Better Code:

Contracts

SEAN PARENT &
DAVE ABRAHAMS

Cppcon
The C++ Conference

2023

October 01 - 06

Better Code: Contracts

Dave Abrahams & Sean Parent

What's holding our software together? Can we do better than duct tape and good intentions?

Artwork by Dan Zucco

Adobe's Software Technology Lab

Sean Parent

Senior Principal Scientist
Manager, Software Technology Lab
Adobe Veteran

Nick DeMarco

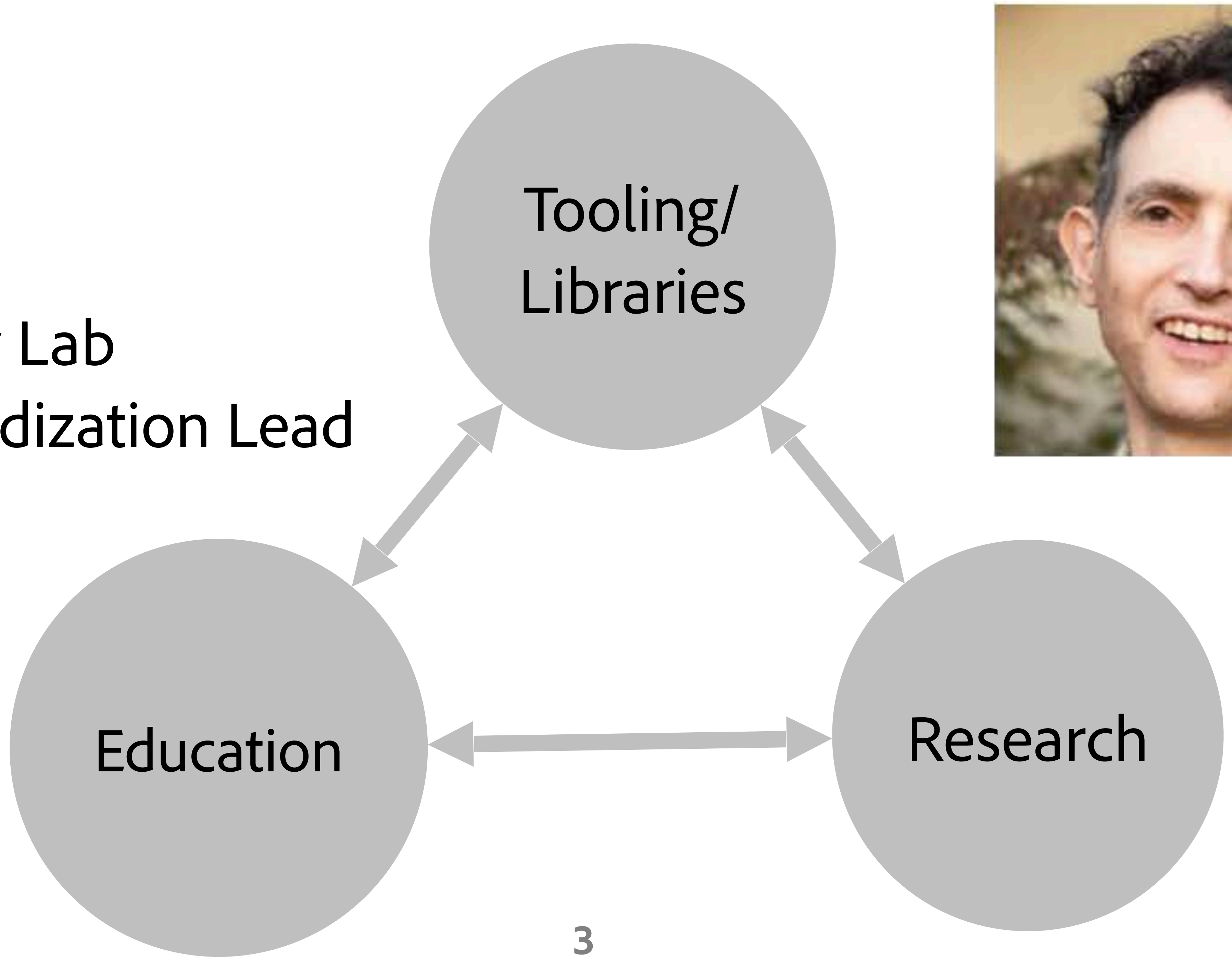
Senior Computer Scientist
Software Technology Lab
Photoshop iPad Async Dev

David Sankel

Principal Scientist
Software Technology Lab
Adobe's C++ Standardization Lead

Dave Abrahams

Principal Scientist
Software Technology Lab
Hylo Language Co-creator



Documentation > Code

Local reasoning

Local reasoning is the idea that the reader can make sense of the code directly in front of them, without going on a journey discovering how the code works.

—Nathan Gitter

(https://medium.com/@nathangitter/local-reasoning-in-swift-6782e459d)

Local reasoning | The tower of abstraction

Top of the tower

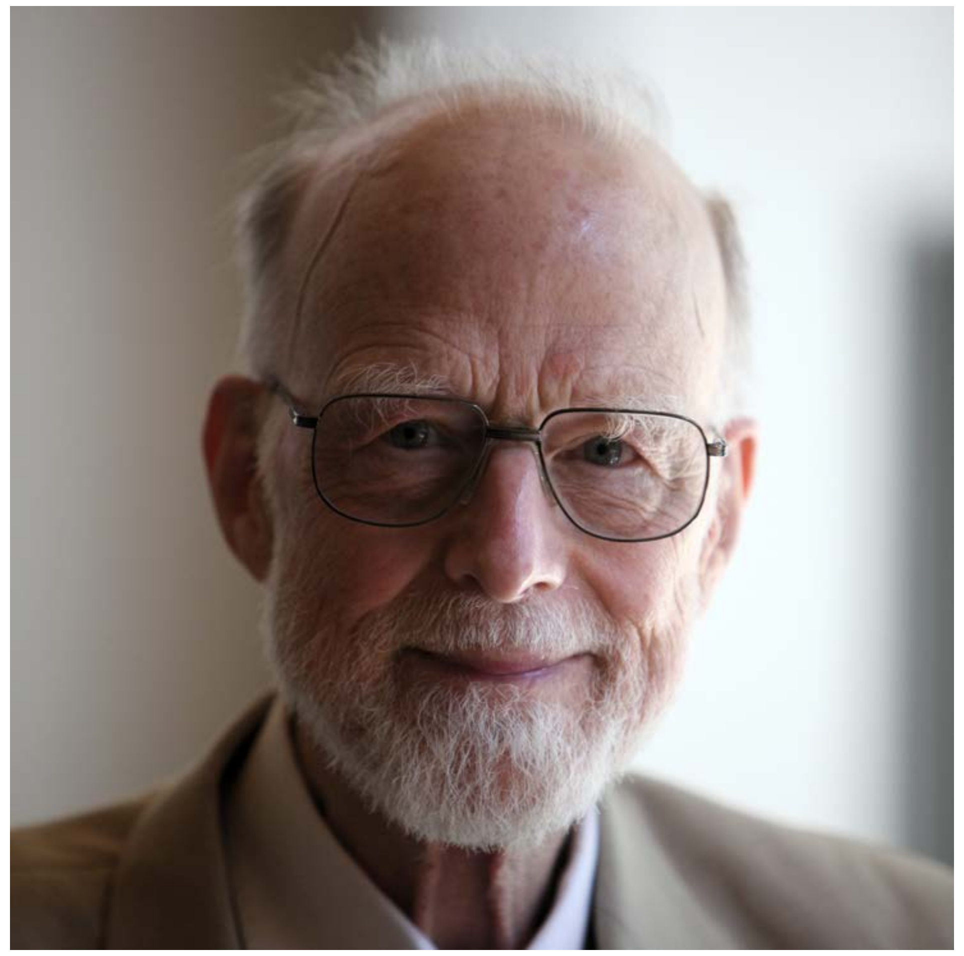
Top of the tower

What's in a contract?

Hoare Logic | Preconditions and Postconditions

$$\{P\}C\{Q\}$$

If precondition P is met, executing C establishes postcondition Q

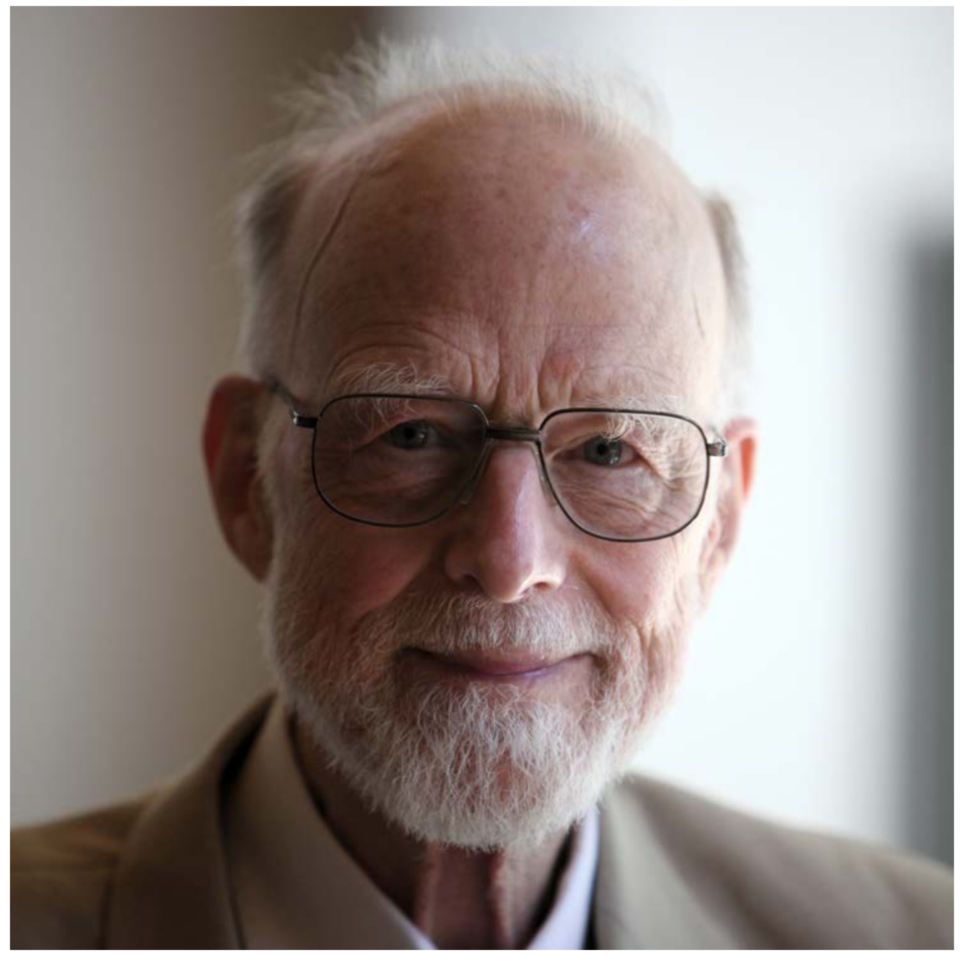


Tony Hoare

Hoare Logic | Preconditions and Postconditions

$$\{P\}C\{Q\}$$

If precondition P is met, executing C establishes postcondition Q

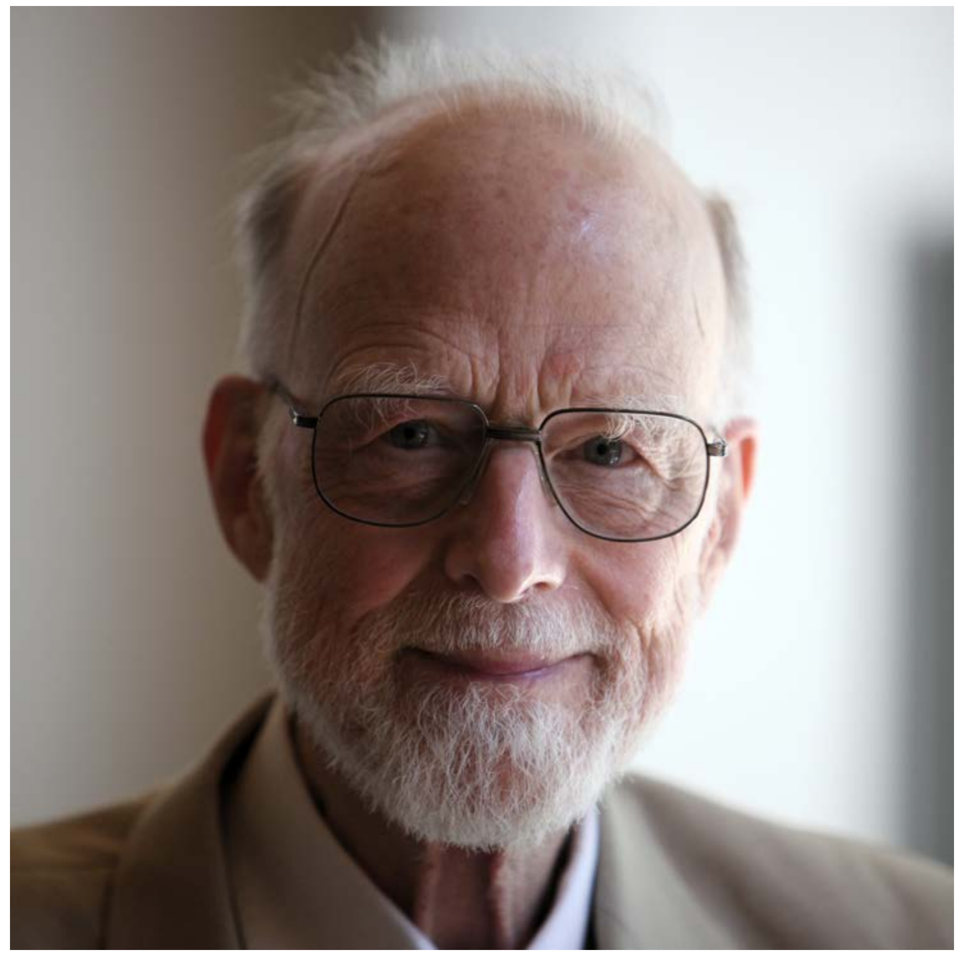


Tony Hoare

Hoare Logic | Preconditions and Postconditions

$$\{P\}C\{Q\}$$

If precondition P is met, executing C establishes postcondition Q



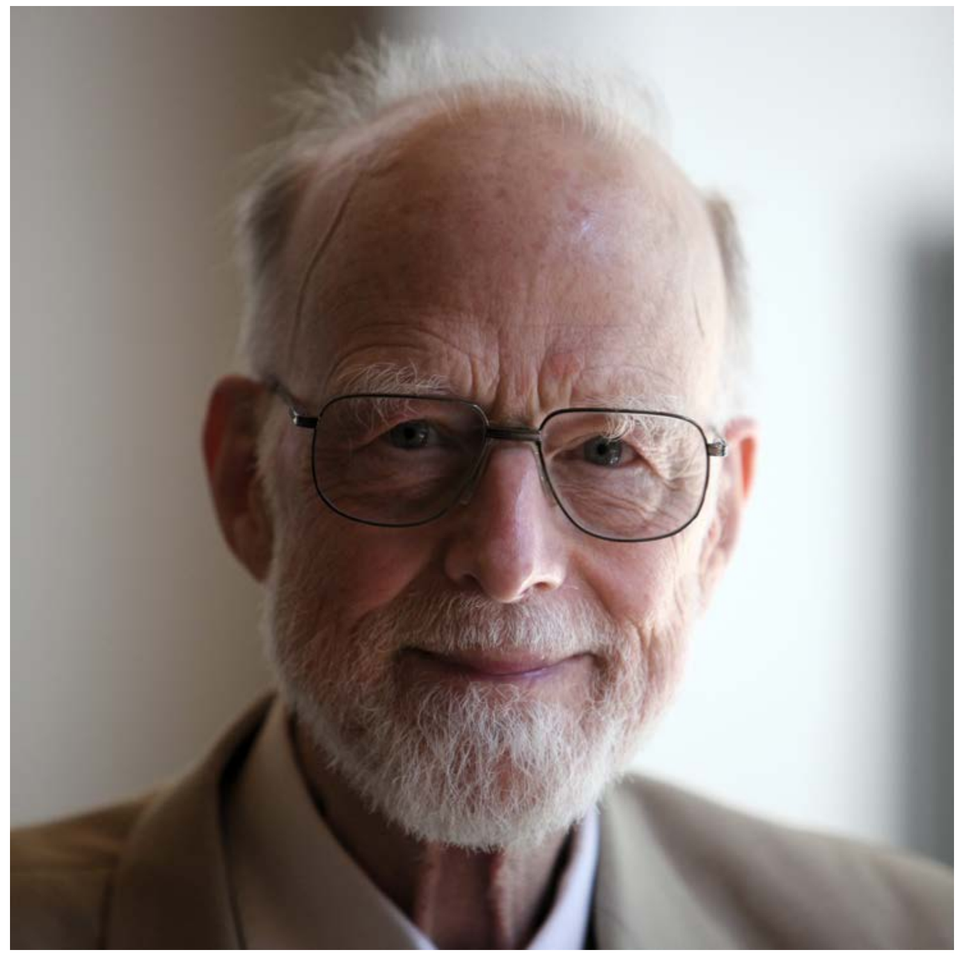
Tony Hoare

{x > 0, x < INT_MAX}   y = x + 1   {y > 1}

Hoare Logic | Preconditions and Postconditions

$$\{P\}C\{Q\}$$

If precondition P is met, executing C establishes postcondition Q



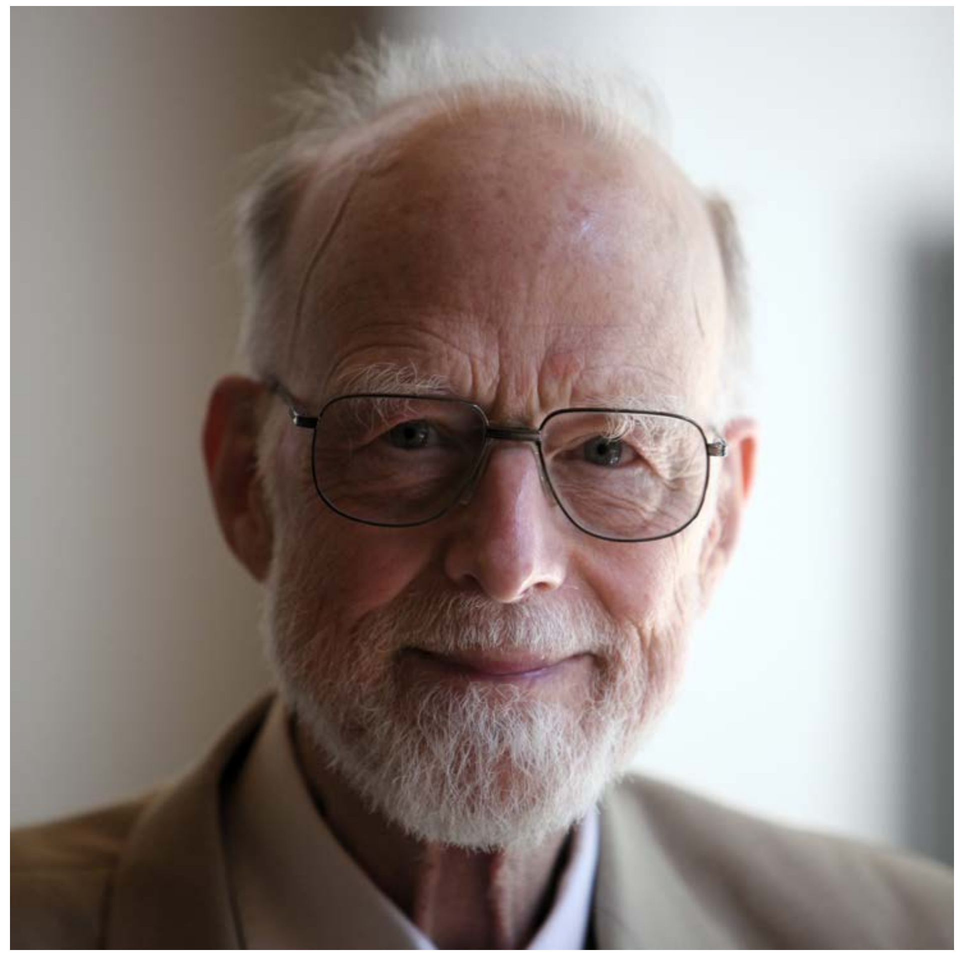
Tony Hoare

```
{x > 0, x < INT_MAX}  y = x + 1  {y > 1}
```

