Preconditions, Postconditions, Invariants: How They Help Write Robust Programs

Andrzej Krzemieński

C++ now

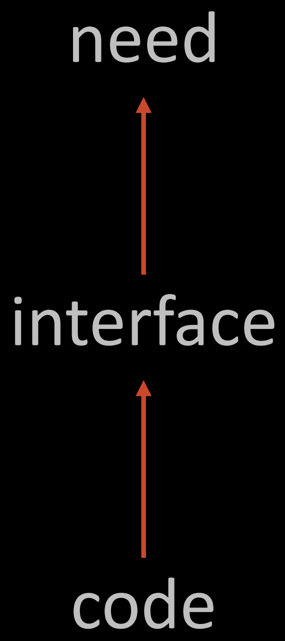
2021
MAY 2-7
Aspen, Colorado, USA

Agenda

- Function contract
- Strong and weak contracts
- Checking contract violation
- Constrained types for enforcing contracts
- Language support
- Bug source vs bug symptom

Function Contract

Function Contract



need

interface

code

Function Contract

The need: sort the collection of aircrafts by their weight.

```
std::vector<Aircraft*> aircrafts;
```

Function Contract

The need: sort the collection of aircrafts by their weight.

```
std::vector<Aircraft*> aircrafts;
```

Abstraction gap:

- We mean aircrafts
- We use memory addresses

Function Contract

The need: sort the collection of aircrafts by their weight.

```
std::vector<Aircraft*> aircrafts;
```

Abstraction gap:

- We mean aircrafts ← abstract notion
- We use memory addresses ← numeric values

Function Contract

Static typing:

```
std::vector<Aircraft*> aircrafts;
```

Dynamic typing:

```
std::vector<void*> aircrafts;
```

Function Contract

Help from the type system:

```
double Aircraft::weight() const;
```

Function Contract

Partial help from the type system:

```
double Aircraft::weight() const
  // weight in kilograms
;
```

Function Contract

Partial help from the type system:

```
double Aircraft::weight() const
  // weight in kilograms
;
```

Abstraction gap:

- We mean weight
- We type `double`

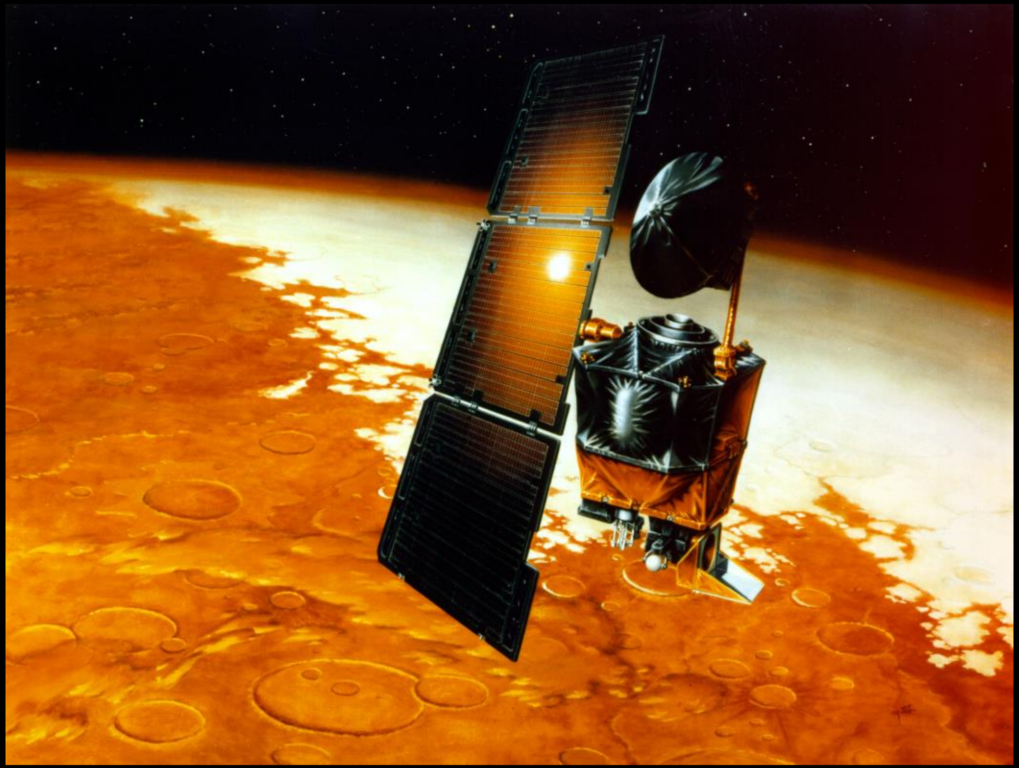
Function Contract

Partial help from the type system:

```
double Aircraft::weight() const
  // weight in kilograms
;
```

Abstraction gap:

- We mean weight
- We type `double`



Source: NASA

Function Contract

```
bool is_lighter(const Aircraft* a, const Aircraft* b);
```

Abstraction gap:

