Programming with Contracts in C++20

Björn Fahller
What is a contract?

<table>
<thead>
<tr>
<th>contract</th>
</tr>
</thead>
<tbody>
<tr>
<td>noun</td>
</tr>
</tbody>
</table>

**Definition of contract**

(Entry 1 of 3)

1:

- a: binding agreement between two or more persons or parties - especially : one legally enforceable
  // If he breaks the contract, he'll be sued.

- b: a business arrangement for the supply of goods or services at a fixed price
  // make parts on contract

- c: the act of marriage or an agreement to marry

2: a document describing the terms of a contract
  // Have you signed the contract yet?

3: the final bid to win a specified number of tricks in bridge

4: an order or arrangement for a hired assassin to kill someone
  // His enemies put out a contract on him.

https://www.merriam-webster.com/dictionary/contract
What is a contract?

**contract**

_noun_  con-tract | \ˈkän-,trakt \_

**Definition of contract**

(Entry 1 of 3)

1:

a: binding agreement between two or more persons or parties - especially : one legally enforceable
   // If he breaks the contract, he'll be sued.

b: a business arrangement for the supply of goods or services at a fixed price
   // make parts on contract

c: the act of marriage or an agreement to marry

2: a document describing the terms of a contract
   // Have you signed the contract yet?

3: the final bid to win a specified number of tricks in bridge

4: an order or arrangement for a hired assassin to kill someone
   // His enemies put out a contract on him.

https://www.merriam-webster.com/dictionary/contract

In SW design:

A formalised agreement, regarding program correctness, between a user and the implementation of a component.
What is a contract?

**contract**

noun  con·tract | \ˈkän-,trakt \  

**Definition of contract**

(Entry 1 of 3)

1:

a:  binding agreement between two or more persons or parties - especially : one legally enforceable  
   // If he breaks the contract, he'll be sued.

b:  a business arrangement for the supply of goods or services at a fixed price  
   // make parts on contract

c:  the act of marriage or an agreement to marry

2:  a document describing the terms of a contract  
    // Have you signed the contract yet?

3:  the final bid to win a specified number of tricks in bridge

4:  an order or arrangement for a hired assassin to kill someone  
    // His enemies put out a contract on him.

https://www.merriam-webster.com/dictionary/contract

In SW design:

A formalised agreement, **regarding program correctness**, between a user and the implementation of a component.
Contracts

- Object-oriented Software Construction
  - Bertrand Meyer - 1988
  - ISBN 978-0136290490
Contracts

- Preconditions
- Postconditions
- Class invariants
Ringbuffer example

```cpp
ringbuffer <int,12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```
Ringbuffer example

```cpp
ingrabuffer <int,12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```
Ringbuffer example

```cpp
ringbuffer <int, 12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```
Ringbuffer example

```cpp
ringbuffer <int,12> b;
b.push_back(1);
**b.push_back(2);**
b.push_back(5);
b.pop_front();  // 1
b.push_back(8);
b.pop_front();  // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```
Ringbuffer example

```cpp
ringbuffer <int, 12> b;
b.push_back(1);
b.push_back(2);
\textcolor{blue}{b.push_back(5)};
b.pop_front();  // 1
b.push_back(8);
b.pop_front();  // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```
Ringbuffer example

```cpp
ingbuffer <int,12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```
Ringbuffer example

```cpp
ringbuffer <int, 12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```
Ringbuffer example

```cpp
ingress <int, 12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```
Ringbuffer example

```cpp
ingithub <int,12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```
Ringbuffer example

```cpp
ringbuffer <int,12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```
Ringbuffer example

```cpp
ringbuffer <int,12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```
Programming with Contracts in C++20 – ACCU 2019 © Björn Fahller

Ringbuffer example

```cpp
ringbuffer <int, 12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```

@bjorn_fahller

18/168
ringbuffer <int, 12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
Ringbuffer example

```cpp
ringbuffer <int,12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```
Ringbuffer example

```cpp
ingbuffer <int,12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
```

```cpp
b.push_back(23);
b.push_back(24);
```
Ringbuffer example

```cpp
ringbuffer <int,12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front();  // 1
b.push_back(8);
b.pop_front();  // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```
template <typename T, int N>
class ringbuffer {
public:

    ringbuffer();

    int size() const;

    void push_back(T);

    T pop_front();

};
Ringbuffer example

template <typename T, int N>
class ringbuffer {
public:

    ringbuffer();
    int size() const;

    void push_back(T);

    T pop_front();

};

Precondition:
Ringbuffer example

template <typename T, int N>
class ringbuffer {
public:

    ringbuffer();

    int size() const;

    void push_back(T);

    T pop_front();

};

Precondition:
An obligation that the caller must fulfill for the program to be correct.
Ringbuffer example

template <typename T, int N>
class ringbuffer {
public:

    ringbuffer();

    int size() const;

    void push_back(T);

    T pop_front();

};

Precondition:
An obligation that the caller must fulfill for the program to be correct.

A precondition may refer to parameter values or the objects state, or both.
Ringbuffer example

template <typename T, int N>
class ringbuffer {
public:

    ringbuffer();

    int size() const;

    void push_back(T);

    T pop_front();

};

Precondition:

An obligation that the caller must fulfill for the program to be correct.
Ringbuffer example

template <typename T, int N>
class ringbuffer {
public:
    ringbuffer();
    int size() const;
    void push_back(T);
    T pop_front();
};

Precondition:
An obligation that the caller must fulfill for the program to be correct.

It almost never makes sense to have a precondition on a default constructor!
Ringbuffer example

template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();

  int size() const;

  void push_back(T);

  T pop_front();

};

Precondition:

An obligation that the caller must fulfill for the program to be correct.
Ringbuffer example

template <typename T, int N>
class ringbuffer {
public:

    ringbuffer();

    int size() const;

    void push_back(T);

    T pop_front();

};

Precondition:
An obligation that the caller must fulfill for the program to be correct.

Functions that query the state of an object rarely has any preconditions.
Ringbuffer example

```cpp
template <typename T, int N>
class ringbuffer {
public:
    ringbuffer();
    int size() const;

    void push_back(T);

    T pop_front();
};
```

**Precondition:**
An obligation that the caller must fulfill for the program to be correct.
Ringbuffer example

template <typename T, int N>
class ringbuffer {
public:
    ringbuffer();
    int size() const;
    void push_back(T);
    T pop_front();
};

Precondition:
An obligation that the caller must fulfill for the program to be correct.

Choose between: Define behaviour when full, or make not-full a precondition.
Ringbuffer example

template <typename T, int N>
class ringbuffer {
public:

    ringbuffer();
    int size() const;

    void push_back(T);
    // requires: size() < N

    T pop_front();
};

Precondition:
An obligation that the caller must fulfill for the program to be correct.
Ringbuffer example

template <typename T, int N>
class ringbuffer {
public:

    ringbuffer();

    int size() const;

    void push_back(T);
    // requires: size() < N

    T pop_front();

};
Ringbuffer example

template <typename T, int N>
class ringbuffer {
public:

    ringbuffer();
    int size() const;
    void push_back(T);

    // requires: size() < N

    T pop_front();

};

Precondition:
An obligation that the caller must fulfill for the program to be correct.

Choose between: Define behaviour when empty, or make not-empty a precondition.
Ringbuffer example

template <typename T, int N>
class ringbuffer {
public:

    ringbuffer();
    int size() const;

    void push_back(T);
    // requires: size() < N

    T pop_front();
    // requires: size() > 0

};

Precondition:
An obligation that the caller must fulfill for the program to be correct.
Ringbuffer example

template <typename T, int N>
class ringbuffer {
public:

    ringbuffer();

    int size() const;

    void push_back(T);
    // requires: size() < N

    T pop_front();
    // requires: size() > 0

};

Postcondition:
Ringbuffer example

template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();

  int size() const;

  void push_back(T);
  // requires: size() < N

  T pop_front();
  // requires: size() > 0

};

Postcondition:
A guarantee from the implementation regarding the effect of a legal call.
Ringbuffer example

template <typename T, int N>
class ringbuffer {
public:

    ringbuffer();

    int size() const;

    void push_back(T);
    // requires: size()

    T pop_front();
    // requires: size() > 0

};

Postcondition:
A guarantee from the implementation regarding the effect of a legal call.