Hoare Logic | Preconditions and Postconditions

$$\{P\}C\{Q\}$$

If precondition P is met, executing C establishes postcondition Q



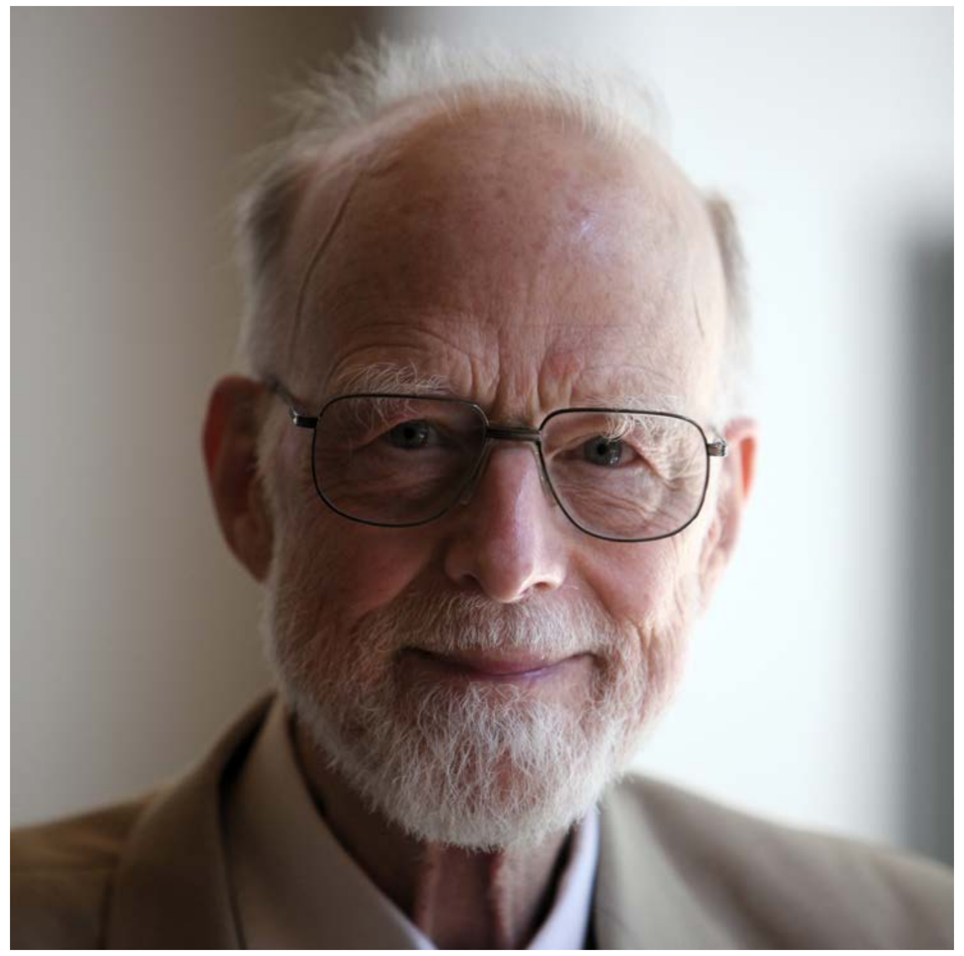
Tony Hoare

{`x > 0`, `x < INT_MAX`} `y = x + 1` {`y > 1`}

Hoare Logic | Preconditions and Postconditions

$$\{P\}C\{Q\}$$

If precondition P is met, executing C establishes postcondition Q



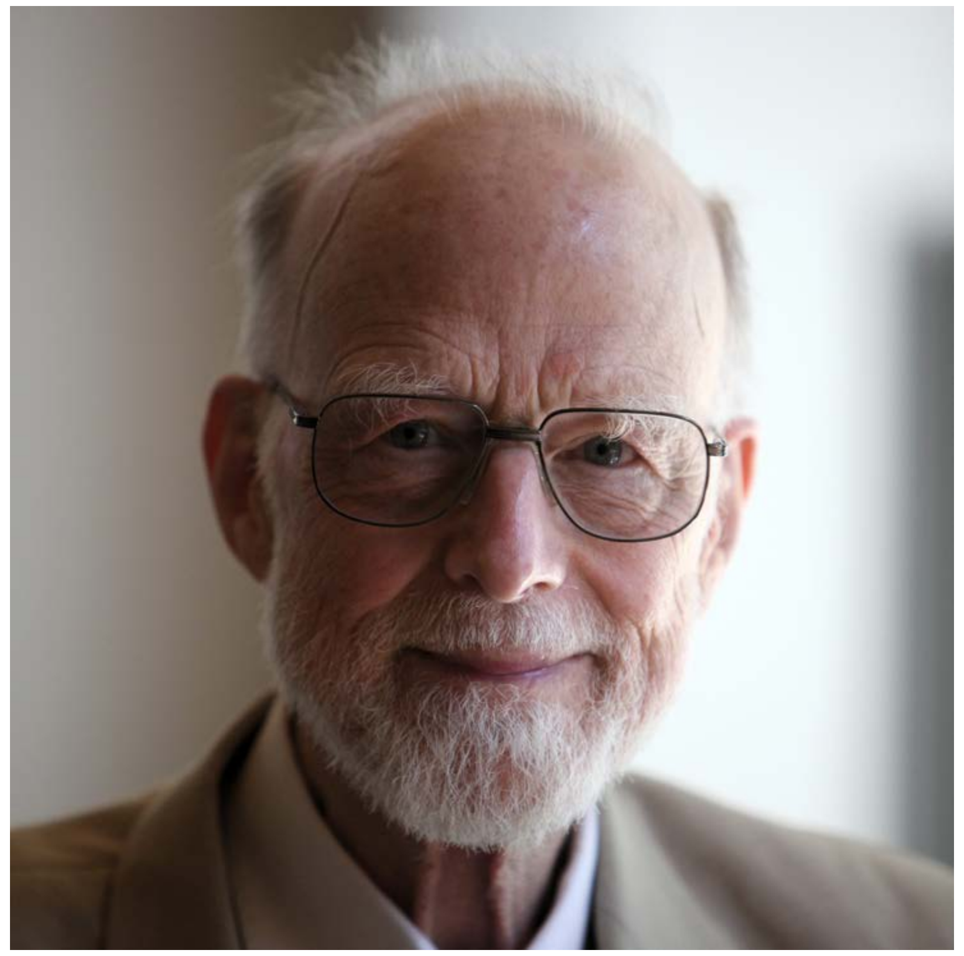
Tony Hoare

{x > 0, x < INT_MAX} `y = x + 1` {y > 1}

Hoare Logic | Preconditions and Postconditions

$$\{P\}C\{Q\}$$

If precondition P is met, executing C establishes postcondition Q



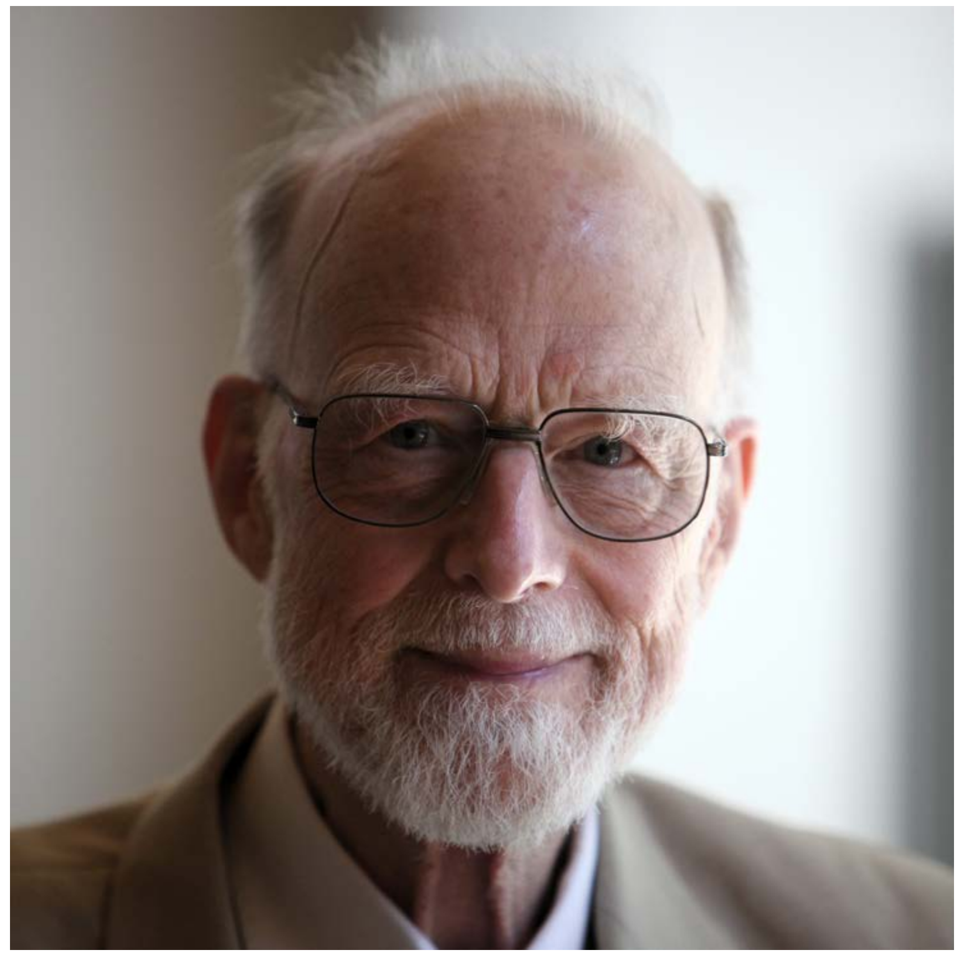
Tony Hoare

{`x > 0`, `x < INT_MAX`} `y = x + 1` {`y > 1`}

Hoare Logic | Preconditions and Postconditions

$$\{P\}C\{Q\}$$

If precondition P is met, executing C establishes postcondition Q



Tony Hoare

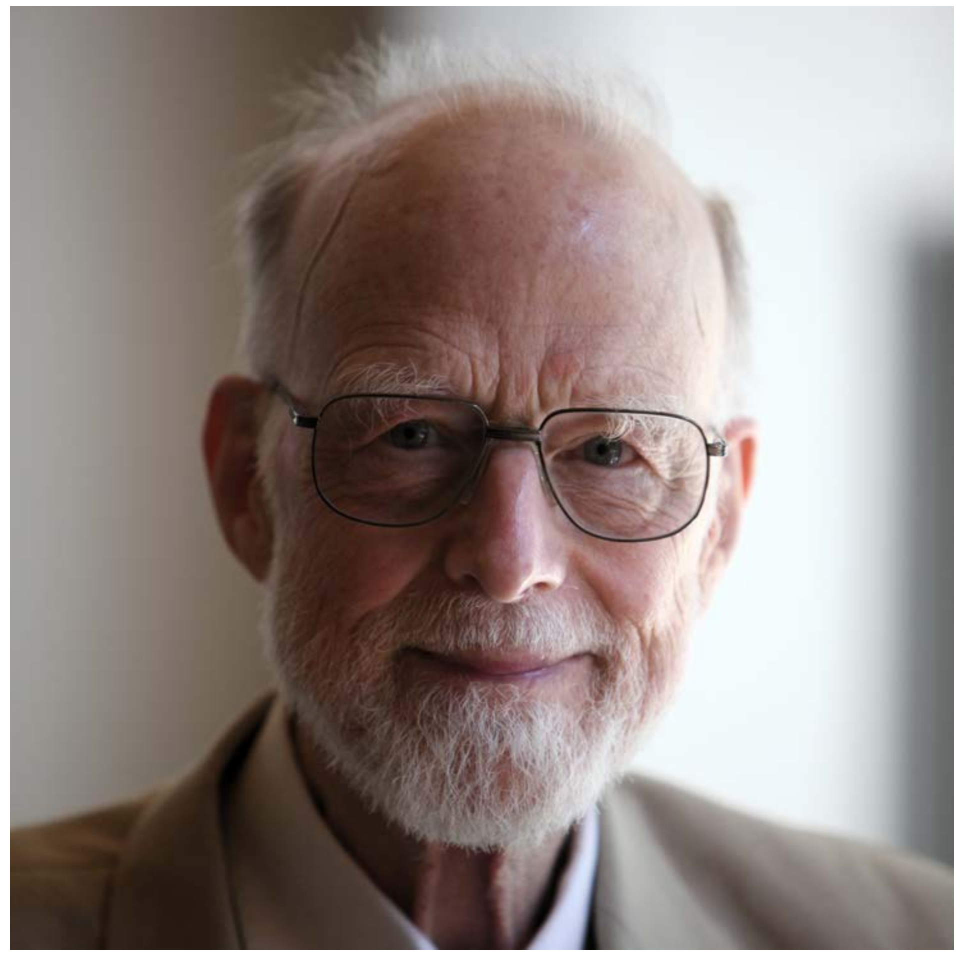
{x > 0, x < INT_MAX} `y = x + 1` {y > 1}

$$\frac{\{P\}S\{Q\},\{Q\}T\{R\}}{\{P\}S;T\{R\}}$$

Hoare Logic | Preconditions and Postconditions

$$\{P\}C\{Q\}$$

If precondition P is met, executing C establishes postcondition Q



Tony Hoare

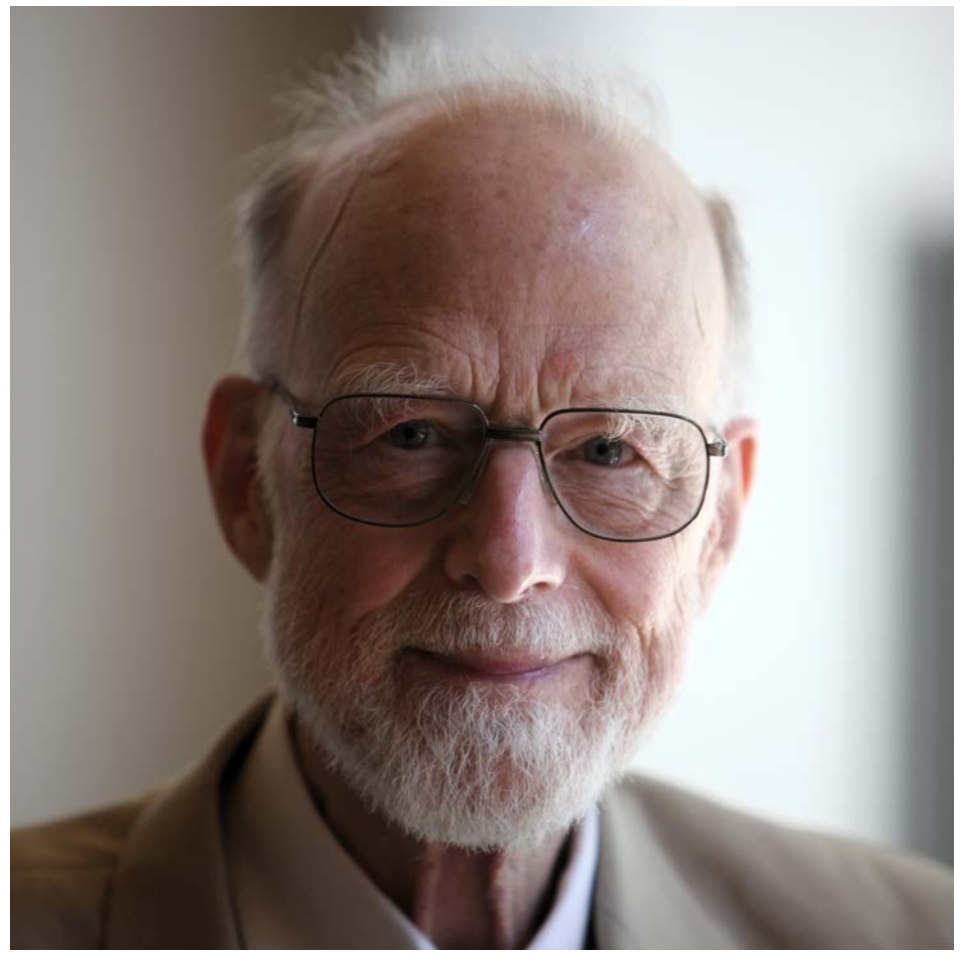
{x > 0, x < INT_MAX}   y = x + 1   {y > 1}

$$\frac{\{P\}S\{Q\},\ \{Q\}T\{R\}}{\{P\}S;T\{R\}}$$

Hoare Logic | Preconditions and Postconditions

$$\{P\}C\{Q\}$$

If precondition P is met, executing C establishes postcondition Q

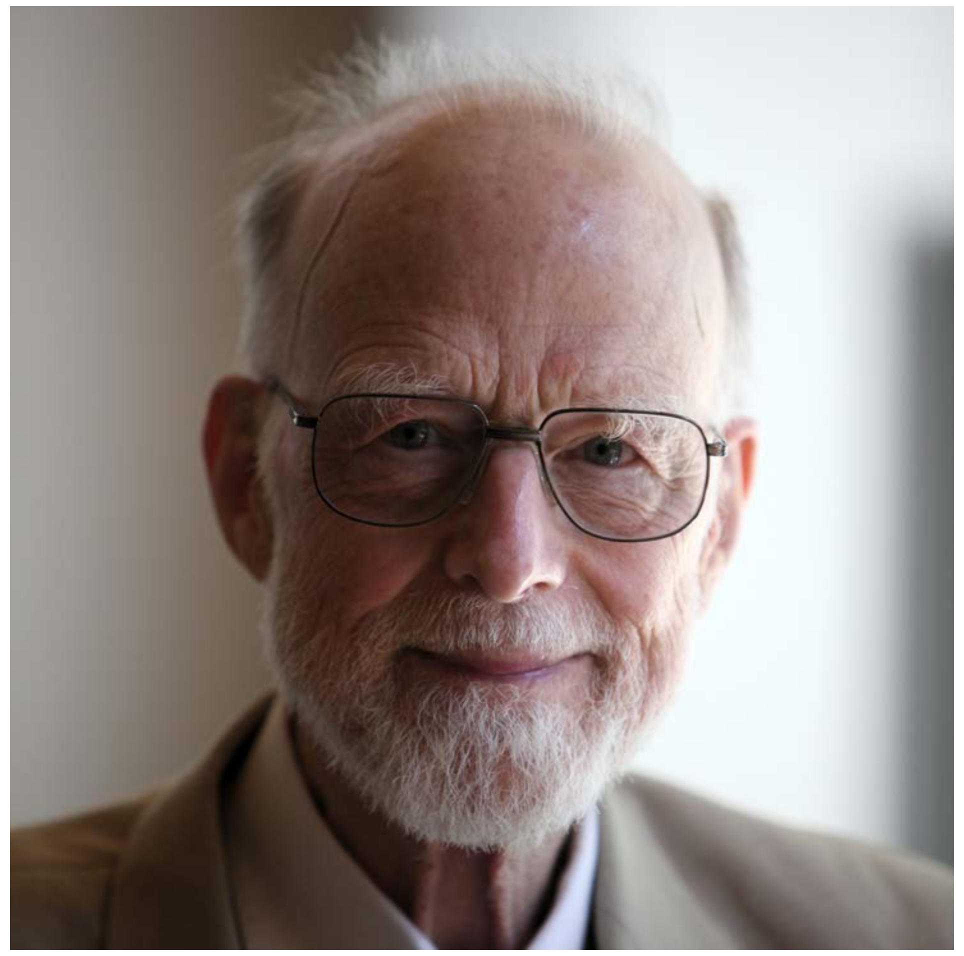


Tony Hoare

$\{$`x > 0`, `x < INT_MAX`$\}$ `y = x + 1` $\{$`y > 1`$\}$

$$\frac{\{P\}S\{Q\},\ \{Q\}T\{R\}}{\{P\}S;T\{R\}}$$

Hoare Logic | Preconditions and Postconditions



Tony Hoare

$$\{P\}C\{Q\}$$

If precondition P is met, executing C establishes postcondition Q

```
{x > 0, x < INT_MAX}  y = x + 1  {y > 1}
```

$$\frac{\{P\}S\{Q\}, \{Q\}T\{R\}}{\{P\}S;T\{R\}}$$

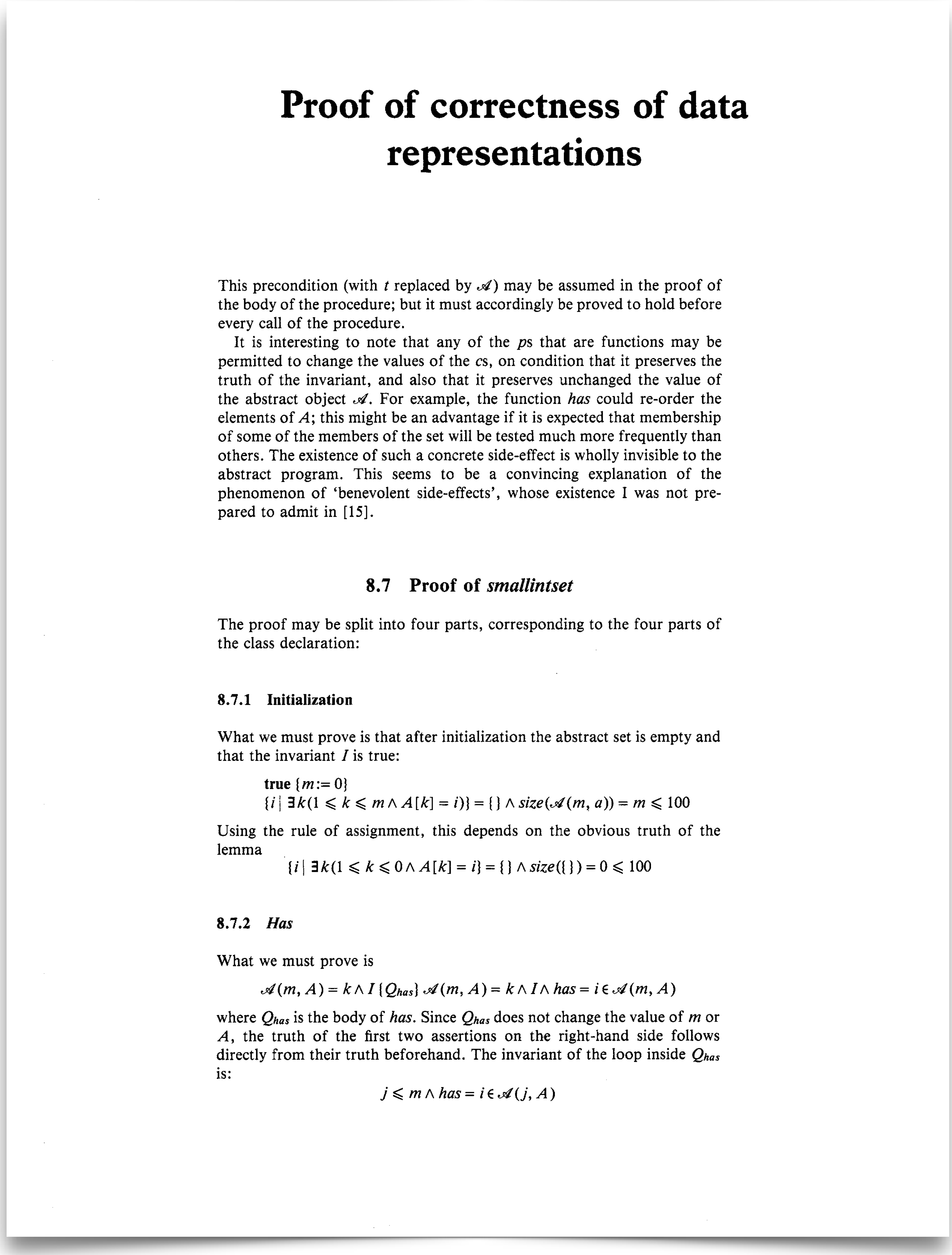
Class Invariants

Hoare 1972 "Proof of Correctness of Data Representations"

Acta Informatica Vol 1, issue 4 pp 271–281

https://doi.org/10.1007/BF00289507

https://dl.acm.org/doi/pdf/10.5555/63445.C1104363



Proof of correctness of data representations

This precondition (with t replaced by $\mathscr{A}$) may be assumed in the proof of the body of the procedure; but it must accordingly be proved to hold before every call of the procedure.

It is interesting to note that any of the *ps* that are functions may be permitted to change the values of the *cs*, on condition that it preserves the truth of the invariant, and also that it preserves unchanged the value of the abstract object $\mathscr{A}$. For example, the function *has* could re-order the elements of A; this might be an advantage if it is expected that membership of some of the members of the set will be tested much more frequently than others. The existence of such a concrete side-effect is wholly invisible to the abstract program. This seems to be a convincing explanation of the phenomenon of 'benevolent side-effects', whose existence I was not prepared to admit in [15].

8.7 Proof of *smallintset*

The proof may be split into four parts, corresponding to the four parts of the class declaration:

8.7.1 Initialization

What we must prove is that after initialization the abstract set is empty and that the invariant I is true:

$$\textbf{true}\ \{m := 0\}$$
$$\{i \mid \exists k(1 \leqslant k \leqslant m \wedge A[k] = i)\} = \{\,\} \wedge \mathit{size}(\mathscr{A}(m, a)) = m \leqslant 100$$

Using the rule of assignment, this depends on the obvious truth of the lemma

$$\{i \mid \exists k(1 \leqslant k \leqslant 0 \wedge A[k] = i\} = \{\,\} \wedge \mathit{size}(\{\,\}) = 0 \leqslant 100$$

8.7.2 *Has*

What we must prove is

$$\mathscr{A}(m, A) = k \wedge I\ \{Q_{has}\}\ \mathscr{A}(m, A) = k \wedge I \wedge has = i \in \mathscr{A}(m, A)$$

where Q_{has} is the body of *has*. Since Q_{has} does not change the value of m or A, the truth of the first two assertions on the right-hand side follows directly from their truth beforehand. The invariant of the loop inside Q_{has} is:

$$j \leqslant m \wedge has = i \in \mathscr{A}(j, A)$$

Design by Contract | Bertrand Meyer

"…a software system is viewed as a set of communicating ***components*** *whose interaction is based on precisely defined specifications of* ***the mutual obligations*** *— contracts."*

Bertrand Meyer

—Building bug-free O-O software: An Introduction to Design by Contract™

https://www.eiffel.com/values/design-by-contract/introduction/

Innovation 1: Each component has a contract

$$\{x > 0, x < \mathtt{INT\_MAX}\} \quad y = x + 1 \quad \{y > 1\}$$

Innovation 1: Each component has a contract

$$\{x > 0, x < \texttt{INT\_MAX}\} \quad y = x + 1 \quad \{y > 1\}$$

Innovation 1: Each component has a contract

```
{x > 0, x < INT_MAX}   y = x + 1   {y > 1}
```

Innovation 1: Each component has a contract

$$\{\texttt{x > 0}, \texttt{x < INT\_MAX}\} \quad \texttt{y = x + 1} \quad \{\texttt{y > 1}\}$$

Innovation 1: Each component has a contract

$$\{\texttt{x < INT\_MAX}\} \quad \texttt{y = x + 1} \quad \{\texttt{y > 1}\}$$

Innovation 1: Each component has a contract

```
{x < INT_MAX}   y = x + 1   {y > 1}
```

Innovation 1: Each component has a contract

```
{x < INT_MAX}   y = x + 1   {y > 1}
```

Innovation 1: Each component has a contract

```
{x < INT_MAX}   y = x + 1   {y == x + 1}
```

Innovation 1: Each component has a contract

```
{x < INT_MAX}  y = x + 1  {y == x + 1}
```

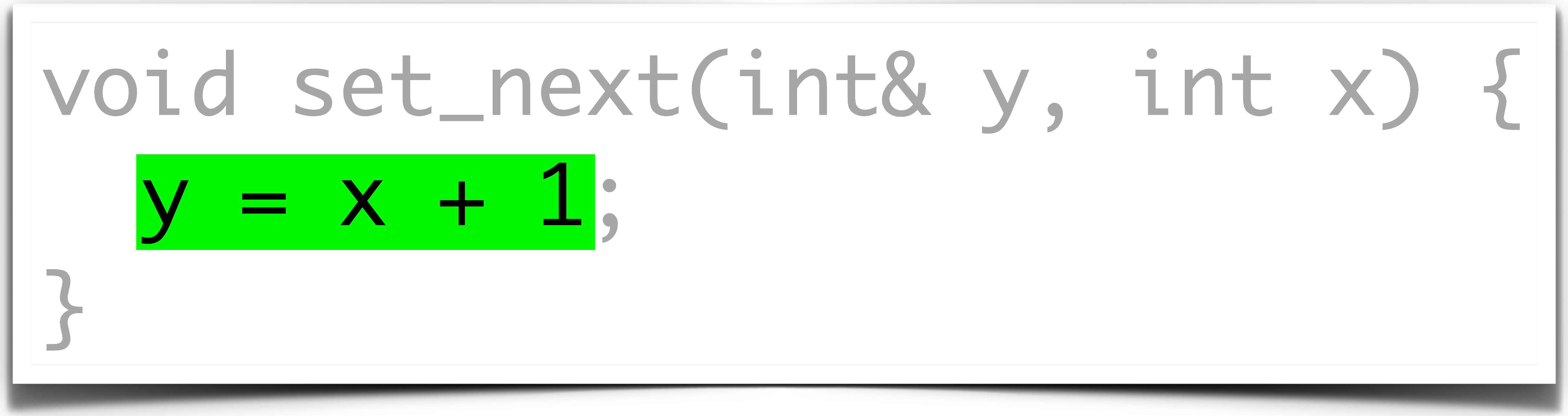
Innovation 1: Each component has a contract

```
{x < INT_MAX}   y = x + 1   {y == x + 1}
```

Innovation 1: Each component has a contract

{x < INT_MAX}

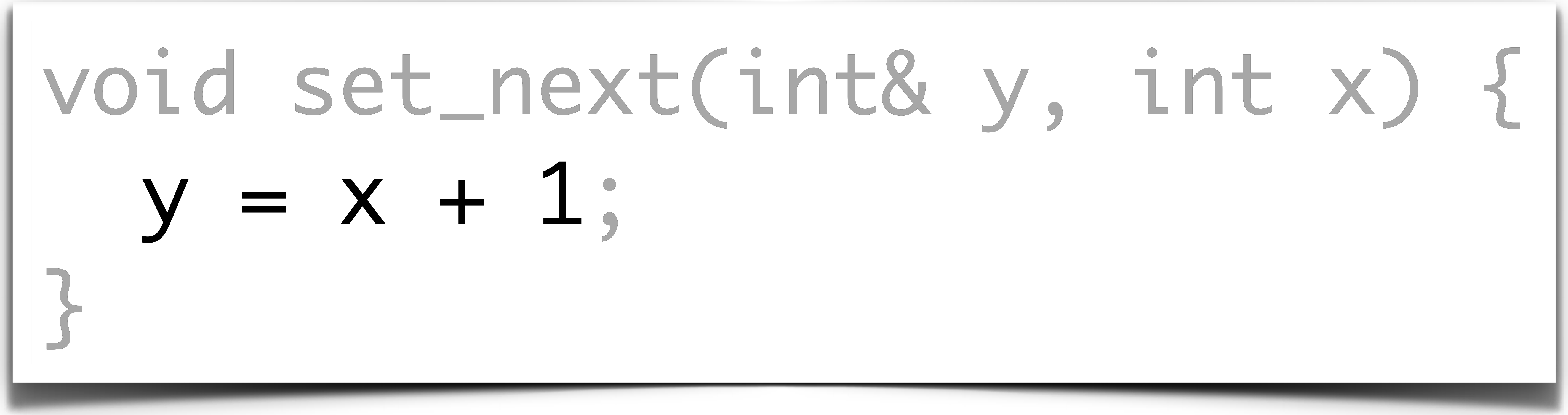
{y == x + 1}



```
void set_next(int& y, int x) {
  y = x + 1;
}
```

Innovation 1: Each component has a contract

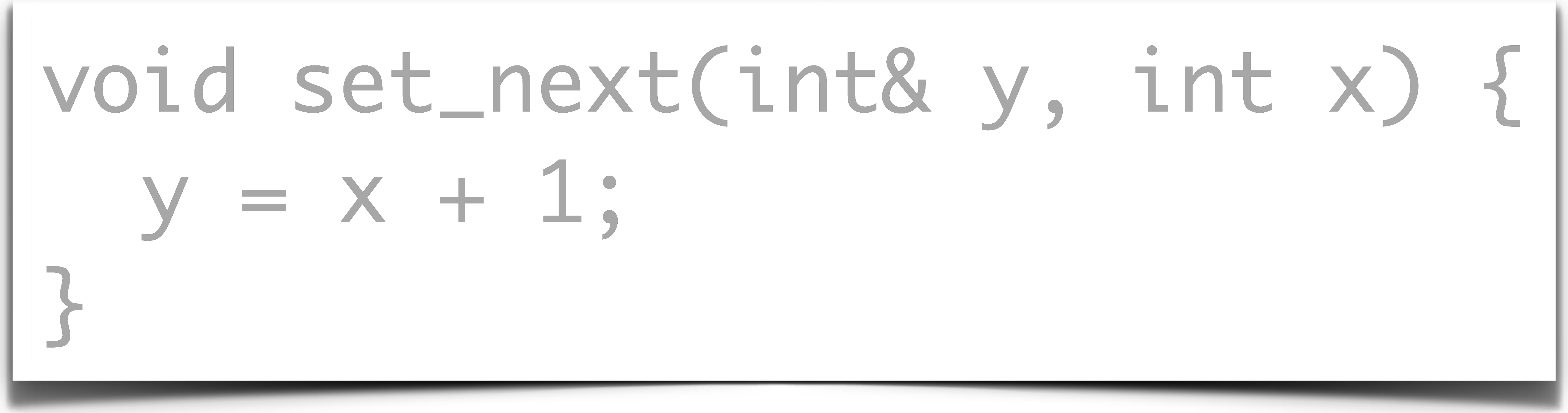
{x < INT_MAX} 🟩 {y == x + 1}



```
void set_next(int& y, int x) {
  y = x + 1;
}
```

Innovation 1: Each component has a contract

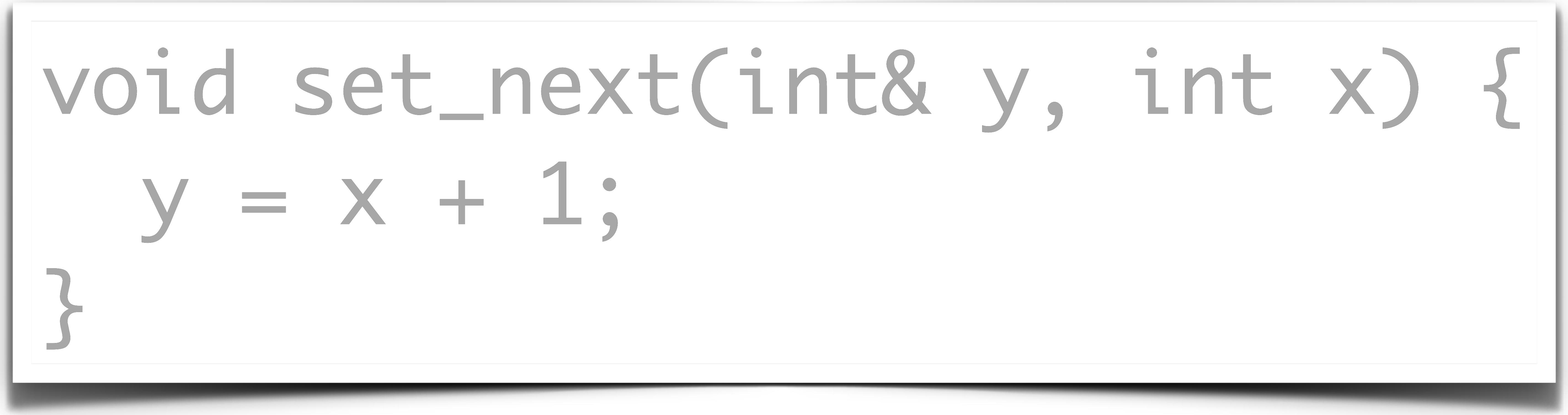
{x < INT_MAX} set_next(y, x) {y == x + 1}



```
void set_next(int& y, int x) {
  y = x + 1;
}
```