- We *mean* aircrafts
- We type `const class Aircraft*`

Function Contract

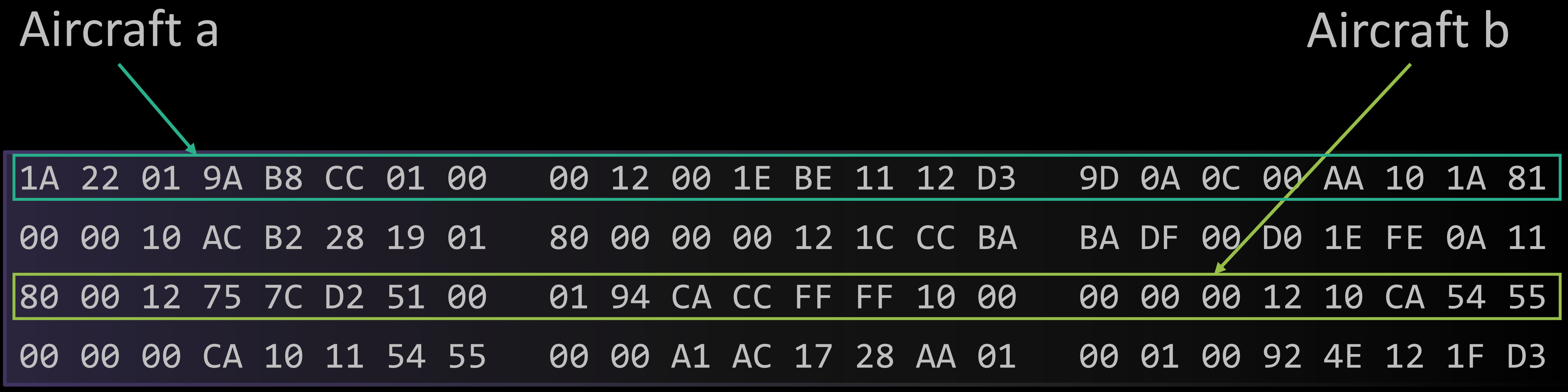
```
bool is_lighter(const Aircraft* a, const Aircraft* b)
  // expects: a and b point to objects of type Aircraft
;
```

Contract of pointer dereference:

- We turn an address into memory accesses
- Assumptions are made

Function Contract

```
bool is_lighter(const Aircraft* a, const Aircraft* b)
  // expects: a and b point to objects of type Aircraft
;
```



Aircraft a

Aircraft b

```
1A 22 01 9A B8 CC 01 00   00 12 00 1E BE 11 12 D3   9D 0A 0C 00 AA 10 1A 81
00 00 10 AC B2 28 19 01   80 00 00 00 12 1C CC BA   BA DF 00 D0 1E FE 0A 11
80 00 12 75 7C D2 51 00   01 94 CA CC FF FF 10 00   00 00 00 12 10 CA 54 55
00 00 00 CA 10 11 54 55   00 00 A1 AC 17 28 AA 01   00 01 00 92 4E 12 1F D3
```

Function Contract

```
bool is_lighter(const Aircraft* a, const Aircraft* b)
  // expects: a and b point to objects of type Aircraft
;
```

Expectation of pointer dereference:

- Address `a` is not null.
- An `Aircraft` object lives there.

Function Contract

```
bool is_lighter(const Aircraft* a, const Aircraft* b)
  // expects: a and b point to objects of type Aircraft
;
```

Expectation of pointer dereference:

- Address `a` is not null. ← Could check
- An `Aircraft` object lives there. ← Cannot check

Function Contract

Function Contract

The need: determine if an integral value is in a closed range.

```
int lower = config.get("LOWER_BOUND");
int upper = config.get("UPPER_BOUND");
```

Function Contract

```
bool is_in_range(int val, int lo, int hi)
  // checks if val is in closed range [lo, hi]
;
```

Function Contract

```
bool is_in_range(int val, int lo, int hi)
  // checks if val is in closed range [lo, hi]
;
```

Abstraction gap:

- We mean a “closed range”
- We use two objects of type `int`.

Function Contract

```
bool is_in_range(int val, int lo, int hi)
  // checks if val is in closed range [lo, hi]
;
```

```
bool f(int a, int b, int c);
```

Function Contract

```
bool is_in_range(int val, int lo, int hi)
  // checks if val is in closed range [lo, hi]
;
```

```
is_in_range(3, 2, 1); // [2, 1] -- not a closed range
```

- Values fit into the type system
- Values don’t fit into the abstraction

Function Contract

Function Contract – all you need to know to call the function correctly

- How the inputs and outputs are interpretted
- Limits (domain)
- What must happen before or after the call
- Effects

Function Contract

Some parts of function contract can be expressed in the language:

- The number and types of the arguments.
- Immutability of variables.

Function Contract

Other parts of function contract cannot:

- Exception safety guarantees.
- Disallowed values.

Function Contract

Two definitions of *Interface*:

- Same as function contract
- Parts of function contract expressible in the language

Function Contract

```
char* setlocale(int category, const char* locale);
```

- `locale` may be null: optional string

```
size_t strlen(const char* s);
```

- `s` must not be null: required string

Strong and Weak Contracts

Strong and Weak Contracts

```
bool is_in_range(int val, int lo, int hi)
  // checks if val is in closed range [lo, hi]
  // expects: lo <= hi
;
```

C++ expression

```
is_in_range(3, 2, 1);
```

Strong and Weak Contracts

```
bool is_in_range(int val, int lo, int hi)
  // checks if val is in closed range [lo, hi]
  // expects: lo <= hi
;
```

C++ expression

```
is_in_range(3, 2, 1);
```

What if `lo > hi`?

Strong and Weak Contracts

```
bool is_in_range(int val, int lo, int hi)
  // if lo <= hi
  //   checks if val is in closed range [lo, hi]
  // otherwise returns false
;
```

Weaken the contract?

Strong and Weak Contracts

Drawbacks of weak contracts:

- Weaker abstractions
- Increased complexity
- No use
- False sense of safety
- Missed opportunity to detect bugs

Strong and Weak Contracts

Weaker abstraction

```
bool is_in_range(int val, int lo, int hi)
  // if lo <= hi
  //   checks if val is in closed range [lo, hi]
  // otherwise returns false
;
```

- No “range” anymore: just `int`s.
- Thinking low-level encourages bugs.

Strong and Weak Contracts

Increased complexity

```
bool is_in_range(int val, int lo, int hi)
  // if lo <= hi
  //   checks if val is in closed range [lo, hi]
  // otherwise returns false
;
```

- Requires additional `if`-statements in code.
- More unit tests

Strong and Weak Contracts

No use

```
bool is_lighter(const Aircraft* a, const Aircraft* b)
  // if a or b is null, returns false
  // otherwise returns whether *a has lower weight than *b
;
```

- Second responsibility: checking for null
- Why would I pass the null poiner here?