A postcondition may refer to return value or the objects state, or both, sometimes dependent on parameter values.
Ringbuffer example

template <typename T, int N>
class ringbuffer {
public:

    ringbuffer();

    int size() const;

    void push_back(T);
    // requires: size() < N

    T pop_front();
    // requires: size() > 0

};

Postcondition:
A guarantee from the implementation regarding the effect of a legal call.
```
template <typename T, int N>
class ringbuffer {
public:
    ringbuffer(); // ensures: size() == 0
    int size() const;

    void push_back(T); // requires: size() < N

    T pop_front(); // requires: size() > 0
};
```

**Postcondition:**
A guarantee from the implementation regarding the effect of a legal call.
template <typename T, int N>
class ringbuffer {
public:

    ringbuffer();
    // ensures: size() == 0
    int size() const;

    void push_back(T);
    // requires: size() < N

    T pop_front();
    // requires: size() > 0

};

Postcondition:
A guarantee from the implementation regarding the effect of a legal call.
Ringbuffer example

template <typename T, int N>
class ringbuffer {
public:

    ringbuffer();  // ensures: size() = 0
    int size() const;

    void push_back(T);  // requires: size() < N

    T pop_front();  // requires: size() > 0

};

Postcondition:
A guarantee from the implementation regarding the effect of a legal call.
Ringbuffer example

template <typename T, int N>
class ringbuffer {
public:

    ringbuffer();
    // ensures: size() == 0
    int size() const;

    void push_back(T);
    // requires: size() < N
    // ensures: size() == old size()+1

    T pop_front();
    // requires: size() > 0

};
Ringbuffer example

```cpp
template <typename T, int N>
class ringbuffer {
public:

    ringbuffer();
    // ensures: size() == 0
    int size() const;

    void push_back(T t);
    // requires: size() < N
    // ensures: size() == old size()+1

    T pop_front();
    // requires: size() > 0

};
```

Postcondition:

A guarantee from the implementation regarding the effect of a legal call.
Ringbuffer example

```cpp
template <typename T, int N>
class ringbuffer {
public:

    ringbuffer();
    // ensures: size() == 0
    int size() const;
    const T& back() const;

    void push_back(T t);
    // requires: size() < N
    // ensures: size() == old size()+1
    // back() == t
    T pop_front();
    // requires: size() > 0

};
```

Postcondition:
A guarantee from the implementation regarding the effect of a legal call.
Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

    ringbuffer();
    // ensures: size() == 0
    int size() const;
    const T& back() const;
    // requires: size() > 0

    void push_back(T t);
    // requires: size() < N
    // ensures: size() == old size()+1
    //         back() == t
    T pop_front();
    // requires: size() > 0

};
```

Postcondition:

A guarantee from the implementation regarding the effect of a legal call.
Ringbuffer example

```cpp
template <typename T, int N>
class ringbuffer {
public:

    ringbuffer();  // ensures: size() /= 0

    int size() const;  // requires: size() > 0

    const T& back() const;  // requires: size() > 0

    void push_back(T t);  // requires: size() < N
                         // ensures: size() = old size() + 1
                         //         back() = t

    T pop_front();  // requires: size() > 0

};
```

Postcondition:
A guarantee from the implementation regarding the effect of a legal call.
What if an exception is thrown?
Ringbuffer example

template <typename T, int N>
class ringbuffer {
public:

    ringbuffer();
    // ensures: size() != 0
    int size() const;
    const T& back() const;
    // requires: size() > 0
    // ensures: size() /= old size()+1
    //          back() /= t
    T pop_front();
    // requires: size() > 0

private:

    void push_back(T t);
    // requires: size()
    // ensures: size() /= old size()+1
    //          back() /= t

};

Postcondition:

A guarantee from the implementation regarding the effect of a legal call.

Postconditions handles return. If an exception is thrown, there is no post condition.
Ringbuffer example

```cpp
template <typename T, int N>
class ringbuffer {
public:

    ringbuffer();
    // ensures: size() == 0
    int size() const;
    const T& back() const;
    // requires: size() > 0

    void push_back(T t);
    // requires: size() < N
    // ensures: size() == old size()+1
    // back() == t

    T pop_front();
    // requires: size() > 0

};
```

Postcondition:

A guarantee from the implementation regarding the effect of a legal call.
Ringbuffer example

template <typename T, int N>
class ringbuffer {
public:

    ringbuffer();  
    // ensures: size() == 0
    int size() const;
    const T& back() const;
    // requires: size() > 0

    void push_back(T t);  
    // requires: size() < N  
    // ensures: size() == old size()+1  
    //     back() == t

    T pop_front();  
    // requires: size() > 0  
    // ensures: size() == old size()-1

};

Postcondition:
A guarantee from the implementation regarding the effect of a legal call.
Ringbuffer example

template <typename T, int N>
class ringbuffer {
public:

ringbuffer();
   // ensures: size() == 0
int size() const;
const T& back() const;
   // requires: size() > 0
const T& front() const;

void push_back(T t);
   // requires: size() < N
   // ensures: size() == old size()+1
   //       back() == t
T pop_front();
   // requires: size() > 0
   // ensures: size() == old size()-1
   //      return == old front();
};

Postcondition:
A guarantee from the implementation regarding the effect of a legal call.
Ringbuffer example

```cpp
template <typename T, int N>
class ringbuffer {
public:

    ringbuffer();
    // ensures: size() == 0

    int size() const;
    // requires: size() > 0

    const T& back() const;
    // requires: size() > 0

    const T& front() const;
    // requires: size() > 0

    void push_back(T t);
    // requires: size() < N
    // ensures: size() == old size()+1
    //         back() == t

    T pop_front();
    // requires: size() > 0
    // ensures: size() == old size()-1
    //         return == old front();
};
```

Postcondition:
A guarantee from the implementation regarding the effect of a legal call.
Ringbuffer example

template <typename T, int N>
class ringbuffer {
public:
    ringbuffer();
    // ensures: size() /= 0
    int size() const;
    const T &top() const;
    // requires: size() > 0
    const T &front() const;
    // requires: size() > 0
    void push_back(T t);
    // requires: size() < N
    // ensures: size() /= old size()+1
    // back() /= t
    T pop_front();
    // requires: size() > 0
    // ensures: size() /= old size()-1
    // return /= old front();
};

It does not make sense to try and express the returned value from the history of pushes and pops as a post condition.

Postcondition:
A guarantee from the implementation regarding the effect of a legal call.
Ringbuffer example

template <typename T, int N>
class ringbuffer {
public:

    ringbuffer();
    // ensures: size() == 0
    int size() const;
    const T& back() const;
    // ensures: size() > 0
    const T& front() const;
    // requires: size() > 0
    void push_back(T t);
    // requires: size() < N
    // ensures: size() == old size()+1
    // back() == t
    T pop_front();
    // requires: size() > 0
    // ensures: size() == old size()-1
    // return == old front();
};
Ringbuffer example

template <typename T, int N>
class ringbuffer {
    public:

    ringbuffer();
        // ensures: size() == 0
    int size() const;
    const T& back() const;
        // ensures: size() > 0
    const T& front() const;
        // requires: size() > 0
    void push_back(T t);
        // requires: size() < N
        // ensures: size() == old size()+1
        //        back() == t
    T pop_front();
        // requires: size() > 0
        // ensures: size() == old size()-1
        //        return == old front();
};

Class invariant:
Something that is always* true for a valid instance
* outside public API
## Ringbuffer example

```cpp
template <typename T, int N>
class ringbuffer {
public:
    ringbuffer();
    // ensures: size() /= 0
    int size() const;
    const T& back() const;
    // ensures: size() > 0
    const T& front() const;
    // requires: size() > 0
    void push_back(T t);
    // requires: size() < N
    // ensures: size() /= old size()+1
    // ensures: back() /= t
    T pop_front();
    // requires: size() > 0
    // ensures: size() /= old size()-1
    // ensures: front() /= old front();
};
```

### Class invariant:

*Something that is always* true for a valid instance *outside public API*

* outside public API
Ringbuffer example

template <typename T, int N>
class ringbuffer {
public:

    // invariant: size() \geq 0 && size() \leq N
    ringbuffer();
    // ensures: size() = 0
    int size() const;
    const T& back() const;
    // ensures: size() > 0
    const T& front() const;
    // requires: size() > 0
    void push_back(T t);
    // requires: size() < N
    // ensures: size() = old size()+1
    // back() = t
    T pop_front();
    // requires: size() > 0
    // ensures: size() = old size()-1
    // return = old front();
};
Ringbuffer example