Innovation 1: Each component has a contract

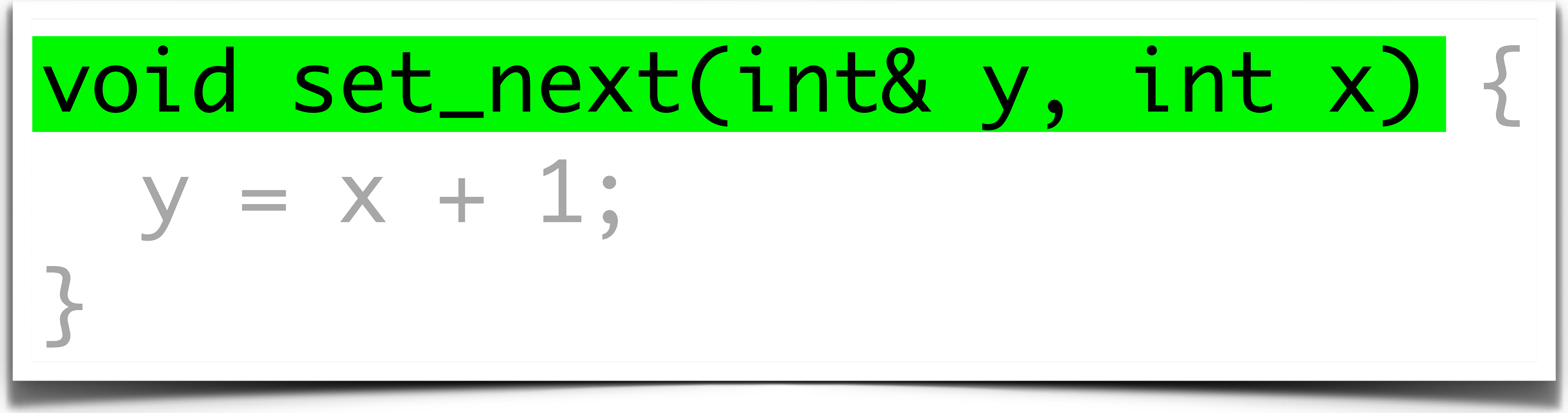
```
{x < INT_MAX} set_next(y, x) {y == x + 1}
```



```
void set_next(int& y, int x) {
  y = x + 1;
}
```

Innovation 1: Each component has a contract

```
{x < INT_MAX} set_next(y, x) {y == x + 1}
```

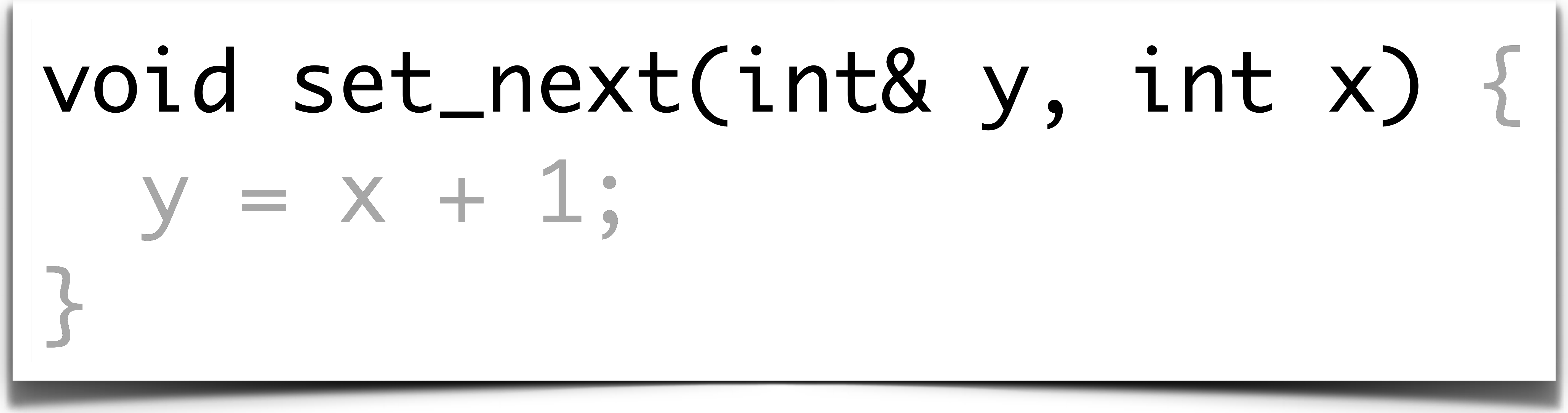


```
void set_next(int& y, int x) {
    y = x + 1;
}
```

Innovation 1: Each component has a contract

precondition **of set_next()**

```
{x < INT_MAX} set_next(y, x) {y == x + 1}
```



```
void set_next(int& y, int x) {
   y = x + 1;
}
```

Innovation 1: Each component has a contract

precondition **of set_next()**

postcondition **of set_next()**

```
{x < INT_MAX} set_next(y, x) {y == x + 1}
```

```
void set_next(int& y, int x) {
  y = x + 1;
}
```

Innovation 1: Each component has a contract

Correct clients provide
Incoming state and argument values
satisfying

precondition

Correct implementation ensures
Outgoing state and return value
satisfying

postcondition

```
auto any_function(parameters…) -> R
```

Innovation 2: An ethos of blame (for code not people)

If preconditions are violated, **that's a bug in the client**.

Otherwise, if the operation returns normally without fulfilling postconditions, **that's a bug in the operation**.

If software malfunctions and you can't clearly assign blame, a contract is missing somewhere.

* stay tuned

Innovation 3: Language support in Eiffel

```
class interface
  COUNTER
feature
  value: INTEGER  -- Counter's value.
  invariant
      value >= 0

  decrement is  -- Decrease counter by one.
    require
      value > 0
    ensure
      value = old value - 1
end
```

Innovation 3: Language support in Eiffel

```
class interface
  COUNTER
feature
  value: INTEGER  -- Counter's value.
  invariant
      value >= 0

  decrement is  -- Decrease counter by one.
    require
      value > 0
    ensure
      value = old value - 1
end
```

Innovation 3: Language support in Eiffel

```
class interface
  COUNTER
feature
  value: INTEGER  -- Counter's value.
  invariant
      value >= 0

  decrement is  -- Decrease counter by one.
    require
      value > 0
    ensure
      value = old value - 1
end
```

Innovation 3: Language support in Eiffel

```
class interface
  COUNTER
feature
  value: INTEGER  -- Counter's value.
  invariant
      value >= 0

  decrement is  -- Decrease counter by one.
    require
      value > 0
    ensure
      value = old value - 1
end
```

Innovation 3: Language support in Eiffel

```
class interface
  COUNTER
feature
  value: INTEGER  -- Counter's value.
  invariant
      value >= 0

  decrement is  -- Decrease counter by one.
    require
      value > 0
    ensure
      value = old value - 1
end
```

Innovation 3: Language support in Eiffel

```
class interface
  COUNTER
feature
  value: INTEGER  -- Counter's value.
  invariant
      value >= 0

  decrement is  -- Decrease counter by one.
    require
      value > 0
    ensure
      value = old value - 1
end
```

Innovation 3: Language support in Eiffel

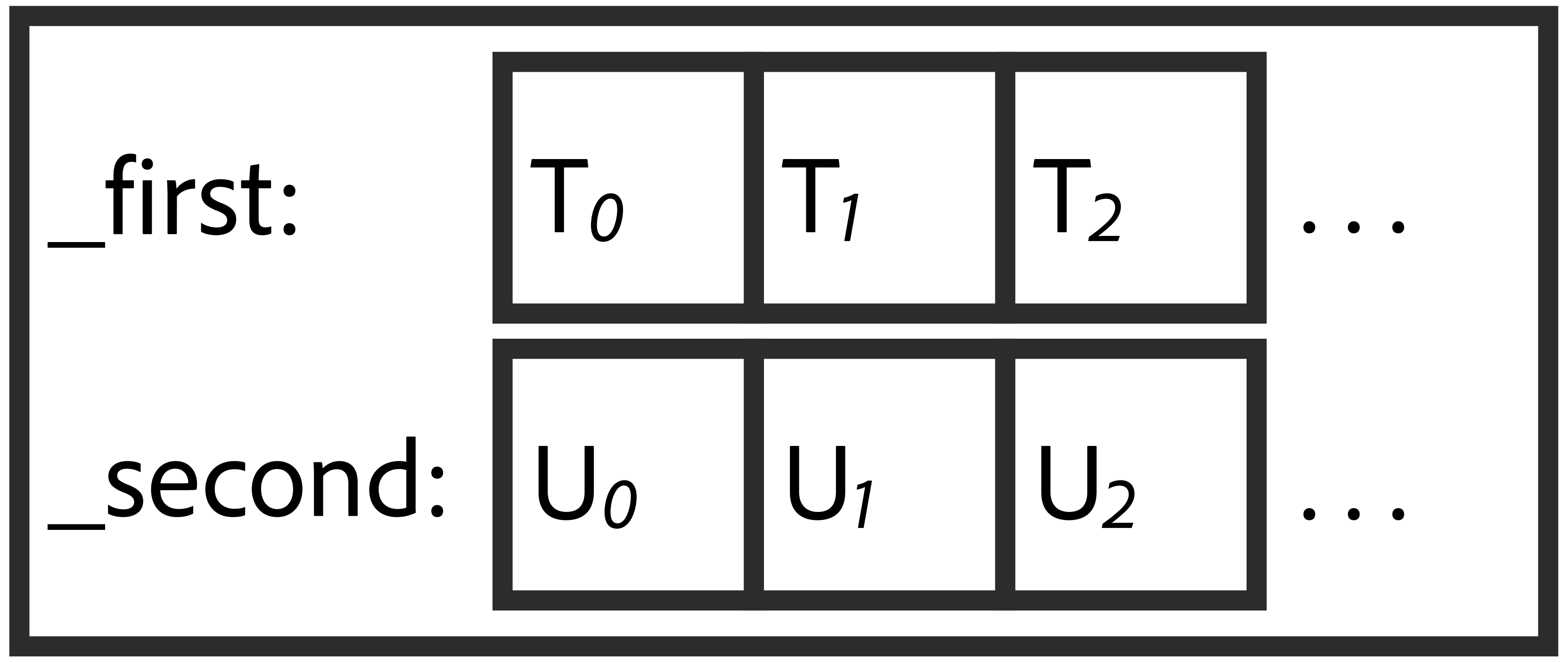
```
class interface
  COUNTER
feature
  value: INTEGER  -- Counter's value.
  invariant
      value >= 0

  decrement is  -- Decrease counter by one.
    require
      value > 0
    ensure
      value = old value - 1
end
```

zip_vector

```
template <class T, class U>
class zip_vector {
  vector<T> _first;
  vector<U> _second;
public:
  const vector<T>& first() const { return _first; }
  const vector<U>& second() const { return _second; }
  ...
  size_t size() const;
  bool empty() const;
  ...
  void pop_back();
  ...
};
```

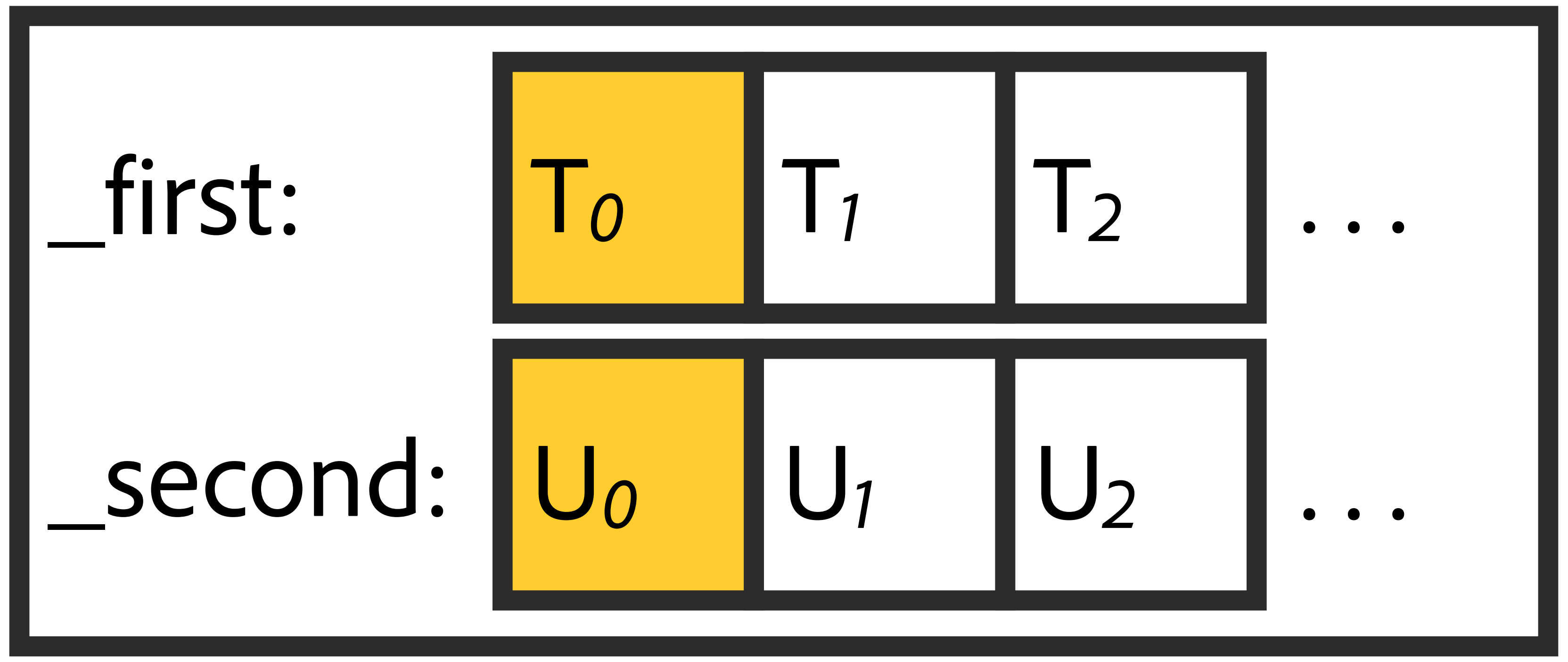
zip_vector



_first:	T_0	T_1	T_2	. . .
_second:	U_0	U_1	U_2	. . .

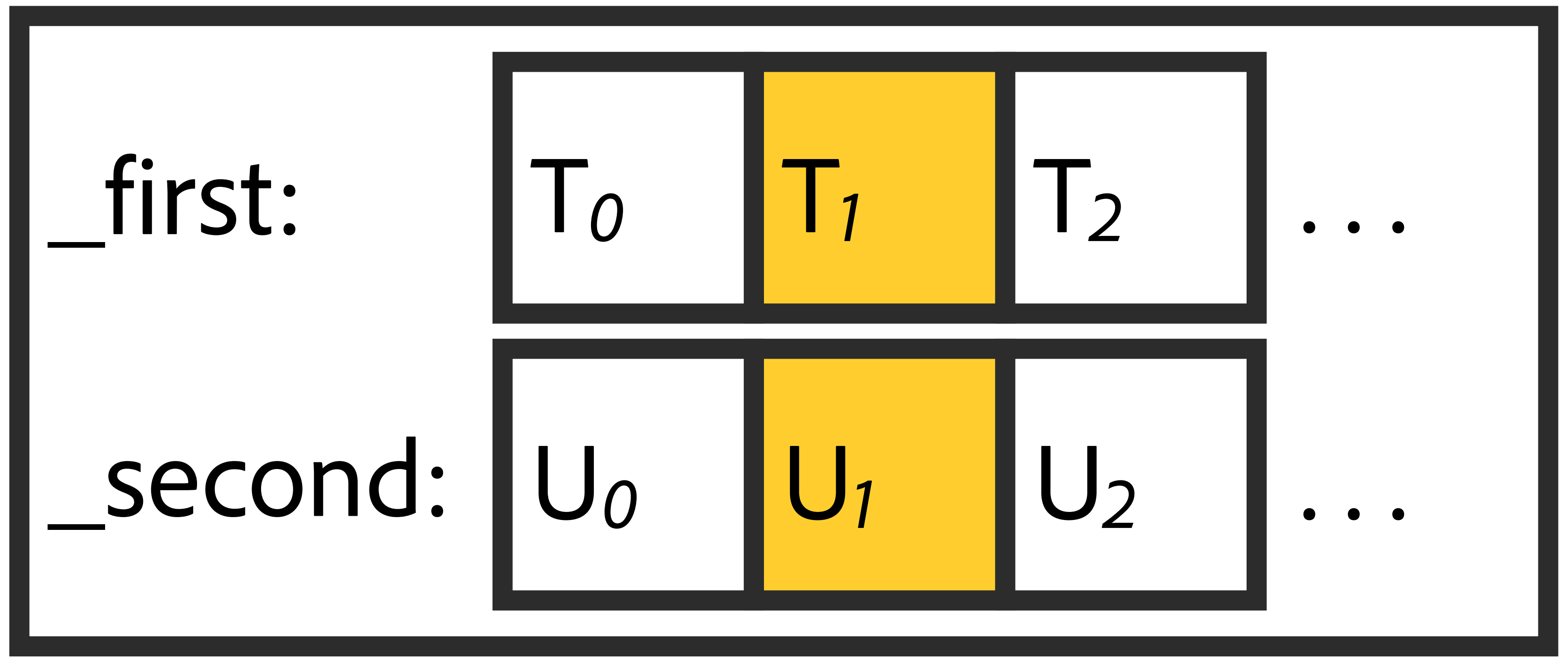
```
template <class T, class U>
class zip_vector {
  vector<T> _first;
  vector<U> _second;
public:
  const vector<T>& first() const { return _first; }
  const vector<U>& second() const { return _second; }
  ...
  size_t size() const;
  bool empty() const;
  ...
  void pop_back();
  ...
};
```

zip_vector



```
template <class T, class U>
class zip_vector {
  vector<T> _first;
  vector<U> _second;
public:
  const vector<T>& first() const { return _first; }
  const vector<U>& second() const { return _second; }
  ...
  size_t size() const;
  bool empty() const;
  ...
  void pop_back();
  ...
};
```

zip_vector



_first: T_0 T_1 T_2 . . .

_second: U_0 U_1 U_2 . . .

```
template <class T, class U>
class zip_vector {
  vector<T> _first;
  vector<U> _second;
public:
  const vector<T>& first() const { return _first; }
  const vector<U>& second() const { return _second; }
  ...
  size_t size() const;
  bool empty() const;
  ...
  void pop_back();
  ...
};
```

zip_vector

```
template <class T, class U>
class zip_vector {
  vector<T> _first;
  vector<U> _second;
public:
  const vector<T>& first() const { return _first; }
  const vector<U>& second() const { return _second; }
  ...
  size_t size() const;
  bool empty() const;
  ...
  void pop_back();
  ...
};
```

zip_vector

```
template <class T, class U>
class zip_vector {
  vector<T> _first;
  vector<U> _second;
public:
  const vector<T>& first() const { return _first; }
  const vector<U>& second() const { return _second; }
  ...
  size_t size() const;
  bool empty() const;
  ...
  void pop_back();
  ...
};
```

zip_vector

```
template <class T, class U>
class zip_vector {
  vector<T> _first;
  vector<U> _second;
public:
  const vector<T>& first() const { return _first; }
  const vector<U>& second() const { return _second; }
  ...
  size_t size() const;
  bool empty() const;
  ...
  void pop_back();
  ...
};
```

zip_vector

```
void pop_back() {
  _first.pop_back();
  _second.pop_back();
}
```

zip_vector

```
void pop_back() {


  _first.pop_back();
  _second.pop_back();
}
```

zip_vector

```
void pop_back() {
  assert("pre " && (size() > 0));

  _first.pop_back();
  _second.pop_back();
}
```

zip_vector

```
void pop_back() {
  assert("pre " && (size() > 0));



  _first.pop_back();
  _second.pop_back();
}
```

zip_vector

```
void pop_back() {
  assert("pre " && (size() > 0));
  #ifndef NDEBUG
  auto old_size = size();
  #endif

  _first.pop_back();
  _second.pop_back();
}
```

zip_vector

```
void pop_back() {
  assert("pre " && (size() > 0));
  #ifndef NDEBUG
  auto old_size = size();
  #endif

  _first.pop_back();
  _second.pop_back();


}
```

zip_vector

```
void pop_back() {
  assert("pre " && (size() > 0));
  #ifndef NDEBUG
  auto old_size = size();
  #endif

  _first.pop_back();
  _second.pop_back();

  assert("post " && (size() == old_size - 1));
}
```

Maybe someday | C++26?

```
void pop_back() {
  assert("pre " && (size() > 0));
  #ifndef NDEBUG
  auto old_size = size();
  #endif

  _first.pop_back();
  _second.pop_back();

  assert("post " && (size() == old_size - 1));
}
```

Maybe someday | C++26?

```
void pop_back() {
  assert("pre " && (size() > 0));
  #ifndef NDEBUG
  auto old_size = size();
  #endif

  _first.pop_back();
  _second.pop_back();

  assert("post " && (size() == old_size - 1));
}
```

Maybe someday | C++26?

```
void pop_back() {

  assert("pre " && (size() > 0));
  #ifndef NDEBUG
  auto old_size = size();
  #endif

  _first.pop_back();
  _second.pop_back();

  assert("post " && (size() == old_size - 1));
}
```

Maybe someday | C++26?

```
void pop_back()
  pre { size() > 0 }
{
  #ifndef NDEBUG
  auto old_size = size();
  #endif

  _first.pop_back();
  _second.pop_back();

  assert("post " && (size() == old_size - 1));
}
```

Maybe someday | C++26?

```
void pop_back()
  pre { size() > 0 }
{
  #ifndef NDEBUG
  auto old_size = size();
  #endif

  _first.pop_back();
  _second.pop_back();

  assert("post " && (size() == old_size - 1));
}
```

Maybe someday | C++26?

```
void pop_back()
  pre { size() < 0 }
  post [old_size = size()] { size() == old_size - 1 }
{
  _first.pop_back();
  _second.pop_back();
}
```

Maybe someday | C++26?

```
void pop_back()
  pre { size() < 0 }
  post [old_size = size()] { size() == old_size - 1 }

{
  _first.pop_back();
  _second.pop_back();
}
```

Maybe someday | C++26?

```
void pop_back()
  pre { size() < 0 }
  post [old_size = size()] { size() == old_size - 1 }
  post [old = *this] { equal(begin(), end(), begin(old)) }
{
  _first.pop_back();
  _second.pop_back();
}
```

Maybe someday | C++26?

```
void pop_back()
  pre { size() < 0 }
  post [old_size = size()] { size() == old_size - 1 }
  post [old = *this] {             equal(begin(), end(), begin(old)) }
{
  _first.pop_back();
  _second.pop_back();
}
```

Maybe someday | C++26?

```
void pop_back()
  pre { size() < 0 }
  post [old_size = size()] { size() == old_size - 1 }
  post [old = *this] { !testing || equal(begin(), end(), begin(old)) }
{
  _first.pop_back();
  _second.pop_back();
}
```

Checking adds generic constraints

```
template <class T, class U>
class zip_vector {
  vector<T> _first;
  vector<U> _second;
public:
  const vector<T>& first() const { return _first; }
  const vector<U>& second() const { return _second; }
  ...
  size_t size() const;
  bool empty() const;
  ...
  void pop_back()
  ...
```

Checking adds generic constraints

```
template <class T, class U>

class zip_vector {
  vector<T> _first;
  vector<U> _second;
public:
  const vector<T>& first() const { return _first; }
  const vector<U>& second() const { return _second; }
  ...
  size_t size() const;
  bool empty() const;
  ...
```

Checking adds generic constraints

```
template <class T, class U>
    requires copyable<T> && assignable<T> && copyable<U> && assignable<U>
class zip_vector {
  vector<T> _first;
  vector<U> _second;
public:
  const vector<T>& first() const { return _first; }
  const vector<U>& second() const { return _second; }
  ...
  size_t size() const;
  bool empty() const;
  ...
```

Specifying and checking invariants

```
template <class T, class U>
    requires copyable<T> && assignable<T> && copyable<U> && assignable<U>
class zip_vector {
  vector<T> _first;
  vector<U> _second;
public:
  const vector<T>& first() const { return _first; }
  const vector<U>& second() const { return _second; }
  ...
  size_t size() const;
  bool empty() const;
  ...
```

Specifying and checking invariants

```
template <class T, class U>
    requires copyable<T> && assignable<T> && copyable<U> && assignable<U>
class zip_vector {
  vector<T> _first;
  vector<U> _second;
public:
  const vector<T>& first() const { return _first; }
  const vector<U>& second() const { return _second; }



  size_t size() const;
  bool empty() const;
  ...
```

Specifying and checking invariants

```
template <class T, class U>
    requires copyable<T> && assignable<T> && copyable<U> && assignable<U>
class zip_vector {
  vector<T> _first;
  vector<U> _second;
public:
  const vector<T>& first() const { return _first; }
  const vector<U>& second() const { return _second; }

  invariant { size(first()) == size(second()) }

  size_t size() const;
  bool empty() const;
  ...
```

Checking invariants automatically

When returning from:

- Constructors

Checking invariants automatically

When returning from:

- Constructors
- Public mutating member functions

Checking invariants automatically

When returning from:

- Constructors
- Public mutating member functions that directly use private mutating API

# Invariant

```
template <class T, class U>
    requires copyable<T> && assignable<T> && copyable<U> && assignable<U>
class zip_vector {
  vector<T> _first;
  vector<U> _second;
public:
  const vector<T>& first() const { return _first; }
  const vector<U>& second() const { return _second; }

  invariant { size(first()) == size(second()) }

  size_t size() const;
  bool empty() const;
  ...
```

# Invariant

```
template <class T, class U>
    requires copyable<T> && assignable<T> && copyable<U> && assignable<U>
class zip_vector {
  vector<T> _first;
  vector<U> _second;
public:
  const vector<T>& first() const { return _first; }
  const vector<U>& second() const { return _second; }

  invariant { size(first()) == size(second()) }

  size_t size() const { return size(first()); }
  bool empty() const;
  ...
```

# Invariant

```
template <class T, class U>
    requires copyable<T> && assignable<T> && copyable<U> && assignable<U>
class zip_vector {
  vector<T> _first;
  vector<U> _second;
public:
  const vector<T>& first() const { return _first; }
  const vector<U>& second() const { return _second; }

  invariant { size(first()) == size(second()) }

  size_t size() const { return min(size(first()), size(second())); }
  bool empty() const;
  ...
```

Strong contracts simplify code

What's in a “strong contract?” | Tradeoffs

	weak	strong
class invariant		

What's in a “strong contract?” | Tradeoffs

	weak	strong
class invariant	high representational flexibility harder to reason about e.g. `xml_document`	

What's in a “strong contract?” | Tradeoffs

	weak	strong
class invariant	high representational flexibility harder to reason about e.g. `xml_document`	low representational flexibility easier to reason about e.g. `bool`

What's in a "strong contract?" | Tradeoffs

	weak	strong
class invariant	high representational flexibility harder to reason about e.g. `xml_document`	low representational flexibility easier to reason about e.g. `bool`
precondition		

What's in a "strong contract?" | Tradeoffs

	weak	strong
class invariant	high representational flexibility harder to reason about e.g. `xml_document`	low representational flexibility easier to reason about e.g. `bool`
precondition	lax rules for clients more uses are legal	

What's in a “strong contract?” | Tradeoffs

	weak	strong
class invariant	high representational flexibility harder to reason about e.g. `xml_document`	low representational flexibility easier to reason about e.g. `bool`
precondition	lax rules for clients more uses are legal	strict rules for clients more uses are bugs

What's in a “strong contract?” | Tradeoffs

	weak	strong
class invariant	high representational flexibility harder to reason about e.g. `xml_document`	low representational flexibility easier to reason about e.g. `bool`
precondition	lax rules for clients more uses are legal	strict rules for clients more uses are bugs
postcondition		

What's in a “strong contract?” | Tradeoffs

	weak	strong
class invariant	high representational flexibility harder to reason about e.g. `xml_document`	low representational flexibility easier to reason about e.g. `bool`
precondition	lax rules for clients more uses are legal	strict rules for clients more uses are bugs
postcondition	fewer guarantees to clients more implementer flexibility	

What's in a “strong contract?” | Tradeoffs

	weak	strong
class invariant	high representational flexibility harder to reason about e.g. `xml_document`	low representational flexibility easier to reason about e.g. `bool`
precondition	lax rules for clients more uses are legal	strict rules for clients more uses are bugs
postcondition	fewer guarantees to clients more implementer flexibility	more guarantees to clients less implementer flexibility

What's in a “strong contract?” | Tradeoffs

	weak	strong
class invariant	high representational flexibility harder to reason about e.g. `xml_document`	low representational flexibility easier to reason about e.g. `bool`
precondition	lax rules for clients more uses are legal	strict rules for clients more uses are bugs
postcondition	fewer guarantees to clients more implementer flexibility	more guarantees to clients less implementer flexibility

What's in a “strong contract?” | Going too far

Where a and b are ints, a / b

- precondition: $b \neq 0$
- postcondition: returns $\lfloor a \div b \rfloor$

What's in a “strong contract?” | Going too far

Where a and b are ints, a / b

- ~~precond[illegible] 0~~

weakened to nothingness

- postcondition: returns ⌊a ÷ b⌋

What's in a “strong contract?” | Going too far

Where a and b are ints, a / b

- ~~precond~~ [illegible] ~~0~~
- postcondition: returns $\lfloor a \div b \rfloor$

weakened to nothingness

What's in a “strong contract?” | Going too far

Where a and b are ints, a / b

- ~~precondition: b ≠ 0~~
- postcondition: returns $b = 0\ ?\ a : \lfloor a \div b \rfloor$

What's in a “strong contract?” | Going too far

Where a and b are ints, a / b

- ~~precon[illegible] 0~~
- postcondition: returns $b = 0 \; ? \; a : \lfloor a \div b \rfloor$

What if sort's spec precisely described which pairs of elements would be compared, and when?