Strong and Weak Contracts

False sense of safety

```
bool is_lighter(const Aircraft* a, const Aircraft* b)
  // if a or b is null, returns false
  // otherwise returns whether *a has lower weight than *b
;
```

- What if `a` is `0xffffffff`? What about `0x12345678`?
- All bad inputs cannot be detected.

Strong and Weak Contracts

Missed opportunity to detect bugs

```
bool validate(int val) {
  int lower = config.get("LOWER_BOUND");
  int upper = config.get("UPPER_BOUND");
  return is_in_range(lower, upper, val);
}
```

```
bool is_in_range(int val, int lo, int hi);
```

Strong and Weak Contracts

Missed opportunity to detect bugs

```
bool validate(int val) {
  int lower = config.get("LOWER_BOUND");
  int upper = config.get("UPPER_BOUND");
  return is_in_range(lower, upper, val);
}
```

- You cannot detect bugs if *every* value is “good”.

Strong and Weak Contracts

Strong contracts:

- Simple abstractions
- Discourages bugs
- Can help detect bugs

Strong and Weak Contracts

Don’t weaken contracts.

Look for language features that enforce strong contracts.

Contract Violation

Contract Violation

```
bool is_lighter(const Aircraft* a, const Aircraft* b)
  // expects: a and b point to objects of type Aircraft
;
```

Contract Violation

```
bool is_lighter(const Aircraft* a, const Aircraft* b)
{
  return a->weight() < b->weight();
}
```

Contract Violation

```
bool is_lighter(const Aircraft* a, const Aircraft* b)
{
  return a->weight() < b->weight();
}
```

Pointer dereference:

- The need: access the object under a given address

Contract Violation

```
bool is_lighter(const Aircraft* a, const Aircraft* b)
{
  return a->weight() < b->weight();
}
```

Language contract for pointer dereference:

- A valid object lives at address `a`.
- Address `a` is not null. ← can be runtime-checked

Contract Violation

```
bool is_lighter(const Aircraft* a, const Aircraft* b)
{ if (!a || !b) std::abort();
  return a->weight() < b->weight();
}
```

← injected by UB-sanitizer

Language contract for pointer dereference:

- A valid object lives at address `a`.
- Address `a` is not null.

Contract Violation

```
inline bool is_lighter(const Aircraft* a, const Aircraft* b)
{
  return a->weight() < b->weight();
}
```

```
Aircraft *p = nullptr, *q = getAircraft(), *r = getAircraft();
return is_lighter(p, r);  // static analyzer warning
```

If a tool can see *both* function definition and the usage, it can warn about a bug.

Contract Violation

```
const char * p = nullptr;
const char * q = "config.cfg";

std::string fname(p);
```

Contract Violation

```
const char * p = nullptr;
const char * q = "config.cfg";
if (!p) std::abort();
std::string fname(p);
```

← UB-sanitizer could inject

Contract for Standard Library functions is known to compilers.

Contract Violation

```
basic_string(CharT* __str) : /* */
{
  assert(__str != nullptr);
}
```

← contract enforcement in STD implementation

Contract for Standard Library functions is known to STD vendors.

Contract Violation

```
bool is_in_range(int val, int lo, int hi)
  // checks if val is in closed range [lo, hi]
;
```

Compiler does not know about your contract.

Contract Violation

```
bool is_in_range(int val, int lo, int hi)
  // checks if val is in closed range [lo, hi]
;
```

```
bool validate(int val) {
  int lower = config.get("LOWER_BOUND");
  int upper = config.get("UPPER_BOUND");
  return is_in_range(lower, upper, val);
}
```

Contract Violation

```
bool is_in_range(int val, int lo, int hi)
  // checks if val is in closed range [lo, hi]
  // expects: lo <= hi
;
```

C++ expression

Contract Violation

Contract Check – it is derived from function contract.

- It can determine that a program has a bug.

Precondition

Precondition

```
bool is_in_range(int val, int lo, int hi) {

  return lo <= val && val <= hi;
}
```

Current implementation produces value for any inputs.

- No UB to be affraid of inside.
- You might forget about the bugs outside.

Precondition

```
bool exceeds_weight(const Aircraft* a, double limit_kg) {
  // risk of UB
  return a->weight() > limit_kg;
}
```

The urge to avoid UB at *any* cost:

- Assumes that bugs are more desireable than UB.
- Assumes that we can avoid UB while leaving bugs alone.

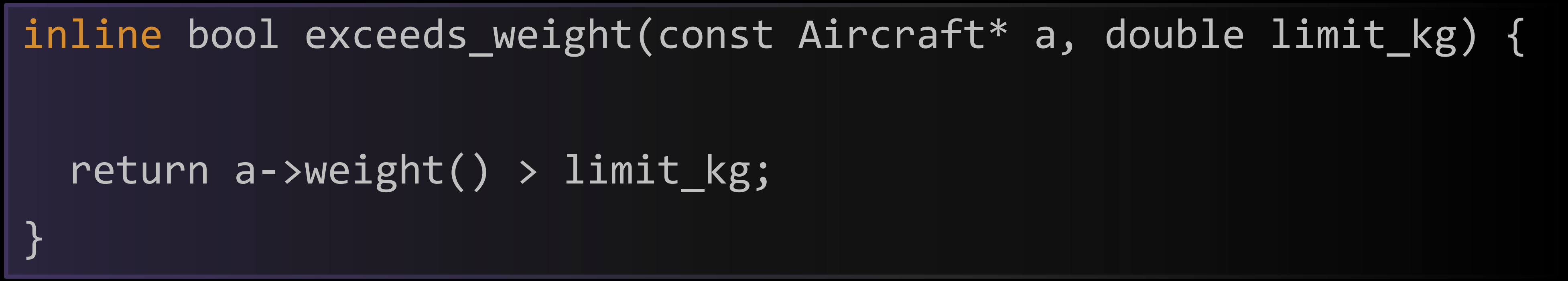
Precondition

```
bool exceeds_weight(const Aircraft* a, double limit_kg) {
  if (!a) REACT();
  return a->weight() > limit_kg;
}
```

The urge to avoid UB at any cost:

- Assumes that bugs are more desireable than UB.
- Assumes that we can avoid UB while leaving bugs alone.