```cpp
template <typename T, int N>
class ringbuffer {
public:
    // invariant: size() \geq 0 \&\& size() \leq N
    ringbuffer();
    // ensures: size() =
    int size() const;
    const T& back() const;
    // ensures: size() > 0
    const T& front() const;
    // requires: size() > 0
    void push_back(T t);
    // requires: size() < N
    // ensures: size() = old size()
    // back() = t
    T pop_front();
    // requires: size() > 0
    // ensures: size() = old size()-1
    // return = old front();
};
```

Class invariant:

Something that is always* true for a valid instance

* outside public API
Ringbuffer example

template <typename T, int N>
class ringbuffer {
public:
    // invariant: size() \geq 0 \&\& size() \leq N
    ringbuffer();
    // ensures: size() = 0
    int size() const;
    const T& back() const;
    // ensures: size() > 0
    const T& front() const;
    // requires: size() > 0
    void push_back(T t);
    // requires: size() < N
    // ensures: size() = old size()+1
    // back() = t
    T pop_front();
    // requires: size() > 0
    // ensures: size() = old size()-1
    // return = old front();
};

Contracts and templates
Ringbuffer example

template <typename T, int N>
class ringbuffer {
public:
    // invariant: size() ≥ 0 && size() ≤ N
    ringbuffer();
    // ensures: size() ≥ 0
    int size() const;
    const T& back() const;
    // ensures: size() > 0
    const T& front() const;
    // requires: size() > 0
    void push_back(T t);
    // requires: size() < N
    // ensures: size() == old size()
    // back() == t
    T pop_front();
    // requires: size() > 0
    // ensures: size() == old size()-1
    // return == old front();
};
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() ≥ 0 && size() ≤ N
  ringbuffer();
  // ensures: size() == 0
  virtual int size() const = 0;
  virtual const T& back() const = 0;
  // requires: size() > 0
  virtual const T& front() const = 0;
  // requires: size() > 0
  virtual void push_back(T t) = 0;
  // requires: size() < N
  // ensures: size() == old size()+1
  // back() == t
  virtual T pop_front() = 0;
  // requires: size() > 0
  // ensures: size() == old size()-1
  // return == old front();
};
Ringbuffer example

template <typename T, int N>
class ringbuffer {

public:

    // invariant: size() \geq 0 \&\& size() \leq N
    ringbuffer();

    // ensures: size() = 0
    virtual int size() const = 0;

    // requires: size() > 0
    virtual const T& back() const = 0;

    // requires: size() > 0
    virtual const T& front() const = 0;

    // requires: size() < N
    // ensures: size() = old size()+1
    //         back() = t
    virtual T pop_front() = 0;

    // requires: size() > 0
    // ensures: size() = old size()-1
    //         return = old front();

};

Contracts and inheritance:

A subcontractor may have more relaxed pre-conditions
template<typename T, int N>
class ringbuffer {
public:
  // invariant: size() \geq 0  \&\&  size() \leq N
  ringbuffer();
  // ensures: size() = 0
  virtual int size() const = 0;
  virtual const T& back() const = 0;
  // requires: size() \geq 0
  virtual const T& front() const = 0;
  // requires: size() \geq 0
  virtual void push_back(T t) = 0;
  // requires: size() < N
  // ensures: size() = old size()+1
  // \hspace{0.5cm} back() = t
  virtual T pop_front() = 0;
  // requires: size() > 0
  // ensures: size() = old size()-1
  // \hspace{0.5cm} return = old front();
};

Contracts and inheritance:

A subcontractor may have more relaxed pre-conditions
and stricter post-conditions
Why bother?
Why bother?

1) It can make interfaces much clearer
Why bother?

1) It can make interfaces much clearer
2) It can make debugging much easier
Why bother?

1) It can make interfaces much clearer

2) It can make debugging much easier

3) It removes defensive checks
Who dunnit?

<table>
<thead>
<tr>
<th>violation</th>
<th>client</th>
<th>implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>precondition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>postcondition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>invariant</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Who dunnit?

Elementary, Dr. Watson

<table>
<thead>
<tr>
<th>violation</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>precondition</td>
<td></td>
</tr>
<tr>
<td></td>
<td>postcondition</td>
<td></td>
</tr>
<tr>
<td></td>
<td>invariant</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>client</th>
<th>implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>guilty</td>
<td></td>
</tr>
</tbody>
</table>

Programming with Contracts in C++20 – ACCU 2019 © Björn Fahlber
Who dunnit?

<table>
<thead>
<tr>
<th>violation</th>
<th>precondition</th>
<th>postcondition</th>
<th>invariant</th>
</tr>
</thead>
</table>
|            | guilty client | guilty
| implementation | | | |

**Invariants**

- A program invariant is a property that holds true for all valid states of a program.

**Preconditions**

- A precondition is a condition that must be true before a function or method is called.

**Postconditions**

- A postcondition is a condition that must be true after a function or method is called.

**Guilty**

- The client is guilty if the precondition is violated.
- The implementation is guilty if the postcondition is violated.
- The invariant is violated if the invariant holds true for an invalid state of the program.
Who dunnit?

<table>
<thead>
<tr>
<th>violation</th>
<th>client</th>
<th>implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>precondition</td>
<td>guilty</td>
<td></td>
</tr>
<tr>
<td>postcondition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>invariant</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Who dunnit?

<table>
<thead>
<tr>
<th>violation</th>
<th>precondition</th>
<th>postcondition</th>
<th>invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>client</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>guilty</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>implementation</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Who dunnit?

<table>
<thead>
<tr>
<th>violation</th>
<th>precondition</th>
<th>postcondition</th>
<th>invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>guilty</td>
<td>client</td>
<td>implementation</td>
<td></td>
</tr>
</tbody>
</table>

---

*Programming with Contracts in C++20 – ACCU 2019 © Björn Fahlle*
### Who dunnit?

<table>
<thead>
<tr>
<th>Violation</th>
<th>Precondition</th>
<th>Postcondition</th>
<th>Invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td><img src="image1.png" alt="Crickets" /></td>
<td><img src="image2.png" alt="Crickets" /></td>
<td></td>
</tr>
<tr>
<td>Implementation</td>
<td><img src="image3.png" alt="Crickets" /></td>
<td><img src="image4.png" alt="Crickets" /></td>
<td></td>
</tr>
</tbody>
</table>
Who dunnit?

<table>
<thead>
<tr>
<th>violation</th>
<th>guilty</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>client</td>
</tr>
<tr>
<td>precondition</td>
<td><img src="image1" alt="Cockroach" /></td>
</tr>
<tr>
<td>postcondition</td>
<td><img src="image3" alt="Cockroach" /></td>
</tr>
<tr>
<td>invariant</td>
<td><img src="image5" alt="Cockroach" /></td>
</tr>
</tbody>
</table>
Who dunnit?

<table>
<thead>
<tr>
<th>violation</th>
<th>guilty</th>
</tr>
</thead>
<tbody>
<tr>
<td>precondition</td>
<td>client</td>
</tr>
<tr>
<td>postcondition</td>
<td></td>
</tr>
<tr>
<td>invariant</td>
<td></td>
</tr>
</tbody>
</table>
Contracts in C++20
Contracts in C++20
Description of contract attribute declarations and semantics here (4 pages)
Contracts in C++20

Description of contract violation handlers
Contracts in C++20

Meaning for virtual functions (1 paragraph)
Contracts in C++20

Programing with Contracts in C++20 – ACCU 2019 © Björn Fahlle
Contract attributes in C++20

9.11.4.1 Syntax

Contract attributes are used to specify preconditions, postconditions, and assertions for functions.

contract-attribute-specifier:
   [ [ expects contract-level_opt : conditional-expression ] ]
   [ [ ensures contract-level_opt identifier_opt : conditional-expression ] ]
   [ [ assert contract-level_opt : conditional-expression ] ]

contract-level:
   default
   audit
   axiom

An ambiguity between a contract-level and an identifier is resolved in favor of contract-level.

http://eel.is/c++draft/dcl.attr.contract#syn-1
Contract attributes in C++20

9.11.4.1 Syntax

Contract attributes are used to specify preconditions, postconditions, and assertions for functions.

Syntax:

```
contract-attribute-specifier:
[[ [ expects contract-level_opt : conditional-expression ] ]
[[ [ ensures contract-level_opt identifier_opt : conditional-expression ] ]
[[ [ assert contract-level_opt : conditional-expression ] ]
```

contract-level:
- default
- audit
- axiom

An ambiguity between a `contract-level` and an `identifier` is resolved in favor of `contract-level`.

http://eel.is/c++draft/dcl.attr.contract#syn-1
# Contract attributes in C++20

## 9.11.4.1 Syntax

Contract attributes are used to specify preconditions, postconditions, and assertions for functions.

**contract-attribute-specifier:**

- `[[ expects contract-level_opt : conditional-expression ]]
- `[[ ensures contract-level_opt identifier_opt : conditional-expression ]]
- `[[ assert contract-level_opt : conditional-expression ]]

**contract-level:**
- `default`
- `audit`
- `axiom`

An ambiguity between a `contract-level` and an `identifier` is resolved in favor of `contract-level`.