Strong contracts are simple and relevant

Strong contracts are simple and relevant

Corrolary: a complex contract is a sign of poor API design

Advice for API designers (that's you)

Support the use cases you're certain are needed

- Use the strongest preconditions
- Use the weakest postconditions
- But keep the contract simple

zip_vector | push_back

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  _second.push_back(e.second);
}
```

zip_vector | push_back

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  _second.push_back(e.second);
}
```

zip_vector | push_back

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  _second.push_back(e.second);
}
```

zip_vector | push_back

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  _second.push_back(e.second);
}
```

zip_vector | push_back

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  _second.push_back(e.second);
}
```

zip_vector | push_back

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  _second.push_back(e.second);
}
```

broken invariant

zip_vector | push_back

```
void push_back(const pair<T, U>& e, function<void()> callback())
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  _second.push_back(e.second);
}
```

broken invariant

zip_vector | push_back

```
void push_back(const pair<T, U>& e, function<void()> callback())
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);

  _second.push_back(e.second);
}
```

broken invariant

zip_vector | push_back

```
void push_back(const pair<T, U>& e, function<void()> callback())
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  callback();
  _second.push_back(e.second);
}
```

broken invariant

zip_vector | push_back

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  _second.push_back(e.second);
}
```

broken invariant

zip_vector | push_back

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  _second.push_back(e.second);
}
```

broken invariant

```
struct bad { bad(const bad&); };
zip_vector<int, bad> v;
bad::bad(const bad&) { print("{}", v.back()); }
...
v.push_back({42, bad{}});
```

zip_vector | push_back

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  _second.push_back(e.second);
}
```

broken invariant

```
struct bad { bad(const bad&); };
zip_vector<int, bad> v;
bad::bad(const bad&) { print("{}", v.back()); }
...
v.push_back({42, bad{}});
```

zip_vector | push_back

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  _second.push_back(e.second);
}
```

broken invariant

```
struct bad { bad(const bad&); };
zip_vector<int, bad> v;
bad::bad(const bad&) { print("{}", v.back()); }
...
v.push_back({42, bad{}});
```

zip_vector | push_back

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  _second.push_back(e.second);
}
```

broken invariant

```
struct bad { bad(const bad&); };
zip_vector<int, bad> v;
bad::bad(const bad&) { print("{}", v.back()); }
...
v.push_back({42, bad{}});
```

Meaningless values

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  _second.push_back(e.second);
}
```

broken invariant

Errors

A short rant about (our misunderstanding of) exceptions

Lots of people are still uncomfortable with exceptions.

A short rant about (our misunderstanding of) exceptions

Lots of people are still uncomfortable with exceptions.

Lots of people still don't understand exceptions

A short rant about (our misunderstanding of) exceptions

Lots of people are still uncomfortable with exceptions.

Lots of people still don't understand exceptions

Sensible-sounding but meaningless advice abounds:

- Don't use exceptions for control flow
- Only use exceptions for exceptional / unexpected conditions

A short rant about (our misunderstanding of) exceptions

Lots of people are still uncomfortable with exceptions.

Lots of people still don't understand exceptions

Sensible-sounding but meaningless advice abounds:

- Don't use exceptions for control flow
- Only use exceptions for exceptional / unexpected conditions

The key to power with exceptions is focusing on contracts and invariants, not control flow

- Welcome, you're in the right place for that.

A short rant about (our misunderstanding of) exceptions

Lots of people are still uncomfortable with exceptions.

Lots of people still don't understand exceptions

Sensible-sounding but meaningless advice abounds:

- Don't use exceptions for control flow
- Only use exceptions for exceptional / unexpected conditions

The key to power with exceptions is focusing on contracts and invariants, not control flow

- Welcome, you're in the right place for that.

Understanding the engineering tradeoffs helps too

Definition | Error (without qualification)

Error, n. An indication that a correct function, correctly called, could not uphold its postcondition.

—Dave and Sean

Definition | Error (without qualification)

Error, n. An indication that a correct function, correctly called, could not uphold its postcondition.

—Dave and Sean

not a bug

Definition | Error (without qualification)

Error, n. An indication that a correct function, correctly called, could not uphold its postcondition.

—Dave and Sean

not a bug

Bugs

- 🐞 programming error
- 🐞 syntax error
- 🐞 bounds error
- 🐞 memory error

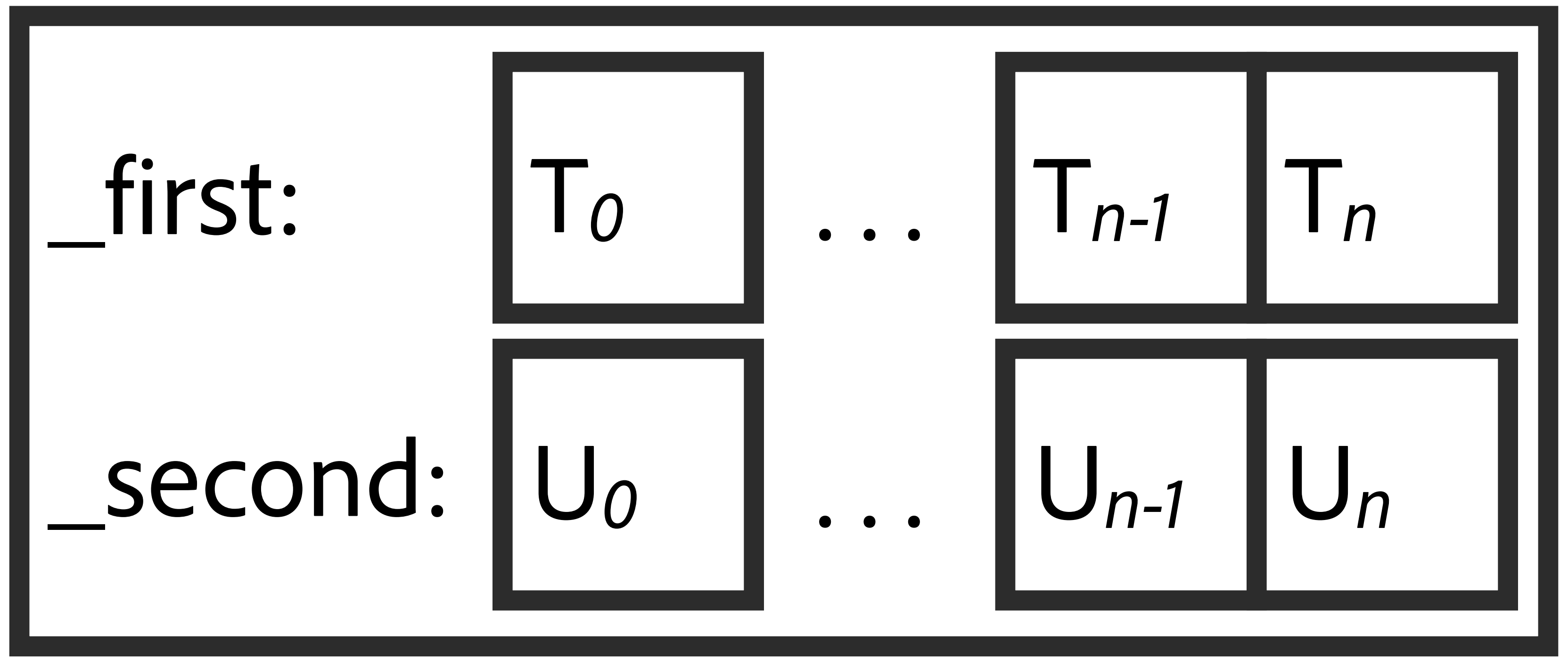
Three useful guarantees regarding errors

The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

Three useful guarantees regarding errors



_first: T_0 . . . T_{n-1} T_n

_second: U_0 . . . U_{n-1} U_n

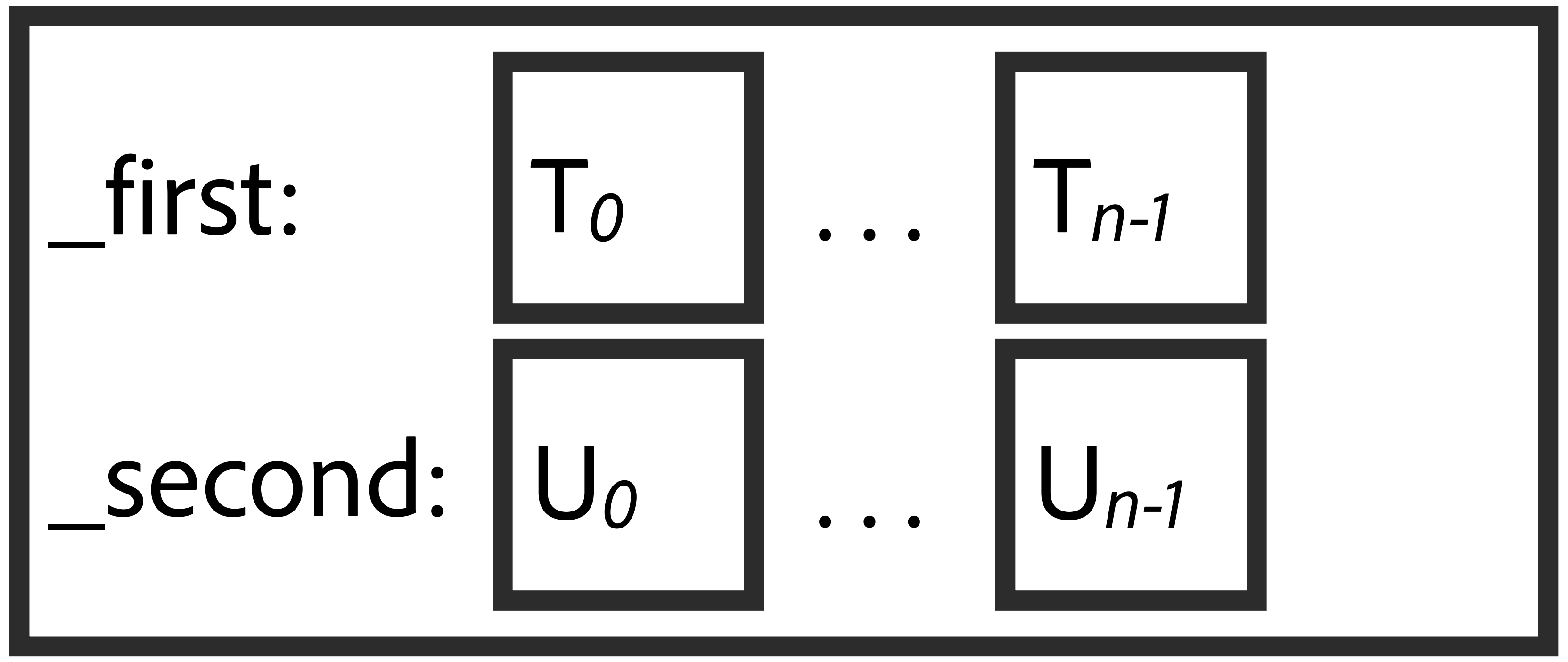
The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void pop_back()
  pre { size() < 0 }
  post [old_size = size()] { size() == old_size - 1 }
  post [old = *this] { !testing || equal(begin(), end(), begin(old)) }
{
  _first.pop_back();
  _second.pop_back();
}
```

Three useful guarantees regarding errors



_first: T_0 . . . T_{n-1}

_second: U_0 . . . U_{n-1}

The nothrow guarantee: no errors can occur.

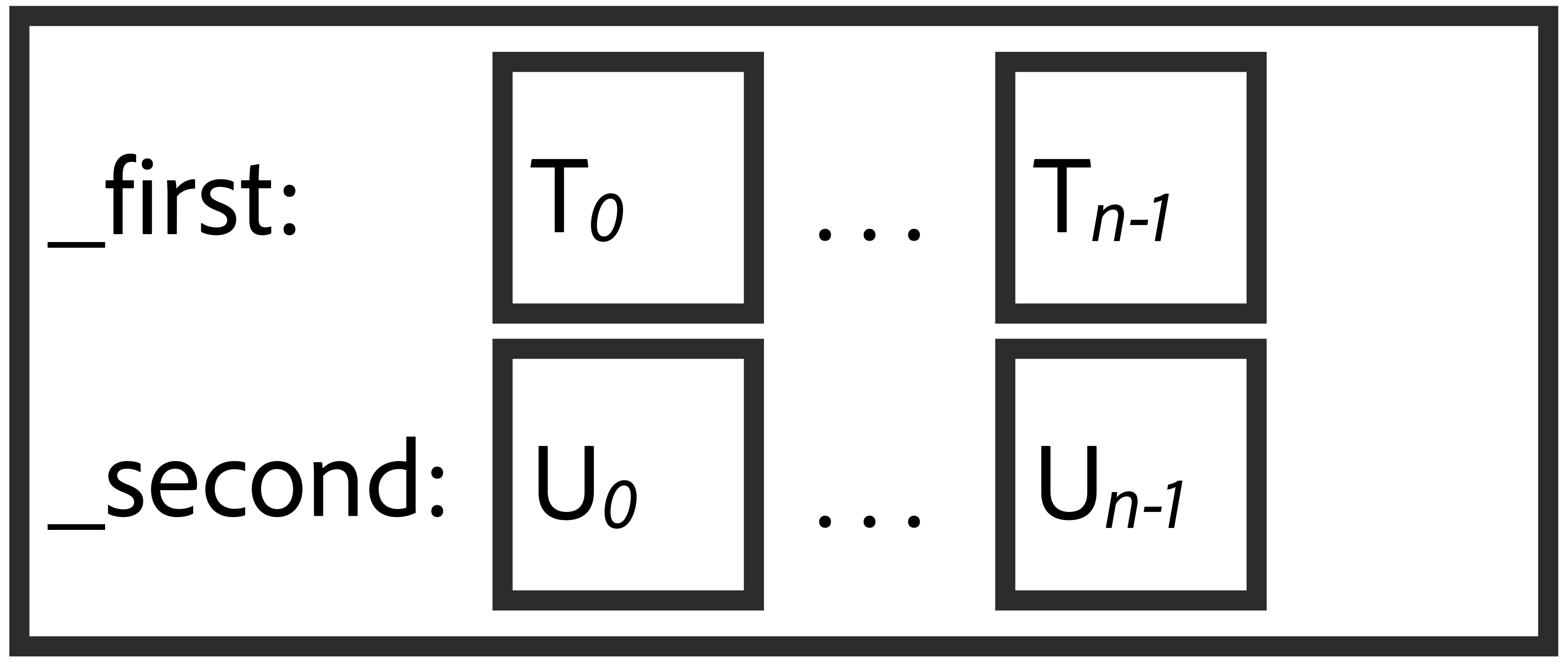
The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void pop_back()
  pre { size() < 0 }
  post [old_size = size()] { size() == old_size - 1 }
  post [old = *this] { !testing || equal(begin(), end(), begin(old)) }
{
  _first.pop_back();
  _second.pop_back();
}
```

nothrow guarantee (x2) - composes

Three useful guarantees regarding errors



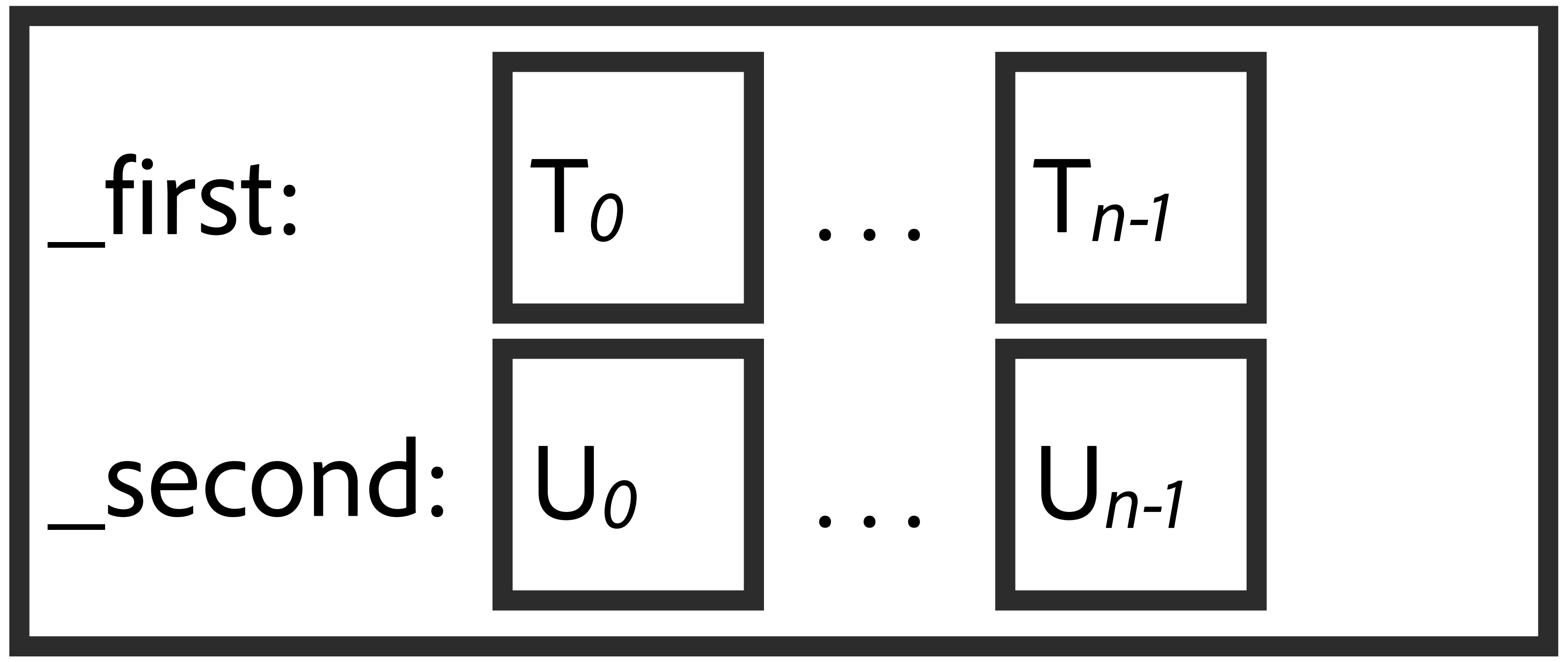
The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void pop_back()
  pre { size() < 0 }
  post [old_size = size()] { size() == old_size - 1 }
  post [old = *this] { !testing || equal(begin(), end(), begin(old)) }
{
  _first.pop_back();
  _second.pop_back();
}
```

Three useful guarantees regarding errors



_first: T_0 . . . T_{n-1}

_second: U_0 . . . U_{n-1}

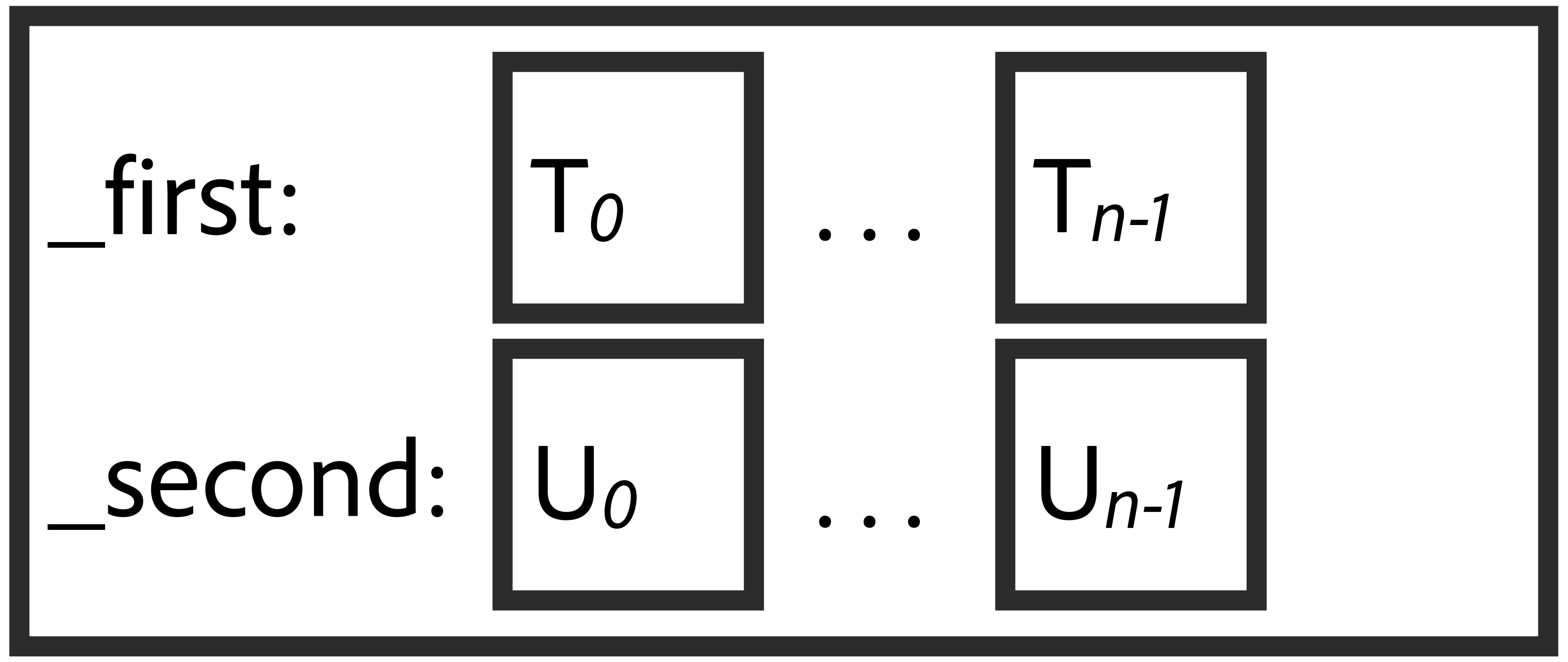
The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void pop_back() noexcept
  pre { size() < 0 }
  post [old_size = size()] { size() == old_size - 1 }
  post [old = *this] { !testing || equal(begin(), end(), begin(old)) }
{
  _first.pop_back();
  _second.pop_back();
}
```

