions to its alternative types
```

No std::variant ? No Problem!

- boost::variant works, and if we really need to, we can go DIY…

```
struct Result {
    union {
        int    i_res;
        double d_res;
    };
    enum { IS_INT, IS_DOUBLE } u_tag;

    Result(Result const&) = default;
    Result(int i)  : i_res{i}, u_tag{IS_INT} {}
    Result(double d) : d_res{d}, u_tag{IS_DOUBLE} {}
};
```

Questions?