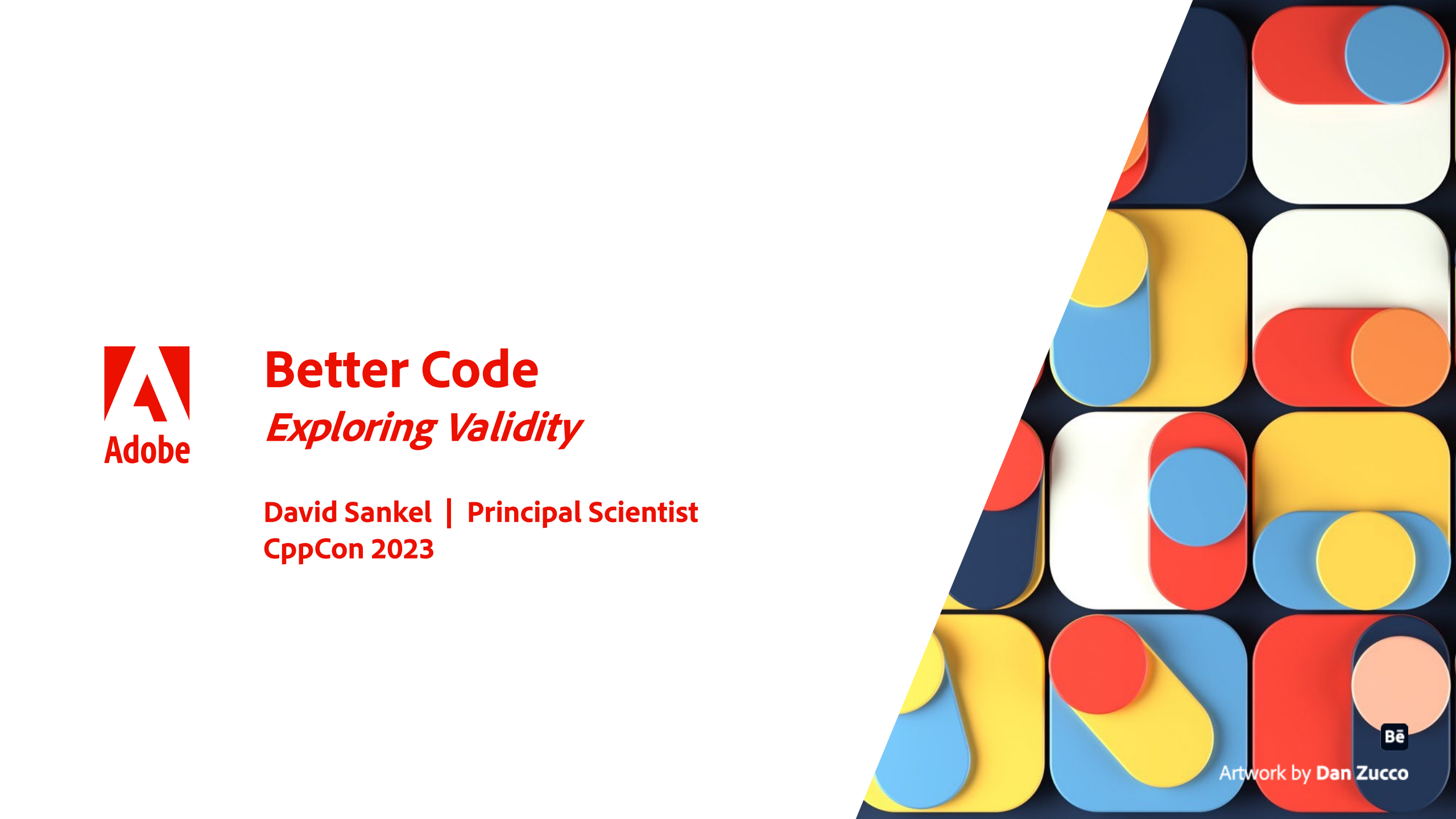
Better Code:
Exploring Validity

DAVID SANKEL

Cppcon
The C++ Conference

20
23

October 01 - 06


Adobe
Better Code
Exploring Validity
David Sankel | Principal Scientist
CppCon 2023
Artwork by Dan Zucco

Adobe's Software Technology Lab

Sean Parent

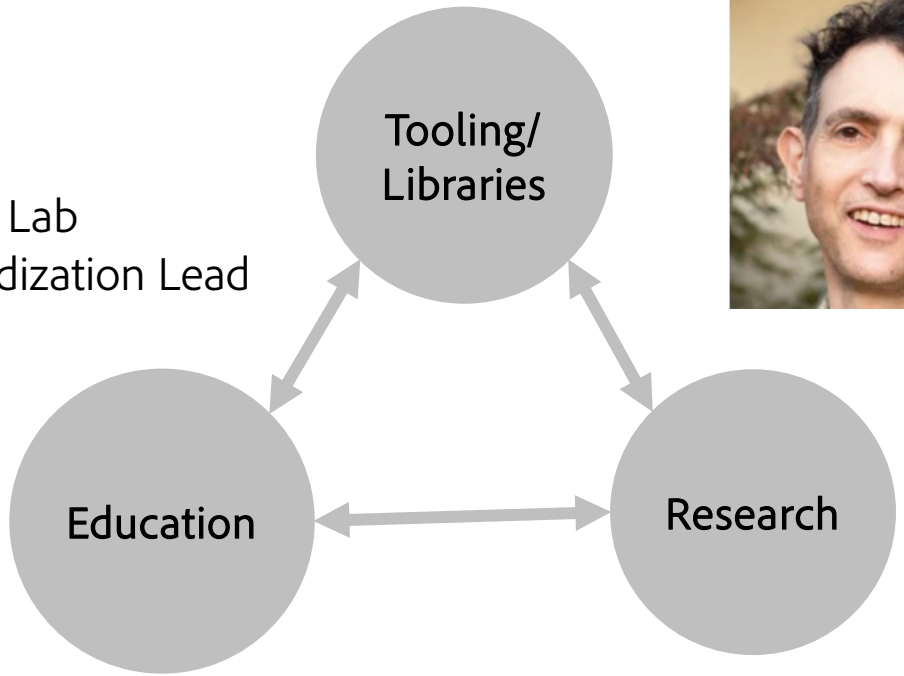
Senior Principal Scientist
Manager, Software Technology Lab
Adobe Veteran

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

Words

Type-and-resource safety

CS@CU

- Every object is accessed according to the type with which it was defined (type safety)
- Every object is properly constructed and destroyed (resource safety)
- Every pointer either points to a valid object or is the **nullptr** (memory safety)
- Every reference through a pointer is not through the **nullptr** (often a run-time check)
- Every access through a subscripted pointer is in-range (often a run-time check)

- That
  - Implies range checking and elimination of dangling pointers ("memory safety")
  - Is just what C++ requires
  - Is what most programmers have tried to ensure since the dawn of time
- The rules are more deduced than invented

Enforcement rules are mutually dependent.
Don't judge individual rules in isolation

ed according to the type with whic

ly constructed and destroyed (reso

oints to a valid object or is the **nu**

gh a pointer is not through the **nu**

a subscripted pointer is in-range (

Unsuitable

Suitable

Incorrect

Correct

Undesired

Desired

Incomplete

Complete

Validity

Two categories

One attracts and the other repels

Validity

Two categories

One attracts and the other repels

Validity is with respect to a beholder

Validity

Two categories

One attracts and the other repels

Validity is with respect to a beholder

```
// - Precondition: x >= 0
double sqrt(double x);
```

- Negative numbers are invalid arguments to sqrt
- A declaration of argument validity is called a precondition

DEFINITION: An argument list $(o_1, o_2, \ldots, o_i)$ is **valid** with respect to a function f if the function call $f(o_1, o_2, \ldots, o_i)$ does not violate any of f's preconditions.

Argument lists not meeting that criteria are said to be **invalid** with respect to f.

What's an invalid pointer?

Invalid pointers

```
int * i = 0;
{
  int j = 3;
  i = &j;
}
```

- At the end of this snippet, we say “i is an invalid pointer”
- What is the beholder?

Invalid pointers

```
int * i = 0;
{
  int j = 3;
  i = &j;
}
```

Invalid pointers are invalid with respect to dereference

Key Observation: so-called "invalid pointers" are valid with respect to other operations.

Best practices on argument validity

- Document preconditions
- When discussing object validity, clarify the associated function
- When checking for validity is possible, name the query after the associated function

Instead of *is_valid* consider	*is_dereferencable*	for a smart pointer
	is_decodable	for a UTF-8 string
	is_readable	for an input stream

ed according to the type with whic

ly constructed and destroyed (reso

oints to a valid object or is the **nu**

gh a pointer is not through the **nu**

a subscripted pointer is in-range (

ed according to the type with whic

ly constructed and destroyed (reso

oints to a valid object or is the **nu**

gh a pointer is not through the **nu**

a subscripted pointer is in-range (

DEFINITION: An **object** is an instance of a type.


State?

010010100111010100101011010110111110101010101001010010101010010011

101001011101101001010101001110101001010101010100100101001010011101

011010101010010011100000100101101110101001010010010000101001001010

100101001110101001010110101101111101010101010010100101010100100111

010010111011010010101010011101010010101010101001001010010100111010

110101010100100111000001001011011101010010100100100001010010010100

100101001110101001010110101101111101010101010010100101010100100111

010010111011010010101010011101010010101010101001001010010100111010

110101010100100111000001001011011101010010100100100001010010010100

100101001110101001010110101101111101010101010010100101010100100111

010010111011010010101010011101010010101010101001001010010100111010

110101010100100111000001001011011101010010100100100001010010010100

100101001110101001010110101101111101010101010010100101010100100111

010010111011010010101010011101010010101010101001001010010100111010

110101010100100111000001001011011101010010100100100001010010010100

100101001110101001010110101101111101010101010010100101010100100111

010010111011010010101010011101010010101010101001001010010100111010

110101010100100111000001001011011101010010100100100001010010010100


0100101001110101001010110101101111101010101001010010101010010011
1010010111011010010101010011101010010101010101001001010010100111101
0110101010100100111000001001011011101010010100100100001010010010100
1001010011101010010101101011011111010101010010100100101010010011101
0100101110110100101010100111010100101010101001001010010100111010
1101010101001001110000010010110111010100101001001000010100100101000
1001010011101010010101101
0110111111010101010100101001
0101010010011101001010110101101111010101010010100100101010010011101
0100101110110100101010100111010100101010101001001010010100111010
1101010101001001110000010010110111010100101001001000010100100101000
1001010011101010010101101011011111010101010010100100101010010011101
0100101110110100101010100111010100101010101001001010010100111010
1101010101001001110000010010110111010100101001001000010100100101000
1001010011101010010101101011011111010101010010100100101010010011101
0100101110110100101010100111010100101010101001001010010100111010
1101010101001001110000010010110111010100101001001000010100100101000
1001010011101010010101101011011111010101010010100100101010010011101
0100101110110100101010100111010100101010101001001010010100111010
1101010101001001110000010010110111010100101001001000010100100101000

```
struct P {
  int32_t x;
  int32_t y;
};
```

An object *o* of type *P*'s state consists of the bit configuration of *o.x* and *o.y*

```
struct P {
  P() {
    x = new int32_t();
    try { y = new int32_t(); }
    catch(...) { delete x; throw; }
  }
  ~P() { delete x; delete y; }
private:
  int32_t * x;
  int32_t * y;
};
```

```
struct P {
  P() {
    x = new int32_t();
    try { y = new int32_t(); }
    catch(...) { delete x; throw; }
  }
  ~P() { delete x; delete y; }
private:
  int32_t * x;
  int32_t * y;
};
```

```
struct P {
  P() {
    x = new int32_t();
    try { y = new int32_t(); }
    catch(...) { delete x; throw; }
  }
  ~P() { delete x; delete y; }
private:
  int32_t * x;
  int32_t * y;
};
```


0100101001110101001010110101101111101010101001010010101010010011
1010010111011010010101010011101010010101010101001001010010100111101
0110101010100100111000001001011011101010010100100100001010010010100
1001010011101010010101101011011111010101010010100100101010010011101
0100101110110100101010100111010100101010101001001010010100111010
1101010101001001110000010010110111010100101001001000010100100101000
1001010011101010010101101
0110111111010101010100101001
0101010010011101001010110101101111010101010010100100101010010011101
0100101110110100101010100111010100101010101001001010010100111010
1101010101001001110000010010110111010100101001001000010100100101000
1001010011101010010101101011011111010101010010100100101010010011101
0100101110110100101010100111010100101010101001001010010100111010
1101010101001001110000010010110111010100101001001000010100100101000
1001010011101010010101101011011111010101010010100100101010010011101
0100101110110100101010100111010100101010101001001010010100111010
1101010101001001110000010010110111010100101001001000010100100101000
1001010011101010010101101011011111010101010010100100101010010011101
0100101110110100101010100111010100101010101001001010010100111010
1101010101001001110000010010110111010100101001001000010100100101000

Objects and State

DEFINITION: An object's **state** is the bit configuration of memory that is under that object's ownership.

```
struct P {
  P() {
    x = new int32_t();
    try { y = new int32_t(); }
    catch(...) { delete x; throw; }
  }
  ~P() { delete x; delete y; }
private:
  int32_t * x;
  int32_t * y;
};
```

An object *o* of type P's state consists of:

- The bit configuration of *o.x*
- The bit configuration of *o.y*
- The bit configuration of **o.x*
- The bit configuration of **o.y*

```
struct P {
  P() {
    x = new int32_t();
    try { y = new int32_t(); }
    catch(...) { delete x; throw; }
  }
  ~P() { delete x; delete y; }
private:
  int32_t * x;
  int32_t * y;
};
```


What is **this*'s state here?

```
struct P {
  P() {
    x = new int32_t();
    try { y = new int32_t(); }
    catch(...) { delete x; throw; }
  }
  ~P() { delete x; delete y; }
private:
  int32_t * x;
  int32_t * y;
};
```


What is **this*'s state here?

- The bit configuration of *x*
- The bit configuration of *y*
- The bit configuration of **x*

An object's **state** is the bit configuration of memory that is under that object's ownership.


State?

