Why Loops End

Lisa Lippincott

C++now, May 2023

No. 1

The heat death of the universe

Will this loop ever end?

Yes! Execution increases the entropy of the universe, turning free energy into useless thermal energy.

Each iteration consumes some free energy, and the free energy of the universe is eventually exhausted.

👿

😈 Will this loop end for some non-cosmological reason?

I don’t know any reason beyond the cosmos. 🙂

Perhaps you'd like to be more specific? 🙂

Will this loop end for some reason that:

- is expressed on every code path that enters the loop,
- is complete before any part of the loop is repeated, and
- is not separated from the loop by a function interface?

That *is* very specific! But you'll have to explain the part about the function interfaces.

result_type function_name (*parameter_list*)
interface
{

// preconditions…

implementation;

// postconditions…

}

The calling function is responsible for the top part of the interface.

The called function is responsible for the bottom part of the interface.

Let’s call that sort of reason a **local** reason.
Does the loop end for some local reason?

frantically types a comment

OK, now there's a local reason.

Now can you write it in a way my robot pal can understand?

Sorry, I don't read comments.

I’m very formal, and just read the compilable code.

So when you're reading the code, what do you look for?

I look for periods of **stability,** during which the state of an object doesn't change.

I look for **substitutability:** times when two different objects have the same value.

And I look for **repetition,** when one operation is so like a previous operation that it must produce a similar result.

Function interfaces tell me about all these things.

🤖 ✅

```
while ( false )
  {}
```

🤖 ❌

```
while ( true )
  {}
```

constexpr bool **false** interface

```
  {
    implementation;

    transfer_stability result;
    discern result;

    if ( result )
       std::unreachable();
    else
       {}
  }
```

constexpr bool **true** interface

```
  {
    implementation;

    transfer_stability result;
    discern result;

    if ( result )
       {}
    else
       std::unreachable();
  }
```

```
constexpr bool false
interface
  {
   implementation;

   transfer_stability result;
   discern result;

   if ( result )
      std::unreachable();
   else
      {}
  }
```

This function gives its caller a right to the stability of the **result** object it creates.

😈 If it changes while you hold the right, *it’s not your fault*.

```
constexpr bool false
interface
  {
   implementation;

   transfer_stability result;
   discern result;

   if ( result )
       std::unreachable();
   else
       {}
  }
```

This function promises that every **result** it produces will have the same value.

😈 If it gives you two different values, *it's not your fault.*

```
constexpr bool false
interface
  {
   implementation;

   transfer_stability result;
   discern result;

   if ( result )
      std::unreachable();
   else
      {}
  }
```

If **unreachable** is executed, the program has undefined behavior.

😈 Undefined behavior in somebody else's postcondition is *not your fault.*

✅

```
while ( false )
  {}
```

The branch in the interface is repeated by the loop.

If the two branches go in different directions, *it's not your fault.*

```
constexpr bool false
interface
  {
   implementation;

   transfer_stability result;
   discern result;

   if ( result )
      std::unreachable();
   else
      {}
  }
```

🤖 ✅

```
while ( 0 != 0 )
    {}
```

```
bool operator!=( const int a,
                 const int b )
interface
   {
    extend_stability a, b;
    discern a, b;

    implementation;

    transfer_stability result;
    discern result;

    // . . .

   }
```

```
bool operator!=( const int a,
                 const int b )
interface
  {
   extend_stability a, b;
   discern a, b;

   implementation;

   transfer_stability result;
   discern result;

   // ...

  }
```

The caller is responsible for the stability of the parameters during the operation.

The implementation gives the caller a right to the stability of the result.

```
bool operator!=( const int a,
                 const int b )
interface
  {
   extend_stability a, b;
   discern a, b;

   implementation;

   transfer_stability result;
   discern result;

   // …

  }
```