```cpp
template <typename T>
void func(std::unique_ptr<T> p)
[[ expects : p != nullptr ]];
```

http://eel.is/c++draft/dcl.attr.contract#syn-1
Contract attributes in C++20

9.11.4.1 Syntax

Contract attributes are used to specify preconditions, postconditions, and assertions for functions.

### Syntax

```
contract-attribute-specifier:
  [[ expects contract-level_opt : conditional-expression ] ]
  [[ ensures contract-level_opt identifier_opt : conditional-expression ] ]
  [[ assert contract-level_opt : conditional-expression ] ]
```

**contract-level:**

- default
- audit
- axiom

Optional level

```
template <typename T>
void func(std::unique_ptr<T> p)
[[ expects : p != nullptr ]];
```

An ambiguity between a `contract-level` and an `identifier` is resolved in favor of `contract-level`.

---

http://eel.is/c++draft/dcl.attr.contract#syn-1
Contract attributes in C++20

9.11.4.1 Syntax

Contract attributes are used to specify preconditions, postconditions, and assertions for functions.

**contract-attribute-specifier:**

```
[[ expects contract-level_opt : conditional-expression ]]
[[ ensures contract-level_opt identifier_opt : conditional-expression ]]
[[ assert contract-level_opt : conditional-expression ]]
```

**contract-level:**

- default
- audit
- axiom

Optional level

```
template <typename T>
void func(std::unique_ptr<T> p)
[[ expects expects axiom : p != nullptr ]];
```

An ambiguity between a `contract-level` and an `identifier` is resolved in favor of `contract-level`.

http://eel.is/c++draft/dcl.attr.contract#syn-1
9.11.4.1 Syntax

Contract attributes are used to specify preconditions, postconditions, and assertions for functions.

```
contract-attribute-specifier:
    [ [ expects contract-level_opt : conditional-expression ] ]
    [ [ ensures contract-level_opt : conditional-expression ] ]
    [ [ assert contract-level_opt : conditional-expression ] ]
```

An ambiguity between a `contract-level` and an `identifier` is resolved in favor of `contract-level`.

http://eel.is/c++draft/dcl.attr.contract#syn-1
Contract attributes in C++20

9.11.4.1 Syntax

Contract attributes are used to specify preconditions, postconditions, and assertions for functions.

\textbf{contract-attribute-specifier:}

\begin{itemize}
  \item \texttt{[ [ expects contract-level\_opt : conditional-expression ] ]}
  \item \texttt{[ [ ensures contract-level\_opt identifier\_opt : conditional-expression ] ]}
  \item \texttt{[ [ assert contract-level\_opt : conditional-expression ] ]}
\end{itemize}

\textbf{contract-level:}

- default
- audit
- axiom

An ambiguity between a \texttt{contract-level} and an \texttt{identifier} is resolved in favor of \texttt{contract-level}.

\begin{verbatim}
template <typename T>
T prev(T v)
[[ expects : v > 0 ]]
[[ ensures audit r : r + 1 != v ]];
\end{verbatim}

http://eel.is/c++draft/dcl.attr.contract#syn-1
Contract attributes in C++20

9.11.4.1 Syntax

Contract attributes are used to specify preconditions, postconditions, and assertions for functions.

Syntax:

```
contract-attribute-specifier:
    [[ [ expects contract-level_opt : conditional-expression ] ]]
    [[ [ ensures contract-level_opt identifier_opt : conditional-expression ] ]]
    [[ [ assert contract-level_opt : conditional-expression ] ]]
```

contract-level:
  default
  audit
  axiom

An ambiguity between a `contract-level` and an `identifier` is resolved in favor of `contract-level`.

Example:

```
template <typename T>
T prev(T v)
[[ expects : v > 0 ]]
[[ ensures audit r : r + 1 == v ]];
```

http://eel.is/c++draft/dcl.attr.contract#syn-1

Name for return value to use in conditional expression

Post condition

@bjorn_fahller
Contract attributes in C++20

9.11.4.1 Syntax

Contract attributes are used to specify preconditions, postconditions, and assertions for functions.

contract-attribute-specifier:

- [[ expects contract-level_opt : conditional-expression ]]
- [[ ensures contract-level_opt identifier_opt : conditional-expression ]]
- [[ assert contract-level_opt : conditional-expression ]]

contract-level:
- default
- audit
- axiom

An ambiguity between a contract-level and an identifier is resolved in favor of contract-level.

http://eel.is/c++draft/dcl.attr.contract#syn-1
Contract attributes in C++20

9.11.4.1 Syntax

Contract attributes are used to specify preconditions, postconditions, and assertions for functions.

```
contract-attribute-specifier:
  [ [ expects contract-level_opt : conditional-expression ] ]
  [ [ ensures contract-level_opt identifier_opt : conditional-expression ] ]
  [ [ assert contract-level_opt : conditional-expression ] ]
```

An ambiguity between a `contract-level` and an `identifier` is resolved in favor of `contract-level`.

```
for (auto p : pointers) {
  [[ assert audit: p != nullptr ]];
  func(p);
}
```

http://eel.is/c++draft/dcl.attr.contract#syn-1
Contract attributes in C++20

9.11.4.1 Syntax

Contract attributes are used to specify preconditions, postconditions, and assertions for functions.

```
contract-attribute-specifier:
  [ [ expects contract-level opt : conditional-expression ] ]
  [ [ ensures contract-level opt : conditional-expression ] ]
  [ [ assert contract-level opt : conditional-expression ] ]
```

```
contract-level:
  default
  audit
  axiom
```

An ambiguity between a `contract-level` and an `identifier` is resolved in favor of `contract-level`.

http://eel.is/c++draft/dcl.attr.contract#syn-1
Using C++20 contract attributes for ringbuffer

template <typename T, int N>
class ringbuffer {
public:
    // invariant: size() ≥ 0 && size() ≤ N
    ringbuffer();
    // ensures: size() == 0
    int size() const;
    const T& back() const;
    // requires: size() > 0
    const T& front() const;
    // requires: size() > 0
    void push_back(T t);
    // requires: size() < N
    // ensures: size() == old size()+1
    //         back() == t
    T pop_front();
    // requires: size() > 0
    // ensures: size() == old size()-1
    //         return == old front();
};
Using C++20 contract attributes for ringbuffer

```cpp
template <typename T, int N>
class ringbuffer {
public:
    // invariant: size() ≥ 0 && size() ≤ N
    ringbuffer();
    // ensures: size() == 0
    int size() const;
    const T& back() const;
    // requires: size() > 0
    const T& front() const;
    void push_back(T t);
    // requires: size() < N
    // ensures: size() == old size()+1
    // back() == t
    T pop_front();
    // requires: size() > 0
    // ensures: size() == old size()-1
    // return == old front();
};
```
Using C++20 contract attributes for ringbuffer

```cpp
template <typename T, int N>
class ringbuffer {
public:
    // invariant: size() ≥ 0 && size() ≤ N
    ringbuffer();
    // ensures: size() == 0
    int size() const;
    const T& back() const;
    // requires: size() > 0
    const T& front() const;
    void push_back(T t);
    // requires: size() < N
    // ensures: size() == old size()+1
    // back() == t
    T pop_front();
    // requires: size() > 0
    // ensures: size() == old size()-1
    // return == old front();
};
```

No support for class invariants, so might as well leave as comment.
Using C++20 contract attributes for ringbuffer

```cpp
template <typename T, int N>
class ringbuffer {
public:
    // invariant: size() \geq 0 && size() \leq N
    ringbuffer();

    // ensures: size() == 0
    int size() const;
    const T& back() const;
    // requires: size() > 0
    const T& front() const;
    // requires: size() > 0
    void push_back(T t);
    // requires: size() < N
    // ensures: size() == old size()+1
    //      back() == t
    T pop_front();
    // requires: size() > 0
    // ensures: size() == old size()-1
    //      return == old front();
};
```
Using C++20 contract attributes for ringbuffer

template <typename T, int N>
class ringbuffer {
public:
    // invariant: size() ≥ 0 && size() ≤ N
    ringbuffer() [[ ensures: size() == 0 ]];

    int size() const;
    const T& back() const;
    // requires: size() > 0
    const T& front() const;
    // requires: size() > 0
    void push_back(T t);
    // requires: size() < N
    // ensures: size() == old size()+1
    //         back() == t
    T pop_front();
    // requires: size() > 0
    // ensures: size() == old size()-1
    //          return == old front();
};
Using C++20 contract attributes for ringbuffer