String with small buffer optimization (SBO)

```
class string {
  char data_[32];
  char * begin_;
  char * end_;
};
```

String with small buffer optimization (SBO)

```
class string {
  char data_[32];
  char * begin_;
  char * end_;

  ~string() {
    if( begin_ != &data_[0] )  delete[] begin_;
  }
};
```

Hash map using global seed

```
inline const unsigned global_seed = []{
    std::random_device rd;
    std::uniform_int_distribution<unsigned> dist(0, UINT32_MAX);
    return dist(rd);
}();

template<Hashable Key, typename Value>
class secure_hash_map{
  public:
    void insert(Key key, Value value) {
        std::size_t hash = global_seed ^ std::hash<Value>{}(key);
        //...
    }
};
```

Hash map using global seed

```
inline const unsigned global_seed = []{
    std::random_device rd;
    std::uniform_int_distribution<unsigned> dist(0, UINT32_MAX);
    return dist(rd);
}();
```

```
template<Hashable Key, typename Value>
class secure_hash_map{
  public:
    void insert(Key key, Value value) {
        std::size_t hash = global_seed ^ std::hash<Value>{}(key);
        //...
    }
};
```

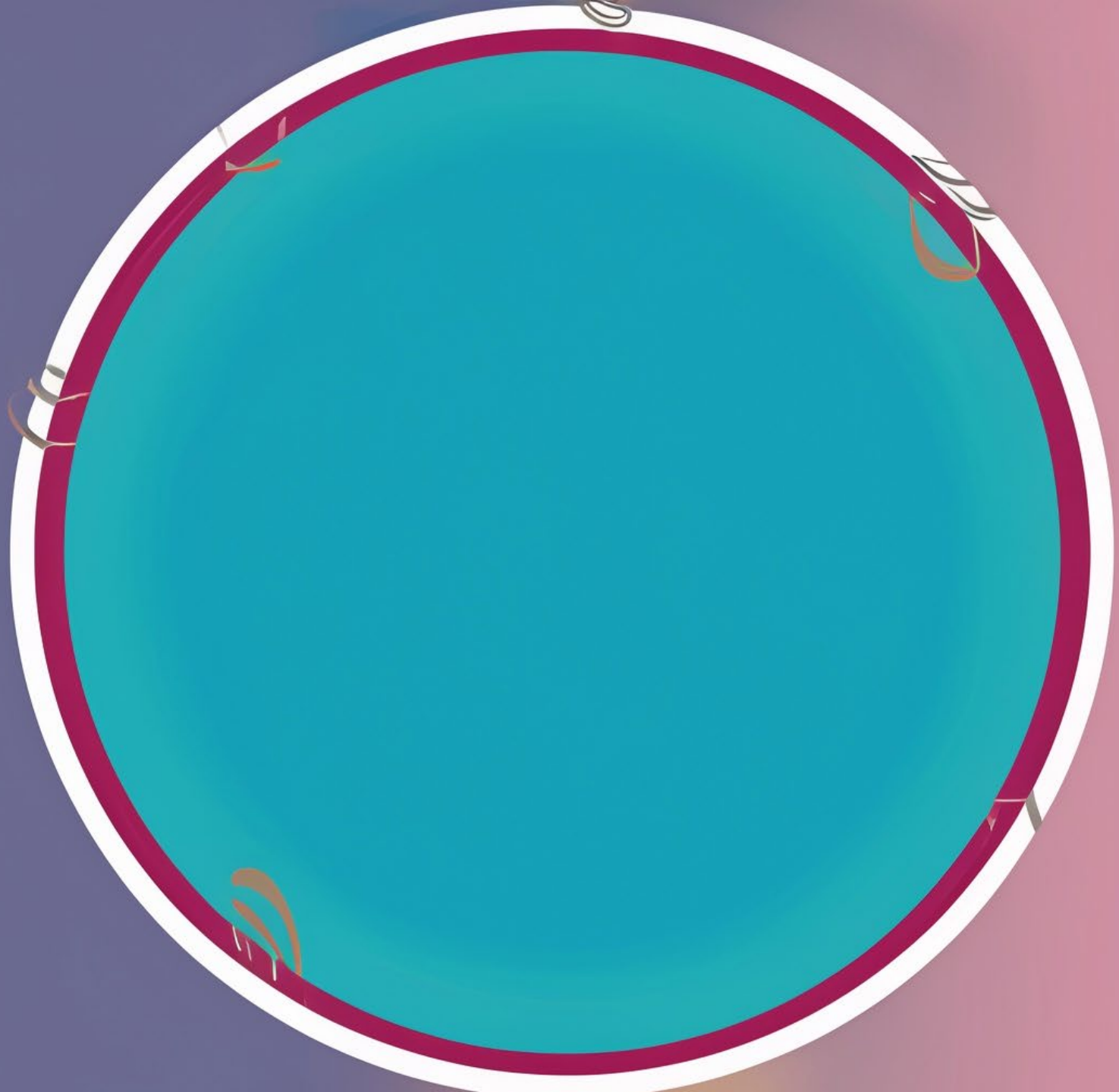
Hash map using global seed

```
inline const unsigned global_seed = []{
    std::random_device rd;
    std::uniform_int_distribution<unsigned> dist(0, UINT32_MAX);
    return dist(rd);
}();

template<Hashable Key, typename Value>
class secure_hash_map{
  public:
    void insert(Key key, Value value) {
        std::size_t hash = global_seed ^ std::hash<Value>{}(key);
        //...
    }
};
```

The “stuff” of an object

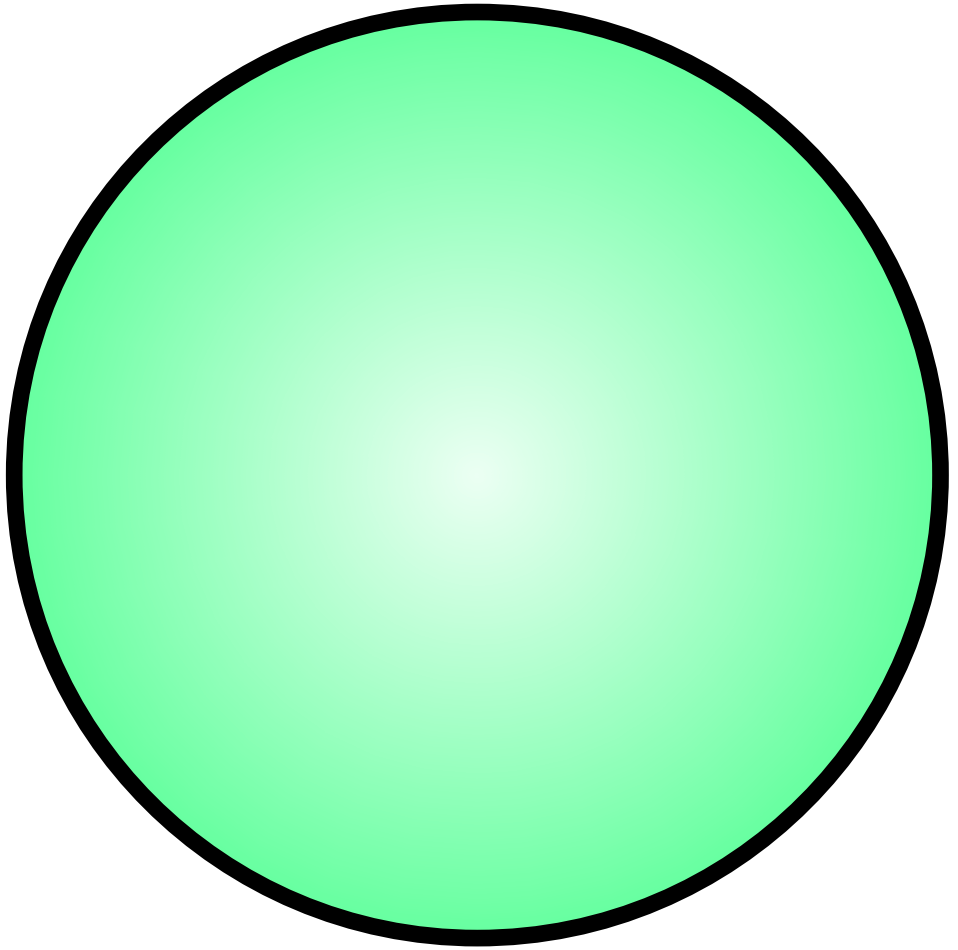
- Includes state
- May include its address or relations thereof
- May include bit configurations not owned by the object

DEFINITION: An object's **substance** is the information on a computer necessary to determine an object's behavior.



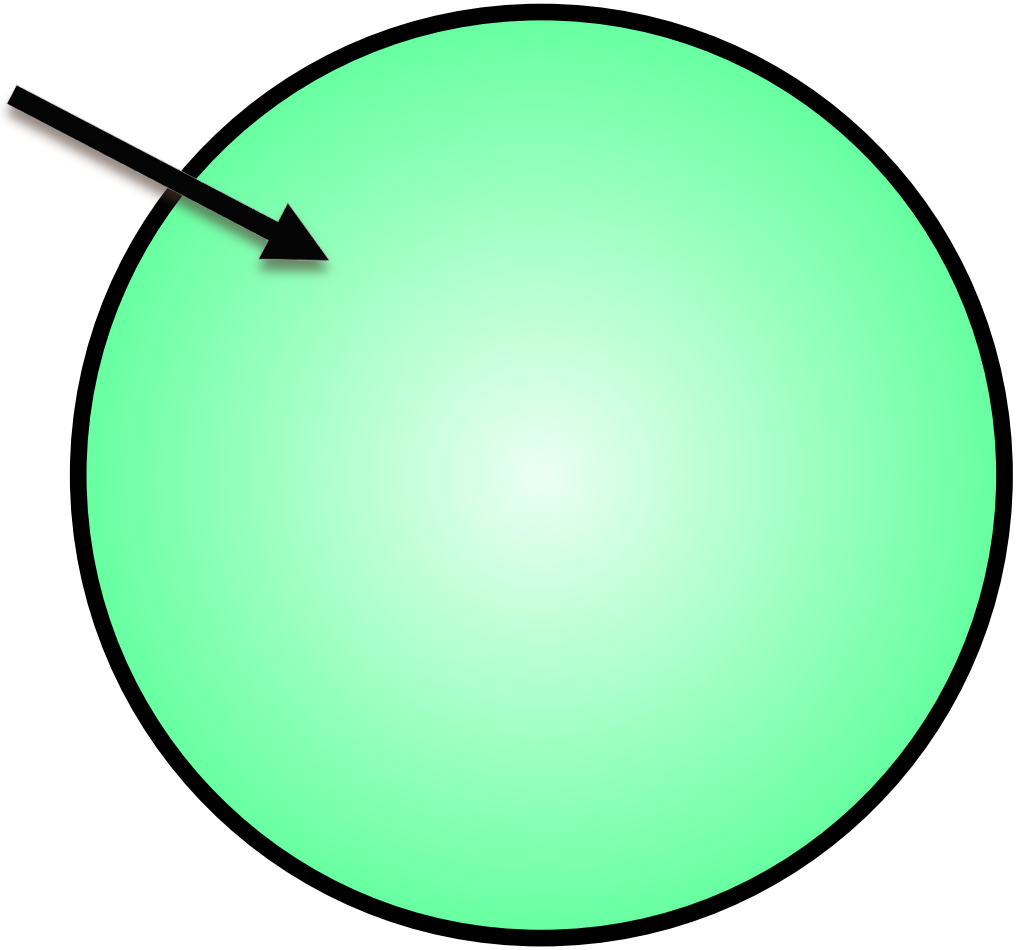
Substance can take on many forms

```
struct P {
  int32_t x;
  int32_t y;
};
```



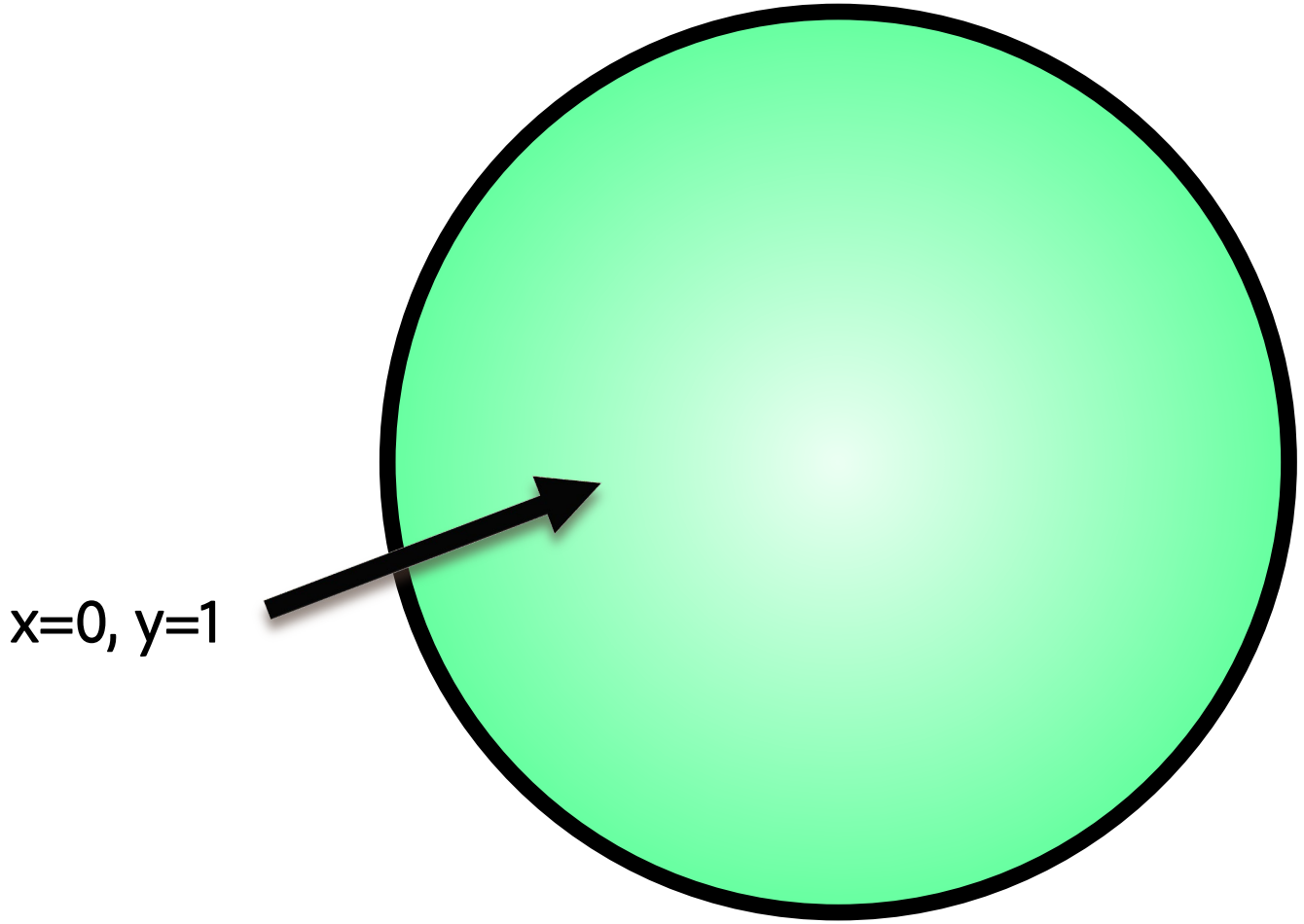
Substance can take on many forms

```
struct P {
  int32_t x;
  int32_t y;
};
```