Precondition



```
inline bool exceeds_weight(const Aircraft* a, double limit_kg) {


  return a->weight() > limit_kg;
}
```

Precondition

```cpp
inline bool exceeds_weight(const Aircraft* a, double limit_kg) {

  return a->weight() > limit_kg;
}
```

```cpp
return is_lighter(nullptr, 12'000);
```

← Static analyzer warning:

null pointer dereference

Precondition

```
bool exceeds_weight(const Aircraft* a, double limit_kg) {
  if (!a) return true; // "safe default"
  return a->weight() > limit_kg;
}
```

This assumes that there is no difference between

- Exceeding a weight limit
- Unable to compute the result

Precondition

```
bool exceeds_weight(const Aircraft* a, double limit_kg) {
  if (!a) return true; // "safe default"
  return a->weight() > limit_kg;
}
```

```
bool enough_fuel = exceeds_weight(&a, required_fuel);
if (!enough_fuel)
  report_danger();
```

Precondition

```
bool exceeds_weight(const Aircraft* a, double limit_kg) {
  if (!a) throw Bug{}; // skip me and my caller
  return a->weight() > limit_kg;
}
```

Precondition

```
bool exceeds_weight(const Aircraft* a, double limit_kg) {
  if (!a) throw Bug{}; // skip me and my caller
  return a->weight() > limit_kg;
}
```

```
exceeds_weight(nullptr, 120'000);
```

← No help from static analyzer

Bug detection postponed to runtime.

Precondition

```
bool is_in_range(int val, int lo, int hi) {
  if (lo > hi) throw Bug{}; // skip me and my caller
  return lo <= val && val <= hi;
}
```

Precondition

```
bool is_in_range(int val, int lo, int hi) {
  if (lo > hi) throw Bug{}; // skip me and my caller
  return lo <= val && val <= hi;
}
```

```
bool validate(int val) {
  int lower = config.get("LOWER_BOUND");
  int upper = config.get("UPPER_BOUND");
  return is_in_range(lower, upper, val);
}
```

Precondition

```
bool is_in_range(int val, int lo, int hi) {
  if (lo > hi) throw Bug{}; // skip me and my caller
  return lo <= val && val <= hi;
}
```

```
bool validate(int val) {
  int lower = 0
  int upper = std::max(100, config.get("LIMIT"));
  return is_in_range(val, lower, upper);
}
```

Precondition

Upon UB your *code* no longer corresponds with the *binary*.

- You cannot draw any conclusions from the code.

```
bool validate(int val) {
  int lower = 0
  int upper = std::max(100, config.get("LIMIT"));
  return is_in_range(val, lower, upper);
}
```

Precondition

Precondition Violation can be caused by prior Undefined Behavior:

- Dangling pointers
- Bad usage of `memset`
- Data races

Undefined Behavior

Undefined Behavior

```
bool is_small(int * p) {
  return *p < 10;
}
```

```
is_small(nullptr);
```

What does it mean in the language?

Undefined Behavior

```
bool is_small(int * p) {
  return *p < 10;
}
```

```
is_small(nullptr);
```

Contract:

- `*p` – access memory *under* address p.
- `nullptr` – *no* memory address.

Undefined Behavior

```
bool is_small(int * p) {
  return *p < 10;
}
```

```
is_small(nullptr);
```

Contract violation:

- No behavior guaranteed by the contract
- Bug

Undefined Behavior

```
bool is_small(int * p) {
  return *p < 10;
}
```

```
is_small(nullptr);
```

Why no behavior?

- No use case
- It would sanction bugs

Undefined Behavior

```
bool is_small(int * p) {
  return *p < 10;
}
```

```
is_small(nullptr);
```

Mechanical outcomes:

- Random number
- Program shutdown

Precondition

```
bool is_in_range(int val, int lo, int hi) {
  if (lo > hi) std::abort();
  return lo <= val && val <= hi;
}
```

Precondition

```
bool is_in_range(int val, int lo, int hi) {
  if (lo > hi) std::abort();
  return lo <= val && val <= hi;
}
```

“works” for *any* values

Trading a returned value for a crash?

Precondition

```
bool is_in_range(int val, int lo, int hi) {
  if (lo > hi) std::abort();
  return lo <= val && val <= hi;
}
```

```
bool in_danger(Aircraft const& ac) {
  return is_in_range(danger_zone.lower(),
                     danger_zone.upper(),
                     ac.stress());
}
```

Precondition

```
bool is_in_range(int val, int lo, int hi) {
  if (lo > hi) std::abort();

  return lo <= val && val <= hi;
}
```

- The bug is outside the function
- The implementation may change

Precondition

```
bool is_in_range(int val, int lo, int hi) {
  if (lo > hi) std::abort();
  _stats[hi - lo] += 1; // for logging
  return lo <= val && val <= hi;
}
```

- The bug is outside the function
- The implementation may change

Precondition

```
bool is_in_range(int val, int lo, int hi) {
  if (lo > hi) std::abort();

  return lo <= val && val <= hi;
}
```

Predictable behavior:

- No further damage (caused by the caller)
- Core dump for post-mortem analysis
- Remains stable as implementation changes

Precondition

Is crashing a good option?