When the operation is repeated with the same parameter values…

…the same **result** value is returned.

```
bool operator!=( const int a,
                 const int b )
interface
  {
   extend_stability a, b;
   discern a, b;

   implementation;

   transfer_stability result;
   discern result;

   claim (a != a) == false;

  }
```

```
bool operator!=( const int a,
                 const int b )
interface
  {
   extend_stability a, b;
   discern a, b;

   implementation;

   transfer_stability result;
   discern result;

   claim primitive_not_equal(a, b) == result;
   claim primitive_not_equal(a, a) == false;
  }
```

```
bool primitive_not_equal( const int a,
                          const int b )
interface
  {
   extend_stability a, b;
   discern a, b;

   implementation;

   transfer_stability result;
   discern result;
  }
```

```
bool operator!=( const int a,
                 const int b )
interface
  {


   primitive_interface;




   using_primitive operator!=;
   claim (a != a) == false;
  }
```

```
bool operator!=( const int a,
                 const int b )
primitive_interface
  {
   extend_stability a, b;
   discern a, b;

   implementation;

   transfer_stability result;
   discern result;
  }
```

```
bool operator!=( const int a,
                 const int b )
interface
  {
   primitive_interface;

   claim (a != a) == false;
  }
```

```
bool operator!=( const int a,
                 const int b )
primitive_interface
  {
   extend_stability a, b;
   discern a, b;

   implementation;

   transfer_stability result;
   discern result;
  }
```

```
bool operator!=( const int a,
                 const int b )
interface
  {
   primitive_interface
     {
      extend_stability a, b;
      discern a, b;

      implementation;

      transfer_stability result;
      discern result;
     }

   claim (a != a) == false;
  }
```

```
bool operator!=( const int a,
                 const int b )
interface
  {
   primitive_interface = default;

   claim (a != a) == false;
  }
```

```
bool operator!=( const int a,
                 const int b )
interface
  {
   primitive_interface = default;

   claim (a != a) == false;
   claim (b != a) == result;

   if ( result )
     {
      using_primitive operator<=, operator>=;

      claim (a == b) == false;
      claim (a  <  b) == (a <= b);
      claim (a  >  b) == (a >= b);
     }
   else
      claim a == b;
  }
```

```
bool operator!=( const int a,
                 const int b )
interface
  {
   primitive_interface = default;

   claim (a != a) == false;
   claim (b != a) == result;

   if ( result )
     {
      using_primitive operator<=, operator>=;

      claim (a == b) == false;
      claim (a  <  b) == (a <= b);
      claim (a  >  b) == (a >= b);
     }
   else
      claim a == b;
  }
```

```
while ( 0 != 0 )
   {}
```

✅

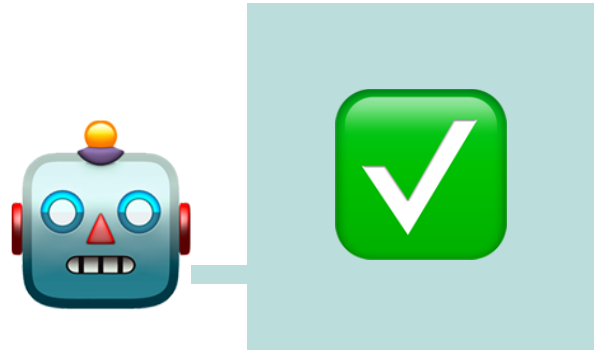
This comparison in the interface repeats the loop condition.

```
int i = 0;

while ( i != 0 )
    ++i;
```

🤖 ✅

🤖 The interface for **0** provides a right to the stability of **i**. We still hold that right when the condition is evaluated.

🤖 We would have to give up that right in order to increment **i**, but this loop never reaches **++i**.



```
int i = 0;

while ( i != 1 )
    ++i;
```

```
int i = 0;

while ( i != 1 )
    ++i;
```

The interface for **operator++** tells me that it sets **i** to **0+1**…

```
int& operator++( const int& a )
interface
  {
   // ...
   const int expected = a + 1;

   transfer_stability a;

   implementation;

   transfer_stability a;
   substitutable &result, &a;

   claim a == expected;
   // ...
  }
```

```
int i = 0;

while ( i != 1 )
    ++i;
```

✅

The interface for **operator++** tells me that it sets **i** to **0+1**…

and the interface for **operator+** tells me that **0+1 == 1**.

```
int operator+( const int a, const int b )
interface
  {
   primitive_interface = default;

   // …

   claim a + 0 == a;
   claim 0 + b == b;
  }
```

✅

```
claim 1 == 0 + 1;

int i = 0;
```

✅

```
while ( i != 1 )
    ++i;
```

The interface for **operator++** tells me that it sets **i** to **0+1**…

and the interface for **operator+** tells me that **0+1 == 1**.

```
int operator+( const int a, const int b )
interface
  {
   primitive_interface = default;

   // …

   claim a + 0 == a;
   claim 0 + b == b;
  }
```

```
claim 2 == 0 + 1 + 1;

int i = 0;

while ( i != 2 )
    ++i;
```

```
template <>
constexpr int operator""< '2' >()
interface
  {
   primitive_interface = default;

   claim result == 1 + 1;

   // ...
  }
```

```
claim 9 == 0 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1;

int i = 0;

while ( i != 9 )
    ++i;
```

```
template <>
constexpr int operator""< '9' >()
interface
  {
   primitive_interface = default;

   claim result == 8 + 1;

   // ...
  }
```

🤖 ✅

```
claim 10 == 0 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1;

int i = 0;
```