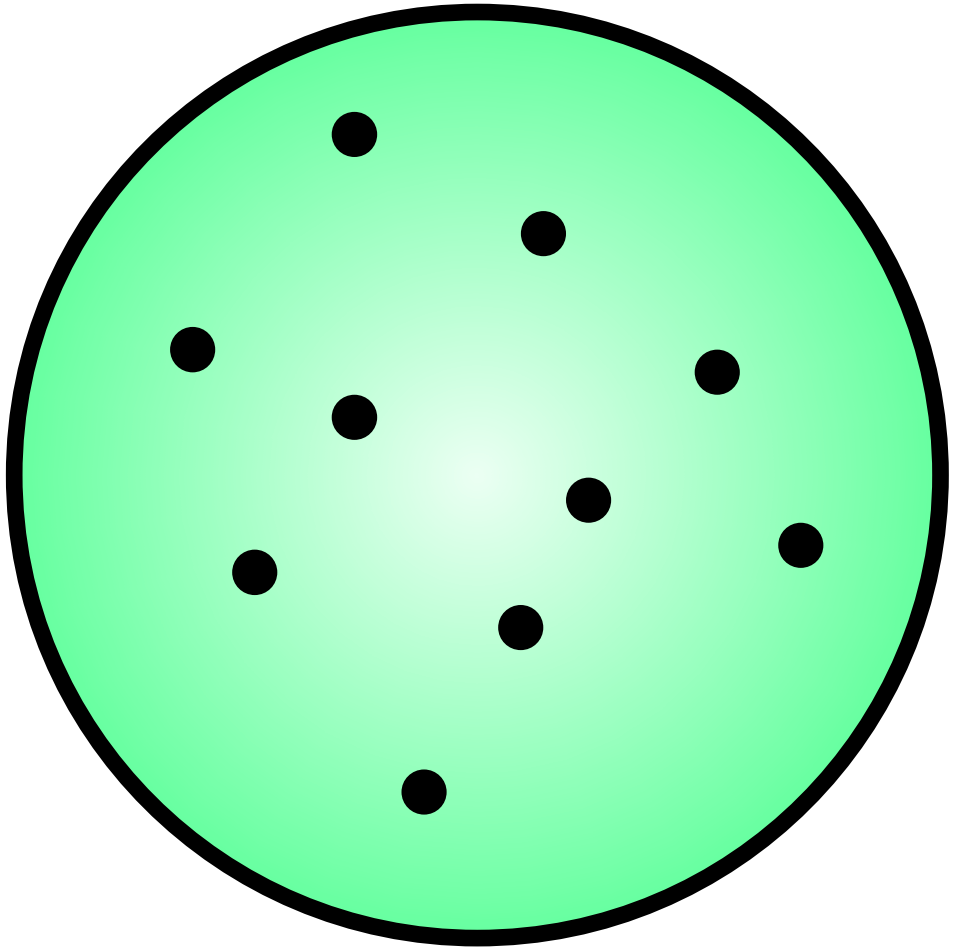
Substance can take on many forms

```
struct P {
  int32_t x;
  int32_t y;
};
```



Substance can take on many forms

```
struct P {
  int32_t x;
  int32_t y;
};
```



