Contracts and Strong Types

Presenter: Rob Keelan

What is a Contract?

- Functions often require input values to meet some precondition(s) to produce valid output.
- Given valid input a function should be able to make some guarantees about the state of the program after the function has run (postconditions).
- Today I'm most concerned with preconditions.
- Great talk on contracts
  - https://youtu.be/Dzk1frUXq10

```
template <class T, size_t size>
class Array
{
public:

// @pre index < size (this one gives UB on violation)
T& operator[](size_t index);

// @pre index < size (this one throws an exception on violation)
T& at(size_t index);

// other functions...
};
```

What about performance?

- As a developer of a function I should check my inputs for contract violations if possible.
- That could get really expensive if the same input is used over and over in different functions.
- All the contract checks could start adding up.
- Also the implementer of foo() and Array need to make a decision whether to check at runtime or leave it as UB.

```
template <class T, size_t size>
std::pair<T, T> foo(Array<T, size> const& x,
Array<T, size> const& y, size_t index)
{
  // The same check is performed twice!
  return std::pair{x.at(index), y.at(index)};
}
```

What's the alternative?

- Strong Types offer a potential solution
  - Usually implemented as a wrapper around some underlying type
  - Can maintain some invariant around the underlying data.
- Great references on Strong Types
  - https://foonathan.net/2016/10/strong-typedefs/
  - https://www.fluentcpp.com/2016/12/08/strong-types-for-strong-interfaces/

```
template <class Tag, class Invariant, class T>
class StrongTypedef
{
public:
    explicit StrongTypedef(const T& value) :
      value_(value)
    {
      if(!F{}(value_))
      {
        std::terminate();
      }
    }

    operator T const&() const noexcept
    {
        return value_;
    }

private:
    T value_;
};
```

How do Strong Types apply to our example?

- Let's have the array's operator[] take in a Strong Type, let's call it BoundedIndex.
- BoundedIndex can only contain indices between zero and some max size inclusive.

```
template <size_t max>
struct MaxValue
{
  constexpr bool operator()(size_t value) const noexcept
  {
    return value < max;
  }
};

template <size_t max>
class BoundedIndex : private
StrongTypedef<BoundedIndex<max>, MaxValue, size_t>
{
  using StrongTypedef<>::StrongTypedef;
  using StrongTypedef<>::
                    operator size_t const&() const noexcept;
};
```

How do Strong Types apply to our example?

- Let's update the Array class to use Strong Types
- Then let's see how foo() is updated.
- Caller of foo() now gets to choose how / when the invariant is checked.

```
template <class T, size_t size>
class Array
{
public:

// @pre index < size (this one gives UB on
violation)
T& operator[](BoundedIndex<size> index);

// other functions...
};
```

```
template <class T, size_t size>
std::pair<T, T> foo(Array<T, size> const& x,
   Array<T, size> const& y, BoundedIndex<size> index)
{
  // No runtime checks are performed!
  return std::pair{x[index], y[index]};
}
```

So we don’t need contracts anymore?

- Well no think about std::vector
  - We can’t create a type that encodes the value is less than the size
  - So here we still need contracts
- Some invariants aren’t known at compile time so they can’t be encoded in types.

```
template <class T>
class vector
{
public:

// @pre index < size (this one gives UB on violation)
T& operator[](size_t index);

// @pre index < size (this one throws an exception on violation)
T& at(size_t index);

// other functions...
};
```

What’s the take away?

- Use Strong Types to encode contracts when they are known at compile time
- Prevents you from having to make decisions about whether certain contracts should be checked or not
- They can reduce the need to write duplicate checks
- Strong Types have other benefits too!
  - Put contracts in the type system
  - Separate types can help eliminate some easy mistakes.

```
class Image
{
public:
// Assume X and Y are Strong Types!
// The compiler will yell if you call this function with
// parameters in the wrong order
Pixel get(X x, Y y) const;
};
```