Three useful guarantees regarding errors



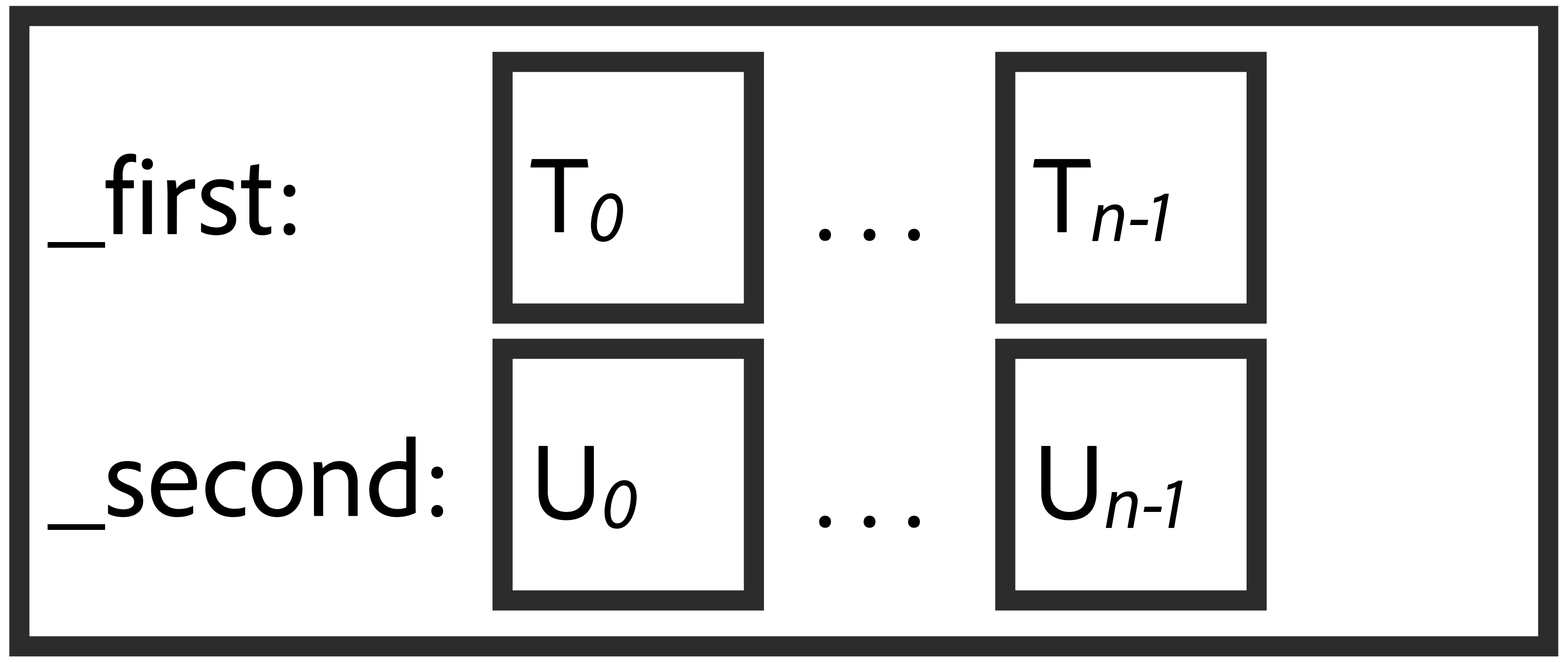
The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void pop_back() noexcept
  pre { size() < 0 }
  post [old_size = size()] { size() == old_size - 1 }
  post [old = *this] { !testing || equal(begin(), end(), begin(old)) }
{
  _first.pop_back();
  _second.pop_back();
}
```

Three useful guarantees regarding errors



_first: T_0 . . . T_{n-1}

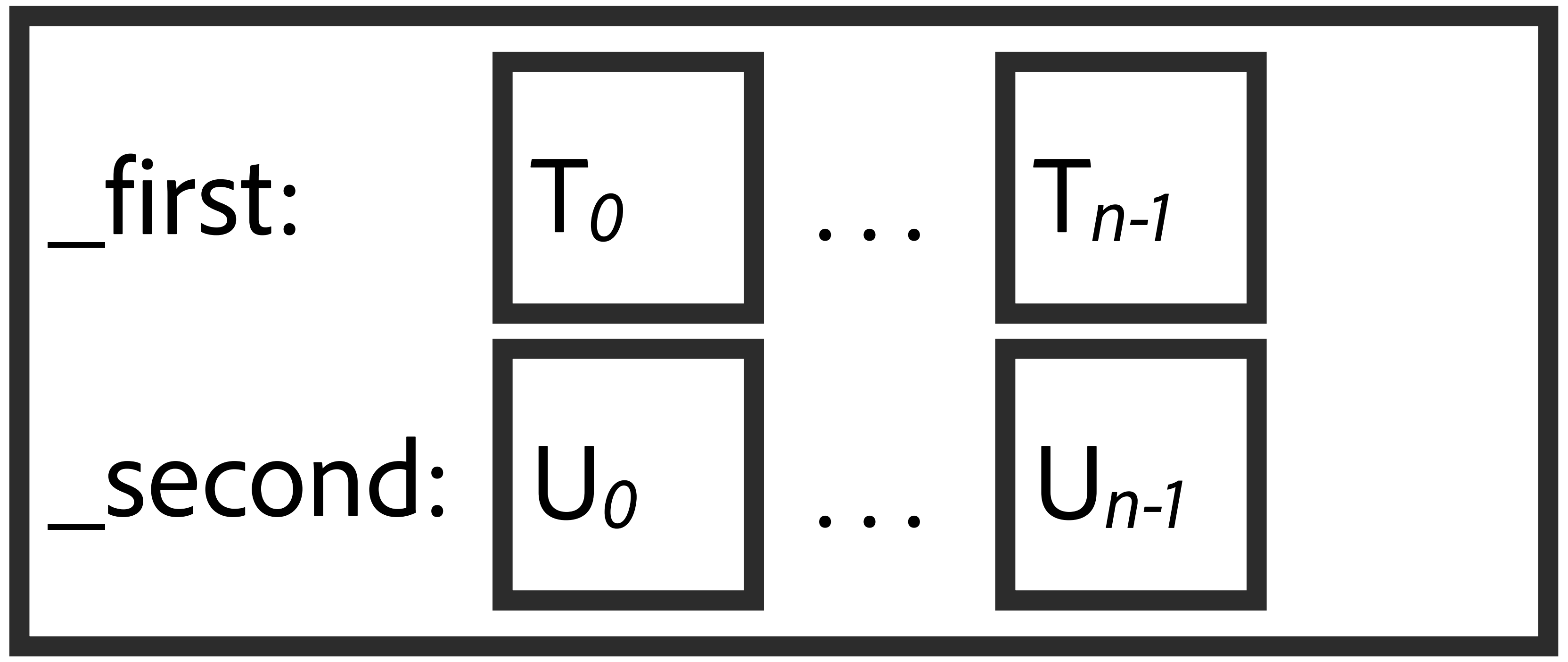
_second: U_0 . . . U_{n-1}

The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

Three useful guarantees regarding errors



_first: T_0 . . . T_{n-1}

_second: U_0 . . . U_{n-1}

The nothrow guarantee: no errors can occur.

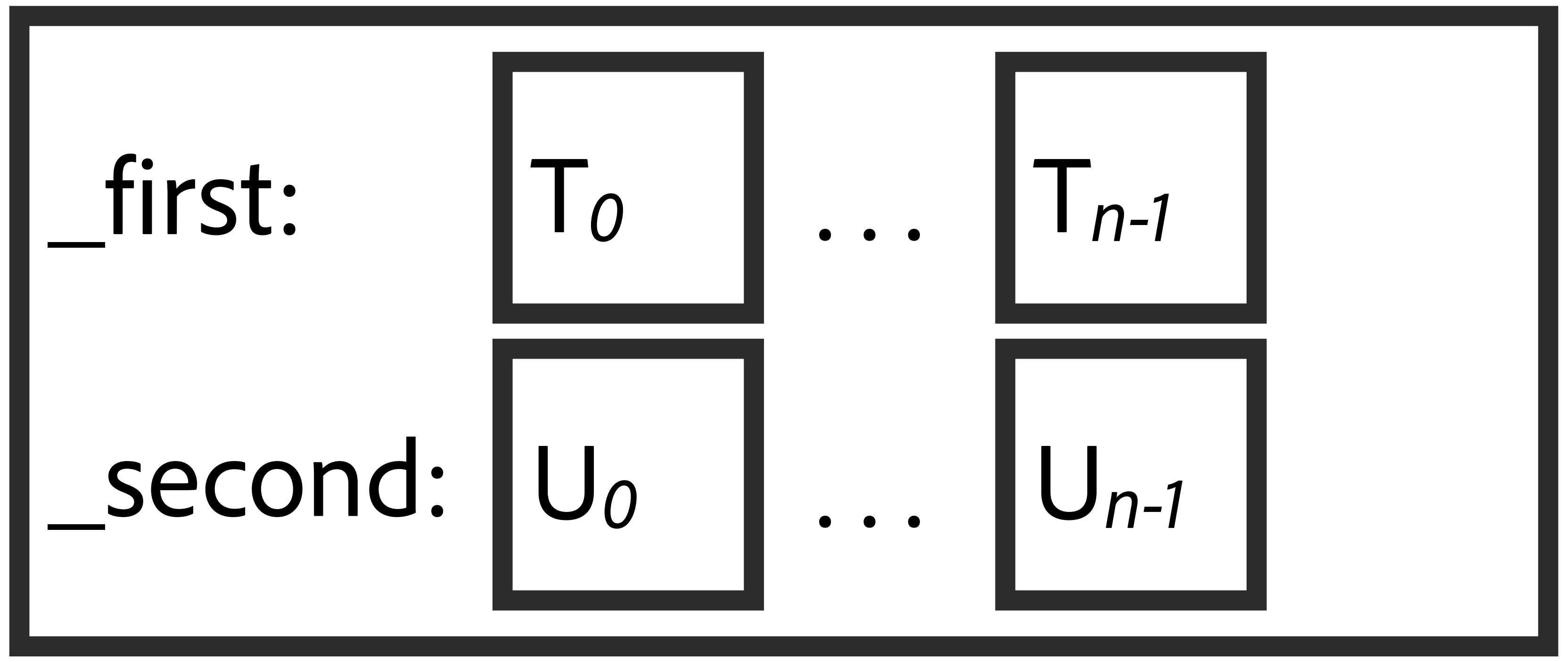
The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  _second.push_back(e.second);
}
```

strong guarantee (x2) - does not compose

Three useful guarantees regarding errors



_first: T_0 . . . T_{n-1}

_second: U_0 . . . U_{n-1}

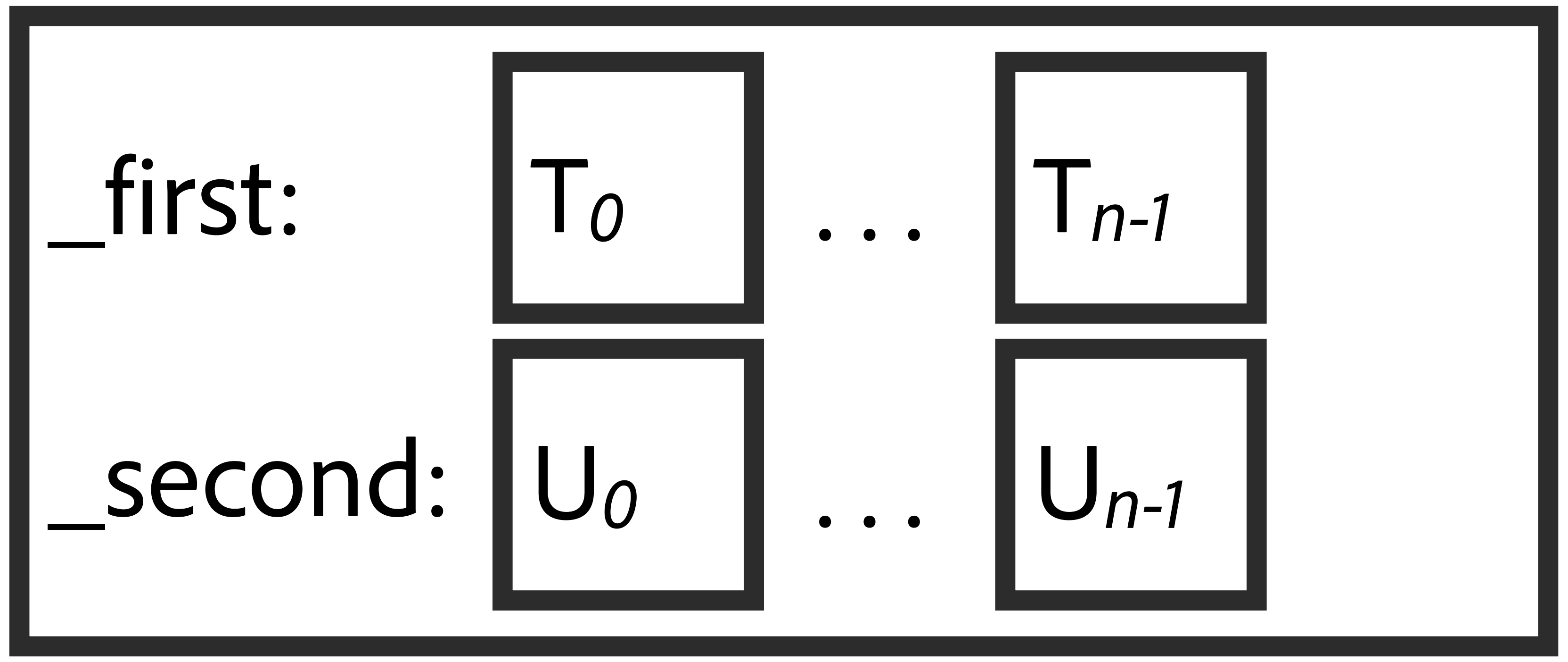
The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  _second.push_back(e.second);
}
```

Three useful guarantees regarding errors



_first: T_0 . . . T_{n-1}

_second: U_0 . . . U_{n-1}

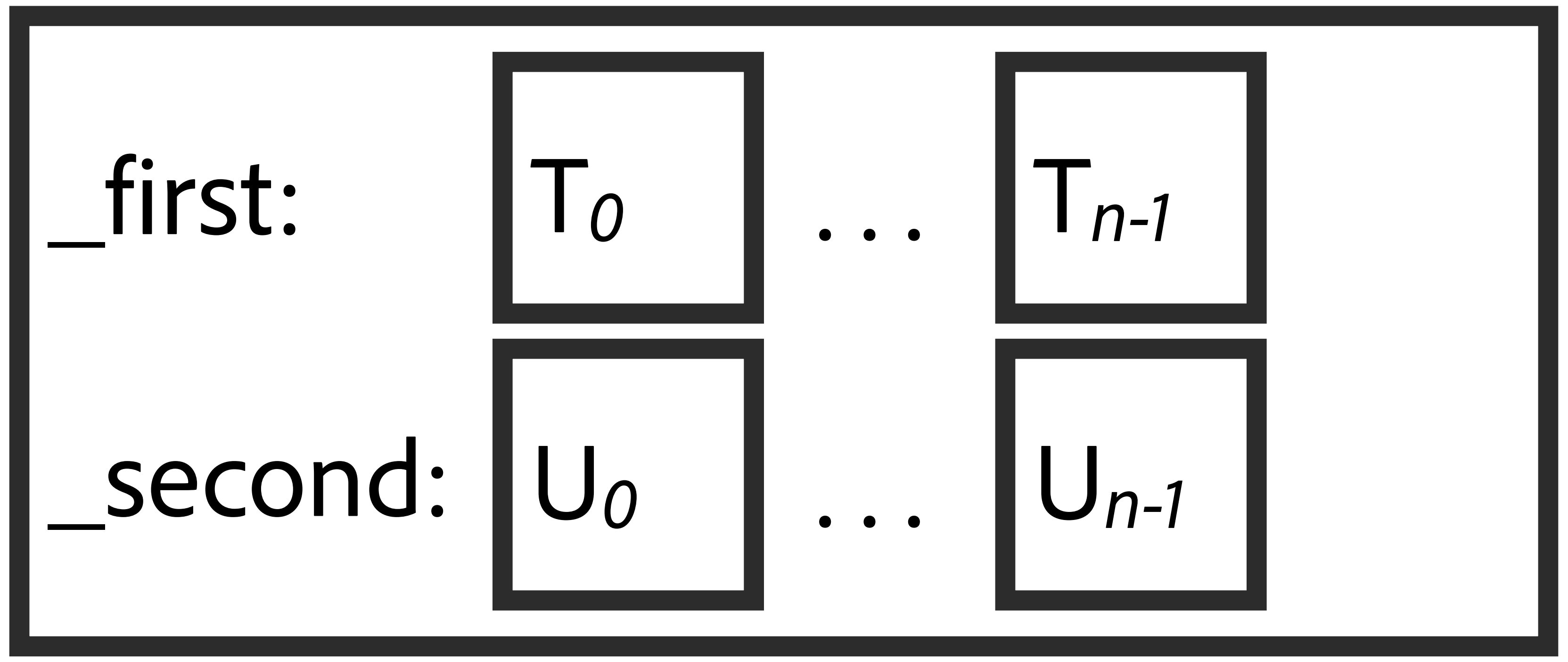
The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  _second.push_back(e.second);
}
```

Three useful guarantees regarding errors



_first: T_0 . . . T_{n-1}

_second: U_0 . . . U_{n-1}

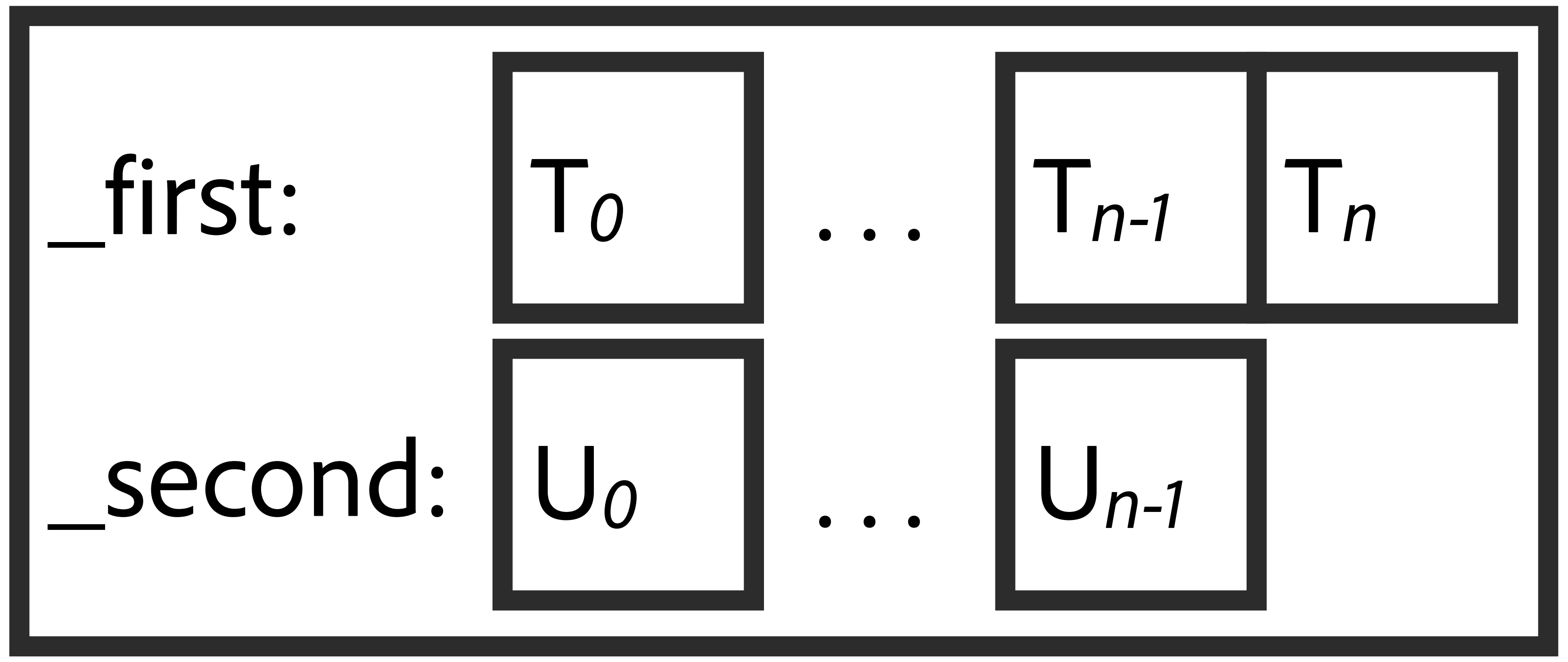
The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  _second.push_back(e.second);
}
```

Three useful guarantees regarding errors



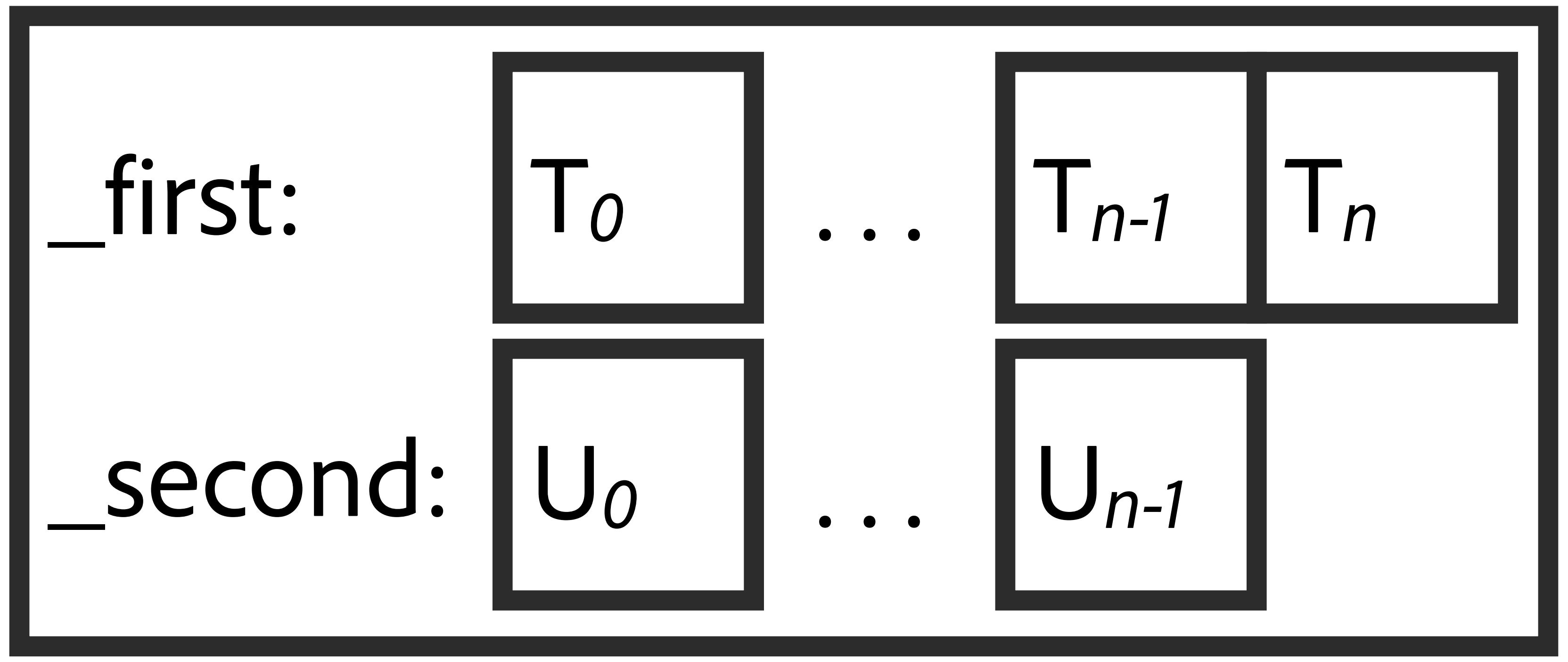
The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  _second.push_back(e.second);
}
```

Three useful guarantees regarding errors



_first: T_0 . . . T_{n-1} T_n

_second: U_0 . . . U_{n-1}

The nothrow guarantee: no errors can occur.

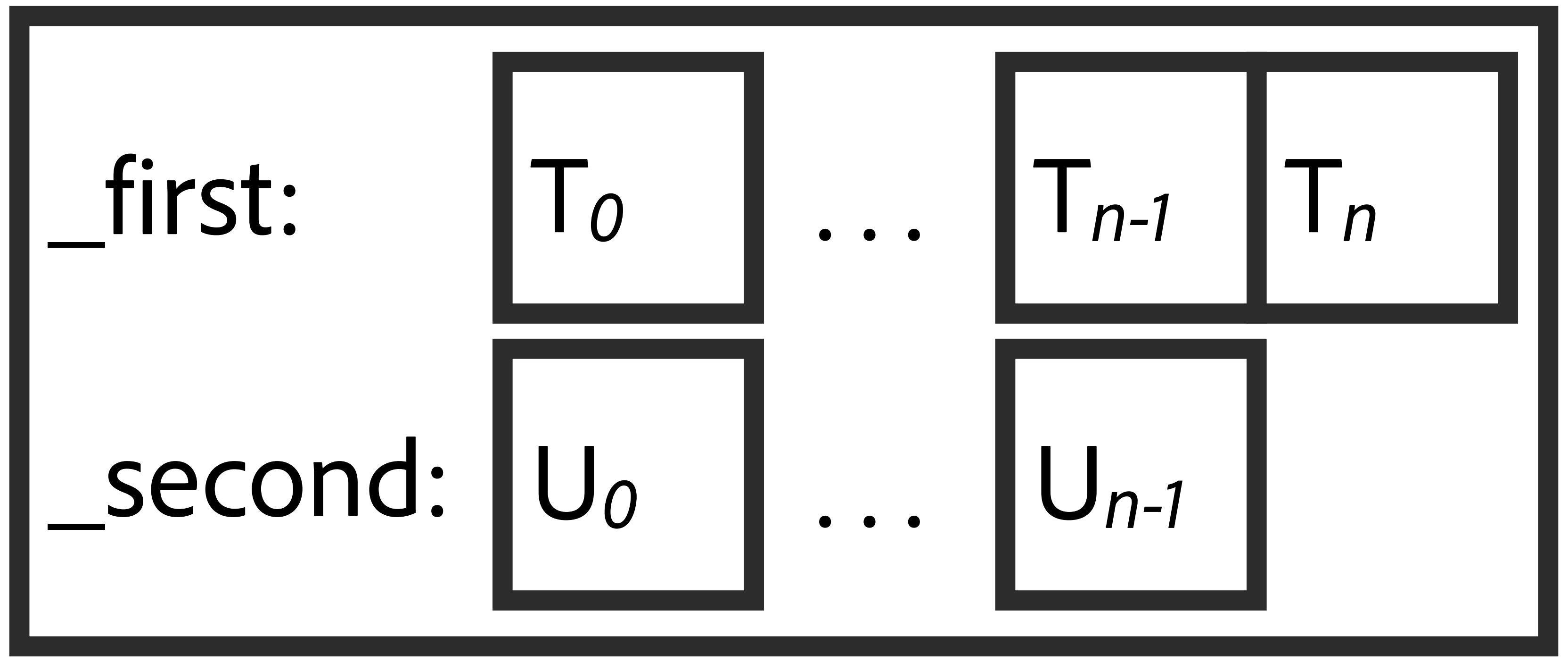
The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  _second.push_back(e.second);
}
```

Adobe

Three useful guarantees regarding errors



_first: T_0 . . . T_{n-1} T_n

_second: U_0 . . . U_{n-1}

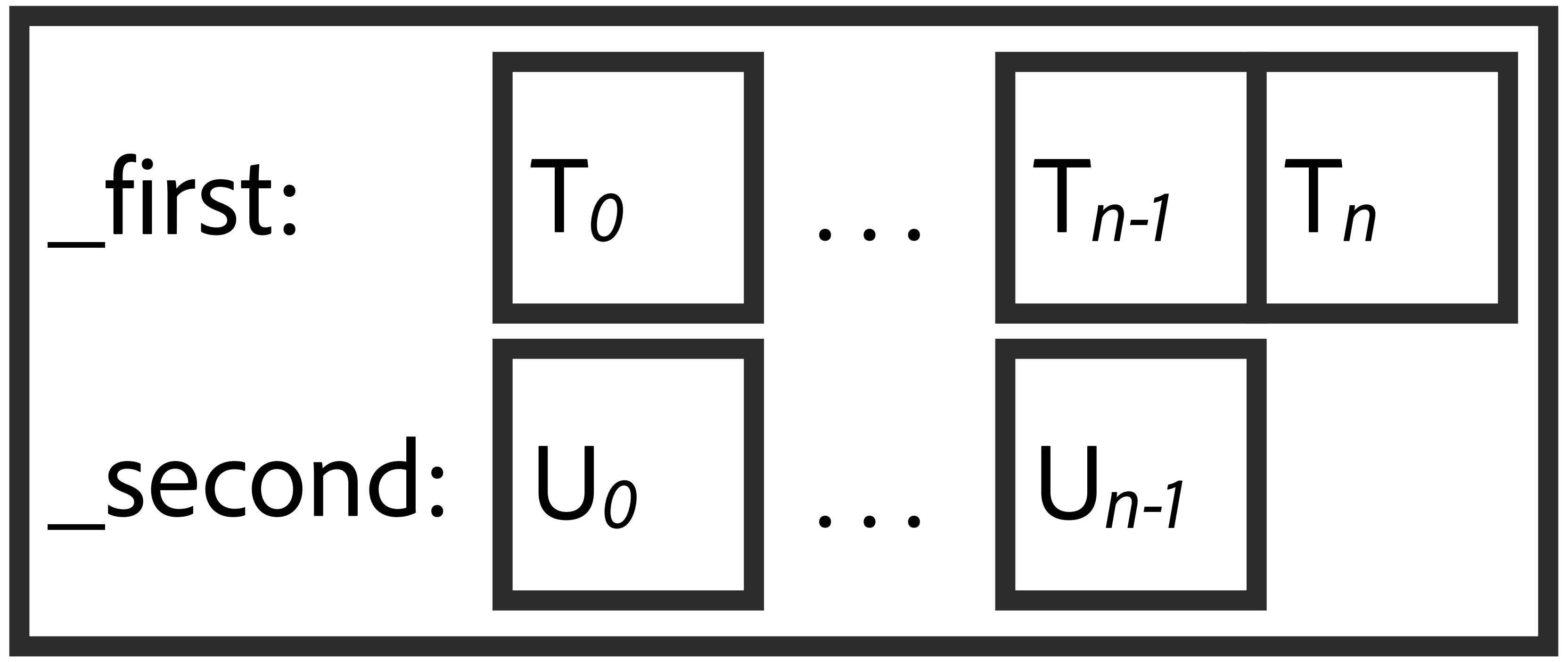
The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  _second.push_back(e.second);
}
```