ed according to the type with whic

ly constructed and destroyed (reso

oints to a valid object or is the **nu**

gh a pointer is not through the **nu**

a subscripted pointer is in-range (

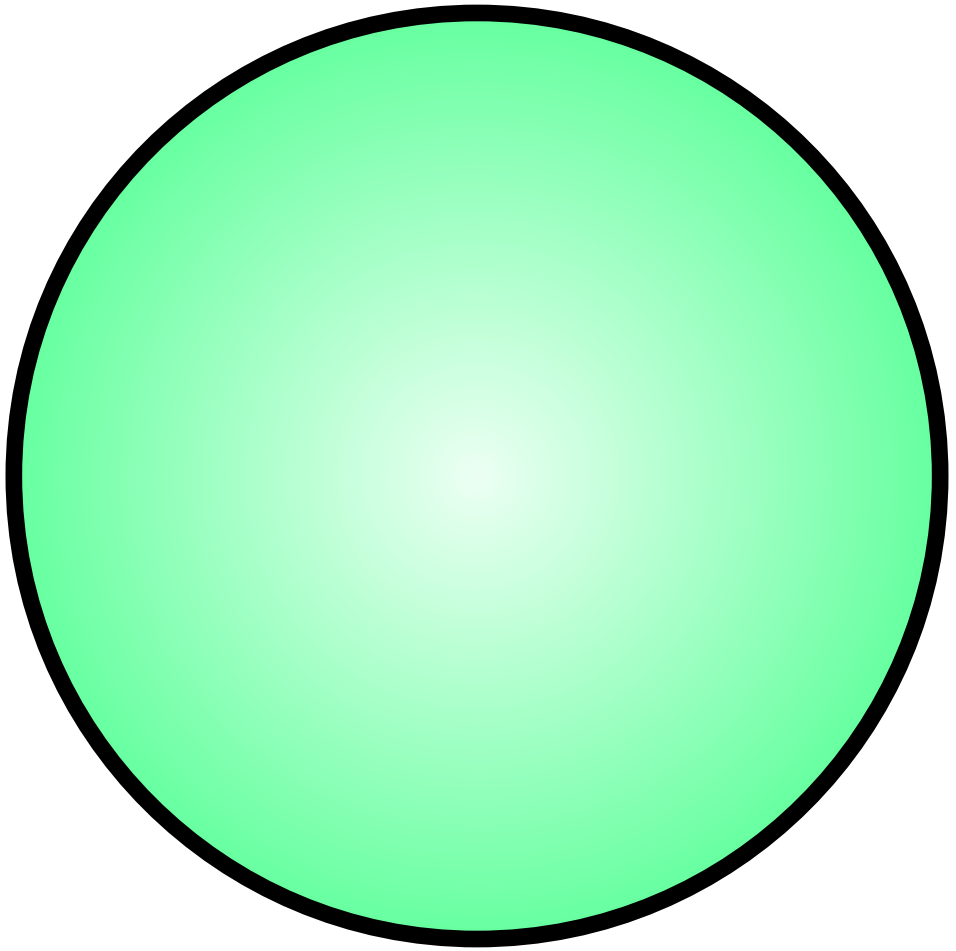
ed according to the type with whic

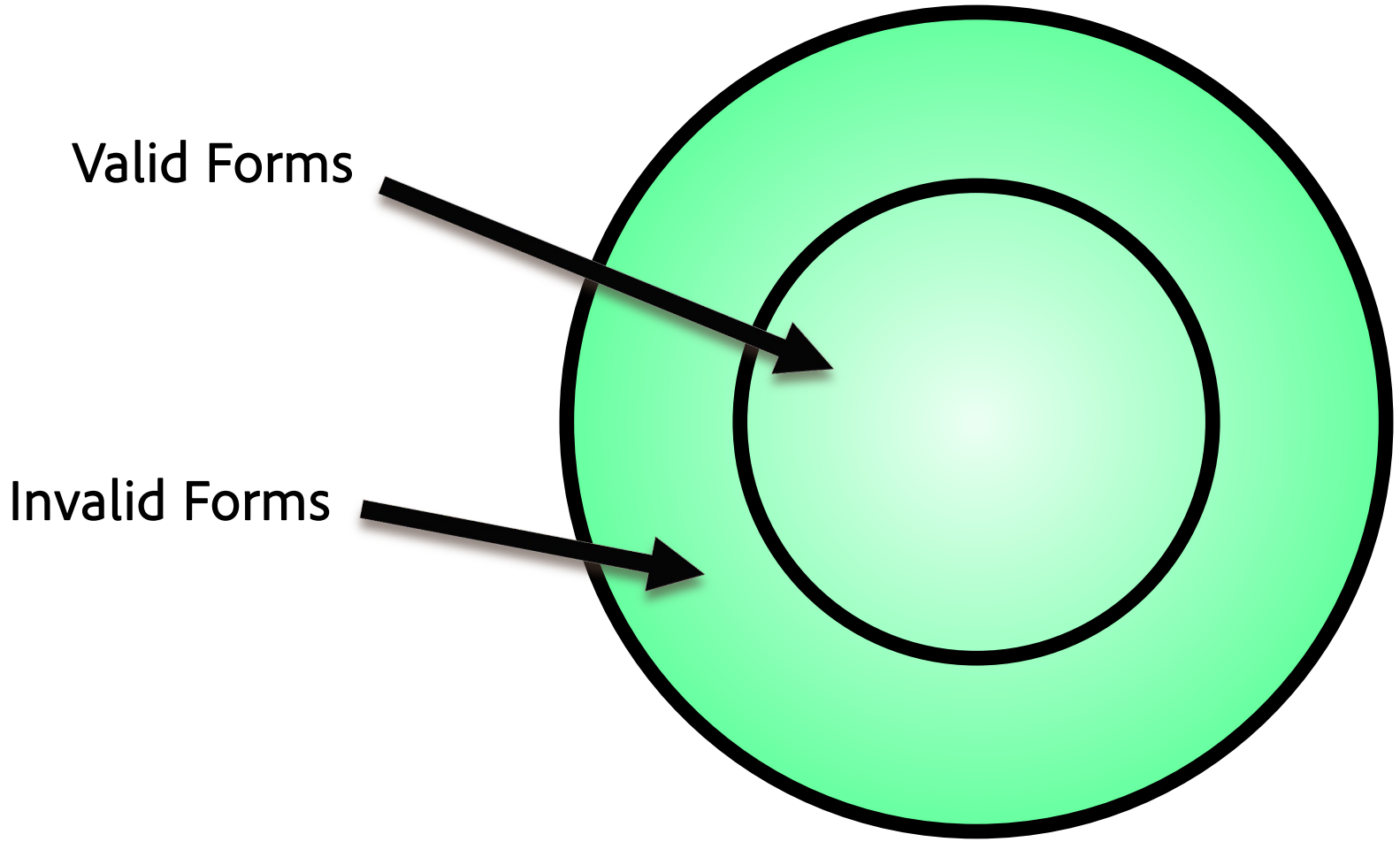
ly constructed and destroyed (reso

oints to a valid object or is the **nu**

gh a pointer is not through the **nu**

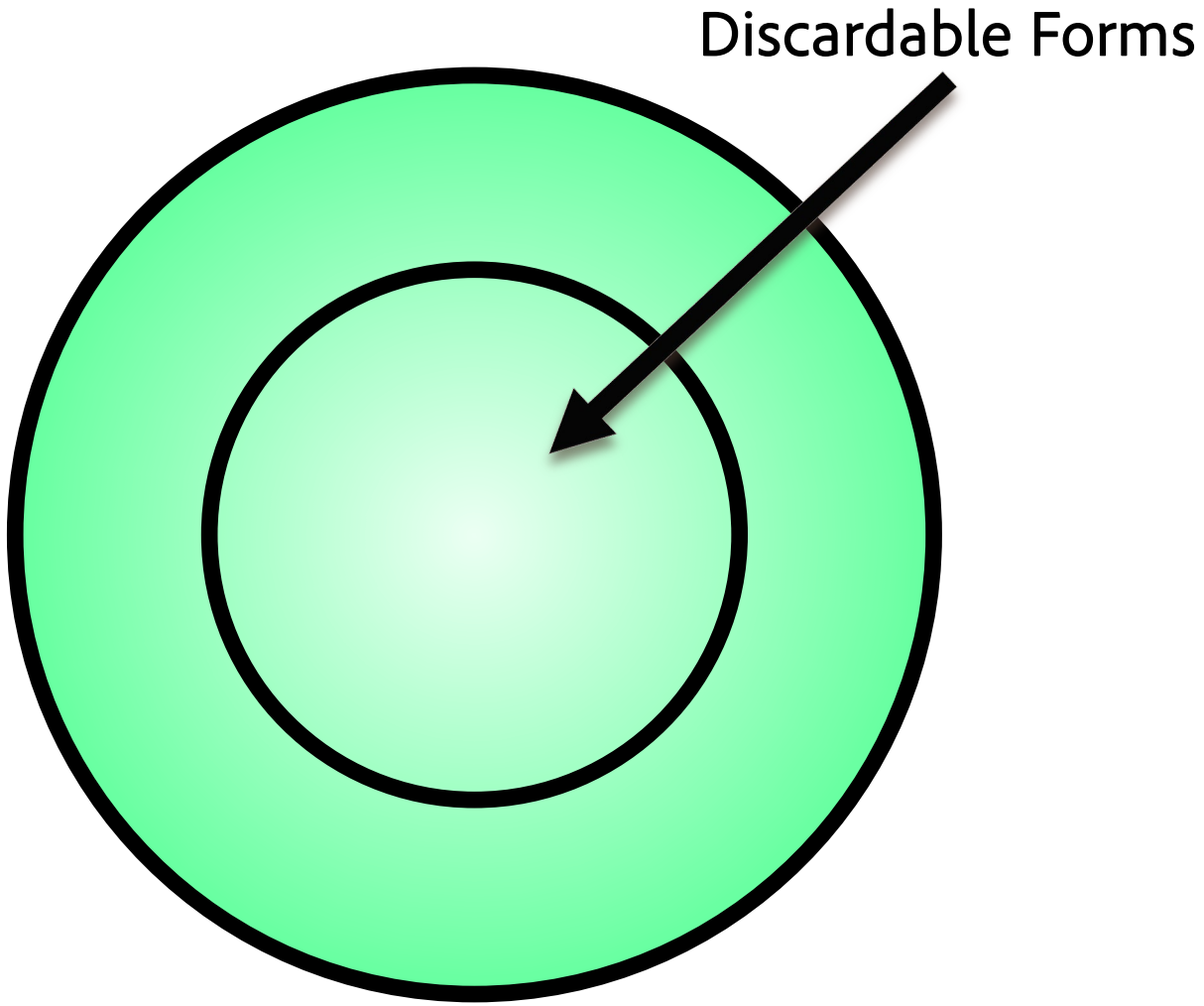
a subscripted pointer is in-range (





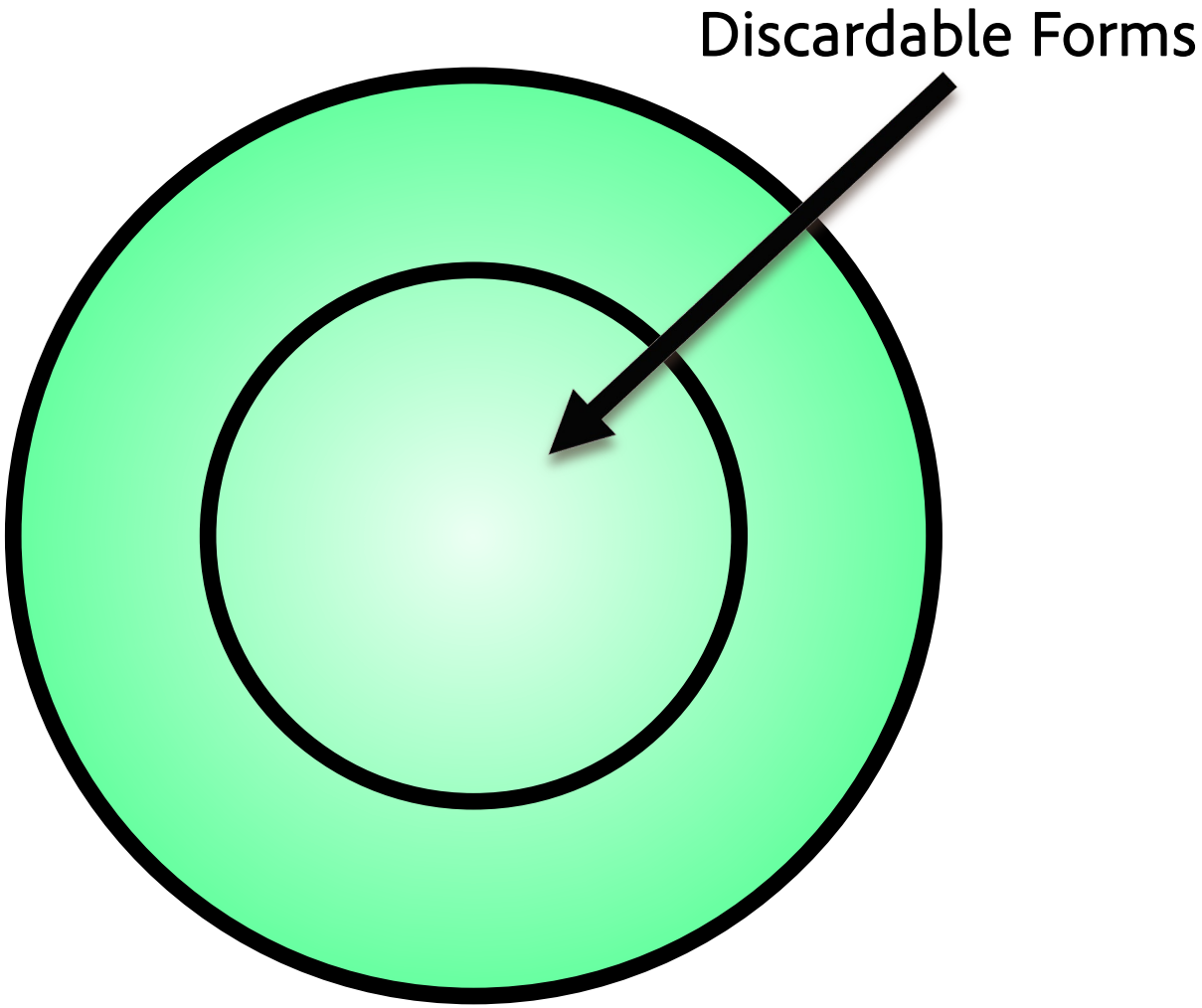
```
unique_ptr<int> p = make_unique<int>( 3 );
delete p.get();
p.release();
```

Discardable



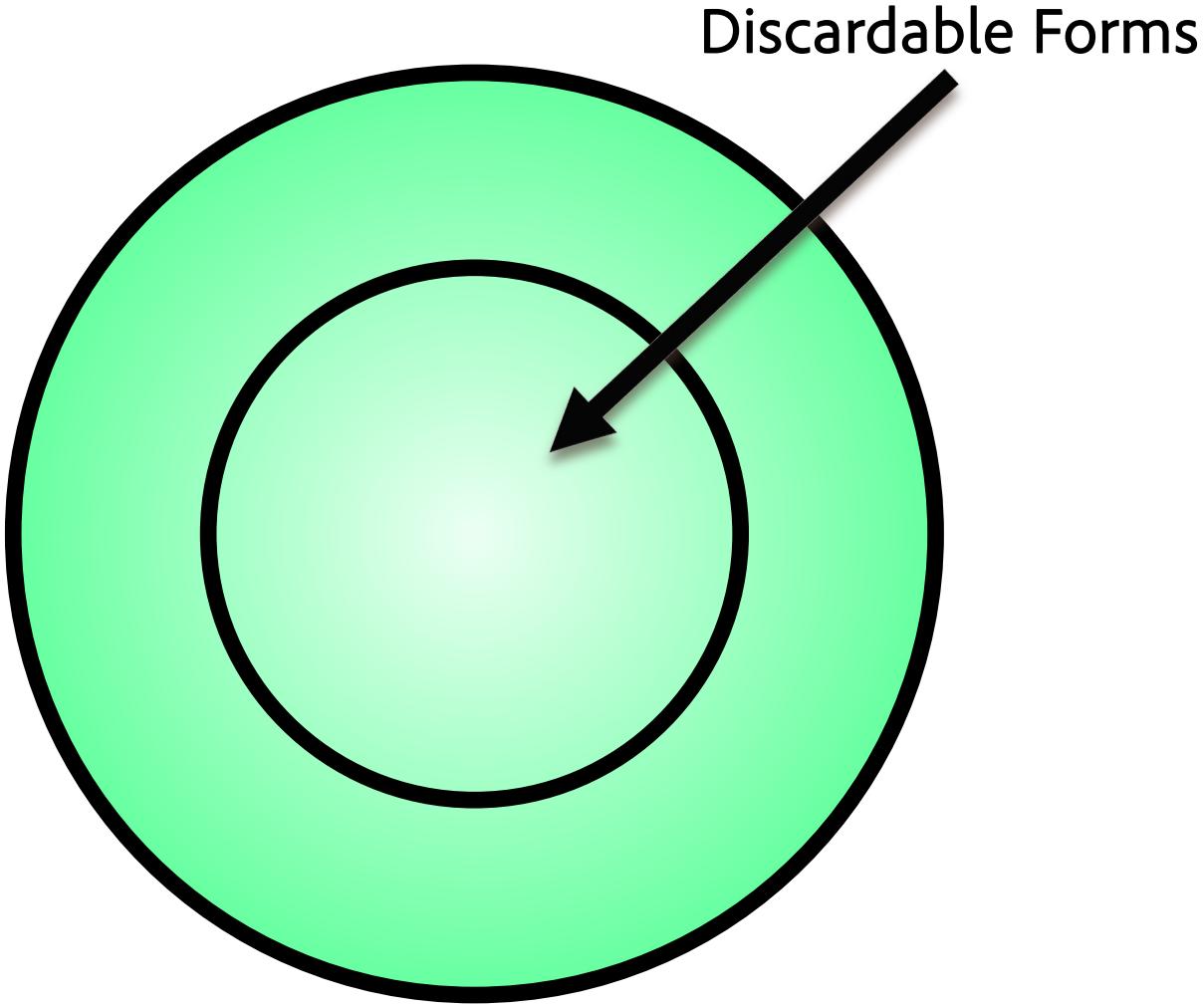
DEFINITION: A **discardable** object can be destructed or assigned to.

Discardable



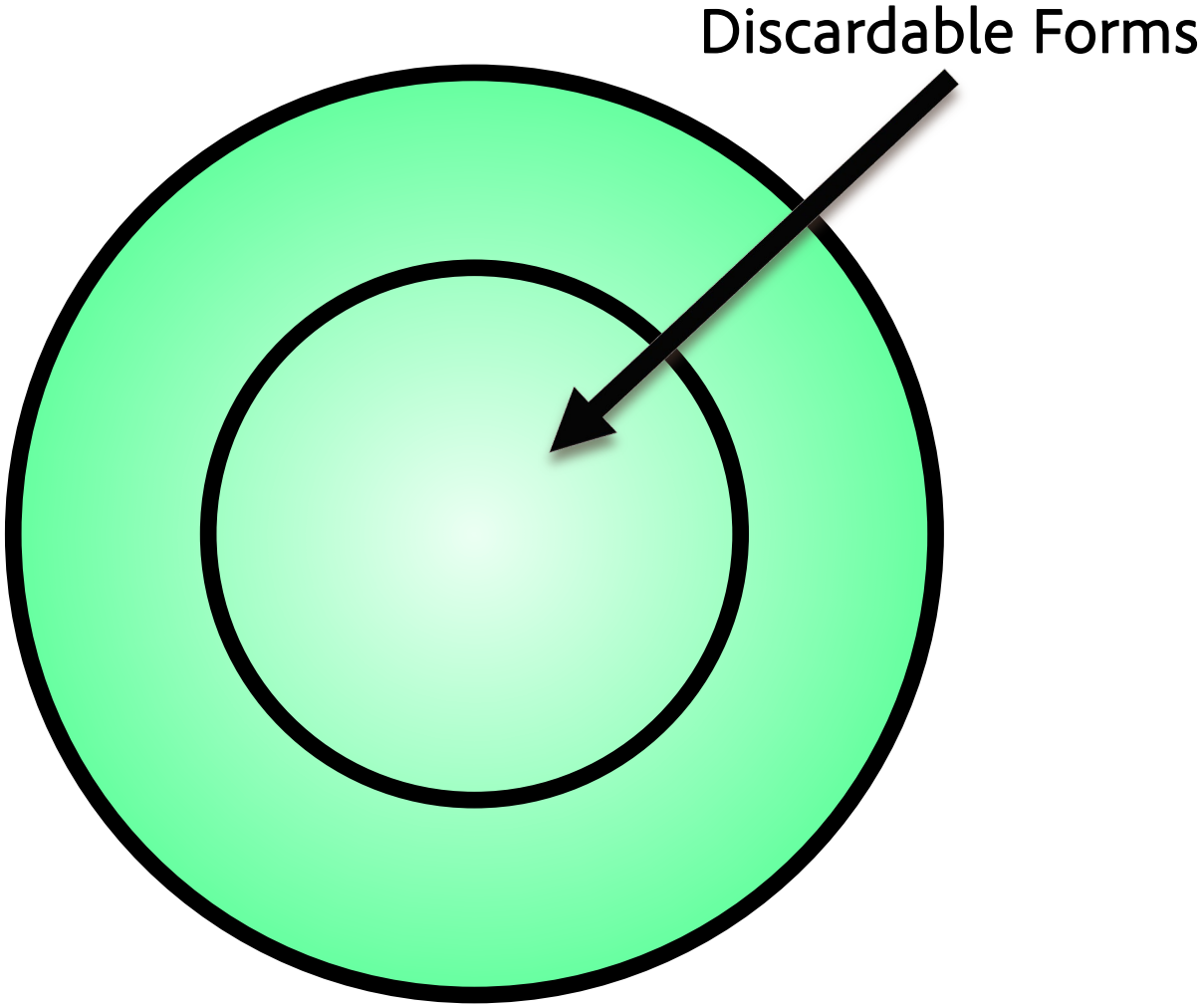
DEFINITION: A **discardable** object's *substance has a form that* can be destructed or assigned to.

Discardable

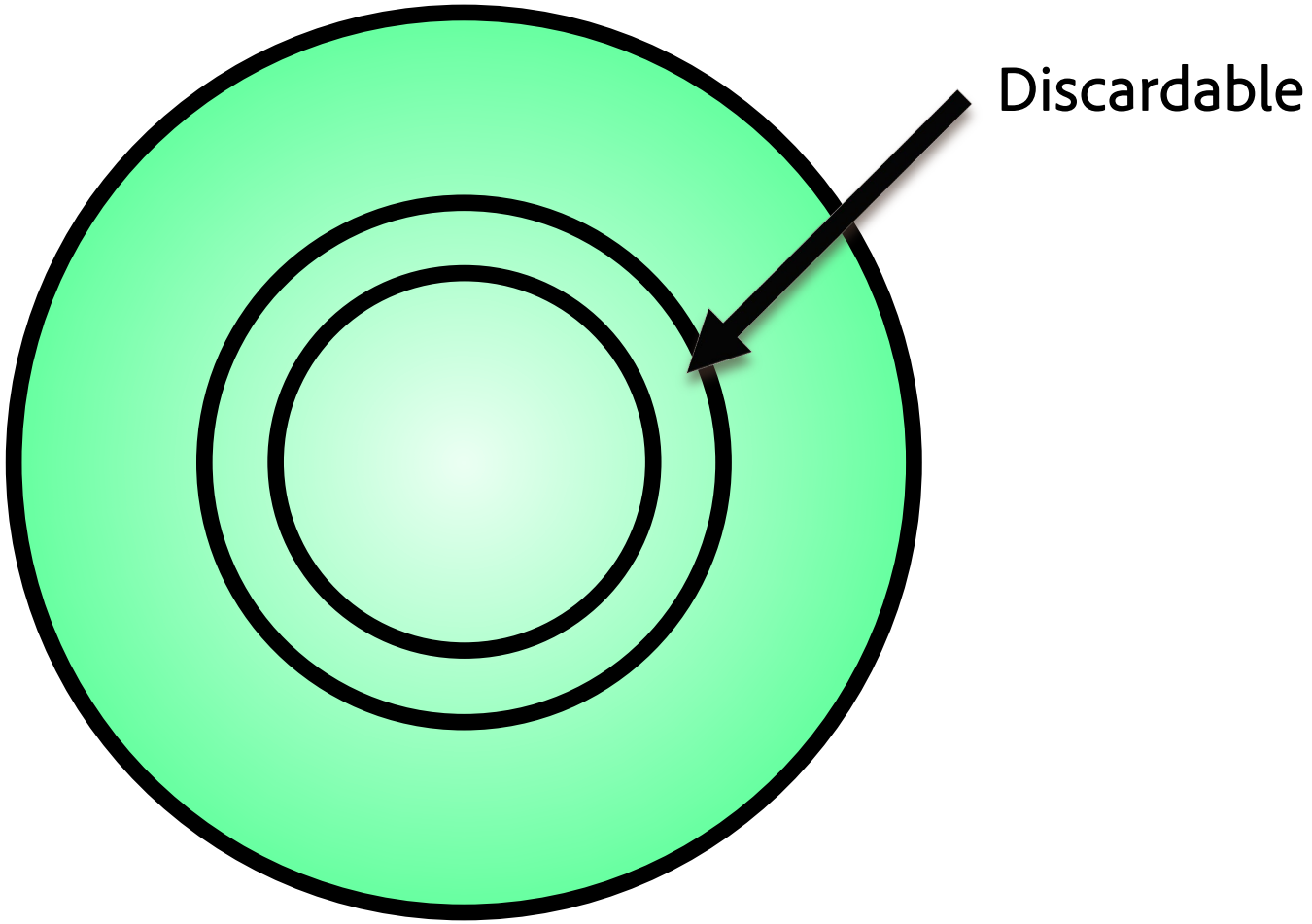


DEFINITION: A **discardable** object's *substance has a form that meets the preconditions for destruction and assignment.*