```cpp
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() ≥ 0 && size() ≤ N
  ringbuffer() [[ ensures: size() == 0 ]];
  int size() const;
  const T& back() const;
  // requires: size() > 0
  const T& front() const;
  // requires: size() > 0
  void push(T t);
  // requires: size() < N
  // ensures: size() /= old size()+1
  //          back() /= t
  T pop_front();
  // requires: size() > 0
  // ensures: size() /= old size()-1
  //          return /= old front();
};
```

<sourcex>:6:15: error: use of undeclared identifier 'size'
  [[ ensures: size() == 0 ]];
```
```
Using C++20 contract attributes for ringbuffer

```cpp
template <typename T, int N>
class ringbuffer {
    public:
        // invariant: size() >= 0 && size() <= N
        ringbuffer() [[ ensures: size() == 0 ]];
        int size() const;
        const T& back() const;
        // requires: size() > 0
        const T& front() const;
        // requires: size() > 0
        void push(T t);
        // requires: size() < N
        // ensures: size() /= old size()+1
        //          back() /= t
        T pop_front();
        // requires: size() > 0
        // ensures: size() /= old size()-1
        //          return /= old front();
};
```

Contract attributes are declarations that can only refer to identifiers seen earlier.
Using C++20 contract attributes for ringbuffer

```cpp
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() \geq 0 && size() \leq N
  int size() const;

  ringbuffer();
  [[ ensures: size() == 0 ]];

  const T& back() const;
  // requires: size() > 0

  const T& front() const;
  // requires: size() > 0

  void push_back(T t);
  // requires: size() < N
  // ensures: size() = old size()+1
  //         back() = t

  T pop_front();
  // requires: size() > 0
  // ensures: size() = old size()-1
  //         return = old front();
};
```

Contract attributes are declarations that can only refer to identifiers seen earlier.
Using C++20 contract attributes for ringbuffer

template <typename T, int N>
class ringbuffer {
public:
// invariant: size() \geq 0 \&\& size() \leq N
int size() const;
ringbuffer()
[[ ensures: size() == 0 ]];
const T& back() const;
// requires: size() > 0
const T& front() const;
// requires: size() > 0
void push_back(T t);
// requires: size() < N
// ensures: size() == old size()+1
// back() == t
T pop_front();
// requires: size() > 0
// ensures: size() == old size()-1
// return == old front();
};
Using C++20 contract attributes for ringbuffer

template <typename T, int N>
class ringbuffer {
public:
    // invariant: size() ≥ 0 && size() ≤ N
    int size() const;
    ringbuffer()
        [[ ensures: size() == 0 ]];
    const T& back() const
        [[ expects: size() > 0 ]];
    const T& front() const;
    // requires: size() > 0
    void push_back(T t);
    // requires: size() < N
    // ensures: size() == old size()+1
    //          back() == t
    T pop_front();
    // requires: size() > 0
    // ensures: size() == old size()-1
    //          return == old front();
};
Using C++20 contract attributes for ringbuffer

template <typename T, int N>
class ringbuffer {
public:
    // invariant: size() ≥ 0 && size() ≤ N
    int size() const;
    ringbuffer()
        [[ ensures: size() == 0 ]];
    const T& back() const
        [[ expects: size() > 0 ]];
    const T& front() const;
    // requires: size() > 0
    void push_back(T t);
    // requires: size() < N
    // ensures: size() == old size()+1
    // back() == t
    T pop_front();
    // requires: size() > 0
    // ensures: size() == old size()-1
    // return == old front();
};
Using C++20 contract attributes for ringbuffer

```cpp
template <typename T, int N>
class ringbuffer {
public:
    // invariant: size() >= 0 && size() <= N
    int size() const;
    ringbuffer()
        [[ ensures: size() == 0 ]];
    const T& back() const
        [[ expects: size() > 0 ]];
    const T& front() const
        [[ expects: size() > 0 ]];
    void push_back(T t);
        // requires: size() < N
        // ensures: size() == old size()+1
        //         back() == t
    T pop_front();
        // requires: size() > 0
        // ensures: size() == old size()-1
        //         return == old front();
};
```
Using C++20 contract attributes for ringbuffer

template <typename T, int N>
class ringbuffer {
public:
    // invariant: size() \geq 0 \&\& size() \leq N
    int size() const;
    ringbuffer()
        [[ ensures: size() = 0 ]];
    const T\& back() const
        [[ expects: size() > 0 ]];
    const T\& front() const
        [[ expects: size() > 0 ]];
    void push_back(T t);

    // requires: size() < N
    // ensures: size() = old size()+1
    //         back() = t
    T pop_front();

    // requires: size() > 0
    // ensures: size() = old size()-1
    //         return = old front();
};
Using C++20 contract attributes for ringbuffer

```cpp
template <typename T, int N>
class ringbuffer {
public:
    // invariant: size() \geq 0 \&\& size() \leq N
    int size() const;

    ringbuffer()
        [[ ensures: size() == 0 ]];

    const T& back() const
        [[ expects: size() > 0 ]];

    const T& front() const
        [[ expects: size() > 0 ]];

    void push_back(T t)
        [[ expects: size() < N ]];

    // ensures: size() == old size()+1
    //          back() == t
    T pop_front();

    // requires: size() > 0
    // ensures: size() == old size()-1
    //          return == old front();
};
```
Using C++20 contract attributes for ringbuffer

template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() ≥ 0 && size() ≤ N
  int size() const;
  ringbuffer()
      [[ ensures: size() == 0 ]];
  const T& back() const
      [[ expects: size() > 0 ]];
  const T& front() const
      [[ expects: size() > 0 ]];
  void push_back(T t)
      [[ expects: size() < N ]];
      // ensures: size() == old size()+1
      // back() == t
  T pop_front();
      // requires: size() > 0
      // ensures: size() == old size()-1
      // return == old front();
};
Using C++20 contract attributes for ringbuffer

```cpp
template <typename T, int N>
class ringbuffer {
public:
    // invariant: size() ≥ 0 && size() ≤ N
    int size() const;
    ringbuffer()
        [[ ensures: size() == 0 ]];
    const T& back() const
        [[ expects: size() > 0 ]];
    const T& front() const
        [[ expects: size() > 0 ]];
    void push_back(T t)
        [[ expects: size() < N ]];
        // ensures: size() == old size()+1
        // back() == t
    T pop_front();
        // requires: size() > 0
        // ensures: size() == old size()-1
        // return == old front();
};
```

There is no way to refer to previous state so this cannot be expressed!
Using C++20 contract attributes for ringbuffer

template <typename T, int N>
class ringbuffer {
  public:
    // invariant: size() ≥ 0 && size() ≤ N
    int size() const;
    ringbuffer()
      [[ ensures: size() == 0 ]];
    const T& back() const
      [[ expects: size() > 0 ]];
    const T& front() const
      [[ expects: size() > 0 ]];
    void push_back(T t)
      [[ expects: size() < N ]]
      [[ ensures: size() > 0 ]]; // incremented
      // back() == t
    T pop_front();
    // requires: size() > 0
    // ensures: size() == old size()-1
    // return == old front();
};
Using C++20 contract attributes for ringbuffer

template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() \geq 0 \&\& size() \leq N
  int size() const;

  ringbuffer()
  [[ ensures: size() == 0 ]];

  const T& back() const
  [[ expects: size() > 0 ]];

  const T& front() const
  [[ expects: size() > 0 ]];

  void push_back(T t)
  [[ expects: size() < N ]];
  [[ ensures: size() \geq 0 ]]; // incremented
  // back() = t

  T pop_front();
  // requires: size() > 0
  // ensures: size() = old size()-1
  // return = old front();
};
Using C++20 contract attributes for ringbuffer

```cpp
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() /= 0 /& size() /= N
  int size() const;
  ringbuffer();
[[ ensures: size() /= 0 ]] const T& back() const;
[[ expects: size() > 0 ]] const T& front() const;
[[ expects: size() < N ]] void push_back(T t);
[[ ensures: size() >/= 0 ]] T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  // return == old front();
};
```

6# If a function has multiple preconditions, their evaluation (if any) will be performed in the order they appear lexically. If a function has multiple postconditions, their evaluation (if any) will be performed in the order they appear lexically. [ Example: 

```cpp
void f(int * p)
[[ expects: p != nullptr ]] // #1
[[ ensures: *p == 1 ]] // #3
[[ expects: *p == 0 ]] // #2
{
  *p = 1;
}
```

— end example ]

http://eel.is/c++draft/dcl.attr.contract#cond-6
Using C++20 contract attributes for ringbuffer

```cpp
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() /= 0 /& size() /= N
  int size() const;
  ringbuffer();
  [[ ensures: size() /= 0 ]] const T& back() const;
  [[ expects: size() ]] const T& front() const;
  void push_back(T t) [[ expects: size() < N ]]
      [[ ensures: size() /= 0 ]] // incremented
      //          back() /= t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return /= old front();
};
```

6# If a function has multiple preconditions, their evaluation (if any) will be performed in the order they appear lexically. If a function has multiple postconditions, their evaluation (if any) will be performed in the order they appear lexically. [ Example:

```cpp
void f(int * p) [[ expects: p != nullptr ]] // #1
    [[ ensures: *p == 1 ]] // #3
    [[ expects: *p == 0 ]] // #2
{
  *p = 1;
}
```

— end example ]

http://eel.is/c++draft/dcl.attr.contract#cond-6
Using C++20 contract attributes for ringbuffer

```cpp
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() /= 0 /& size() /= N
  int size() const;
  ringbuffer();
  [[ ensures: size() ]] const T& back() const;
  [[ expects: size() ]] const T& front() const;
  [[ expects: size() ]] [[ ensures: size() > 0 ]] void push_back(T t); // incremented
  // requires: size() > 0
  // ensures: size() == old size()-1
  // return == old front();
  T pop_front();
};
```

7# If a postcondition odr-uses ([basic.def.odr]) a parameter in its predicate and the function body makes direct or indirect modifications of the value of that parameter, the behavior is undefined. [ Example :

```cpp
int f(int x)
  [[ensures r: r == x]]
{
  return ++x;
  // undefined behavior
}
```

http://eel.is/c++draft/dcl.attr.contract#cond-7
Using C++20 contract attributes for ringbuffer

```cpp
template <typename T, int N>
class ringbuffer {
  public:
    // invariant: size() /= 0 /& size() /= N
    int size() const;
    ringbuffer();
    [[ ensures: size() ]]
    const T& back() const;
    [[ expects: size() ]]
    const T& front() const;
    [[ expects: size() ]]
    [[ ensures: size() > 0 ]]; // incremented
    //          back() /= t
    T pop_front();
    // requires: size() > 0
    // ensures: size() /= old size()-1
    //          return /= old front();
};
```

7# If a postcondition odr-uses ([basic.def.odr]) a parameter in its predicate and the function body makes direct or indirect modifications of the value of that parameter, the behavior is undefined. [ Example:

```cpp
int f(int x)
  [[ensures r: r == x]]
{
  return ++x;
}
```

So the validity of the post condition declaration depends on how the function is implemented

http://eel.is/c++draft/dcl.attr.contract#cond-7
Using C++20 contract attributes for ringbuffer

```cpp
template <typename T, int N>
class ringbuffer {
public:
    // invariant: size() ≥ 0 && size() ≤ N
    int size() const;
    ringbuffer()
        [[ ensures: size() == 0 ]];
    const T& back() const
        [[ expects: size() > 0 ]];
    const T& front() const
        [[ expects: size() > 0 ]];
    void push_back(T t)
        [[ expects: size() < N ]]
        [[ ensures: size() > 0 ]] // incremented
        [[ ensures: back() == t ]];
    T pop_front();
    // requires: size() > 0
    // ensures: size() == old size()-1
    // return == old front();
};
```

Potentially dangerous
Using C++20 contract attributes for ringbuffer

template <typename T, int N>
class ringbuffer {
public:
   // invariant: size() ≥ 0 && size() ≤ N
   int size() const;
   ringbuffer()
   [[ ensures: size() = 0 ]];
   const T& back() const
   [[ expects: size() > 0 ]];
   const T& front() const
   [[ expects: size() > 0 ]];
   void push_back(T t)
   [[ expects: size() < N ]]
   [[ ensures: size() > 0 ]] // incremented
   [[ ensures: back() = t ]];
   T pop_front();
   // requires: size() > 0
   // ensures: size() = old size()-1
   // return = old front();
};
Using C++20 contract attributes for ringbuffer

```cpp
#include <assert.h>

using namespace std;

template <typename T, int N>
class ringbuffer {
public:
    // invariant: size() ≥ 0 && size() ≤ N
    int size() const;

    ringbuffer()
        [[ ensures: size() = 0 ]];

    const T& back() const
        [[ expects: size() > 0 ]];

    const T& front() const
        [[ expects: size() > 0 ]];

    void push_back(T t)
        [[ expects: size() < N ]]
        [[ ensures: size() > 0 ]] // incremented
        [[ ensures: back() = t ]];

    T pop_front()
        [[ ensures: size() > 0 ]];

    // ensures: size() = old size()-1
    //         return = old front();
};
```
Using C++20 contract attributes for ringbuffer

template <typename T, int N>
class ringbuffer {
public:
    // invariant: size() \geq 0 \&\& size() \leq N
    int size() const;
    ringbuffer()
        [[ ensures: size() == 0 ]];
    const T& back() const
        [[ expects: size() > 0 ]];
    const T& front() const
        [[ expects: size() > 0 ]];
    void push_back(T t)
        [[ expects: size() < N ]]
        [[ ensures: size() > 0 ]] // incremented
        [[ ensures: back() == t ]];
    T pop_front()
        [[ expects: size() > 0 ]];
        // ensures: size() == old size()-1
        // return == old front();
};
Using C++20 contract attributes for ringbuffer

```cpp
template <typename T, int N>
class ringbuffer {
public:
    // invariant: size() \geq 0 \&\& size() \leq N
    int size() const;
    ringbuffer() [[ ensures: size() == 0 ]];
    const T& back() const [[ expects: size() > 0 ]];
    const T& front() const [[ expects: size() > 0 ]];
    void push_back(T t) [[ expects: size() < N ]]
    [[ ensures: size() > 0 ]] // incremented
    [[ ensures: back() == t ]];
    T pop_front() [[ expects: size() > 0 ]]
    [[ ensures: size() < N ]]; // decremented
    // return = old front();
};
```
Using C++20 contract attributes for ringbuffer

```cpp
template <typename T, int N>
class ringbuffer {
public:
    // invariant: size() ≥ 0 && size() ≤ N
    int size() const;
    ringbuffer();
    [[ ensures: size() == 0 ]] const;
    const T& back() const
    [[ expects: size() > 0 ]];
    const T& front() const
    [[ expects: size() > 0 ]];
    void push_back(T t)
    [[ expects: size() < N ]]
    [[ ensures: size() > 0 ]] // incremented
    [[ ensures: back() == t ]];
    T pop_front()
    [[ expects: size() > 0 ]]
    [[ ensures: size() < N ]]; // decremented
    //          return /= old front();
};
```

Cannot express condition with previous state so might as well leave as comment

Programming with Contracts in C++20  –  ACCU 2019  © Björn Fahller
Virtual functions and contracts in C++20
Virtual functions and contracts in C++20

If an overriding function specifies contract conditions ([dcl.attr.contract]), it shall specify the same list of contract conditions as its overridden functions; no diagnostic is required if corresponding conditions will always evaluate to the same value. Otherwise, it is considered to have the list of contract conditions from one of its overridden functions; ...

http://eel.is/c++draft/class.virtual#19
Virtual functions and contracts in C++20

If an overriding function specifies contract conditions ([dcl.attr.contract]), it shall specify the same list of contract conditions as its overridden functions; no diagnostic is required if corresponding conditions will always evaluate to the same value. Otherwise, it is considered to have the list of contract conditions from one of its overridden functions; ...

http://eel.is/c++draft/class.virtual#19
Virtual functions and contracts in C++20

If an overriding function specifies contract conditions ([dcl.attr.contract]), it shall specify the same list of contract conditions as its overridden functions; no diagnostic is required if corresponding conditions will always evaluate to the same value. Otherwise, it is considered to have the list of contract conditions from one of its overridden functions; ...
Function pointers and contracts in C++20
# Note: A function pointer cannot include contract conditions. [Example:]