Three useful guarantees regarding errors



_first: T_0 . . . T_{n-1} T_n

_second: U_0 . . . U_{n-1}

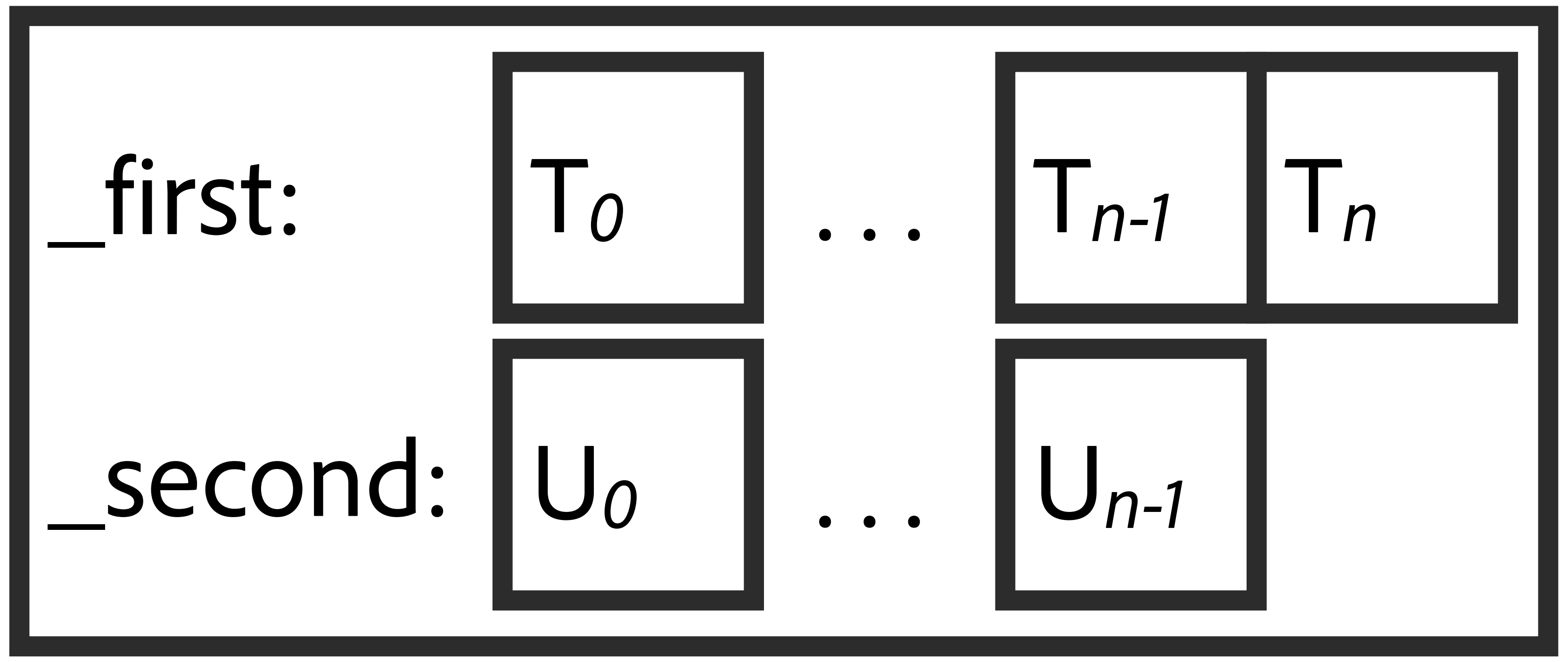
The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  _second.push_back(e.second);
}
```

Three useful guarantees regarding errors



_first: T_0 . . . T_{n-1} T_n

_second: U_0 . . . U_{n-1}

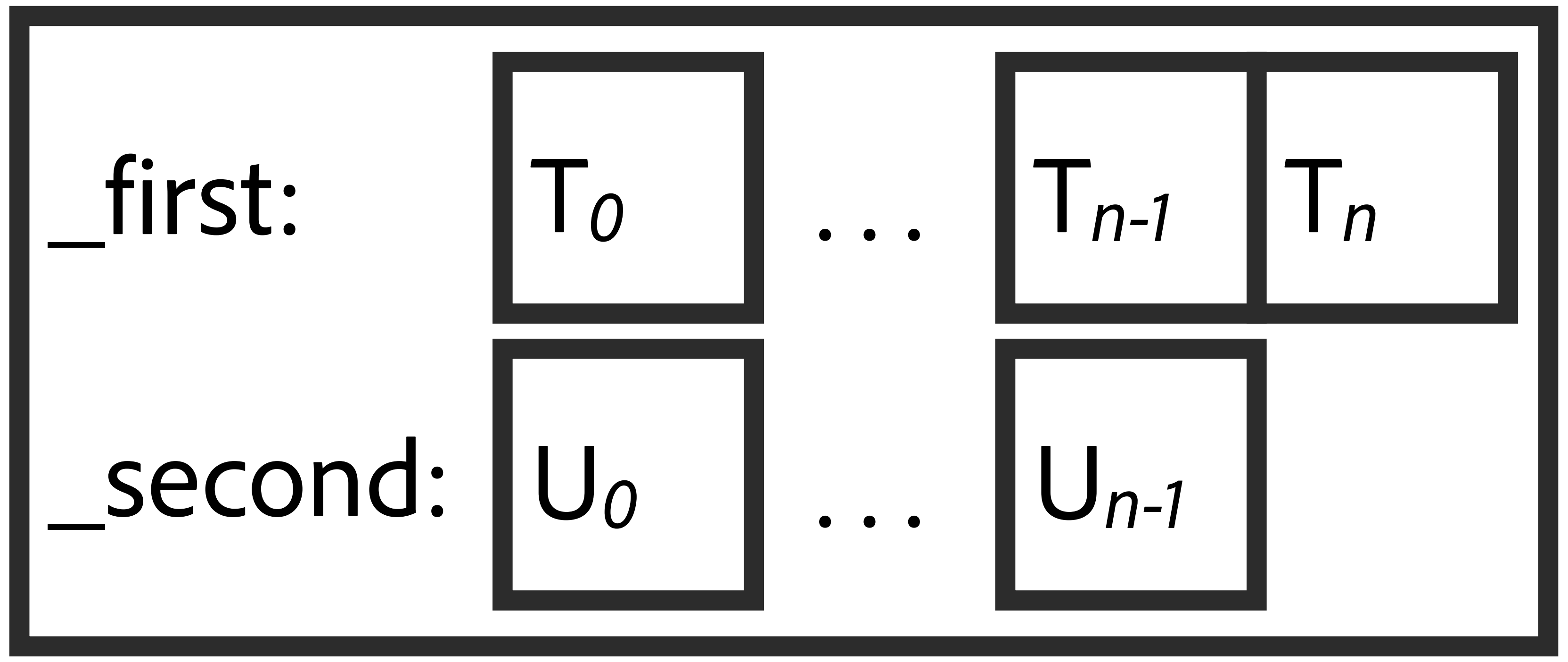
The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  _second.push_back(e.second);
}
```

Three useful guarantees regarding errors



_first: T_0 . . . T_{n-1} T_n

_second: U_0 . . . U_{n-1}

The nothrow guarantee: no errors can occur.

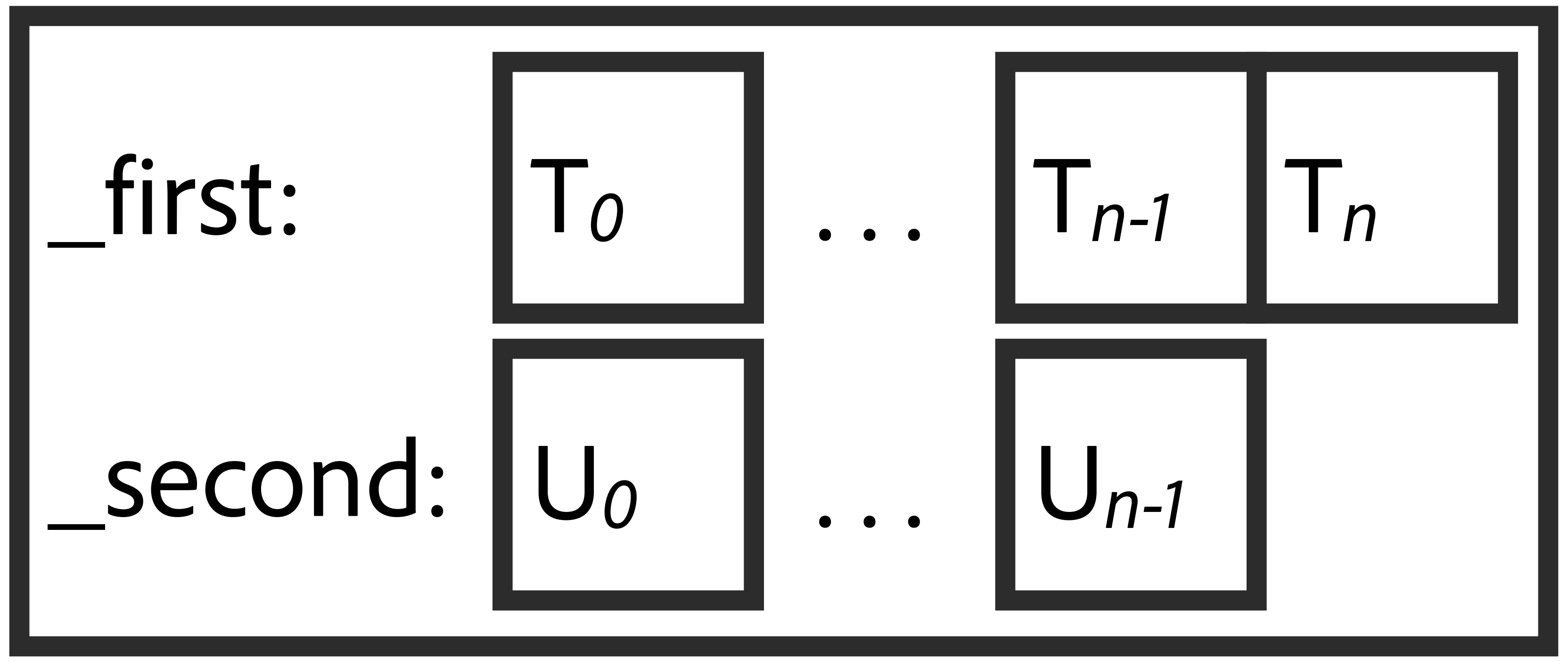
The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
        _second.push_back(e.second);

}
```

Three useful guarantees regarding errors



_first: T_0 . . . T_{n-1} T_n

_second: U_0 . . . U_{n-1}

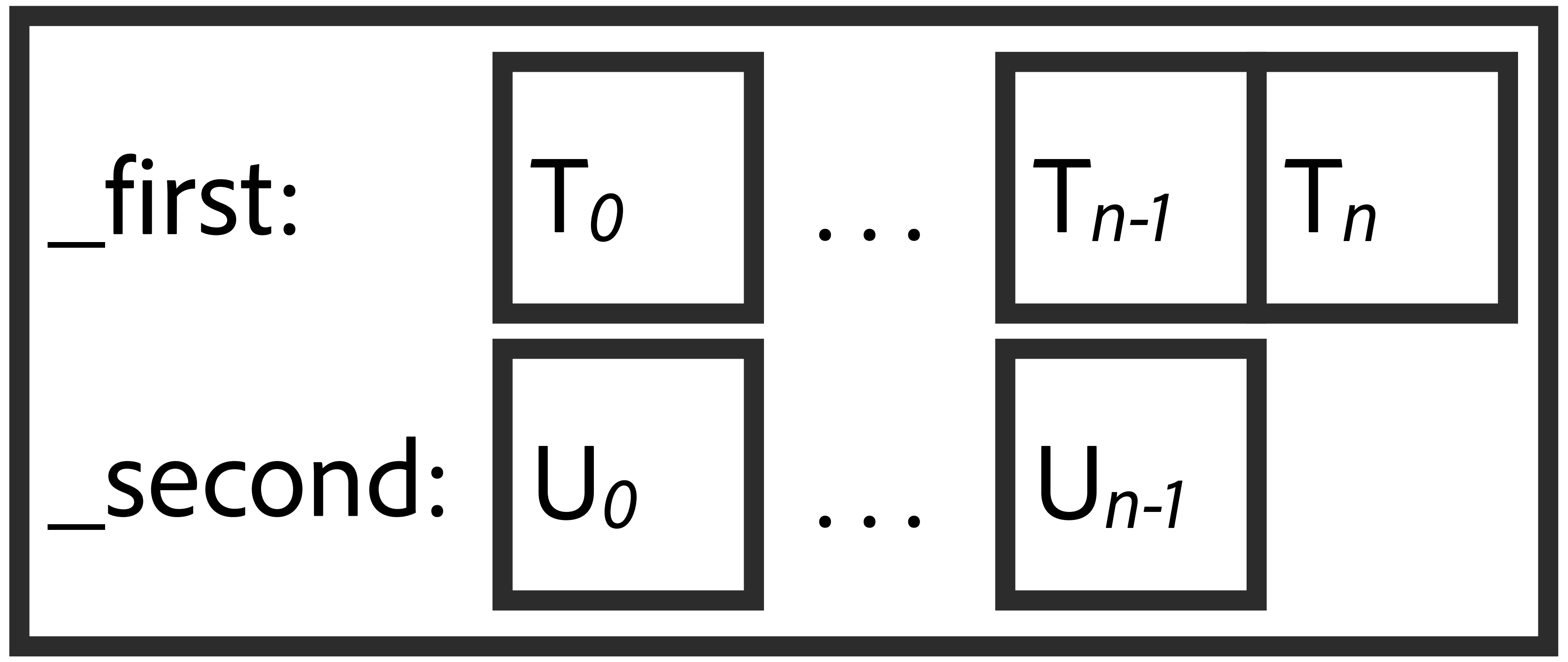
The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  try { _second.push_back(e.second); }
  catch(...) {                    throw; }
}
```

Three useful guarantees regarding errors



_first: T_0 . . . T_{n-1} T_n

_second: U_0 . . . U_{n-1}

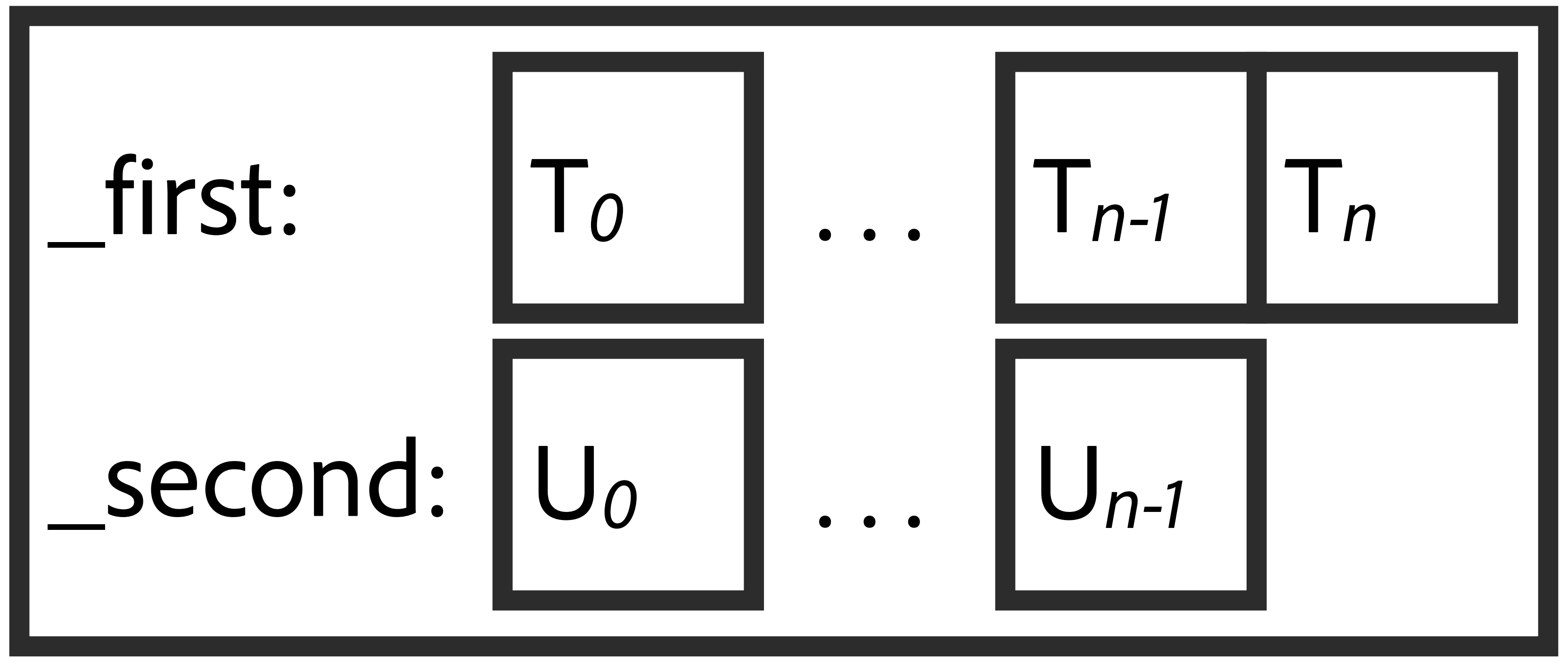
The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  try { _second.push_back(e.second); }
  catch(...) {                    throw; }
}
```

Three useful guarantees regarding errors



_first: T_0 . . . T_{n-1} T_n

_second: U_0 . . . U_{n-1}

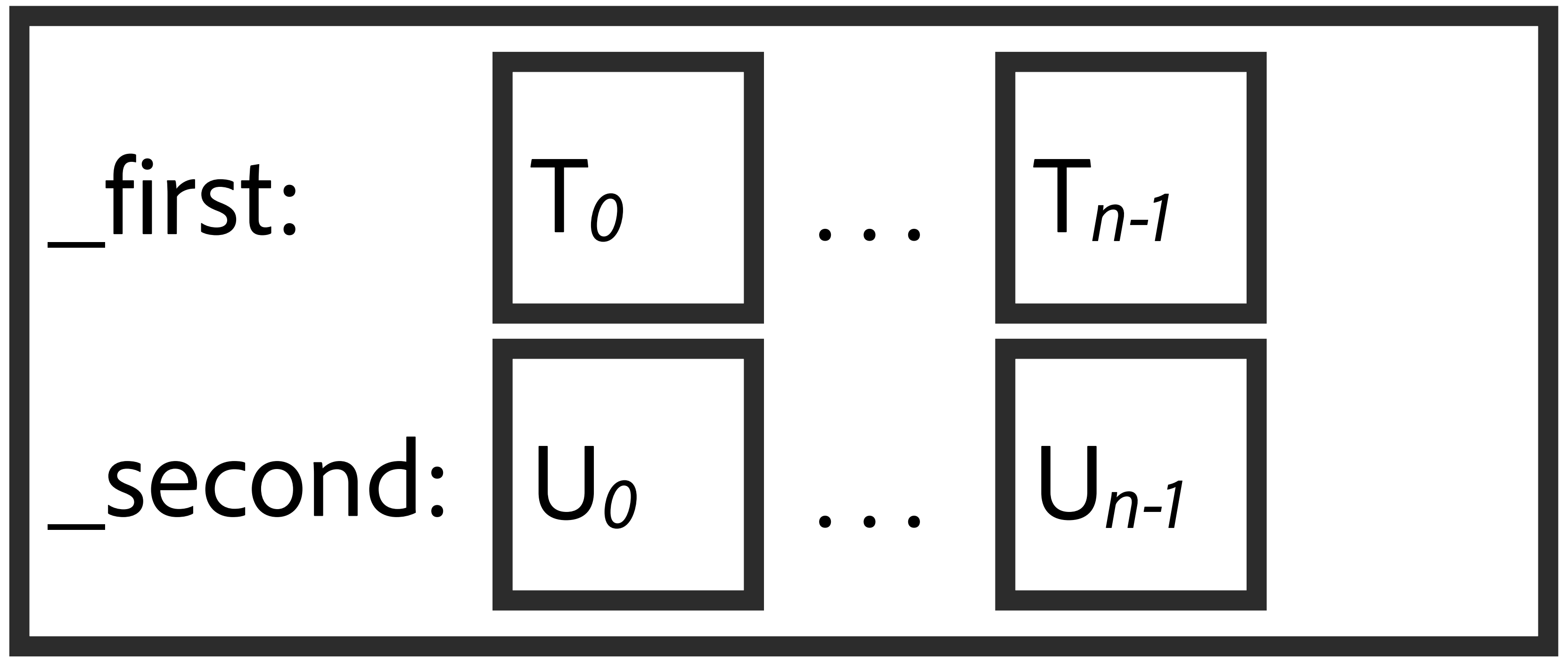
The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  try { _second.push_back(e.second); }
  catch(...) { _first.pop_back(); throw; }
}
```

Three useful guarantees regarding errors



_first: T_0 . . . T_{n-1}

_second: U_0 . . . U_{n-1}

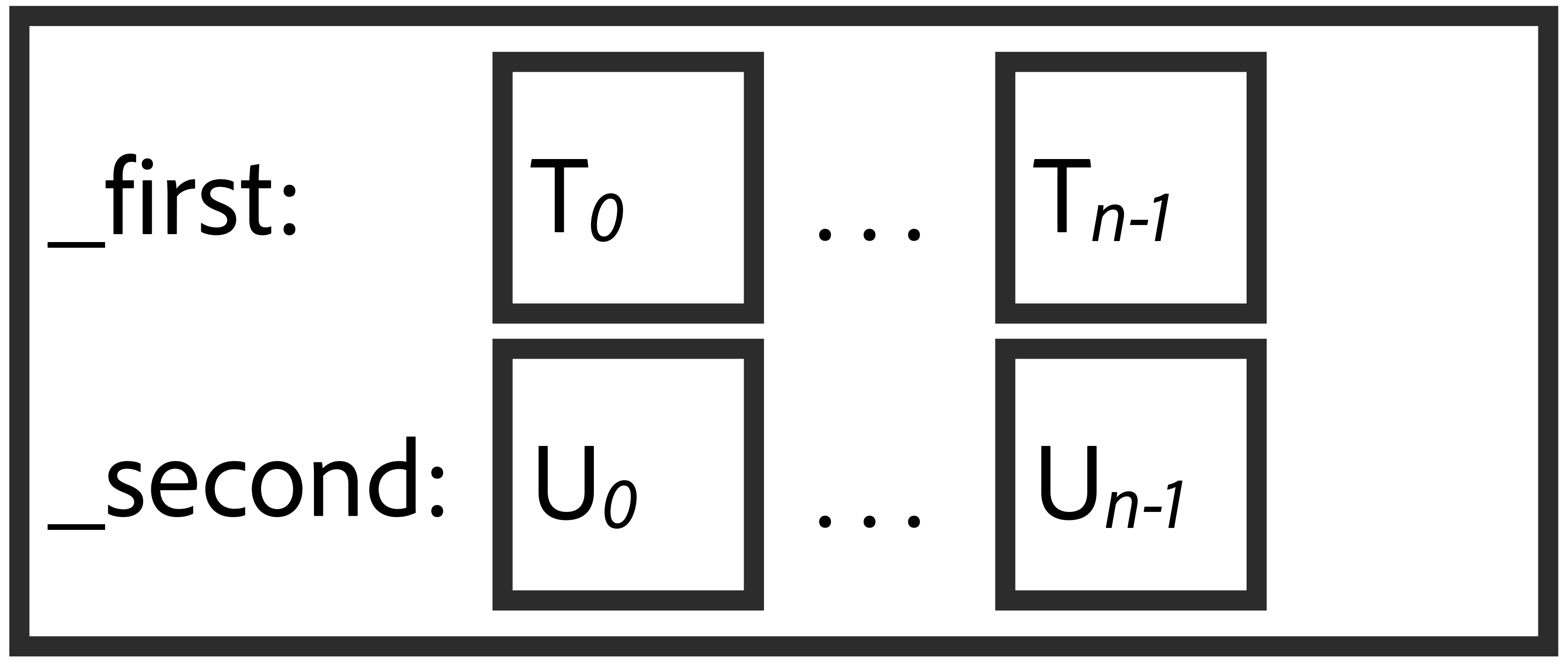
The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  try { _second.push_back(e.second); }
  catch(...) { _first.pop_back(); throw; }
}
```

Three useful guarantees regarding errors



The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

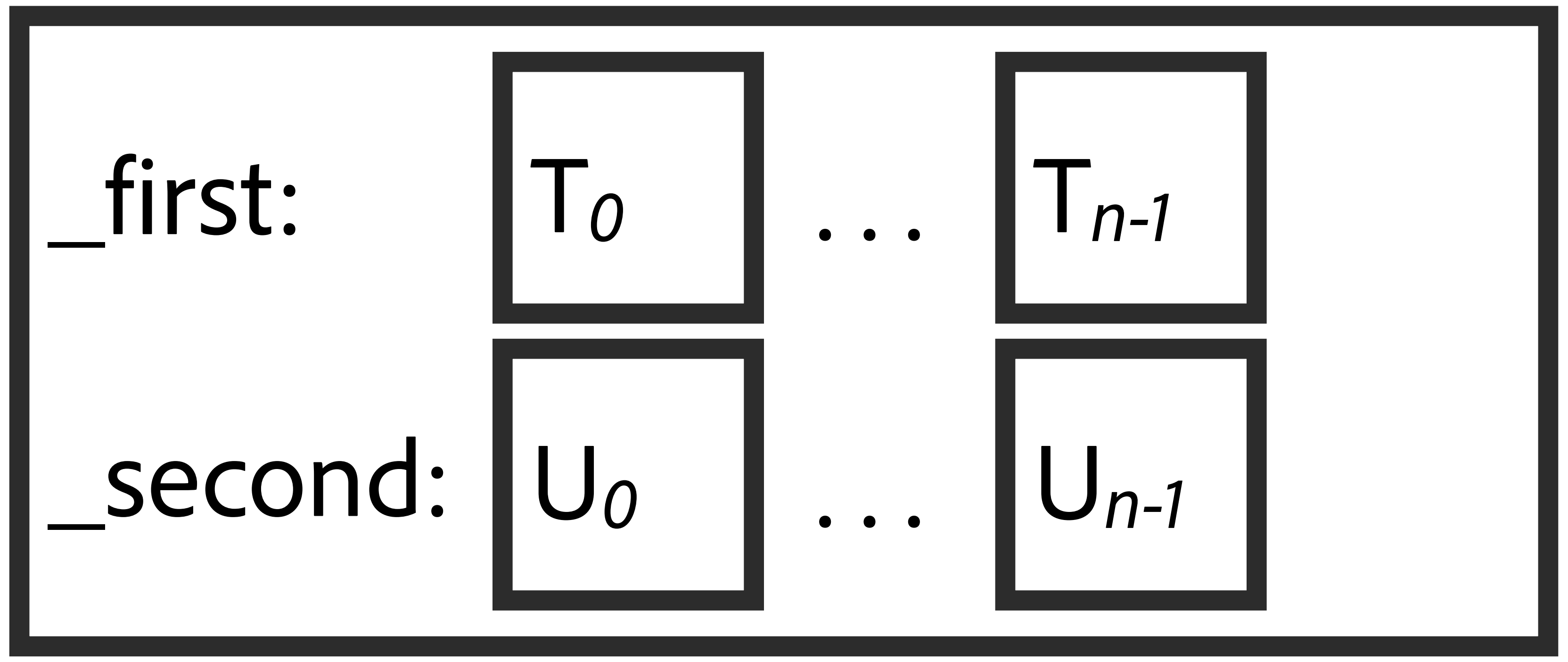
The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void push_back(const pair<T, U>& e)

  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  try { _second.push_back(e.second); }
  catch(...) { _first.pop_back(); throw; }
}
```