Discardable

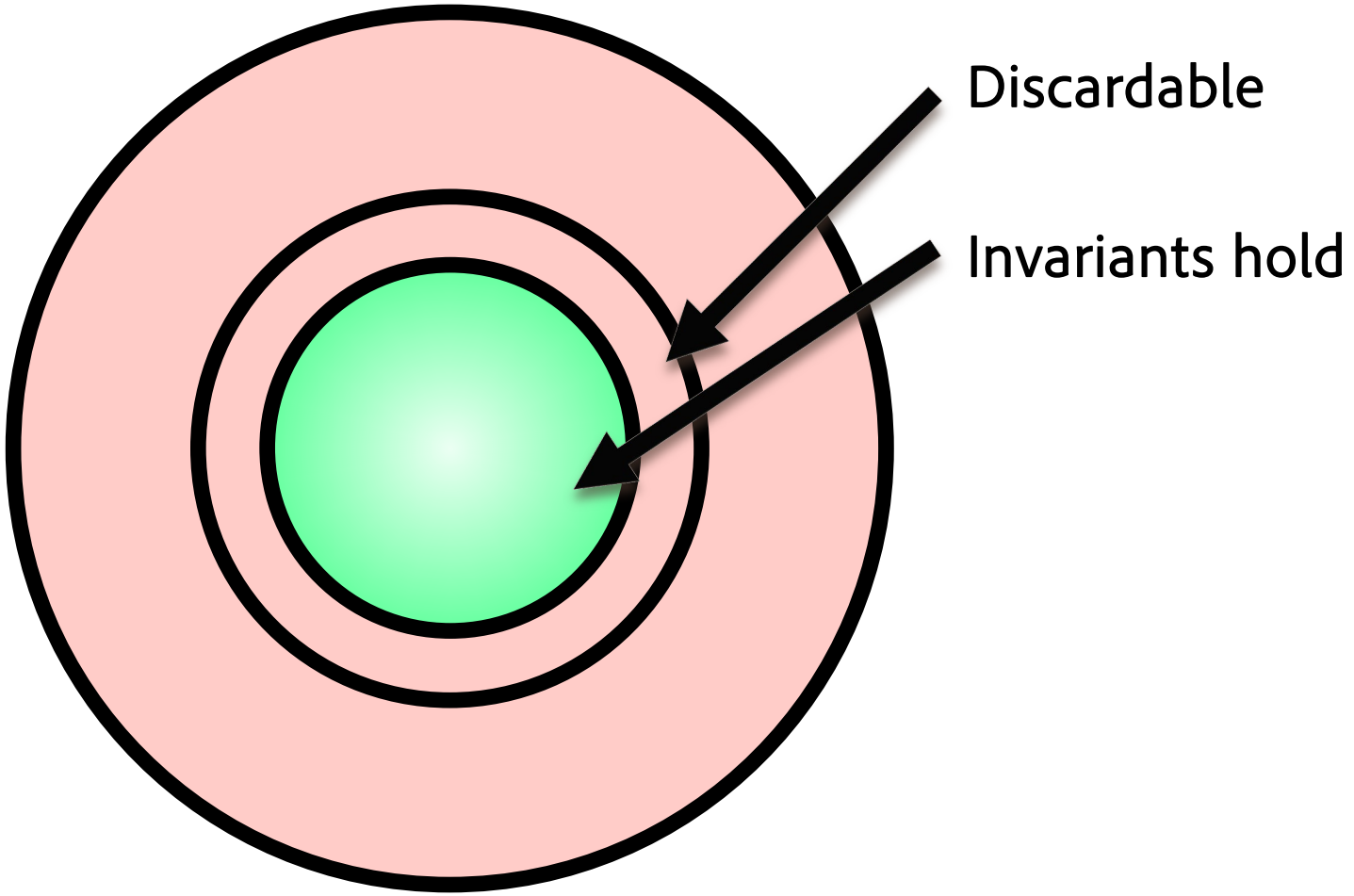


DEFINITION: **Discardable** objects are valid for destruction and assignment.



DEFINITION: A **class invariant** is a predicate over an object's substance for a particular type. A stated invariant for a type implies requirements…

DEFINITION: An object whose class invariants do not hold is called **broken**.



Broken

```
template<typename T, typename U>
class PairSequence {
  vector<T> ts;
  vector<U> us;
};
```

```
template<typename T, typename U>
class PairSequence {
  vector<T> ts;
  vector<U> us;

  // invariant: ts.size() == us.size()
};
```

What does a class invariant imply?

1. Unless a function's documentation says so, broken arguments are invalid. [implicit precondition]
   *(*this of destructor and assignment are notable exceptions)*

2. Class APIs may not give callers broken objects. [implicit postcondition]

3. Mutating class APIs may not leave objects broken that were found unbroken. [implicit postcondition]

First invariant requirement

Unless a function's documentation says so, broken arguments are invalid.
*(*this of destructor and assignment are notable exceptions)*

```
template<typename T, typename U>
class PairSequence {
  //...
  // invariant: ts.size() == us.size()
public:
  PairSequence(const PairSequence &);

  void push_back(pair<T,U>);
  void resize(size_t x);


 ~PairSequence();
  PairSequence& operator=(const PairSequence&);
};
```

First invariant requirement

Unless a function's documentation says so, broken arguments are invalid.
*(*this of destructor and assignment are notable exceptions)*

```
template<typename T, typename U>
class PairSequence {
  //...
  // invariant: ts.size() == us.size()
public:
  PairSequence(const PairSequence &);

  void push_back(pair<T,U>);
  void resize(size_t x);


 ~PairSequence();
  PairSequence& operator=(const PairSequence&);
};
```

First invariant requirement

Unless a function's documentation says so, broken arguments are invalid.
*(*this of destructor and assignment are notable exceptions)*

```
template<typename T, typename U>
class PairSequence {
  //...
  // invariant: ts.size() == us.size()
public:
  PairSequence(const PairSequence &);

  void push_back(pair<T,U>);
  void resize(size_t x);

 ~PairSequence();
  PairSequence& operator=(const PairSequence&);
};
```

First invariant requirement

Unless a function's documentation says so, broken arguments are invalid.
*(*this of destructor and assignment are notable exceptions)*

```
template<typename T, typename U>
class PairSequence {
  //...
  // invariant: ts.size() == us.size()
public:
  PairSequence(const PairSequence &);

  void push_back(pair<T,U>);
  void resize(size_t x);

  ~PairSequence();
  PairSequence& operator=(const PairSequence&);
};
```

First invariant requirement

Unless a function's documentation says so, broken arguments are invalid.
*(*this of destructor and assignment are notable exceptions)*

```
template<typename T, typename U>
class PairSequence {
  //...
  // invariant: ts.size() == us.size()
public:
  PairSequence(const PairSequence &);

  void push_back(pair<T,U>);
  void resize(size_t x);


 ~PairSequence();
  PairSequence& operator=(const PairSequence&);
};
```

First invariant requirement

Unless a function's documentation says so, broken arguments are invalid.
*(*this of destructor and assignment are notable exceptions)*

```
void foo( PairSequence<int,int> );
```

Second invariant requirement

Class APIs may not give callers broken objects.

```
template<typename T, typename U>
class PairSequence {
  //...
  // invariant: ts.size() == us.size()
public:
  PairSequence(const PairSequence &);

  void push_back(pair<T,U>);
  void resize(size_t x);


 ~PairSequence();
  PairSequence& operator=(const PairSequence&);
};
```

Third invariant requirement

Mutating class APIs may not leave objects broken that were found unbroken.

```
template<typename T, typename U>
class PairSequence {
  //...
  // invariant: ts.size() == us.size()
public:
  PairSequence(const PairSequence &);

  void push_back(pair<T,U>);
  void resize(size_t x);

 ~PairSequence();
  PairSequence& operator=(const PairSequence&);
};
```

Third invariant requirement

Mutating class APIs may not leave objects broken that were found unbroken.

```
template<typename T, typename U>
class PairSequence {
  //...
  // invariant: ts.size() == us.size()
public:
  PairSequence(const PairSequence &);

  void push_back(pair<T,U>);
  void resize(size_t x);


 ~PairSequence();
  PairSequence& operator=(const PairSequence&);
};
```

Class invariant best practices

- Document class invariants
- Check invariant preservation of your APIs in debug builds
- Make it impossible (or at least difficult) for a user of your API to get exposed to broken objects

Substance

Substance

Essence

Objects and their values

- Objects usually have a meaning that goes beyond substance and invariants
- What do we mean when we say two objects are “the same”?

Rational class

```
struct rational {
  unsigned numerator;
  unsigned denominator;
};
```

Rational class

```
struct rational {
  unsigned numerator;
  unsigned denominator;
};
```

Observation: the denominator cannot be 0

Rational class

```
class rational {
  unsigned _numerator;
  unsigned _denominator;
  // invariant: _denominator != 0
};
```

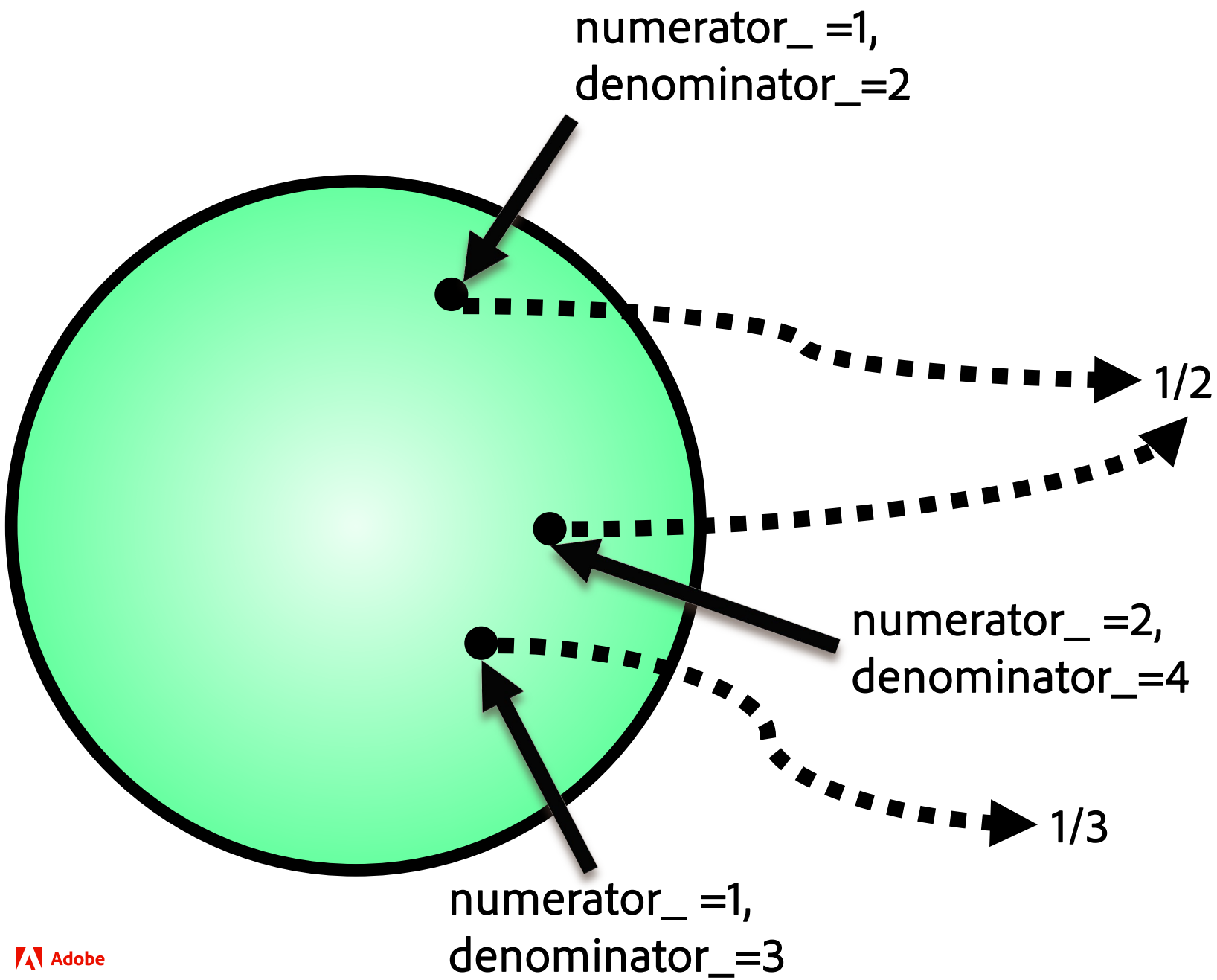
Observation: the denominator cannot be 0

Rational class

```
class rational {
  unsigned _numerator;
  unsigned _denominator;
  // invariant: _denominator != 0
};
```