- Weight & balance calculator: user can go to manual
- Word processor: better to waste 1hr of work that 1 day of work
- Drone: better to restart than do random actions
- Financial server: better go down than make bad decisions
- Assisting troops: better go down than give false sense of security

Precondition

```
inline bool is_in_range(int val, int lo, int hi) {
  if (lo > hi) { *((int*)0) = 0; } // explicit UB
  return lo <= val && val <= hi;
}
```

Give hint to static analyzer

Precondition

```
inline bool is_in_range(int val, int lo, int hi) {
  if (lo > hi) { *((int*)0) = 0; } // explicit UB
  return lo <= val && val <= hi;
}
```

```
return is_in_range(0, 100, 50);
```

Static analyzer warning:

null pointer dereference

Precondition

```
inline bool is_in_range(int val, int lo, int hi) {
  if (lo > hi) { *((int*)0) = 0; } // explicit UB
  return lo <= val && val <= hi;
}
```

Give hint to static analyzer

Is UB more dangerous than a bug?

Precondition

```
inline bool is_in_range(int val, int lo, int hi) {
  if (lo > hi) TRAP();
  return lo <= val && val <= hi;
}
```

```
#if defined STATIC_ANALYSIS
#  define TRAP() { *((int*)0) = 0; }
#else
#  define TRAP() std::abort()
#elif
```

Precondition

```
bool is_in_range(int val, int lo, int hi) {
  assert(lo <= hi);
  return lo <= val && val <= hi;
}
```

assert():

- Declares the bug criterion
- Its effect in code depends on configuration

Precondition

```
bool is_in_range(int val, int lo, int hi) {
  Expects(lo <= hi); // GSL.assert
  return lo <= val && val <= hi;
}
```

Precondition

```
bool is_in_range(int val, int lo, int hi) {
  assert(lo <= hi);
  return lo <= val && val <= hi;
}
```

```
bool check(int lo, int hi, int val) {
  return is_in_range(lo, hi, val); // 2 <= 3 (check passes)
}

return check(1, 2, 3);
```

Precondition

Contract Check – it is derived from function contract.

- It can determine that a program has a bug.
- It cannot determine if a program hasn’t got a bug.

Precondition

```
bool is_lighter(const Aircraft* a, const Aircraft* b)
  // expects: a != nullptr && b != nullptr
;
```

- We don’t mean that `a != nullptr && b != nullptr` is good
- We mean that `a == nullptr || b == nullptr` is *bad*

Precondition

```
bool is_in_range(int val, Range r);
```

```
bool validate(int val) {
  int lower = config.get("LOWER_BOUND");
  int upper = config.get("UPPER_BOUND");
  return is_in_range(val, Range{lower, upper});
}
```

Precondition

```
bool is_in_range(int val, Range r);
```

```
bool validate(int val) {
  int lower = config.get("LOWER_BOUND");
  int upper = config.get("UPPER_BOUND");
  return is_in_range(Range{lower, upper}, val); // type-system error
}
```

Precondition

```
bool is_in_range(int val, Range r)
  // expects: r.lo() <= r.hi()
;
```

What did we gain?

Invariant

Invariant

```
class Range {
  int _lo, _hi; // private

public:
  Range(int l, int h);
  int lo() const { return _lo; }
  int hi() const { return _hi; }
  // invariant: lo() <= hi()
};
```

Invariant

```
class Range {
  int _lo, _hi; // private

public:
  Range(int l, int h);
  int lo() const { return _lo; }
  int hi() const { return _hi; }
  bool invariant() const { return lo() <= hi(); }
};
```

Invariant

```
class Range {
  int _lo, _hi; // private

public:
  Range(int l, int h);
  int lo() const { assert(invariant()); return _lo; }
  int hi() const { assert(invariant()); return _hi; }
  bool invariant() const { return lo() <= hi(); }
};
```

Invariant

```
bool is_in_range(int val, Range r)
  // expects: r.invariant()
;
```

Technically true, but redundant.

Invariant on function parameters is an implied precondition.

Invariant

```
bool is_in_range(int val, Range r)
  // expects: r.invariant()
;
```

```
return is_in_range(val, Range{hi, lo}); // still a bug
```

Invariant

```
class Range {
  int _lo, _hi; // private

public:
  Range(int l, int h); // precondition: l <= h
    : _lo(l), _hi(h) {}
  int lo() const { return _lo; }
  int hi() const { return _hi; }
  bool invariant() const { return lo() <= hi(); }
};
```

Invariant

```
class Range {
  int _lo, _hi; // private

public:
  Range(int l, int h); // precondition: l <= h
    : _lo((assert(l <= h), l)), _hi(h) {}
  int lo() const { return _lo; }
  int hi() const { return _hi; }
  bool invariant() const { return lo() <= hi(); }
};
```

Invariant

An invariant is a *conditional* guarantee.

It depends on the preconditions of member functions.

Invariant

```
return is_in_range(lo, hi, val); // no longer a bug
```

Invariant

```
auto [a, b] = bounds(); // a > b
is_in_range(x, Range{a, b});
```

Invariant

Before

```
auto [a, b] = bounds(); // a > b
is_in_range(x, Range{a, b});
is_in_range(y, Range{a, b});
is_in_range(z, Range{a, b});
```

After

```
Range r = bounds();
is_in_range(x, r);
is_in_range(y, r);
is_in_range(z, r);
```

The scope of range-related bugs is narrowed:

- Potential bugs *only* when you create the Range,
- No potential bugs when you pass the Range around.

Invariant

Not every precondition can be turned into a type.

```
T& vector<T>::operator[](size_t i)
  // expects: i < this->size()
;
```

- The range of allowed `i` may change over time.

Inexpressible Contract

Inexpressible Contract

```
class Kilograms {
  double _value; // weight difference can be negative

public:
  explicit Kilograms(double val) : _value{val} {}
  static Kilograms from_double(double val) { return Kilograms{val}; }
;
```

Inexpressible Contract

```
Kilograms Aircraft::weight() const
```

Inexpressible Contract

Your own class:

- Reflects the invariant
- Reflects how the values are interpretted

Inexpressible Contract

```
size_t length(const char* str)
  // expects: str != nullptr
  // expects: "str is null terminated"
;
```

← inexpressible

Inexpressible Contract

```
size_t length(C_string str);
```

Inexpressible Contract

```
class C_string {
  const char * _str;
public:
  // invariant: _str != nullptr && "str is nul terminated"
};
```

Inexpressible Contract

```
C_string str = get_name();
return length(str);
```

Inexpressible Contract

```
C_string str = get_name();
return length(str);
```

But what if some API uses `const char *`?