Adobe

Three useful guarantees regarding errors



_first: T_0 . . . T_{n-1}

_second: U_0 . . . U_{n-1}

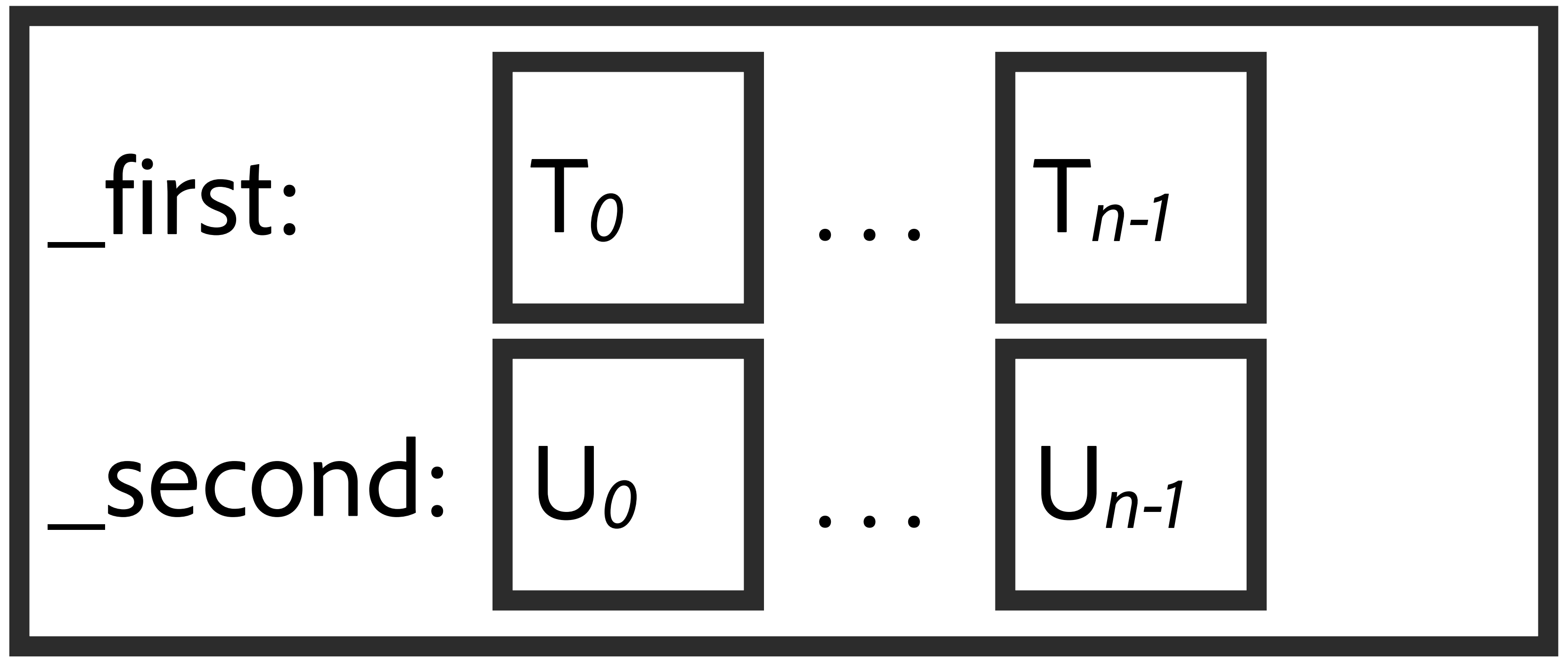
The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void push_back(const pair<T, U>& e)
  // if an exception is thrown there are no effects
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) }
{
  _first.push_back(e.first);
  try { _second.push_back(e.second); }
  catch(...) { _first.pop_back(); throw; }
}
```

Three useful guarantees regarding errors



_first: T_0 . . . T_{n-1}

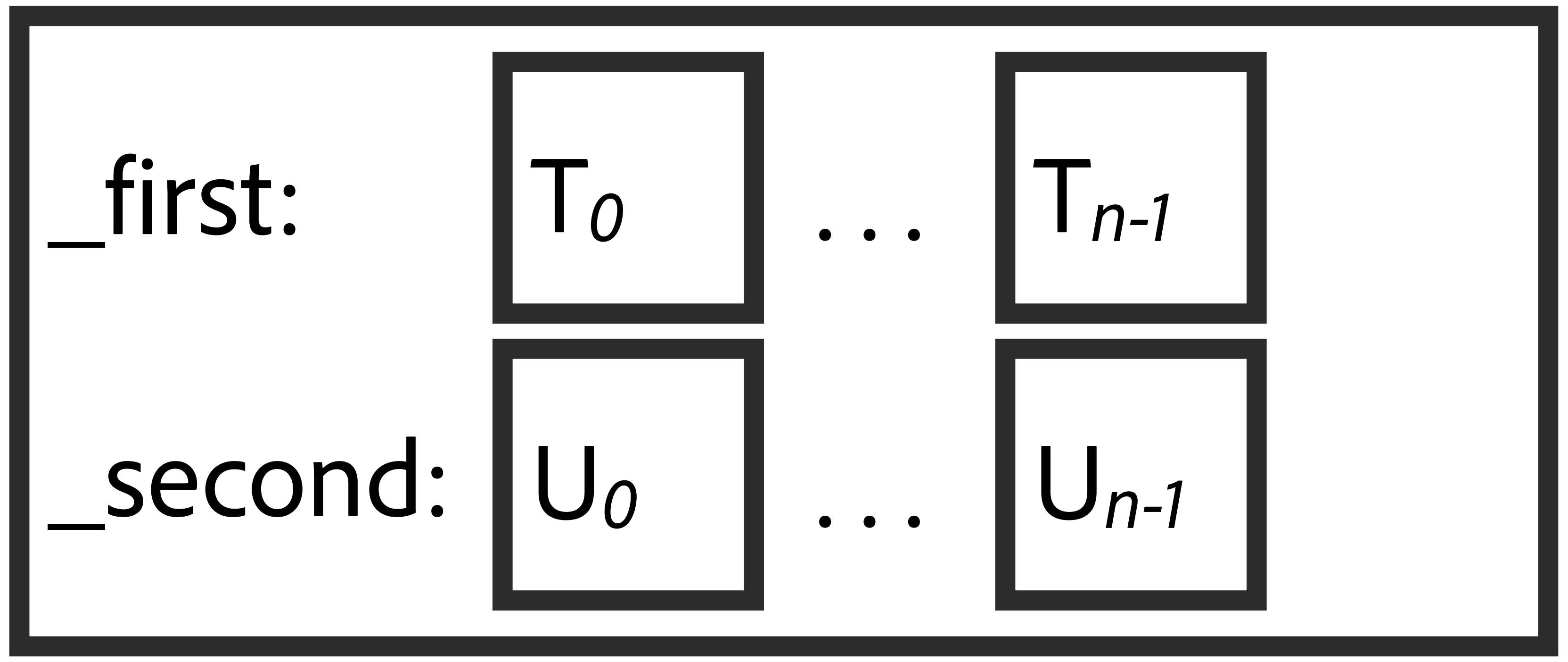
_second: U_0 . . . U_{n-1}

The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

Three useful guarantees regarding errors



_first: T_0 . . . T_{n-1}

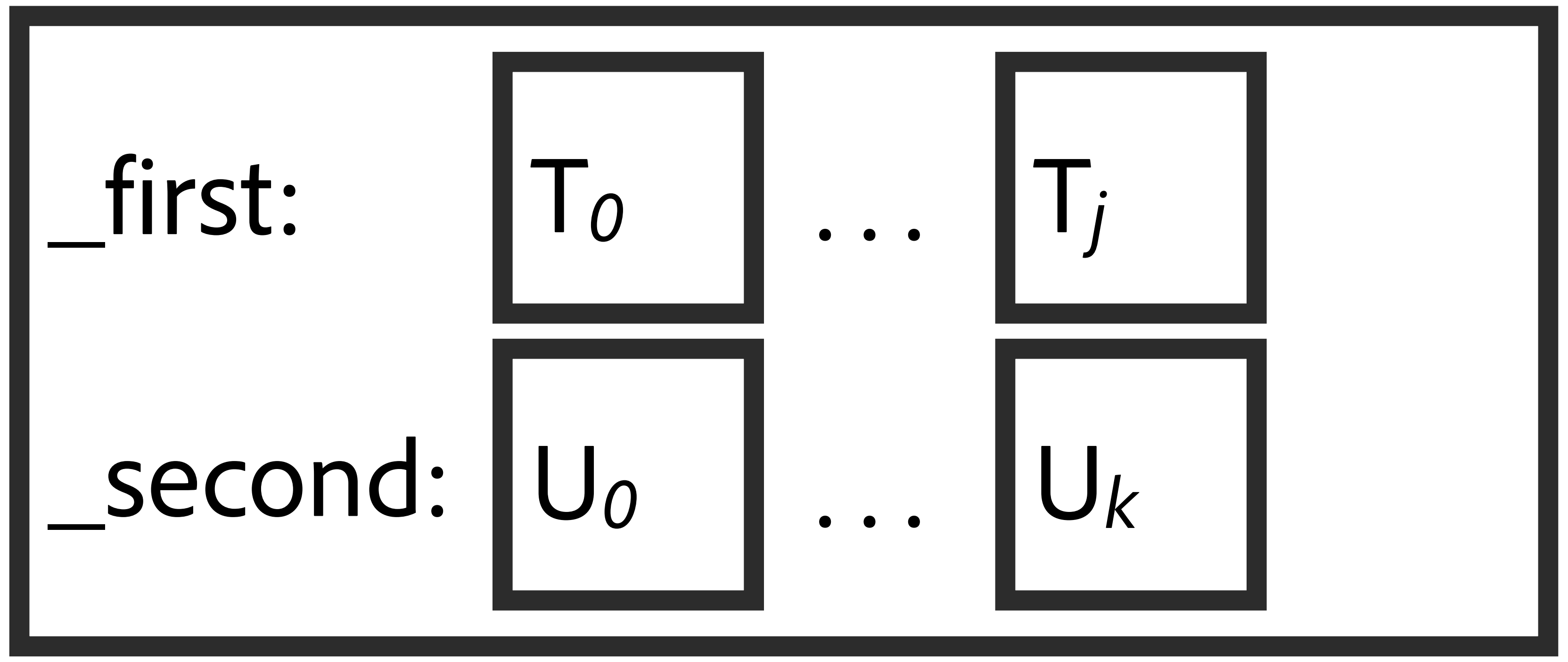
_second: U_0 . . . U_{n-1}

The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

Three useful guarantees regarding errors



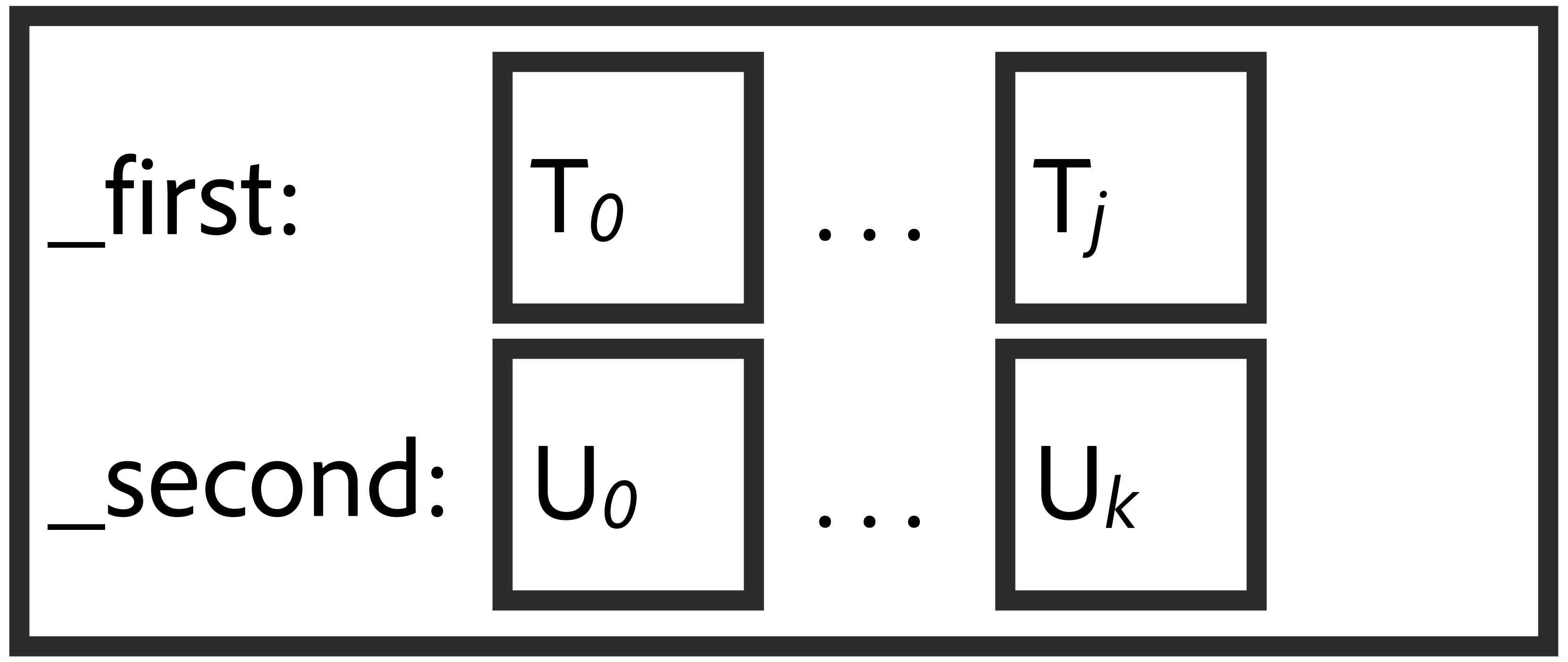
The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void insert(size_t p, const pair<T, U>& e)
{
   _first.insert(begin(_first) + p, e.first);
   _second.insert(begin(_second) + p, e.second);
}
```

Three useful guarantees regarding errors



_first: T_0 . . . T_j

_second: U_0 . . . U_k

The nothrow guarantee: no errors can occur.

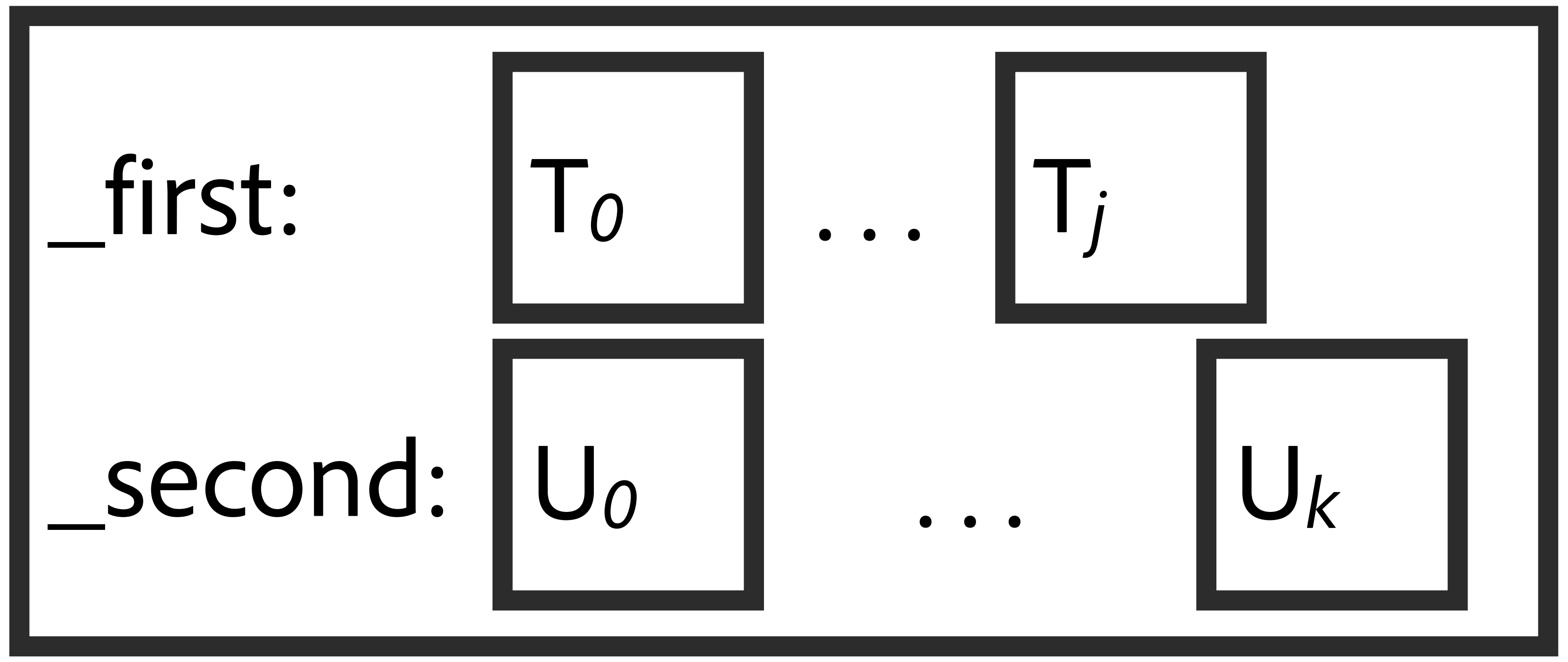
The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void insert(size_t p, const pair<T, U>& e)
{
    _first.insert(begin(_first) + p, e.first);
    _second.insert(begin(_second) + p, e.second);
}
```

basic guarantee (x2) - does not compose

Three useful guarantees regarding errors



_first: T_0 . . . T_j

_second: U_0 . . . U_k

The nothrow guarantee: no errors can occur.

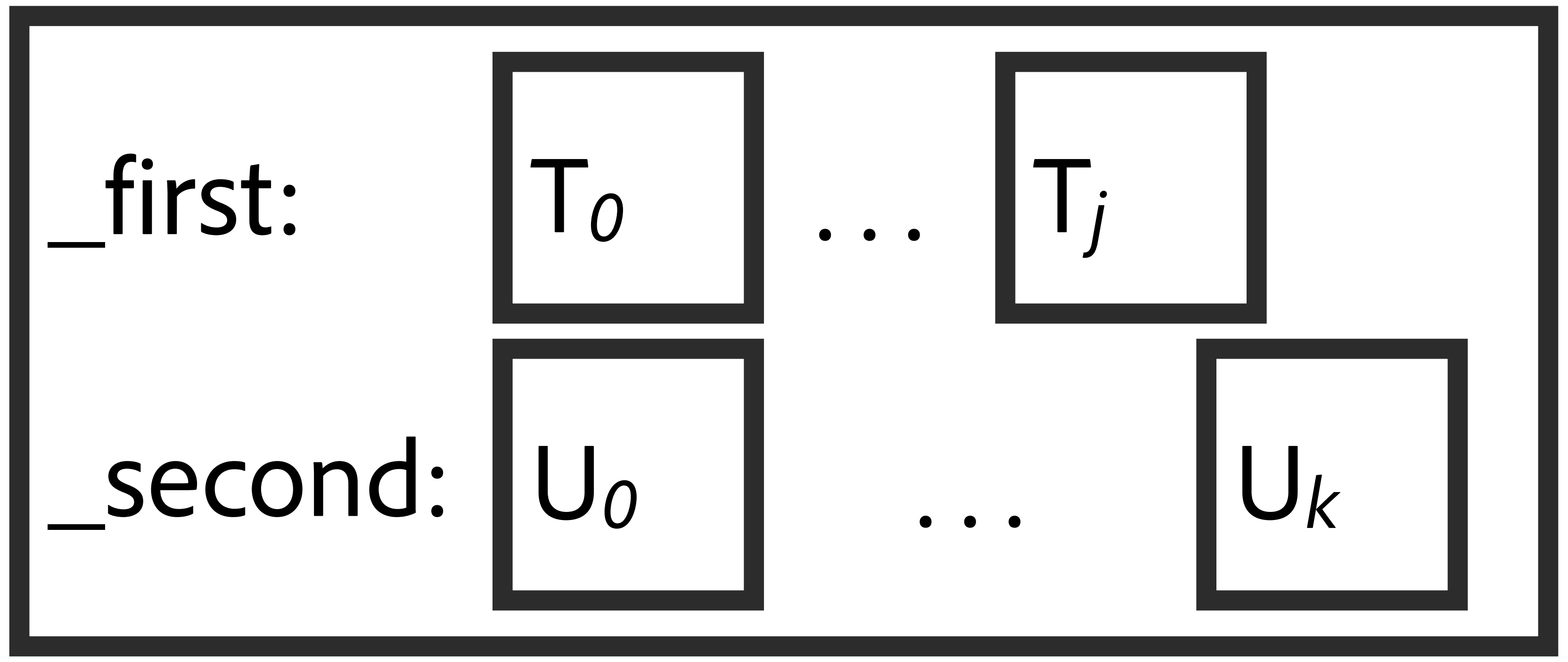
The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void insert(size_t p, const pair<T, U>& e)
{
    _first.insert(begin(_first) + p, e.first);
    _second.insert(begin(_second) + p, e.second);
}
```

basic guarantee (x2) - does not compose

Three useful guarantees regarding errors



_first: T_0 . . . T_j

_second: U_0 . . . U_k

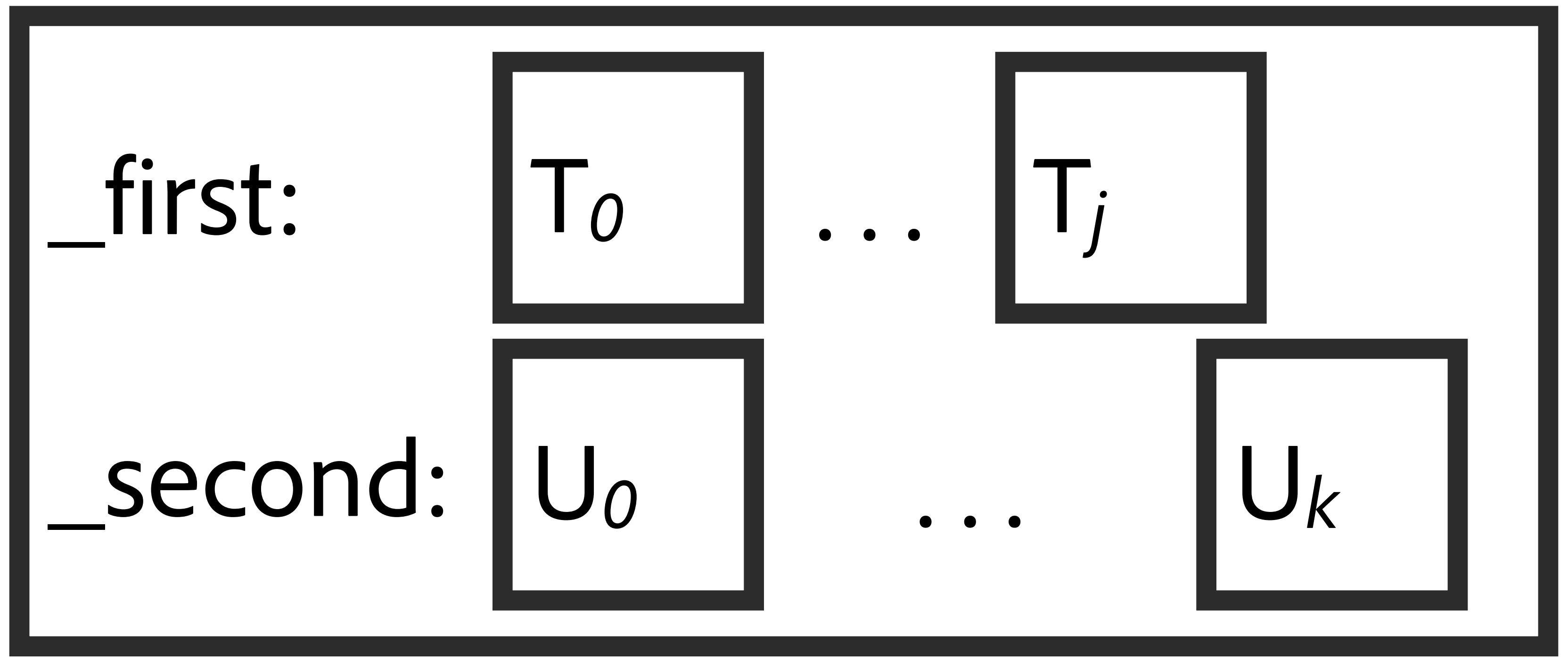
The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void insert(size_t p, const pair<T, U>& e)
{
    _first.insert(begin(_first) + p, e.first);
    _second.insert(begin(_second) + p, e.second);
}
```

Three useful guarantees regarding errors



_first: T_0 . . . T_j

_second: U_0 . . . U_k

The nothrow guarantee: no errors can occur.

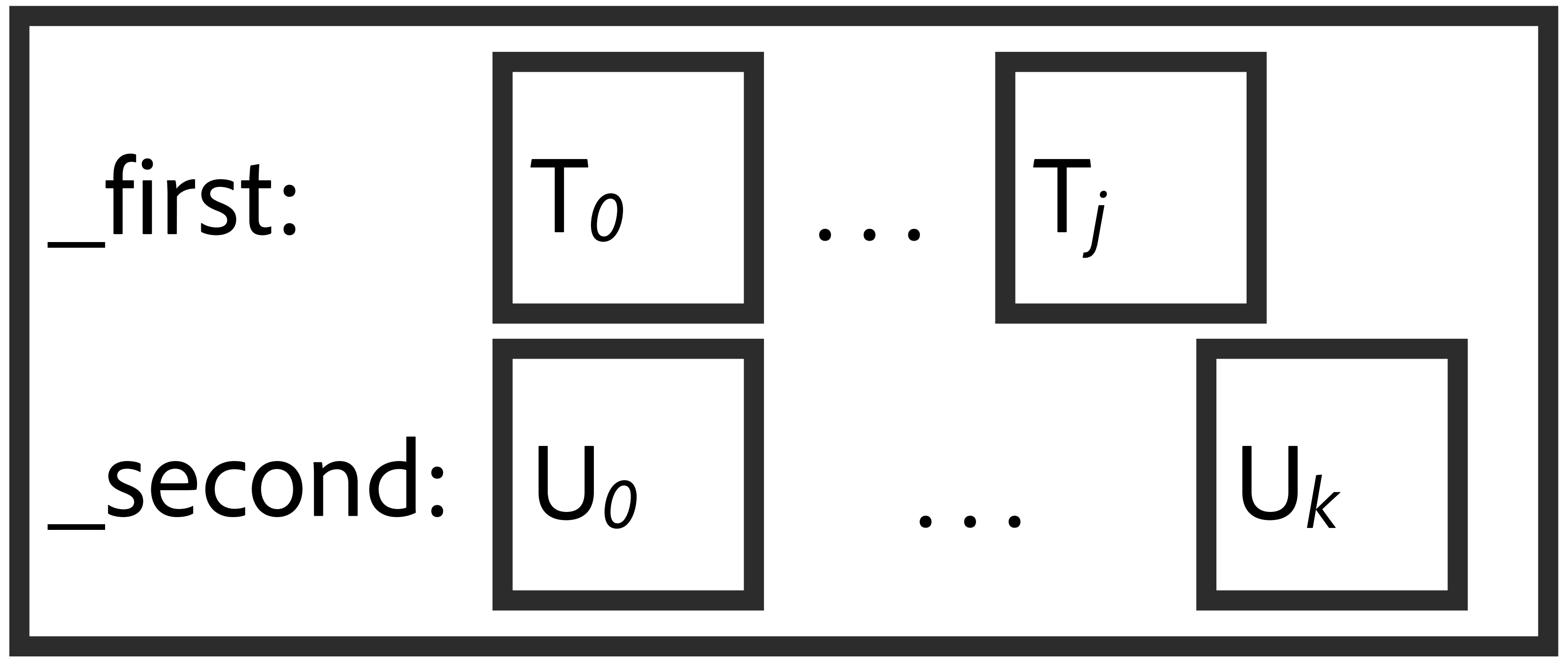
The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void insert(size_t p, const pair<T, U>& e)
{
   try {
     _first.insert(begin(_first) + p, e.first);
     _second.insert(begin(_second) + p, e.second);
   }
   catch(...) {                                 throw; }
}
```

Adobe

Three useful guarantees regarding errors



_first: T_0 . . . T_j

_second: U_0 . . . U_k

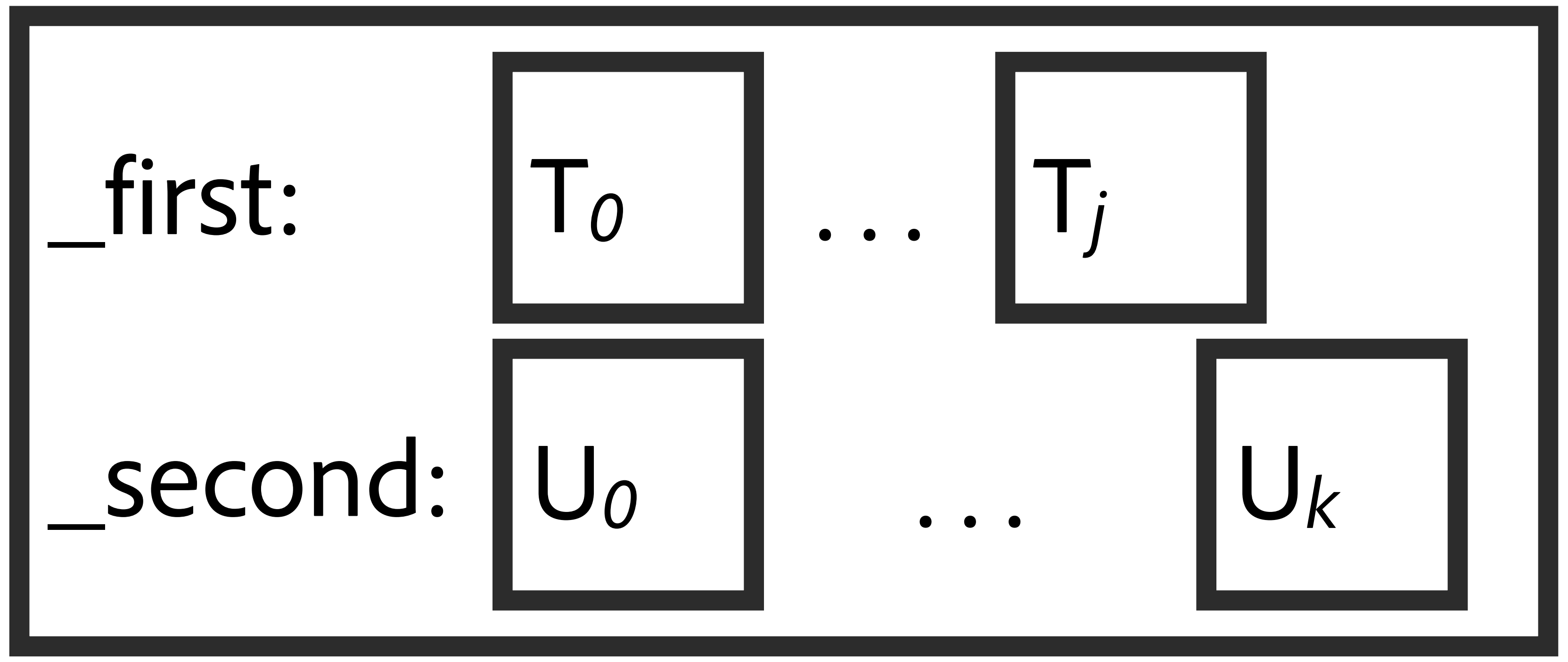
The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void insert(size_t p, const pair<T, U>& e)
{
   try {
     _first.insert(begin(_first) + p, e.first);
     _second.insert(begin(_second) + p, e.second);
   }
   catch(...) {                                     throw; }
}
```

Three useful guarantees regarding errors



_first: T_0 . . . T_j

_second: U_0 . . . U_k

The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void insert(size_t p, const pair<T, U>& e)
{
   try {
     _first.insert(begin(_first) + p, e.first);
     _second.insert(begin(_second) + p, e.second);
   }
   catch(...) { _first.clear(); _second.clear(); throw; }
}
```

