hw

1.

a)

With pass by value, this program would print:

```
baz
bar
1
```

b)

With pass by reference, this program would print:

```
baz
bar
4
```

c)

With pass by object reference, this program would print:

```
baz
bar
1
```

d)

With pass by need, this program would print:

```
bar
1
```

2.

A benefit of the `Optional` approach is improved programmer experience: you know that the function returns an optional based on the function signature. This enables the LSP to give you more intelligent autocompletion and error / type checking. With the second option, you only know that an `int` is returned from the function: you don't know if the function returns `-1`, throws an exception, or does anything different when/if the value is not found. You would have to spend time looking at the function logic to determine this behavior. Users of the `Optional` method must only handle two cases: the `Optional` contains a value or the `Optional` contains `nullptr`. Users of the exception throwing method may assume the value given back is always valid, so long as they have a catcher around the function just in case it is not. This is more dangerous because the `Optional` value must always be "unboxed" from the `struct`, meaning the programmer must always consciously be aware that their return value being vaild is not guaranteed. With the exception method, the value given back is always an `int` that, according to intuition (and your LSP) can always be used normally where any other `int` can be used, so it can be easy to call the function and accidentally forget that you need to set up a catcher for it just in case. I would say that given

all of this the `Optional` is more suitable for this use case, as it seems most idiomatic given that the function has only two possible states: “has it” and “doesn’t have it”.

3.

a)

```
catch 2
I'm done!
that's what I say
Really done!
```

b)

```
catch 1
hurray!
I'm done!
that's what I say
Really done!
```

c)

```
catch 1
hurray!
```

```
I'm done!
that's what I say
Really done!
```

d)

```
catch 3
Really done!
```

e)

```
hurray!
I'm done!
that's what I say
Really done!
```

4.

---

a)

```
template <typename T> class Kontainer {
  private:
    T *min_el;
    T array[100];
```

```
    size_t size = 0;

  public:
    Kontainer() {
        min_el = nullptr;
    }

    void addVal(T val) {
        if (!min_el || val < *min_el) {
            min_el = &array[size];
        }
        array[size++] = val;
    }

    T getMin() {
        if (min_el) {
            return *min_el;
        } else {
            throw runtime_error("No values in Kontainer yet!");
        }
    }
};
```

b)

```
struct Kontainer<T: PartialOrd + Copy> {
    items: [T; 100],
    min_item: Option<T>,
    current: usize,
}
```

```
impl<T: PartialOrd + Copy> Kontainer<T> {
    fn add_val(&mut self, val: T) {
        let len = self.current;

        self.items[len] = val;
        if self.min_item.is_none() || self.min_item.unwrap() > val {
            self.min_item = Some(self.items[len]);
        }
        self.current += 1;
    }

    fn get_min(&self) -> Option<T> {
        return self.min_item;
    }
}

impl<T: PartialOrd + Copy + Default> Default for Kontainer<T> {
    fn default() -> Self {
        Kontainer {
            items: [T::default(); 100],
            min_item: None,
            current: 0,
        }
    }
}
```

c)

Version 1

```
int main() {
    Kontainer<string> *str_kont = new Kontainer<string>();
    str_kont->addVal("hi");
    str_kont->addVal("hiiii");
    str_kont->addVal("zUP");
    cout << str_kont->getMin() << endl; // prints "hi";

    Kontainer<double> *dub_kont = new Kontainer<double>();
    dub_kont->addVal(6.9);
    dub_kont->addVal(0.00023);
    dub_kont->addVal(10000000);
    cout << dub_kont->getMin() << endl; // prints 0.00023
}
```

Version 2

```
fn main() {
    let mut kont: Kontainer<f64> = Default::default();
    kont.add_val(5.999);
    kont.add_val(0.681);
    kont.add_val(900.0);
    let min = kont.get_min().unwrap();
    println!("{}", min); // prints 0.681

    let mut kont: Kontainer<&str> = Default::default();
    kont.add_val("hELLo");
    kont.add_val("sup");
    kont.add_val("ZAMN");
    let min = kont.get_min().unwrap();
    println!("{}", min); // prints "ZAMN"
}
```

d)

In C++, the difficulties (for me) were few and mostly due to syntax regarding how to declare a template class and declare/assign a class field as an array of a generic type. Also figuring out how to handle the error that there was no minimum value was sort of tricky at first because I didn't know what approach I should take, and I eventually decided on just throwing an exception.

With Rust I had much more difficulty. Much of it was still regarding syntax, but I also had a great deal of trouble figuring out how to appease the Rust compliler in the process of making my `Kontainer`. I blame this mostly on my lack of experience and not immediately knowing what approach to take for certain things (E.g. should I return the minimum value as an optional reference and thus force myself to be concerned with lifetimes? Do I return an optional value and then figure out how to specify that my value inherits the `Copy` trait? After struggling a lot with the former I settled on the latter which proved easier than I thought.). Many compiler errors came about that I

didn’t even know existed, so that was unexpected and a good learning experience. Overall the Rust version was much more difficult to implement than the C++ version, but I am glad I did both because the Rust version is certainly more rubust (though more verbose) and I at least take comfort in the fact that the plethora of errors I encountered during compile time mean my Rust code is much more future-proof than its C++ counterpart, which (as I found out the hard way while working on this assignment) is always one typo away from a segfault.