```cpp
typedef int (*fpt)(int) [[ensures r: r ≠ 0]]; 
    // error: contract condition not on a function declaration

int g(int x) [[expects: x ≥ 0]] [[ensures r: r > x]]
{
    return x+1;
}

int (*pf)(int) = g; // OK
int x = pf(5); // contract conditions of g are checked

— end example ] — end note ]
```

http://eel.is/c/++draft/dcl.attr.contract#cond-3
Function pointers and contracts in C++20

3  

#[ Note: A function pointer cannot include contract conditions. [ Example:

typedef int (*fpt)(int) [[ensures r: r ≠ 0]];
  // error: contract condition not on a function declaration

int g(int x) [[expects: x ≥ 0]] [[ensures r: r > x]]
{
    return x+1;
}

int (*pf)(int) = g;
int x = pf(5);
  // OK
  // contract conditions of g are checked

— end example ] — end note ]

http://eel.is/c++draft/dcl.attr.contract#cond-3

In other words, it is the responsibility of a function implementation to enforce its contracts, not the caller.
Let’s explore!

https://github.com/arcosuc3m/clang-contracts

Fork from clang-6

http://fragata.arcos.inf.uc3m.es/#
Policing contracts in C++20
Policing contracts in C++20

3# A translation may be performed with one of the following build levels: **off**, **default**, or **audit**. A translation with build level set to **off** performs no checking for any contract. A translation with build level set to **default** performs checking for default contracts. A translation with build level set to **audit** performs checking for default and audit contracts. If no build level is explicitly selected, the build level is **default**. The mechanism for selecting the build level is implementation-defined. The translation of a program consisting of translation units where the build level is not the same in all translation units is conditionally-supported. There should be no programmatic way of setting, modifying, or querying the build level of a translation unit.

http://eel.is/c++/draft/dcl.attr.contract#check-3
Policing contracts in C++20

A translation may be performed with one of the following build levels: `off`, `default`, or `audit`. A translation with build level set to `off` performs no checking for any contract. A translation with build level set to `default` performs checking for default contracts. A translation with build level set to `audit` performs checking for default and audit contracts. If no build level is explicitly selected, the build level is `default`. The mechanism for selecting the build level is implementation-defined. The translation of a program consisting of translation units where the build level is not the same in all translation units is conditionally-supported. There should be no programmatic way of setting, modifying, or querying the build level of a translation unit.

http://eel.is/c++draft/dcl.attr.contract#check-3
Policing contracts in C++20

A translation may be performed with one of the following build levels: off, default, or audit. A translation with build level set to off performs no checking for any contract. A translation with build level set to default performs checking for default contracts. A translation with build level set to audit performs checking for default and audit contracts. If no build level is explicitly selected, the build level is default. The mechanism for selecting the build level is implementation-defined. The translation of a program consisting of translation units where the build level is not the same in all translation units is conditionally-supported. There should be no programmatic way of setting, modifying, or querying the build level of a translation unit.

http://eel.is/c++draft/dcl.attr.contract#check-3
A translation may be performed with one of the following build levels: off, default, or audit. A translation with build level set to off performs no checking for any contract. A translation with build level set to default performs checking for default contracts. A translation with build level set to audit performs checking for default and audit contracts. If no build level is explicitly selected, the build level is default. The mechanism for selecting the build level is implementation-defined. The translation of a program consisting of translation units where the build level is not the same in all translation units is conditionally-supported. There should be no programmatic way of setting, modifying, or querying the build level of a translation unit.

http://eel.is/c++draft/dcl.attr.contract#check-3
A translation may be performed with one of the following build levels: off, default, or audit. A translation with build level set to off performs no checking for any contract. A translation with build level set to default performs checking for default contracts. A translation with build level set to audit performs checking for default and audit contracts. If no build level is explicitly selected, the build level is default. The mechanism for selecting the build level is implementation-defined. The translation of a program consisting of translation units where the build level is not the same in all translation units is conditionally-supported. There should be no programmatic way of setting, modifying, or querying the build level of a translation unit.

http://eel.is/c++draft/dcl.attr.contract#check-3
A translation may be performed with one of the following build levels: off, default, or audit. A translation with build level set to off performs no checking for any contract. A translation with build level set to default performs checking for default contracts. A translation with build level set to audit performs checking for default and audit contracts. If no build level is explicitly selected, the build level is default. The mechanism for selecting the build level is implementation-defined. The translation of a program consisting of translation units where the build level is not the same in all translation units is conditionally-supported. There should be no programmatic way of setting, modifying, or querying the build level of a translation unit.

http://eel.is/c++draft/dcl.attr.contract#check-3
Policing contracts in C++20

3# A translation may be performed with one of the following build levels: off, default, or audit. A translation with build level set to off performs no checking for any contract. A translation with build level set to default performs checking for default contracts. A translation with build level set to audit performs checking for default and audit contracts. If no build level is explicitly selected, the build level is default. The mechanism for selecting the build level is implementation-defined. The translation of a program consisting of translation units where the build level is not the same in all translation units is conditionally-supported. There should be no programmatic way of setting, modifying, or querying the build level of a translation unit.

http://eel.is/c++draft/dcl.attr.contract#check-3
Policing contracts in C++20

3# A translation may be performed with one of the following build levels:

- off
- default
- audit

A translation with build level set to off performs no checking for any contract. A translation with build level set to default performs checking for default contracts. A translation with build level set to audit performs checking for default and audit contracts. If no build level is explicitly selected, the build level is default. The mechanism for selecting the build level is implementation-defined. The translation of a program consisting of translation units where the build level is not the same in all translation units is conditionally-supported. There should be no programmatic way of setting, modifying, or querying the build level of a translation unit.

http://eel.is/c/+draft/intro.defs#defns.cond.supp

http://eel.is/c/+draft/dcl.attr.contract#check-3
A translation may be performed with one of the following build levels: off, default, or audit. A translation with build level set to off performs no checking for any contract. A translation with build level set to default performs checking for default contracts. A translation with build level set to audit performs checking for default and audit contracts. If no build level is explicitly selected, the build level is default. The mechanism for selecting the build level is implementation-defined. The translation of a program consisting of translation units where the build level is not the same in all translation units is conditionally-supported. There should be no programmatic way of setting, modifying, or querying the build level of a translation unit.

http://eel.is/c++draft/dcl.attr.contract#check-3
Let’s explore!

https://github.com/arcosuc3m/clang-contracts

Fork from clang-6

http://fragata.arcos.inf.uc3m.es/#
Let’s explore!

https://github.com/arcosuc3m/clang-contracts

Fork from clang-6

http://fragata.arcos.inf.uc3m.es/#

-build-level=(off|default|audit)
When contracts are violated in C++20
When contracts are violated in C++20

5# The violation handler of a program is a function of type “noexcept_opt function of (lvalue reference to const std::contract_violation) returning void”. The violation handler is invoked when the predicate of a checked contract evaluates to false (called a contract violation). There should be no programmatic way of setting or modifying the violation handler. It is implementation-defined how the violation handler is established for a program and how the std::contract_violation argument value is set, except as specified below. If a precondition is violated, the source location of the violation is implementation-defined. [Note: Implementations are encouraged but not required to report the caller site. —end note] If a postcondition is violated, the source location of the violation is the source location of the function definition. If an assertion is violated, the source location of the violation is the source location of the statement to which the assertion is applied.

http://eel.is/c++draft/dcl.attr.contract#check-5
When contracts are violated in C++20

The violation handler of a program is a function of type “`noexcept_opt function of (lvalue reference to const std::contract_violation) returning void`”. The violation handler is invoked when the predicate of a checked contract evaluates to `false` (called a contract violation). There should be no programmatic way of setting or modifying the violation handler. It is implementation-defined how the violation handler is established for a program and how the `std::contract_violation` argument value is set, except as specified below. If a precondition is violated, the source location of the violation is implementation-defined. [Note: Implementations are encouraged but not required to report the caller site. — end note] If a postcondition is violated, the source location of the violation is the source location of the function definition. If an assertion is violated, the source location of the violation is the source location of the statement to which the assertion is applied.

http://eel.is/c++draft/dcl.attr.contract#check-5
When contracts are violated in C++20

The violation handler of a program is a function of type “\texttt{\textcolor{purple}{\texttt{noexcept}} \textcolor{green}{\texttt{opt}} \texttt{function of (lvalue reference to \texttt{const} std::contract\_violation) returning \texttt{void}}”

The violation handler is invoked when the predicate of a checked contract evaluates to \texttt{false} (called a contract violation). There should be no programmatic way of setting or modifying the violation handler. It is implementation-defined how the violation handler is established for a program and how the \texttt{std::contract\_violation} argument value is set, except as specified below. If a precondition is violated, the source location of the violation is implementation-defined. [\textbf{Note}: Implementations are encouraged but not required to report the caller site. — \textbf{end note}] If a postcondition is violated, the source location of the violation is the source location of the function definition. If an assertion is violated, the source location of the violation is the source location of the statement to which the assertion is applied.

\begin{verbatim}
16.8.2 Class contract\_violation [support.contract.cviol]
namespace std {
    class contract\_violation {
        public:
            uint\_least32\_t line\_number() const noexcept;
            string\_view file\_name() const noexcept;
            string\_view function\_name() const noexcept;
            string\_view comment() const noexcept;
            string\_view assertion\_level() const noexcept;
    };
}
\end{verbatim}

\url{http://eel.is/c++draft/support.contract.cviol}
\url{http://eel.is/c++draft/dcl.attr.contract#check-5}
When contracts are violated in C++20

5# The violation handler of a program is a function of type “\texttt{\texttt{noexcept}}\textsubscript{opt} \texttt{function of (lvalue reference to \texttt{const std::contract\textunderscore violation}) returning \texttt{void}}”. The violation handler is invoked when the predicate of a checked contract evaluates to \texttt{false} (called a contract violation). There should be no programmatic way of setting or modifying the violation handler. It is implementation-defined how the violation handler is established for a program and how the \texttt{std::contract\textunderscore violation} argument value is set, except as specified below. If a precondition is violated, the source location of the violation is implementation-defined. [\textit{Note}: Implementations are encouraged but not required to report the caller site. — \textit{end note} ] If a postcondition is violated, the source location of the violation is the source location of the function definition. If an assertion is violated, the source location of the violation is the source location of the statement to which the assertion is applied.

http://eel.is/c++draft/dcl.attr.contract#check-5
When contracts are violated in C++20

5# The violation handler of a program is a function of type “\texttt{\texttt{noexcept}}\opt\texttt{ function of (lvalue reference to \texttt{const std::contract\_violation}) returning \texttt{void}}”. The violation handler is invoked when the predicate of a checked contract evaluates to \texttt{false} (called a contract violation). \textbf{There should be no programmatic way of setting or modifying the violation handler.} It is implementation-defined how the violation handler is established for a program and how the \texttt{std::contract\_violation} argument value is set, except as specified below. If a precondition is violated, the source location of the violation is implementation-defined. \textit{[Note: Implementations are encouraged but not required to report the caller site. — \textit{end note}]} If a postcondition is violated, the source location of the violation is the source location of the function definition. If an assertion is violated, the source location of the violation is the source location of the statement to which the assertion is applied.

\texttt{\url{http://eel.is/c++draft/dcl.attr.contract#check-5}}
When contracts are violated in C++20

5# The violation handler of a program is a function of type “`noexcept` function of (lvalue reference to `const std::contract_violation`) returning `void`”. The violation handler is invoked when the predicate of a checked contract evaluates to `false` (called a contract violation). There should be no programmatic way of setting or modifying the violation handler. It is implementation-defined how the violation handler is established for a program and how the `std::contract_violation` argument value is set, except as specified below. If a precondition is violated, the source location of the violation is implementation-defined. [Note: Implementations are encouraged but not required to report the caller site. — end note] If a postcondition is violated, the source location of the violation is the source location of the function definition. If an assertion is violated, the source location of the violation is the source location of the statement to which the assertion is applied.

http://eel.is/c++draft/dcl.attr.contract#check-5
Let’s explore!

https://github.com/arcoSuc3m/clang-contracts

Fork from clang-6

http://fragata.arco.inf.uc3m.es/#

-\texttt{build-level=(off|default|audit)}
Let’s explore!

https://github.com/arcosuc3m/clang-contracts

Fork from clang-6

http://fragata.arcos.inf.uc3m.es/

- build-level=(off|default|audit)
- contract-violation-handler=function
When contracts are violated in C++20
When contracts are violated in C++20

A translation may be performed with one of the following violation continuation modes: off or on. A translation with violation continuation mode set to off terminates execution by invoking the function std::terminate ([except.terminate]) after completing the execution of the violation handler. A translation with a violation continuation mode set to on continues execution after completing the execution of the violation handler. If no continuation mode is explicitly selected, the default continuation mode is off. [Note: A continuation mode set to on provides the opportunity to install a logging handler to instrument a pre-existing code base and fix errors before enforcing checks. — end note]

http://eel.is/c++draft/dcl.attr.contract#check-7
A translation may be performed with one of the following violation continuation modes: \textit{off} or \textit{on}. A translation with violation continuation mode set to \textit{off} terminates execution by invoking the function \texttt{std::terminate ([except.terminate])} after completing the execution of the violation handler. A translation with a violation continuation mode set to \textit{on} continues execution after completing the execution of the violation handler. If no continuation mode is explicitly selected, the default continuation mode is \textit{off}. [\textbf{Note}: A continuation mode set to \textit{on} provides the opportunity to install a logging handler to instrument a pre-existing code base and fix errors before enforcing checks. — \textit{end note}]
When contracts are violated in C++20

A translation may be performed with one of the following violation continuation modes: off or on. A translation with violation continuation mode set to off terminates execution by invoking the function `std::terminate ([except.terminate])` after completing the execution of the violation handler. A translation with a violation continuation mode set to on continues execution after completing the execution of the violation handler. If no continuation mode is explicitly selected, the default continuation mode is off. [Note: A continuation mode set to on provides the opportunity to install a logging handler to instrument a pre-existing code base and fix errors before enforcing checks. — end note ]

http://eel.is/c++draft/dcl.attr.contract#check-7
When contracts are violated in C++20

A translation may be performed with one of the following violation continuation modes: *off* or *on*. A translation with violation continuation mode set to *off* terminates execution by invoking the function `std::terminate` ([except.terminate]) after completing the execution of the violation handler. A translation with a violation continuation mode set to *on* continues execution after completing the execution of the violation handler. If no continuation mode is explicitly selected, the default continuation mode is *off*. [Note: A continuation mode set to *on* provides the opportunity to install a logging handler to instrument a pre-existing code base and fix errors before enforcing checks. — *end note*]
When contracts are violated in C++20

A translation may be performed with one of the following violation continuation modes: off or on. A translation with violation continuation mode set to off terminates execution by invoking the function std::terminate() after completing the execution of the violation handler. A translation with a violation continuation mode set to on continues execution after completing the execution of the violation handler. If no continuation mode is explicitly selected, the default continuation mode is off. [Note: A continuation mode set to on provides the opportunity to install a logging handler to instrument a pre-existing code base before enforcing checks. — end note]

Example:

```cpp
void f(int x) [[expects: x > 0]];

void g() {
  f(0); // std::terminate() after handler if
  // continuation mode is off;
  // proceeds after handler if
  // continuation mode is on
  /* ... */
}
—end example
```

http://eel.is/c++draft/dcl.attr.contract#check-7
Programming with Contracts in C++20

Björn Fahller
Summary

- Design by contract is a way to clarify the responsibility between a function implementation and its callers.
Summary

- Design by contract is a way to clarify the responsibility between a function implementation and its callers.
- Language support is coming in C++20
Summary

- Design by contract is a way to clarify the responsibility between a function implementation and its callers.
- Language support is coming in C++20
  - But it’s lacking class invariants
Summary

• Design by contract is a way to clarify the responsibility between a function implementation and its callers.
• Language support is coming in C++20
  • But it’s lacking class invariants
  • and post conditions cannot refer to pre-call state.
Summary

- Design by contract is a way to clarify the responsibility between a function implementation and its callers.
- Language support is coming in C++20
  - But it’s lacking class invariants
  - and post conditions cannot refer to pre-call state.
- Interesting gotchas:
  - Modifying parameter values, and template specializations comes to mind.
Summary

- Design by contract is a way to clarify the responsibility between a function implementation and its callers.
- Language support is coming in C++20
  - But it’s lacking class invariants
  - and post conditions cannot refer to pre-call state.
- Interesting gotchas:
  - Modifying parameter values, and template specializations comes to mind.
- Contracts can be used by static analysis tools and the optimizer.
Summary

- Design by contract is a way to clarify the responsibility between a function implementation and its callers.
- Language support is coming in C++20
  - But it’s lacking class invariants
  - and post conditions cannot refer to pre-call state.
- Interesting gotchas:
  - Modifying parameter values, and template specializations comes to mind.
- Contracts can be used by static analysis tools and the optimizer.
- Configurable levels of contracts, e.g. full in debug builds, only cheap ones in release.
Summary

- Design by contract is a way to clarify the responsibility between a function implementation and its callers.
- Language support is coming in C++20
  - But it’s lacking class invariants
  - and post conditions cannot refer to pre-call state.
- Interesting gotchas:
  - Modifying parameter values, and template specializations comes to mind.
- Contracts can be used by static analysis tools and the optimizer.
- Configurable levels of contracts, e.g. full in debug builds, only cheap ones in release.
- Prefer to express semantics using the type system, if you can.
Summary

- Design by contract is a way to clarify the responsibility between a function implementation and its callers.
- Language support is coming in C++20.
- But it's lacking class invariants and post conditions cannot refer to pre-call state.
- Interesting gotchas:
  - Modifying parameter values, and template specializations come to mind.
- Contracts can be used by static analysis tools and the optimizer.
- Configurable levels of contracts, e.g. full in debug builds, only cheap ones in release.
- Prefer to express semantics using the type system, if you can.

Play with it!

https://github.com/arcosuc3m/clang-contracts

Fork from clang-6

http://fragata.arcos.inf.uc3m.es/#
Programming with Contracts in C++20

Björn Fahller

bjorn@fahller.se

@bjorn_fahller

@rollbear