Adobe

Three useful guarantees regarding errors

_first:

_second:

The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void insert(size_t p, const pair<T, U>& e)
{
   try {
     _first.insert(begin(_first) + p, e.first);
     _second.insert(begin(_second) + p, e.second);
   }
   catch(...) { _first.clear(); _second.clear(); throw; }
}
```

Too much work

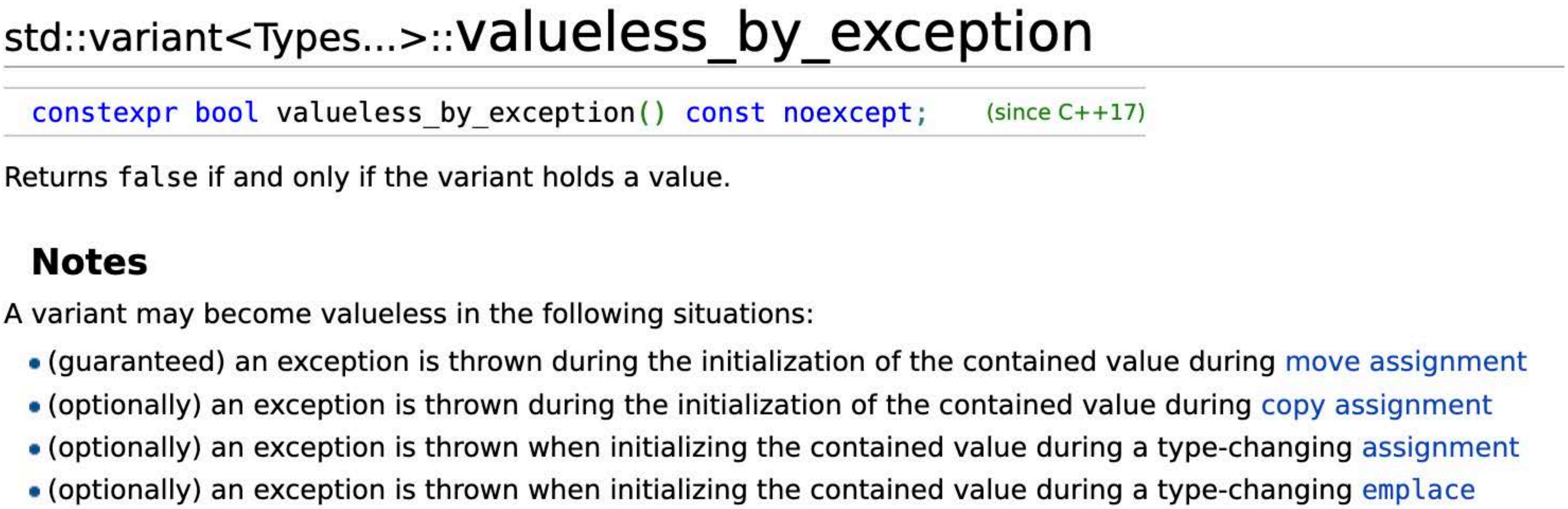
The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

```
void insert(size_t p, const pair<T, U>& e)
{
   try {
     _first.insert(begin(_first) + p, e.first);
     _second.insert(begin(_second) + p, e.second);
   }
   catch(...) { _first.clear(); _second.clear(); throw; }
}
```

Weakened



std::variant<Types...>::valueless_by_exception

```
constexpr bool valueless_by_exception() const noexcept;      (since C++17)
```

Returns `false` if and only if the variant holds a value.

Notes

A variant may become valueless in the following situations:

- (guaranteed) an exception is thrown during the initialization of the contained value during move assignment
- (optionally) an exception is thrown during the initialization of the contained value during copy assignment
- (optionally) an exception is thrown when initializing the contained value during a type-changing assignment
- (optionally) an exception is thrown when initializing the contained value during a type-changing `emplace`

Too much work

```
void insert(size_t p, const pair<T, U>& e)
{
   try {
     _first.insert(begin(_first) + p, e.first);
     _second.insert(begin(_second) + p, e.second);
   }
   catch(...) { _first.clear(); _second.clear(); throw; }
}
```

Too much work

```
void insert(size_t p, const pair<T, U>& e)
  ... // Contracts
{
  try {
    _first.insert(begin(_first) + p, e.first);
    _second.insert(begin(_second) + p, e.second);
  } catch (...) {
    _first.clear();
    _second.clear();
    throw;
  }
}
```

Too much work

```
void insert(size_t p, const pair<T, U>& e)
  ... // Contracts
{
  try {
    _first.insert(begin(_first) + p, e.first);
    _second.insert(begin(_second) + p, e.second);
  } catch (...) {
    _first.clear();
    _second.clear();
    throw;

  }
}
```

Too much work

```
void insert(size_t p, const pair<T, U>& e)
  ... // Contracts
{
  try {
    _first.insert(begin(_first) + p, e.first);
    _second.insert(begin(_second) + p, e.second);
  } catch (...) {
    _first.clear();
    _second.clear();
    throw;
    partially_mutated = true;
  }
}
```

Inspect Try Blocks

```
try {
   a.push_back(e); // a is under mutation
} catch (...) { }
use(a); // BUG - a may be meaningless here
```

Inspect Try Blocks

```
try {
    a.push_back(e); // a is under mutation
} catch (...) { }
use(a); // BUG - a may be meaningless here
```

Inspect Try Blocks

```
try {
   a.push_back(e); // a is under mutation
} catch (...) { }
use(a); // BUG - a may be meaningless here
```

# upshot for insert

```
void insert(size_t p, const pair<T, U>& e)
  ... // Contracts
{
  try {
    _first.insert(begin(_first) + p, e.first);
    _second.insert(begin(_second) + p, e.second);
  } catch (...) {
    _first.clear();
    _second.clear();
    throw;
  }
}
```

# upshot for insert

```
void insert(size_t p, const pair<T, U>& e)
  ... // Contracts
{
  _first.insert(begin(_first) + p, e.first);
  _second.insert(begin(_second) + p, e.second);
}
```

Where try/catch remains

To clean up unmanaged resources

To report the error and recover

Unless the type has all default destruction and assignment operators,
we may need to catch in non-const member functions to ensure the object is discardable

To provide a strong guarantee

zip_vector | push_back discardable

```
void push_back(const pair<T, U>& e)
  ... // Contracts
{
  _first.push_back(e.first);
  _second.push_back(e.second);
}
```

Three useful guarantees regarding errors

The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

The basic guarantee: if an error occurs, invariants are upheld and no resources leak.

Three useful guarantees regarding errors

The nothrow guarantee: no errors can occur.

The strong guarantee: if an error occurs, the operation has no effects.

The minimal guarantee: if an error occurs, the object is discardable, and no resources leak.

Code > Documentation

Contracts | The Reckoning

```
template <class T, class U>
 requires copyable<T> && assignable<T> && copyable<U> && assignable<U>
class zip_vector {
public:
 const vector<T>& first() const
  post (r){ !testing || equal(*this, r,
    [](const auto& a, const auto& b){
     return &a.first == &b;
     }) };
 const vector<U>& second() const
  post (r){ !testing || equal(*this, r,
    [](const auto& a, const auto & b){
     return &second == &b;
    }) };

 invariant { size(first()) == size(second()) }

 size_t size() const
  post (r){ r == size(first()) && r == size(last()) };
 bool empty() const
  post (r){ r == (size() == 0) }
```

```
 void pop_back() noexcept
  pre { size() < 0 }
  post [old_size = size()] { size() == old_size - 1 }
  post [old = *this] { !testing || equal(begin(), end(), begin(old)) };

 void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing
    || equal(begin(old), end(old), begin()) };

 void insert(size_t p, const pair<T, U>& e)
  pre { p <= size() }
  post [old_size = size()] { size() == old_size + 1 }
  post { (*this)[p] == e }
  post [old = *this] { !testing
    || (equal(begin(), begin() + p, begin(old))
    && equal(begin() + p + 1, end(), begin(old) + p)) };

 ...
};
```

Contracts | The Reckoning

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) };
```

Contracts | The Reckoning

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) };
```

Contracts | The Reckoning

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) };
```

Contracts | The Reckoning

```
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) };
```

Contracts | The Reckoning

```
// Appends e
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) };
```

Contracts | The Reckoning

```
// Appends e
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) };
```

Contracts | The Reckoning

```
// Appends e
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) };
```

Contracts | The Reckoning

```
// Appends e
void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing || equal(begin(old), end(old), begin()) };
```

Contracts | The Reckoning

```
template <class T, class U>
class zip_vector {
public:
  // Returns the first element of each pair
  const vector<T>& first() const;
  // Returns the second element of each pair
  const vector<U>& second() const;

  // Returns the number of elements
  size_t size() const;
  // Returns true iff there are no elements
  bool empty() const;

  // Removes the last element
  void pop_back() noexcept;

  // Appends e
  void push_back(const pair<T, U>& e);

  // Injects e at position p
  void insert(size_t p, const pair<T, U>& e);
};
```

Contracts | The Reckoning

```
template <class T, class U>
class zip_vector {
public:
  // Returns the first element of each pair
  const vector<T>& first() const;
  // Returns the second element of each pair
  const vector<U>& second() const;

  // Returns the number of elements
  size_t size() const;

  // Returns true iff there are no elements
  bool empty() const;

  // Removes the last element
  void pop_back() noexcept;

  // Appends e
  void push_back(const pair<T, U>& e);

  // Injects e at position p
  void insert(size_t p, const pair<T, U>& e);
};
```

Contracts | The Reckoning

```
// A collection of T, U pairs whose first
// elements are stored in one vector and
// second elements in a second vector
template <class T, class U>
class zip_vector {
public:
  // Returns the first element of each pair
  const vector<T>& first() const;
  // Returns the second element of each pair
  const vector<U>& second() const;

  // Returns the number of elements
  size_t size() const;

  // Returns true iff there are no elements
  bool empty() const;

  // Removes the last element
  void pop_back() noexcept;

  // Appends e
  void push_back(const pair<T, U>& e);

  // Injects e at position p
  void insert(size_t p, const pair<T, U>& e);
};
```

Contracts | The Reckoning

```
template <class T, class U>
  requires copyable<T> && assignable<T> && copyable<U> && assignable<U>
class zip_vector {
public:
  const vector<T>& first() const
    post (r){ !testing || equal(*this, r,
        [](const auto& a, const auto& b){
          return &a.first == &b;
         }) };
  const vector<U>& second() const
    post (r){ !testing || equal(*this, r,
        [](const auto& a, const auto & b){
          return &second == &b;
        }) };

  invariant { size(first()) == size(second()) }

  size_t size() const
    post (r){ r == size(first()) && r == size(last()) };
  bool empty() const
    post (r){ r == (size() == 0) }

  void pop_back() noexcept
    pre { size() < 0 }
    post [old_size = size()] { size() == old_size - 1 }
    post [old = *this] { !testing || equal(begin(), end(), begin(old)) };

  void push_back(const pair<T, U>& e)
    post [old_size = size()] { size() == old_size + 1 }
    post { back() == e }
    post [old = *this] { !testing
        || equal(begin(old), end(old), begin()) };

  void insert(size_t p, const pair<T, U>& e)
    pre { p <= size() }
    post [old_size = size()] { size() == old_size + 1 }
    post { (*this)[p] == e }
    post [old = *this] { !testing
        || (equal(begin(), begin() + p, begin(old))
        && equal(begin() + p + 1, end(), begin(old) + p)) };

  ...
};
```

Contracts | The Reckoning

```
// A collection of T, U pairs whose first
// elements are stored in one vector and
// second elements in a second vector
template <class T, class U>
class zip_vector {
public:
  // Returns the first element of each pair
  const vector<T>& first() const;
  // Returns the second element of each pair
  const vector<U>& second() const;

  // Returns the number of elements
  size_t size() const;

  // Returns true iff there are no elements
  bool empty() const;

  // Removes the last element
  void pop_back() noexcept;

  // Appends e
  void push_back(const pair<T, U>& e);

  // Injects e at position p
  void insert(size_t p, const pair<T, U>& e);
};
```

Contracts | The Reckoning

```
template <class T, class U>
 requires copyable<T> && assignable<T> && copyable<U> && assignable<U>
class zip_vector {
public:
 const vector<T>& first() const
  post (r){ !testing || equal(*this, r,
    [](const auto& a, const auto& b){
     return &a.first == &b;
     }) };
 const vector<U>& second() const
  post (r){ !testing || equal(*this, r,
    [](const auto& a, const auto & b){
     return &second == &b;
    }) };

 invariant { size(first()) == size(second()) }

 size_t size() const
  post (r){ r == size(first()) && r == size(last()) };
 bool empty() const
  post (r){ r == (size() == 0) }
```

```
 void pop_back() noexcept
  pre { size() < 0 }
  post [old_size = size()] { size() == old_size - 1 }
  post [old = *this] { !testing || equal(begin(), end(), begin(old)) };

 void push_back(const pair<T, U>& e)
  post [old_size = size()] { size() == old_size + 1 }
  post { back() == e }
  post [old = *this] { !testing
    || equal(begin(old), end(old), begin()) };

 void insert(size_t p, const pair<T, U>& e)
  pre { p <= size() }
  post [old_size = size()] { size() == old_size + 1 }
  post { (*this)[p] == e }
  post [old = *this] { !testing
    || (equal(begin(), begin() + p, begin(old))
    && equal(begin() + p + 1, end(), begin(old) + p)) };

 ...
};
```

Contracts | The Reckoning

```
// A collection of T, U pairs whose first
// elements are stored in one vector and
// second elements in a second vector
template <class T, class U>
class zip_vector {
public:
  // Returns the first element of each pair
  const vector<T>& first() const;
  // Returns the second element of each pair
  const vector<U>& second() const;

  // Returns the number of elements
  size_t size() const;

  // Returns true iff there are no elements
  bool empty() const;

  // Removes the last element
  void pop_back() noexcept;

  // Appends e
  void push_back(const pair<T, U>& e);

  // Injects e at position p
  void insert(size_t p, const pair<T, U>& e);
};
```

Insights

Insights

You Can't Check Everything

Strict weak ordering

Pointer validity

The result of sort is a permutation of the input

A move-only result is equal to its prior value

You can't check all your checks

Human Language is More Expressive than Code

Can capture preconditions and postconditions in one phrase e.g. "Removes the last element."

Can describe un-checkable conditions.

Can describe things that would be expensive to check.

Can describe postconditions as effects rather than predicates.

Can describe all the preconditions and postconditions in one place
whether they are efficiently checkable or not.

Checking is super useful

Beginning - intermediate programmers

Fragile code bases

Unit test

Spoiler Alert: It's Documentation

"…a software system is viewed as a set of communicating components whose interaction is based on ***precisely defined specifications*** *of the mutual obligations — contracts."*

Bertrand Meyer

—Building bug-free O-O software: An Introduction to Design by Contract™

https://www.eiffel.com/values/design-by-contract/introduction/

Spoiler Alert: It's Documentation

“…a software system is viewed as a set of communicating components whose interaction is based on ***p*** ***defined specifications*** *of the mutual* *ts”*

Contract review before code review

—Building bug-free O-O software: An Introduction to

https://www.eiffel.com/values/design-by-contract/introduction/

Bertrand Meyer

C++ at Adobe!

developer.adobe.com/cpp

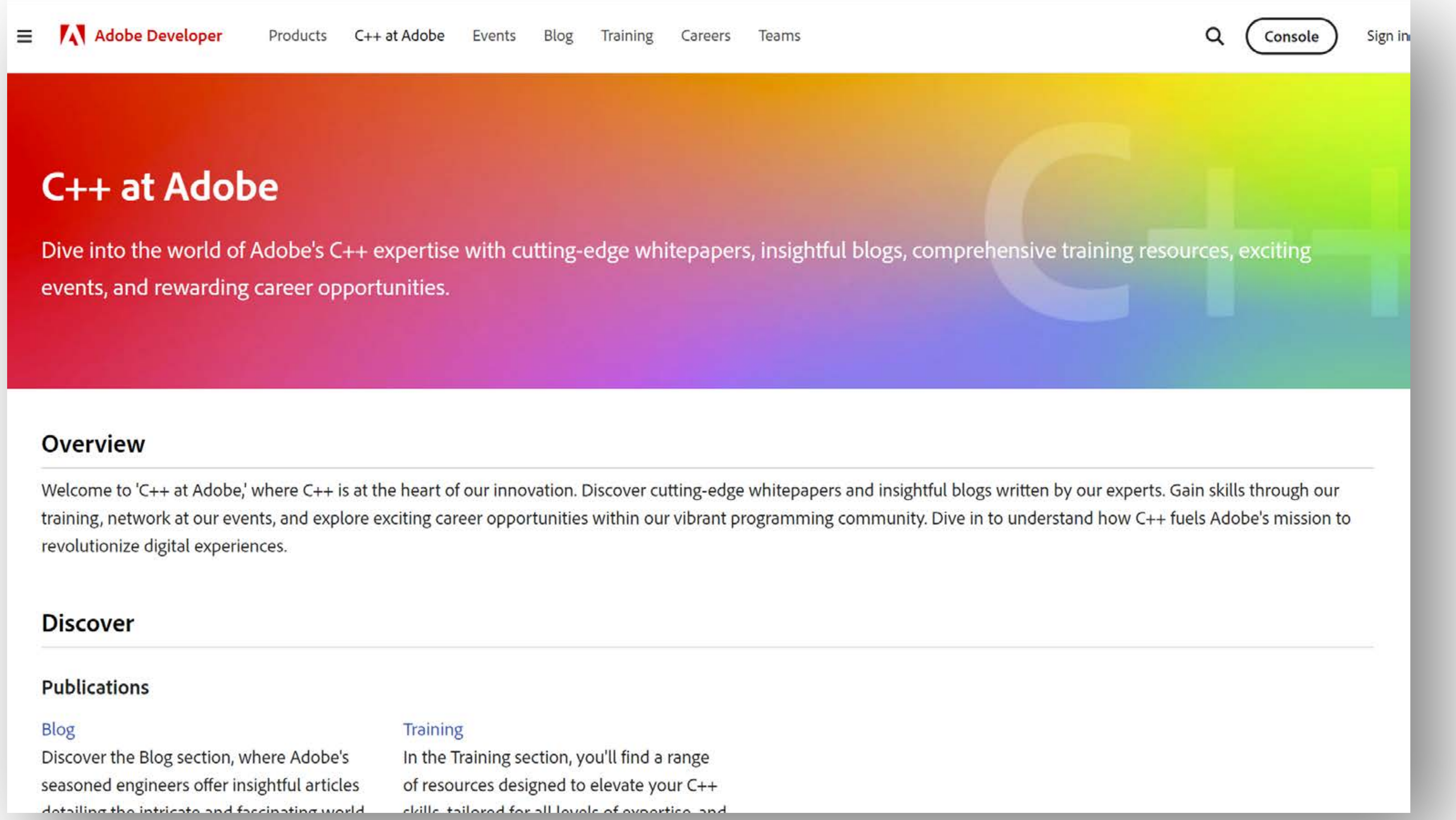
Careers

Events

Training Videos (STLab Better Code Series!)

Blog Posts

…



C++ at Adobe

Dive into the world of Adobe's C++ expertise with cutting-edge whitepapers, insightful blogs, comprehensive training resources, exciting events, and rewarding career opportunities.

Overview

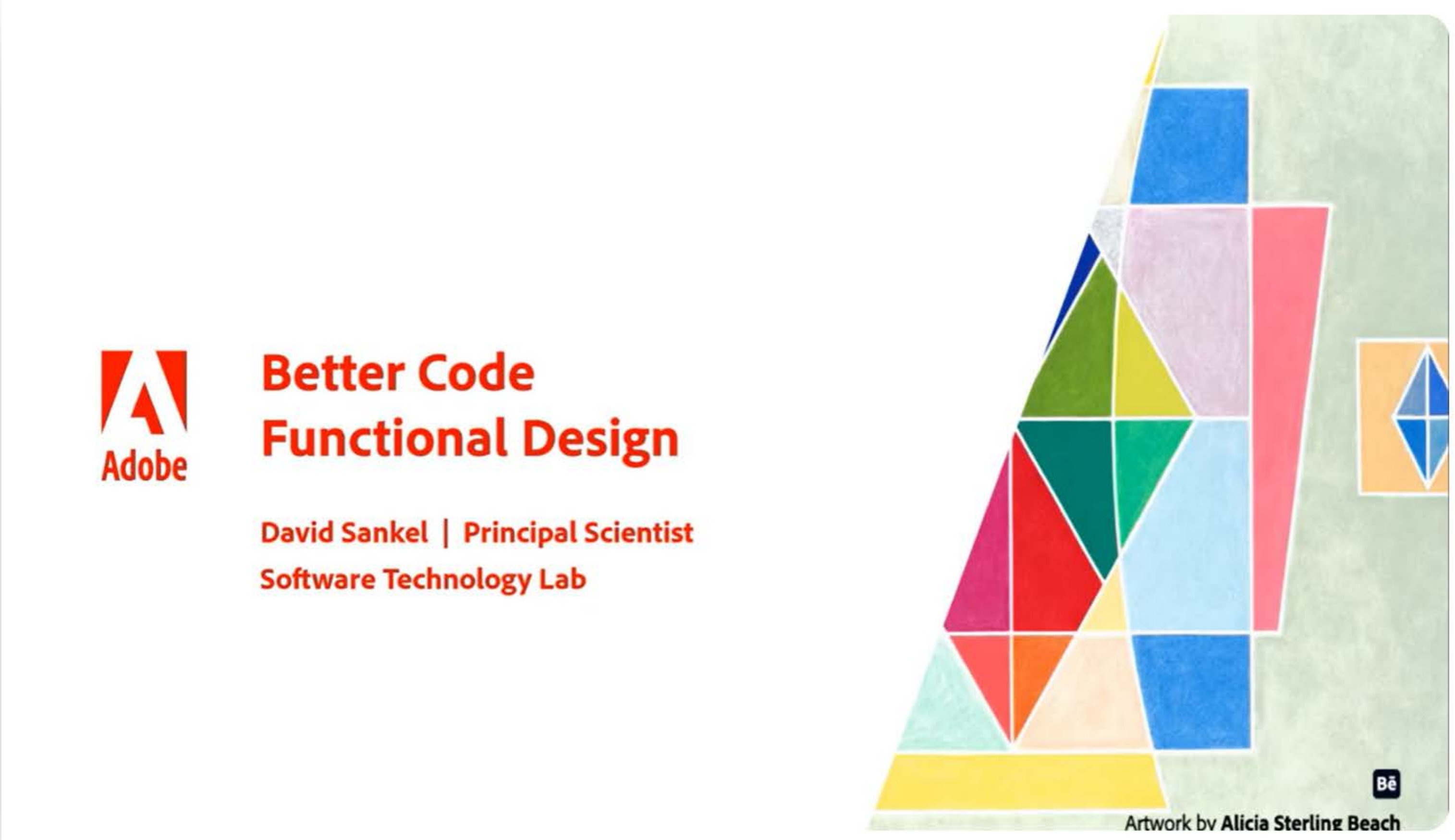
Welcome to 'C++ at Adobe,' where C++ is at the heart of our innovation. Discover cutting-edge whitepapers and insightful blogs written by our experts. Gain skills through our training, network at our events, and explore exciting career opportunities within our vibrant programming community. Dive in to understand how C++ fuels Adobe's mission to revolutionize digital experiences.

Discover

Publications

Blog
Discover the Blog section, where Adobe's seasoned engineers offer insightful articles detailing the intricate and fascinating world

Training
In the Training section, you'll find a range of resources designed to elevate your C++ skills, tailored for all levels of expertise, and

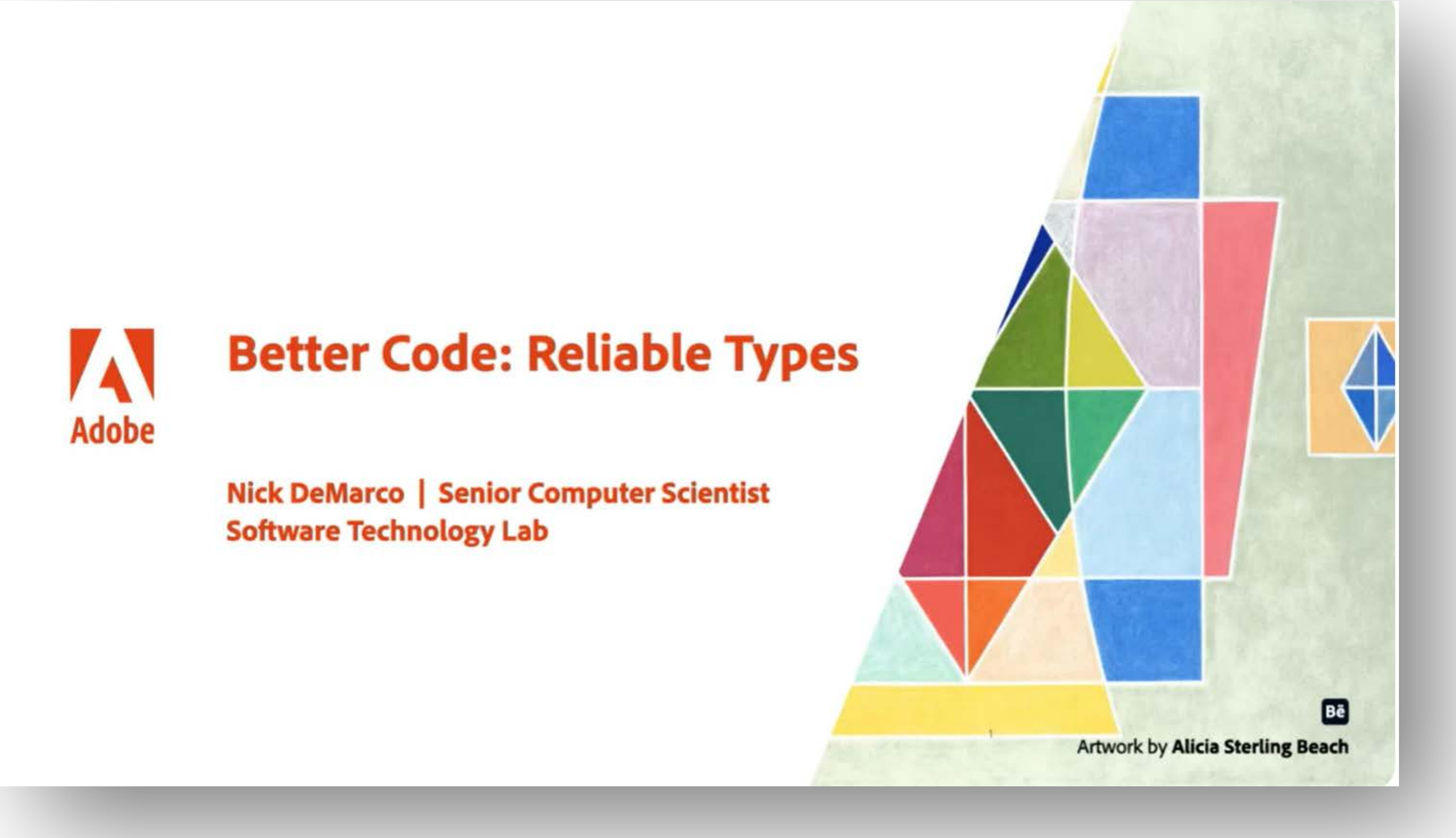


Adobe

Better Code
Functional Design

David Sankel | Principal Scientist
Software Technology Lab

Artwork by Alicia Sterling Beach



Adobe

Better Code: Reliable Types

Nick DeMarco | Senior Computer Scientist
Software Technology Lab

Artwork by Alicia Sterling Beach

Q & A

About the artist

Dan Zucco

London-based 3D art and motion director Dan Zucco creates repeating 2D patterns and brings them to life as 3D animated loops. Inspired by architecture, music, modern art, and generative design, he often starts in Adobe Illustrator and builds his animations using Adobe After Effects and Cinema 4D. Zucco's objective for this piece was to create a geometric design that felt like it could have an infinite number of arrangements.

Made with

Ai Adobe Illustrator Ae Adobe After Effects

Adobe

Artwork by Dan Zucco