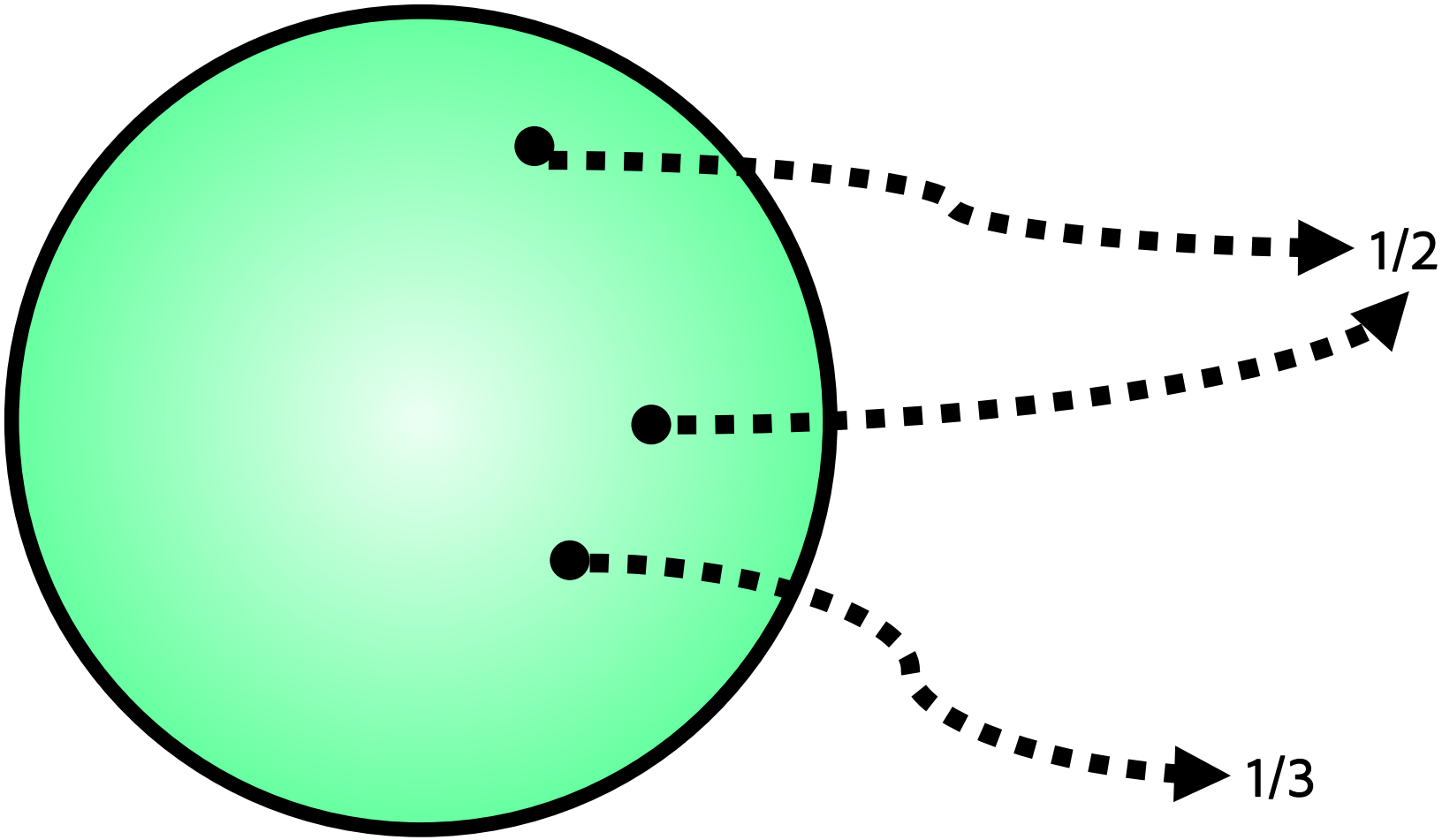
Observation: objects of this class are intended to represent rational numbers

- A *rational* object with *_numerator* set to 1 and *_denominator* set to 2 represents the rational number ½.
- The essence of a *rational* in this state is ½.



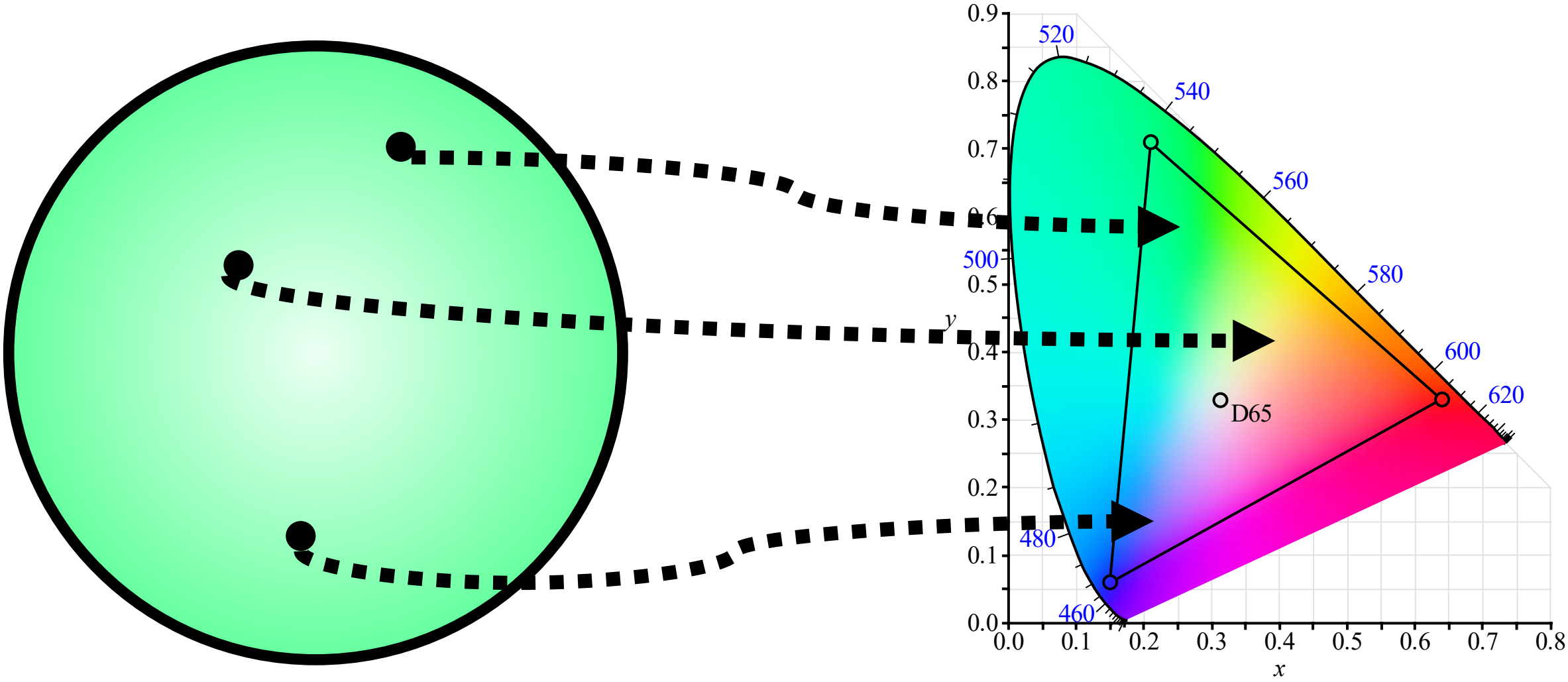
Substance

Essence/Value



Substance

Essence/Value



DEFINITION: **Value types** have substances that map to values.

DEFINITION: An **object has a value** if its substance maps to a value.

Rational Class

```
// A rational number.
// https://en.wikipedia.org/wiki/Rational_number
class rational {
public:
  // Creates an instance with value a/b.
  // - Precondition: b!=0.
  rational(unsigned a, unsigned b);
  bool operator==(const rational & rhs) const;
  // Returns the sum of a and b.
  friend rational operator+(rational a, rational b);
private:
  unsigned _numerator;
  unsigned _denominator;
  // invariant: _denominator != 0
};
```

Rational Class

```
// A rational number.
// https://en.wikipedia.org/wiki/Rational_number
class rational {
public:
  // Creates an instance with value a/b.
  // - Precondition: b!=0.
  rational(unsigned a, unsigned b);
  bool operator==(const rational & rhs) const;
  // Returns the sum of a and b.
  friend rational operator+(rational a, rational b);
private:
  unsigned _numerator;
  unsigned _denominator;
  // invariant: _denominator != 0
};
```

Document the values the class represents.

Rational Class

```
// A rational number.
// https://en.wikipedia.org/wiki/Rational_number
class rational {
public:
  // Creates an instance with value a/b.
  // - Precondition: b!=0.
  rational(unsigned a, unsigned b);
  bool operator==(const rational & rhs) const;
  // Returns the sum of a and b.
  friend rational operator+(rational a, rational b);
private:
  unsigned _numerator;
  unsigned _denominator;
  // invariant: _denominator != 0
};
```

External interfaces should be primarily documented in terms of values.

Rational Class

```
// A rational number.
// https://en.wikipedia.org/wiki/Rational_number
class rational {
public:
  // Creates an instance with value a/b.
  // - Precondition: b!=0.
  rational(unsigned a, unsigned b);
  bool operator==(const rational & rhs) const;
  // Returns the sum of a and b.
  friend rational operator+(rational a, rational b);
private:
  unsigned _numerator;
  unsigned _denominator;
  // invariant: _denominator != 0
};
```

- operator== returns whether or not objects have the same value
- Value types can sometimes represent the same value in multiple ways. Consider

```
auto a = rational{ 1, 2 };
auto b = rational{ 2, 4 };
// a and b have the same value
```

Rational Class

```
// A rational number.
// https://en.wikipedia.org/wiki/Rational_number
class rational {
public:
  // Creates an instance with value a/b.
  // - Precondition: b!=0.
  rational(unsigned a, unsigned b);
  bool operator==(const rational & rhs) const;
  // Returns the sum of a and b.
  friend rational operator+(rational a, rational b);
private:
  unsigned _numerator;
  unsigned _denominator;
  // invariant: _denominator != 0
};
```

Broken objects have no value

Value types

- Most types are value types
- Non-value types (aka mechanisms) are things like traits, metafunctions, etc.

Substance

Substance

Essence

Substance

Essence

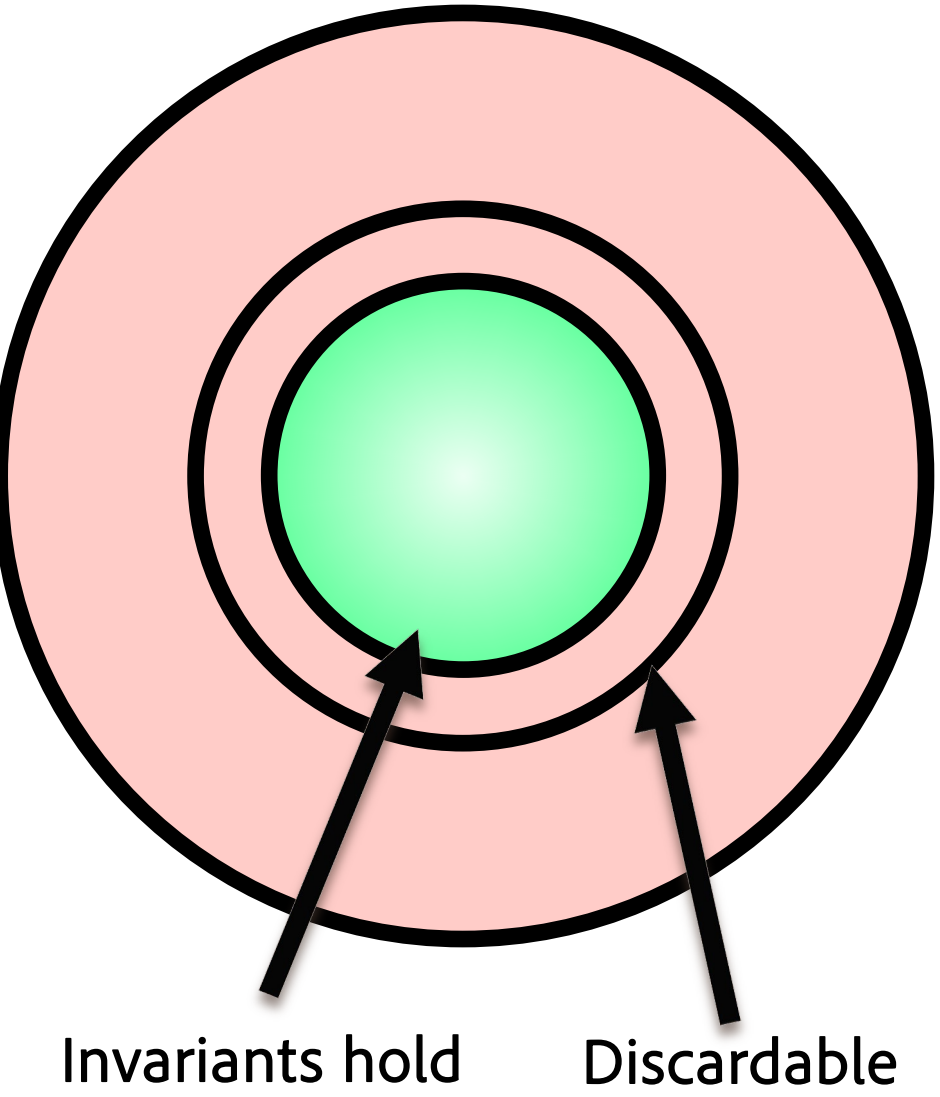
Meaning

DEFINITION: An object is **meaningless** if the most that is known of its substance is that it is discardable.