Inexpressible Contract

```
class C_string {
  const char * _str;

public:
  explicit(false) C_string(const char* s)
    // expects: s != nullptr && "s is nul terminated"
    ;

  // invariant: _str != nullptr && "str is nul terminated"
};
```

Inexpressible Contract

Runtime checking is impossible.

Static analysis might work.

Inexpressible Contract

```
inline bool is_null_terminated(const char *)
  [[symbolic]]  // <-- for static analyzers
{
  return true;  // <-- for run-time checking
};
```

During runtime checks: avoid false positives.

During static analysis: treat as a *symbol*.

Inexpressible Contract

```
size_t length(const char* str)
  // expects: str != nullptr
  // expects: is_null_terminated(str)
;
```

← expressible as *symbol*

Inexpressible Contract

```
const char* name = get_name();
return length(name); // potential contract violation
```

Inexpressible Contract

```
const char* name = get_name();
return length(name); // proven correct!
```

```
const char * get_name()
  // postcondition(r): r != nullptr && is_null_trminated(r)
;
```

name of return value.

Postcondition

Postconditions are useful for matching against preconditions.

Postcondition

```
int better(int a, int b)
  // precondition: a >= 0
  // precondition: b >= 0
  // postcondition(r): r >= 0
;
```

A postcondition is expected to hold if:

- Function does not report failure (e.g., by throwing exception)
- Function’s preconditions are satisfied

Language Support

Language Support

```
int better(int a, int b)
  // @precondition: a >= 0
  // @precondition: b >= 0
  // @postcondition(r): r >= 0
;
```

Comments:

- Only humans can use
- Or individual tools

Language Support

```
int better(int a, int b) {
  Expects(a >= 0);
  Expects(b >= 0);

  return a > b ? a : b;
}
```

Assertions in function bodies:

- Help prevent damage
- Help testing
- Don’t prevent bugs

Language Support

```
int better(int a, int b) {
  Expects(a >= 0);
  Expects(b >= 0);
  // <-- to early for a postcond
  return a > b ? a : b;
} // <-- to late for a postcond
```

Postconditions as assertions:

Language Support

```
int better(int a, int b) {
  Expects(a >= 0);
  Expects(b >= 0);

  int r = a > b ? a : b;
  Ensures(r >= 0);
  return r;
}
```

Postconditions as assertions:

- Make code longer
- No RVO

Language Support

```
int better(int a, int b) {
  Expects(a >= 0);
  Expects(b >= 0);

  if (a > b) {
    Ensures(a >= 0);
    return a;
  }

  Ensures(b >= 0);
  return b;
}
```

Postconditions as assertions:

- Make code longer
- No RVO
- Are repetitive

Language Support

```
int better(int a, int b)
  [[pre: a >= 0]]
  [[pre: b >= 0]]
  [[post r: r >= 0]]
;
```

Contract *anntations*:

- Appear in function declarations
- Compiler checks expressions

Language Support

Standardized contract *anntations*:

- Communicate (parts of) our contracts to different tools
- Provide same bug-detection experience as with language contracts

Language Support

```
int better(int a, int b)
  [[pre: a >= 0]]
  [[pre: b >= 0]]
  [[post r: r >= 0]]
;
```

- Static analyzer can detect bugs without seeing function bodies.

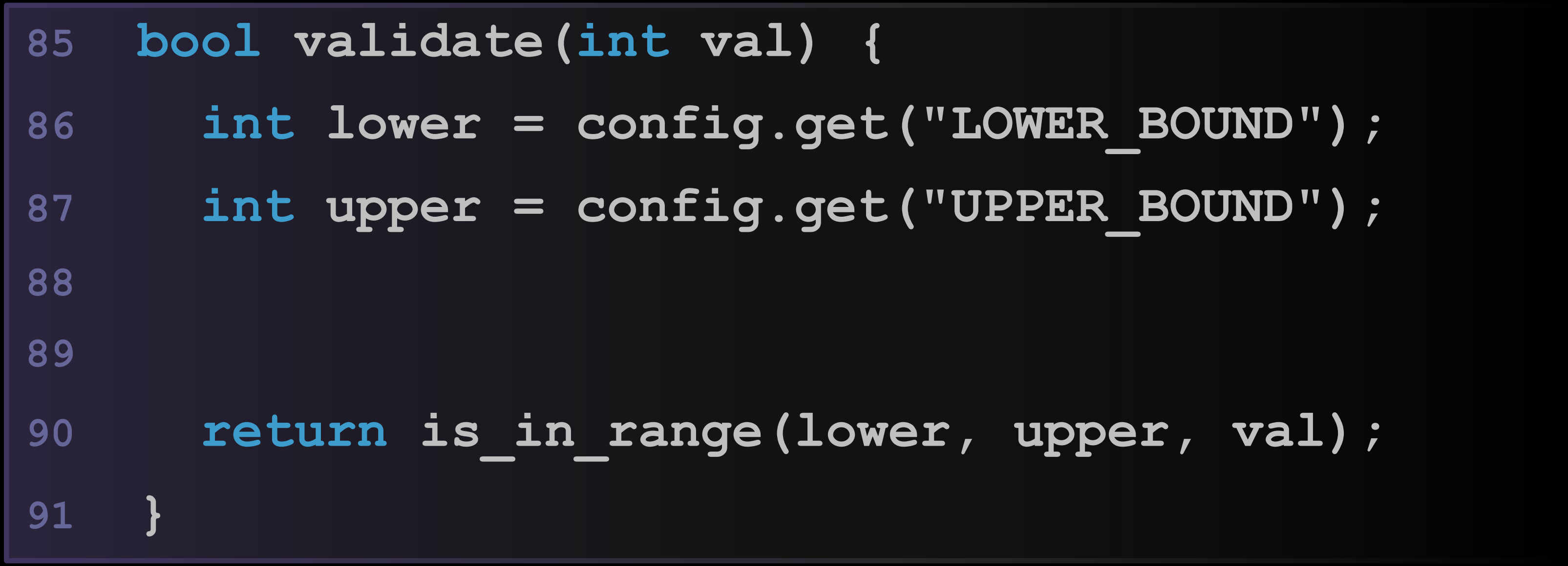
Language Support

```
class Range
{
public:
  Range(int l, int h);
  int lo() const;
  int hi() const;
  [[assert: lo <= hi]];
};
```

