Question 1

a)
```
baz
bar
1
```

b)
```
baz
bar
4
```

c)
```
baz
bar
1
```

d)
```
bar
1
```

Question 2

The `Optional` will allow the program to continue running even if the value is not found, returning `nullptr` instead. This allows for the programmer to simply check if the value of `Optional` is `nullptr` rather than handling an exception explicitly. C++'s native exception handling forces the user to explicitly handle the exception. Consumers of the `Optional` API need to check whether or not the value returned is a `nullptr` whereas the consumers of C++'s native exception handling need to explicitly define an error handler via a `try/catch` block. In this particular use case, I think that the `Optional` struct is better since users are most likely going to be querying for values that probably exist in the array. Additionally, an exception doesn't seem appropriate in this case, so forcing the user to explicitly handle the exception sounds worse than just checking for a `nullptr`.

Question 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
```

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

Question 4

a)
```
template<typename T>
class Kontainer {
public:
  using value_type           = T;
  using reference_type       = T&;
  using pointer_type         = T*;
  using const_reference_type = const T&;
  using const_pointer_type   = const T*;

public:
  constexpr Kontainer();
            ~Kontainer();

  void       add(const_reference_type element);
  value_type minimum_element()                  const;

private:
  value_type m_Data[100];
  size_t       m_Size     = 0;
};

template<typename T>
constexpr Kontainer<T>::Kontainer() : m_Size(0) { }

template<typename T>
Kontainer<T>::~Kontainer() { }

template<typename T>
inline void Kontainer<T>::add(const_reference_type element)
{
  m_Data[m_Size++] = element;
}

template<typename T>
inline typename Kontainer<T>::value_type Kontainer<T>::minimum_element() const
{
  if (m_Size == 0)
      return value_type();

  value_type min = m_Data[0];

  for (size_t i = 0; i < m_Size; i++)
    if (m_Data[i] < min)
      min = m_Data[i];

  return min;
}
```

b)
```
using System;

public class Kontainer<T> where T : IComparable<T>
{
  private T[] m_Data;
  private int m_Size;
```

```
  public Kontainer()
  {
    m_Data = new T[100];
    m_Size = 0;
  }

  public void Add(T element)
  {
      if (m_Size < 100)
        m_Data[m_Size++] = element;
  }

  public T MinimumElement()
  {
    if (m_Size == 0)
      return default(T);

    T min = m_Data[0];

    for (int i = 0; i < m_Size; i++)
      if (m_Data[i].CompareTo(min) < 0)
        min = m_Data[i];
    return min;
  }
}
```

c)
```
#include <iostream>
#include <string>
#include "kontainer.hpp"

int main ()
{
  Kontainer<std::string> strings;

  strings.add("hello");
  strings.add("abc");
  strings.add("lmnop");

  std::cout << strings.minimum_element() << std::endl;

  Kontainer<double> doubles;

  doubles.add(10.0);
  doubles.add(12.32);
  doubles.add(0.1029320);
  doubles.add(0.000139);

  std::cout << doubles.minimum_element() << std::endl;
  return 0;
}
```

d) C++ is more verbose than C#. Other than that, in this particular case, I did not see too much of a difference.