Meaningless objects

```
int i;

i = 3;
```

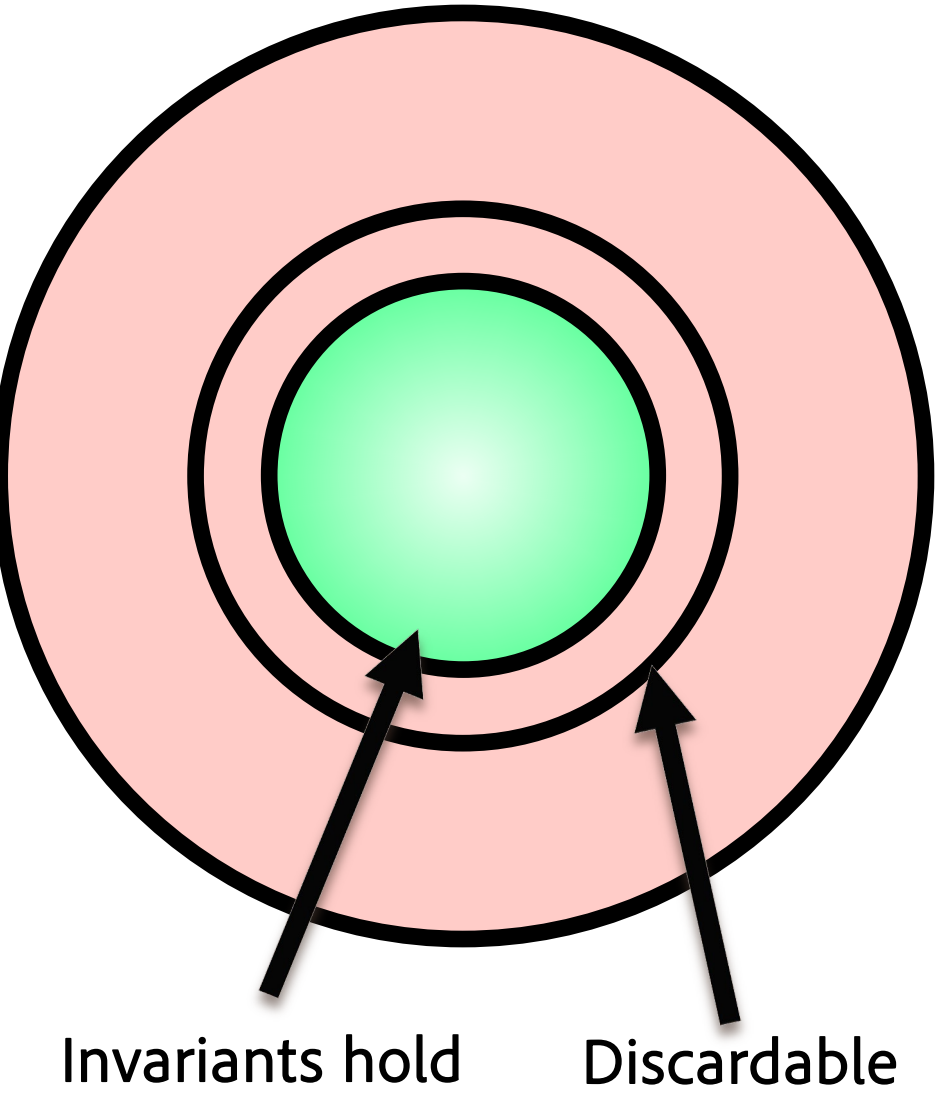
Meaningless objects




```
int i;
i = 3;
```

Here *i* is meaningless

Meaningless objects




```
int i;

i = 3;
```

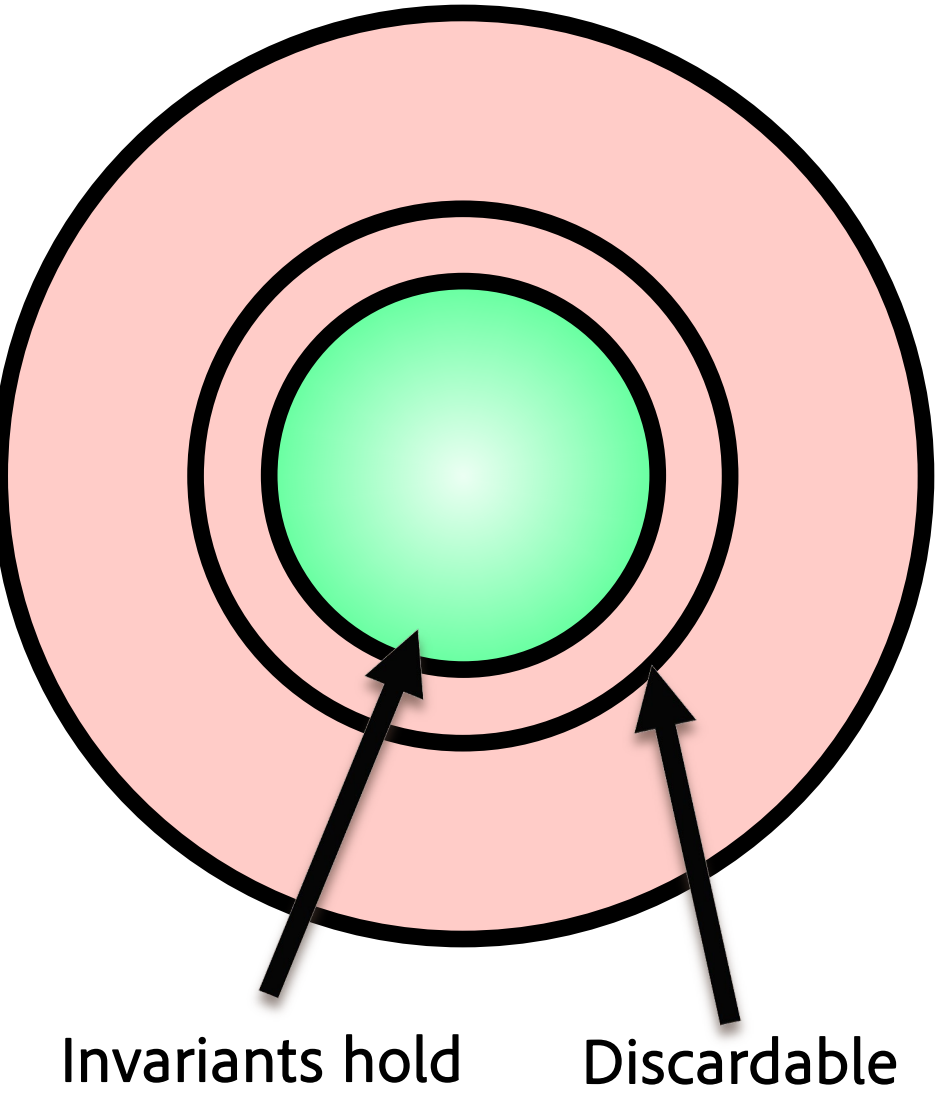
Here *i* is not meaningless

STLab exception safety classes

Minimal exception safety. Partial execution of failed operations can result in side effects, but all objects under mutation must be discardable.

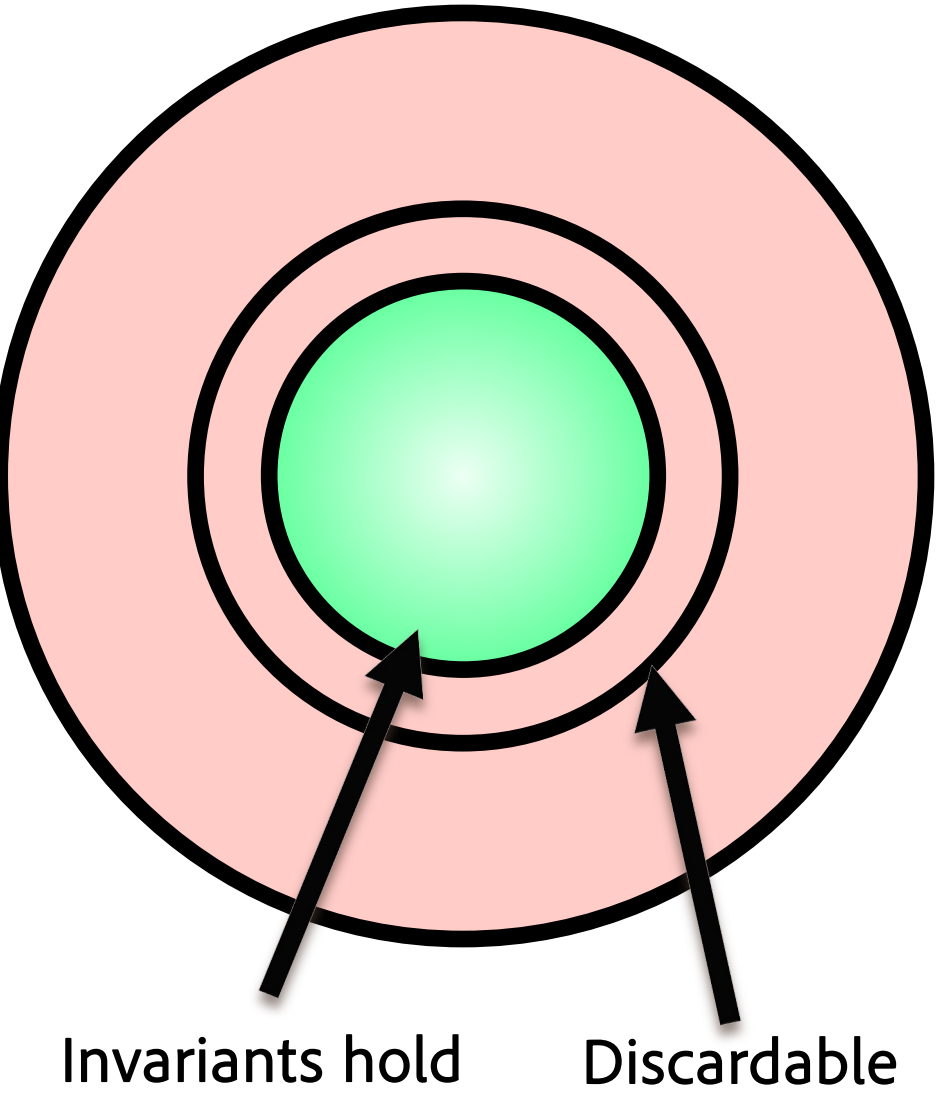
Strong exception safety. Partial execution of failed operations cannot result in side effects. All objects under mutation must have their original values.

Objects becoming meaningless



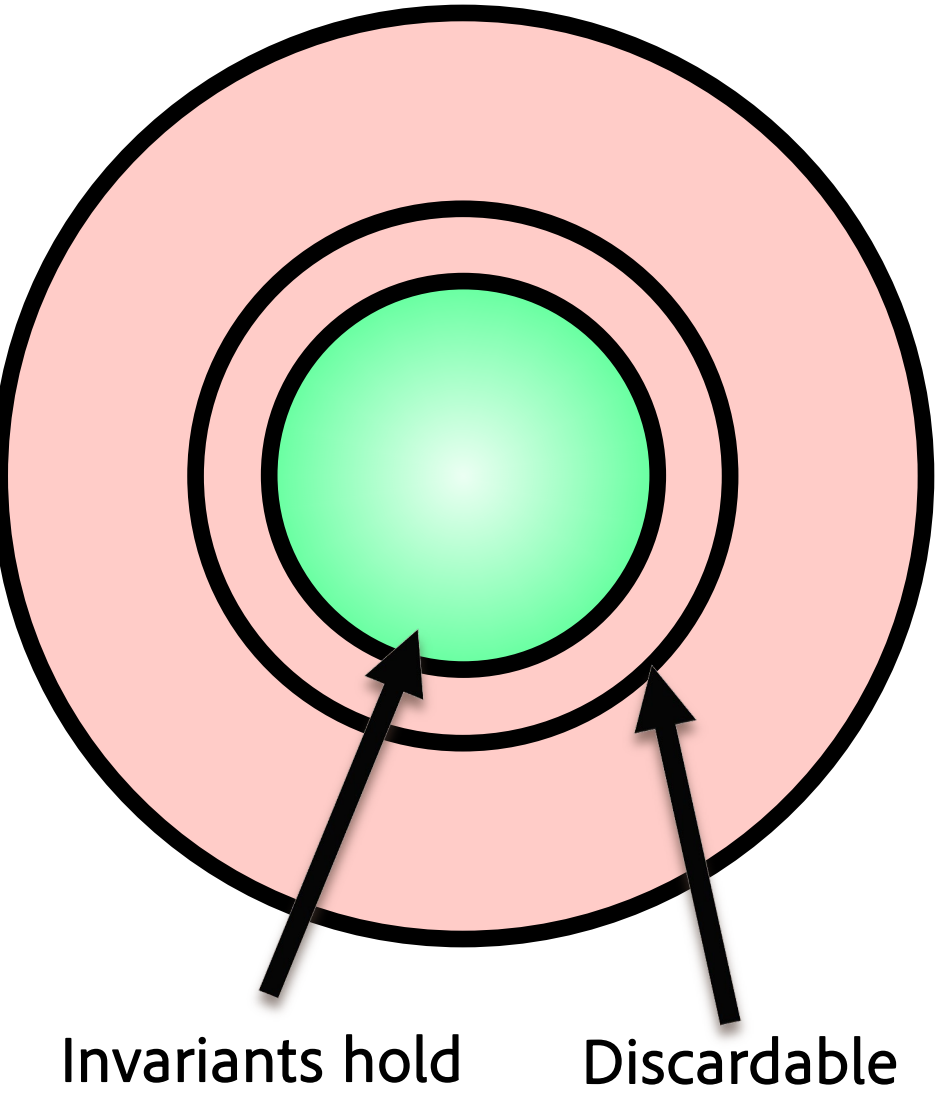

```
// Has the minimal exception safety guarantee
void minimal(Foo&);
// Has the strong exception safety guarantee
void strong(Foo&);

x = /*...*/;
try{ minimal(x); }
catch(...) {

}

x = /*...*/;
try{ strong(x); }
catch(...) {

}
```