Contract *anntations*:

- Appear in class definitions

Language Support

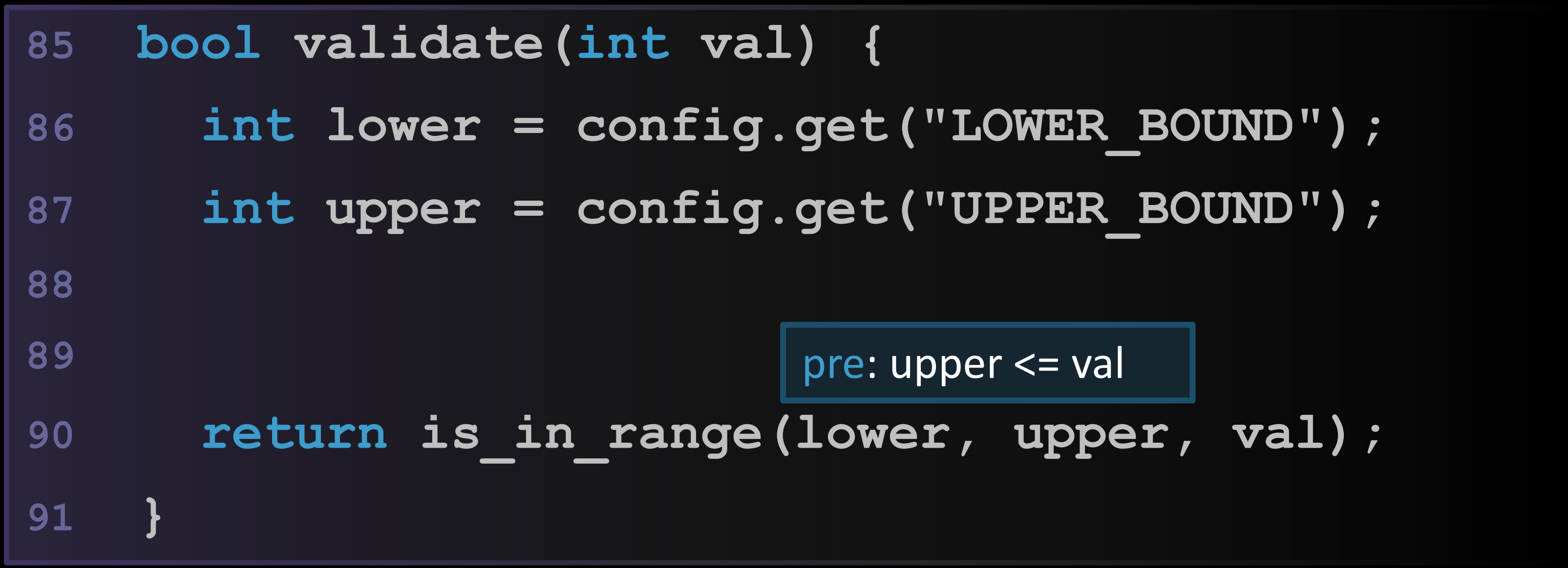


```
bool validate(int val) {
  int lower = config.get("LOWER_BOUND");
  int upper = config.get("UPPER_BOUND");


  return is_in_range(lower, upper, val);
}
```

Standardized notation

- IDE can give a hint

Language Support



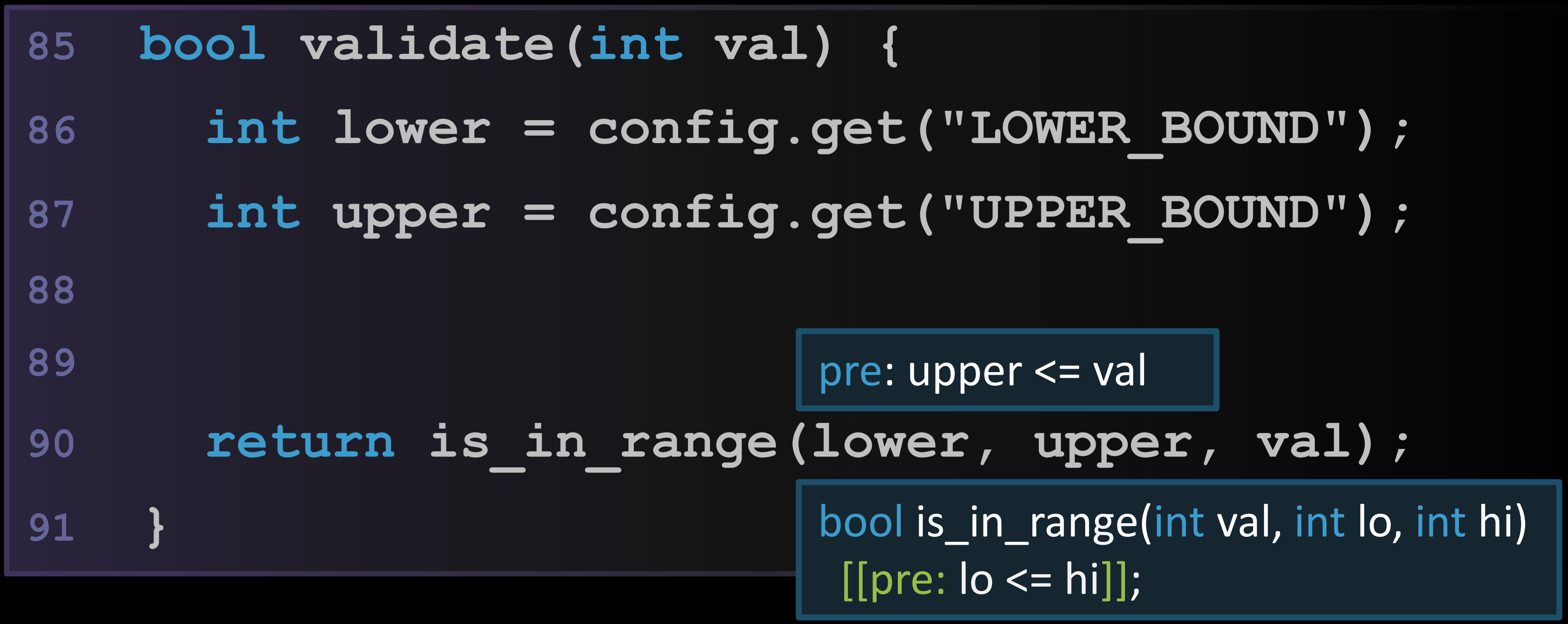
```
bool validate(int val) {
  int lower = config.get("LOWER_BOUND");
  int upper = config.get("UPPER_BOUND");


  return is_in_range(lower, upper, val);
}
```