Objects becoming meaningless



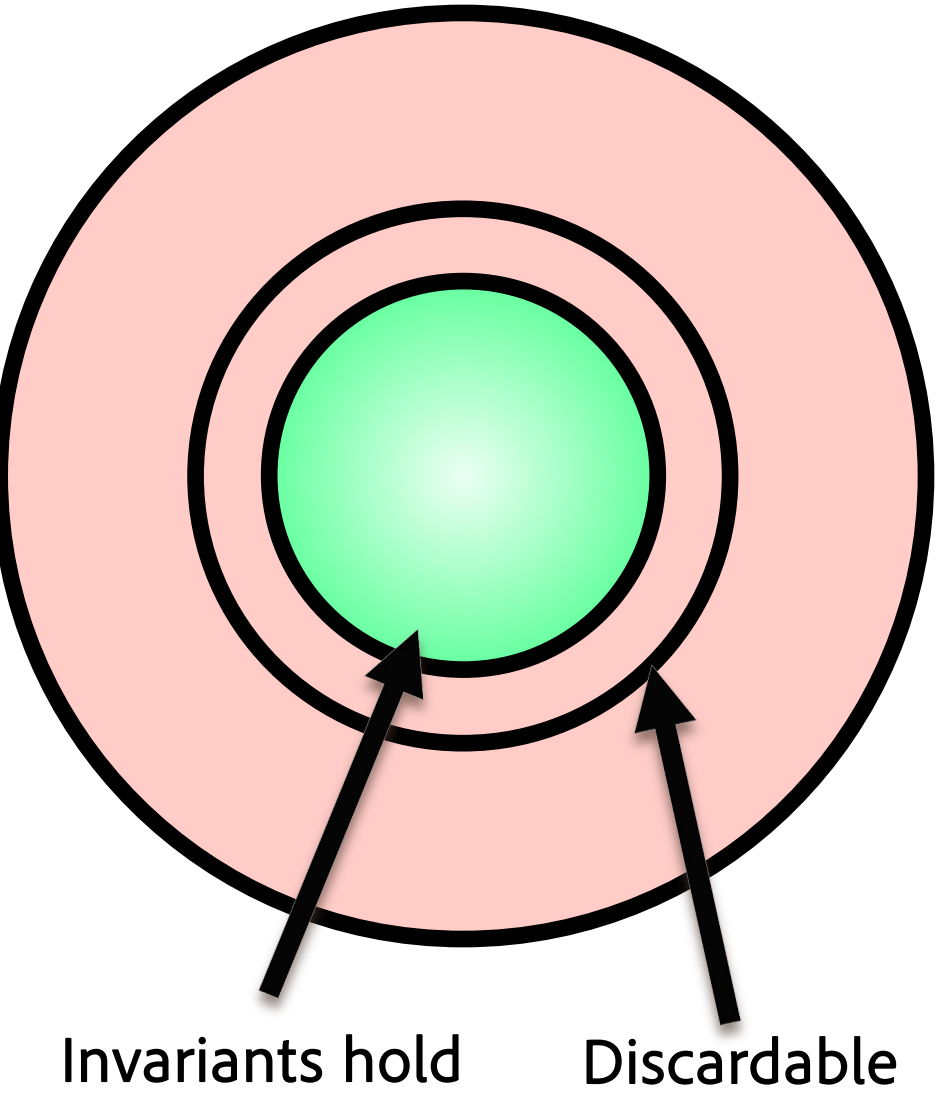

```
// Has the minimal exception safety guarantee
void minimal(Foo&);
// Has the strong exception safety guarantee
void strong(Foo&);

x = /*...*/;
try{ minimal(x); }
catch(...) {

}

x = /*...*/;
try{ strong(x); }
catch(...) {

}
```

Objects becoming meaningless



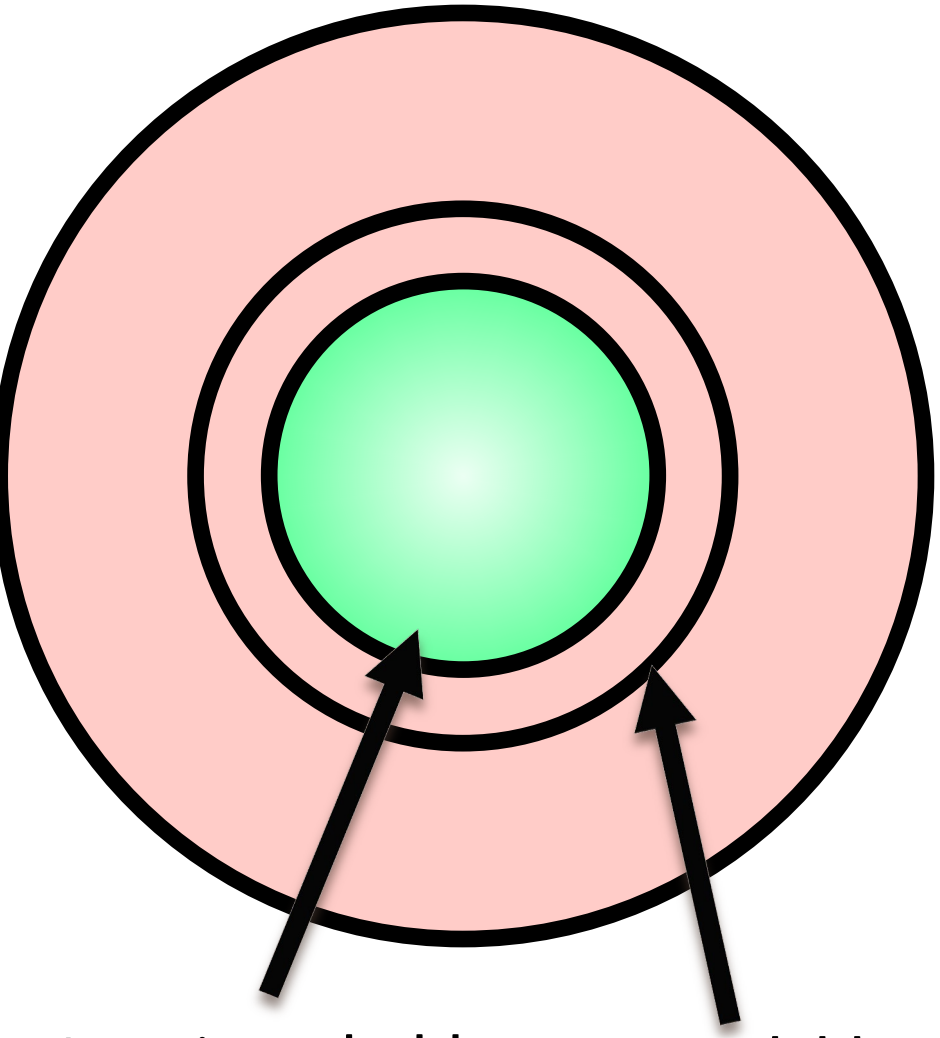

```
// Has the minimal exception safety guarantee
void minimal(Foo&);
// Has the strong exception safety guarantee
void strong(Foo&);

x = /*...*/;
try{ minimal(x); }
catch(...) {

}

x = /*...*/;
try{ strong(x); }
catch(...) {

}
```

Objects becoming meaningless



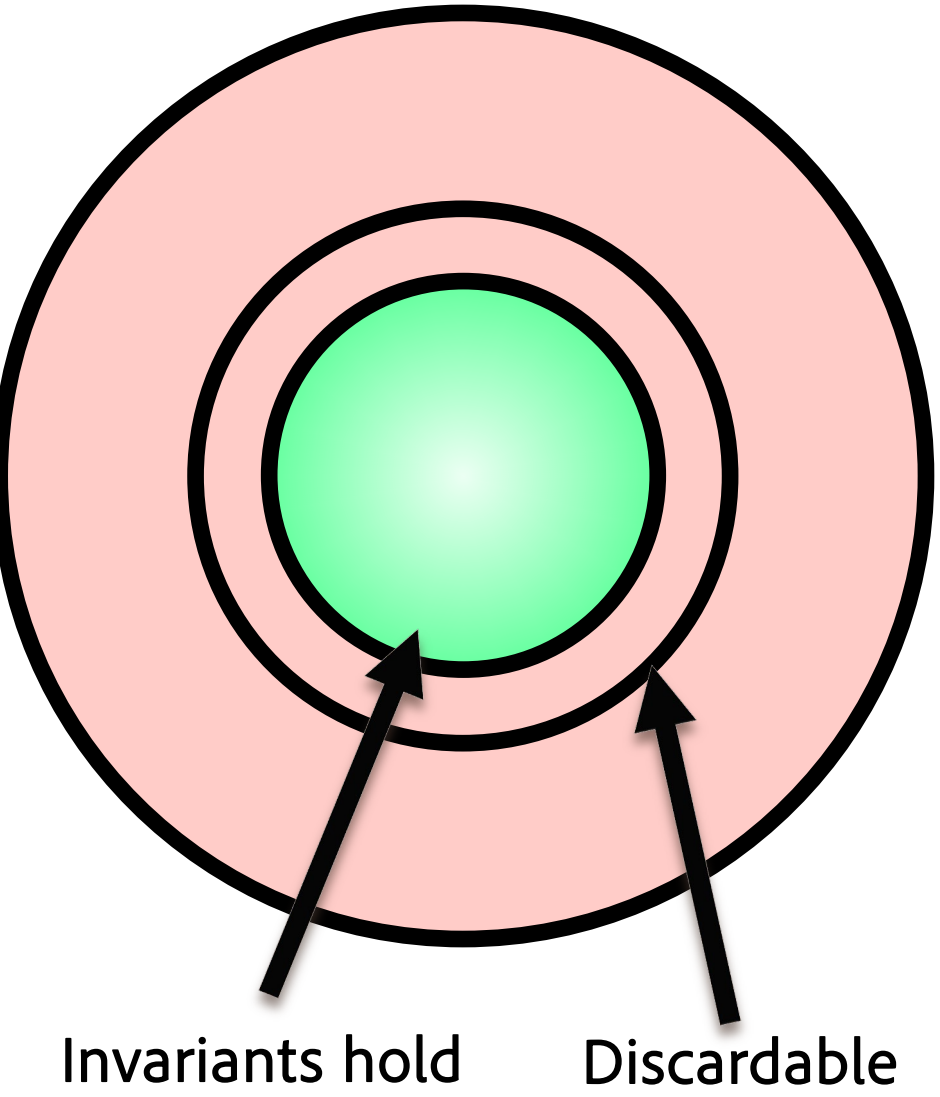

```
// Has the minimal exception safety guarantee
void minimal(Foo&);
// Has the strong exception safety guarantee
void strong(Foo&);

x = /*...*/;
try{ minimal(x); }
catch(...) {
                    <-- Here x is meaningless
}

x = /*...*/;
try{ strong(x); }
catch(...) {

}
```

Objects becoming meaningless




```
// Has the minimal exception safety guarantee
void minimal(Foo&);
// Has the strong exception safety guarantee
void strong(Foo&);

x = /*...*/;
try{ minimal(x); }
catch(...) {

}

x = /*...*/;
try{ strong(x); }
catch(...) {
                    <-- Here x is not meaningless
}
```

Objects becoming meaningless




```
Foo a = /*...*/;
Foo b = std::move(a);
```

Here *a* is meaningless

Meaningless objects

- Have a substance
- Are discardable
- May or may not be broken
- Can be made meaningful through assignment or other setter operations

Intuition: an observation of a meaningless object is a defect

Unless a function's documentation says so, meaningless arguments are invalid. [implicit precondition]

*(*this of destructor and assignment are notable exceptions)*

Valid

- separation
- aesthetics
- subjective

Object

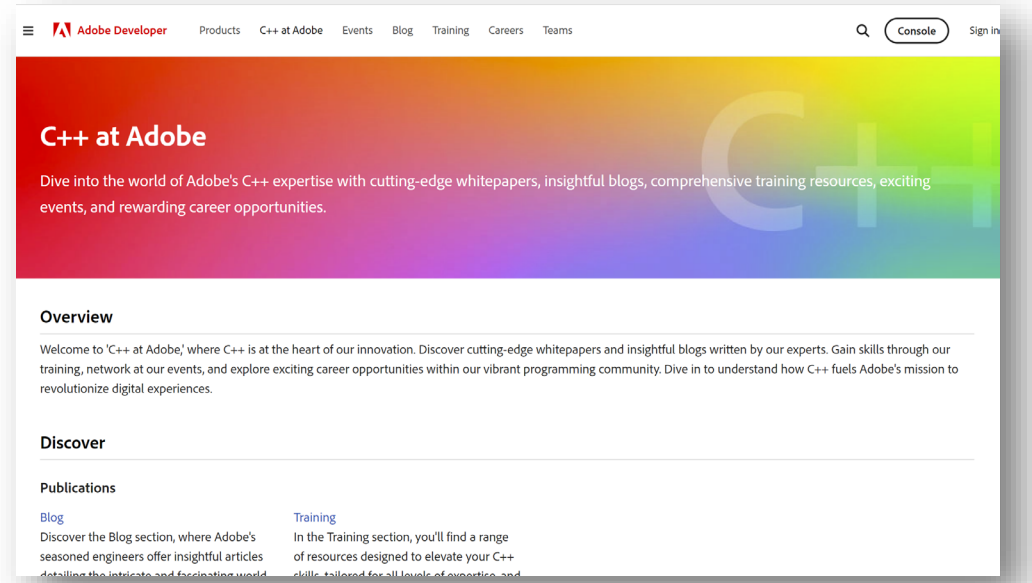
- substance
- essence
- meaning

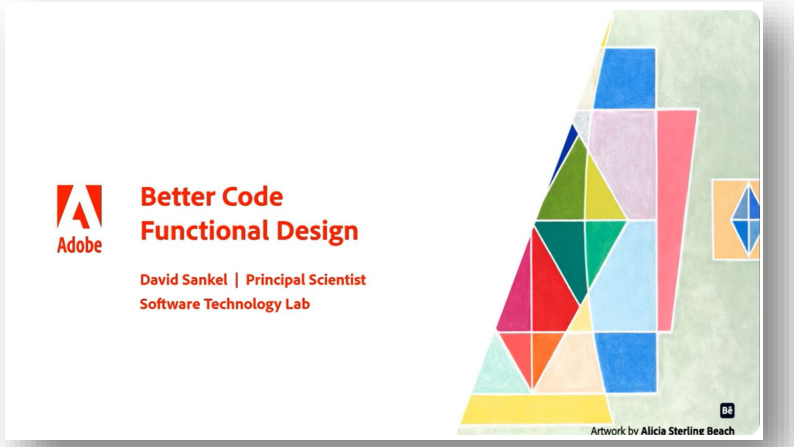
- substance “Am I?”
- essence “What am I?”
- meaning “Why am I?”

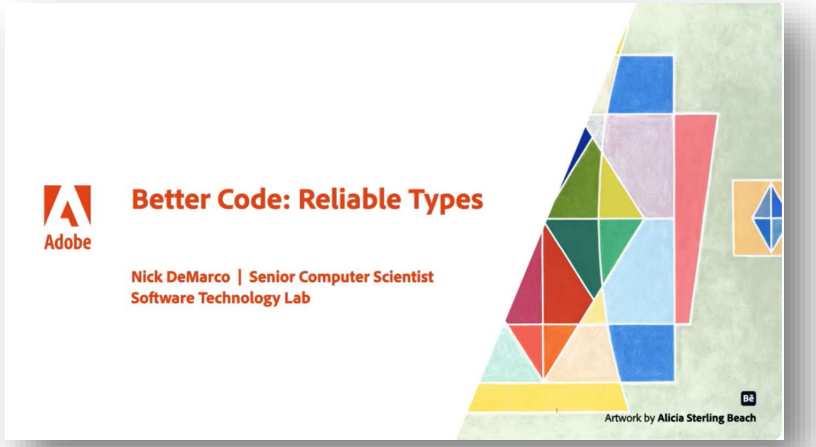
C++ at Adobe!

developer.adobe.com/cpp

- Careers
- Events
- Training Videos (STLab Better Code Series!)
- Blog Posts
- …







Comments/Questions


Adobe
Artwork by Dan Zucco

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

Adobe Illustrator Adobe After Effects