pre: upper <= val

Standardized notation

- IDE can give a hint

Language Support



```
bool validate(int val) {
  int lower = config.get("LOWER_BOUND");
  int upper = config.get("UPPER_BOUND");


  return is_in_range(lower, upper, val);
}
```

pre: upper <= val

```
bool is_in_range(int val, int lo, int hi)
  [[pre: lo <= hi]];
```

Standardized notation

- IDE can give a hint

Language Support

```
bool validate(int val) {
  int lower = config.get("LOWER_BOUND");
  int upper = config.get("UPPER_BOUND");

  return is_in_range(lower, upper, val);
}
```

Language Support

```
bool validate(int val) {
  int lower = config.get("LOWER_BOUND");
  int upper = config.get("UPPER_BOUND");
  if (upper > val) std::abort();
  return is_in_range(lower, upper, val);
}
```

injected based on precondition

Standardized notation

- Compiler can inject runtime checks

Language Support

Language Support

```
class window : public widget {
  [[deprecated("use simpler decl")]] [[nodiscard]] const widget & clone
  (const point & x, const point& y)
  const & noexcept override [[pre: are_rectangle(x, y)]]
  [[pre: ordered([](const auto& a) -> auto && { return a.h; }, x, y) ]]
  [[post x: *this == x]];
};
```

- How many class members?
- What are their names?

Fixing Bugs

Fixing Bugs

```
Aircraft *x = nullptr, *y = nullptr;
try {
  x = get_aircraft("x");
  y = get_aircraft("y");
}
catch(...) {
  // TODO: handle it
}

if (!x)
  return is_lighter(x, y);
```

```
Aircraft* get_aircraft(string_view name)
  [[post a: a != nullptr]];
```

Fixing Bugs

```
Aircraft *x = nullptr, *y = nullptr;
try {
  x = get_aircraft("x");
  y = get_aircraft("y");
}
catch(...) {
  // TODO: handle it
}

if (!x)
  return is_lighter(x, y); // <-- analyzer warning: y might be null
```

Fixing Bugs

Analyzer detects only a *symptom* of a bug.

- Analyzer does not know where the bug is.
- Programmer must look for it.

Fixing Bugs

```
Aircraft *x = nullptr, *y = nullptr;
try {
  x = get_aircraft("x");
  y = get_aircraft("y");
}
catch(...) {
  // TODO: handle it
}

if (!x)
  return is_lighter(x, y); // <-- analyzer warning: y might be null
```

Fixing Bugs

```
Aircraft *x = nullptr, *y = nullptr;
try {
  x = get_aircraft("x");
  y = get_aircraft("y");
}
catch(...) {
  // TODO: handle it
}

if (!x || !y)
  return is_lighter(x, y); // warning silenced; bug still present!
```

Fixing Bugs

```
Aircraft *x = nullptr, *y = nullptr;
try {
  x = get_aircraft("x");
  y = get_aircraft("y");
}
catch(...) { // <-- real bug: premature try-catch
  // TODO: handle it
}

if (!x)
  return is_lighter(x, y);
```

Fixing Bugs

```
Aircraft *x = nullptr, *y = nullptr;
try {
  x = get_aircraft("x");
  y = get_aircraft("y");
  return is_lighter(x, y); // skipped upon a throw
}
catch(...) {
  // TODO: handle it
}
```

Fixing Bugs

```
Aircraft *x = nullptr, *y = nullptr;

  x = get_aircraft("x");
  y = get_aircraft("y");
  return is_lighter(x, y);
```

Fixing Bugs

```
Aircraft *x = nullptr, *y = nullptr;
x = get_aircraft("x");
y = get_aircraft("y");
return is_lighter(x, y);
```

Fixing Bugs

```
Aircraft * x = get_aircraft("x");
Aircraft * y = get_aircraft("y");
return is_lighter(x, y);
```

Fixing Bugs

```
Aircraft * x = get_aircraft("x");
Aircraft * y = get_aircraft("y");
return is_lighter(x, y);
```

- Bug is fixed
- Program is cleaner

Fixing Bugs

Warning → start investigation.

- Time consuming
- Avoid false positives

Summary

Summary

Function contract

- Inexpressible, human-to-human
- Try to express parts of it in the language
  - Classes for providing interpretation of values
  - *Contract annotations* for expressing disallowed values

Summary

Contract annotations – not only about runtime checks

- Provide same tool experience as the language contract
- Static analysis
- IDE hints
- Human understanding

Contact

akrzemi1@gmail.com

akrzemi1.wordpress.com