🤖 ✅

```
while ( i != 10 )
    ++i;
```

```
template <>
constexpr int operator""< '1', '0' >()
interface
  {
   primitive_interface = default;

   claim result == 9 + 1;

   // ...
  }
```

🤖 ✅

```
claim 11 == 0 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1;

int i = 0;
```

🤖 ✅

```
while ( i != 11 )
    ++i;
```

🤖 The interface for **operator*** tells me that **10 * 1 == 10**.

```
template < char d1, char d0 >
constexpr int operator""()
interface
  {
   primitive_interface = default;

   claim result ==     10 * operator""<d1>()
                     +      operator""<d0>();
   // ...
  }
```

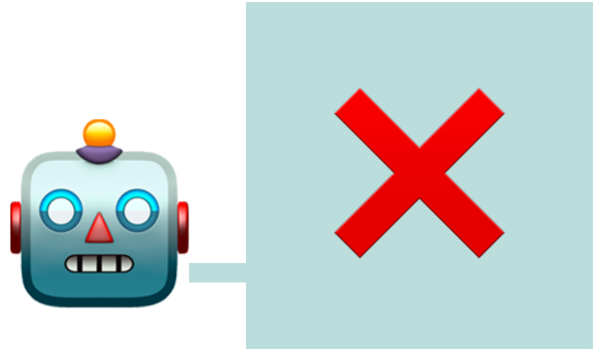
```
claim 12 == 0 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1;

int i = 0;

while ( i != 12 )
    ++i;
```

```
template < char d1, char d0 >
constexpr int operator""()
interface
  {
   primitive_interface = default;

   claim result ==    10 * operator""<d1>()
                    +      operator""<d0>();
   // . . .
  }
```

```
static_assert( 12 == 0 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 );

int i = 0;

while ( i != 12 )
    ++i;
```

✅

If someone runs your function despite a static assertion failing, *it's not your fault.*



```
if ( b <= e )
  {
   int i = b;

   while ( i != e )
       ++i;
  }
```

A **variant** is an integer expression whose value is non-negative after loop initialization, and is decreased by at least one for every execution of the loop body (when the exit condition is not satisfied) but never becomes negative.

Bertrand Meyer,
Object-oriented Software Construction
1988

A **loop variant** is a non-negative integer expression decreased by execution of the loop body.

a natural number

A **loop variant** is ~~a non-negative integer~~ expression decreased by execution of the loop body.

an ordinal

~~a natural number~~

A **loop variant** is ~~a non-negative integer~~ expression decreased by execution of the loop body.

~~an ordinal~~

~~a natural number~~ something

A **loop variant** is ~~a non-negative integer expression~~ decreased by execution of the loop body.

~~an ordinal~~

~~a natural number~~ something

A **loop variant** is ~~a non-negative integer expression~~ ~~decreased~~ by execution of the loop body.

consumed in a way that leads to its exhaustion.

A **loop variant** is something consumed by execution of the loop body in a way that leads to its exhaustion.

😈 Will this loop ever end?

Yes! Execution increases the entropy of the universe, turning free energy into useless thermal energy. 🙂

Each iteration consumes some free energy, and the free energy of the universe is eventually exhausted. 🙂

👿 👿

If the iterations of a loop consume some resource in a way that leads to its exhaustion, the loop must end.

```
static_assert( 12 == 0 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 );

int i = 0;

while ( i != 12 )
    ++i;
```

🤖 ✅

If the iterations of a loop consume some resource in a way that leads to its exhaustion, the loop must end.

```
static_assert( 12 == 0 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 );

int i = 0;

while ( i != 12 )
    ++i;
```

🤖 ✅

If the iterations of a loop repeat a sequence of events that happened before the loop, the loop must end.

```
if ( b <= e )
  {
   int i = b;

   while ( i != e )
      ++i;
  }
```

If the iterations of a loop repeat a sequence of events that happened before the loop, the loop must end.

```
if ( b <= e )
  {
   counting_theorem( b, e );

   int i = b;
   while ( i != e )
      ++i;
  }
```

```
void counting_theorem( const int b,
                       const int e )
interface
  {
   extend_stability b, e;

   claim b <= e;

   claim implementation;

   auto i = b;
   while ( i != e )
     {
      claim i < e;
      ++i;
     }
  }
```

```
if ( b <= e )
  {
   counting_theorem( b, e );

   int i = b;
   while ( i != e )
      ++i;
  }
```

```
void counting_theorem( const int b,
                       const int e )
interface
  {
   extend_stability b, e;

   claim b <= e;

   claim implementation;

   auto i = b;
   while ( i != e )
     {
      claim i < e;
      ++i;
     }
  }
```

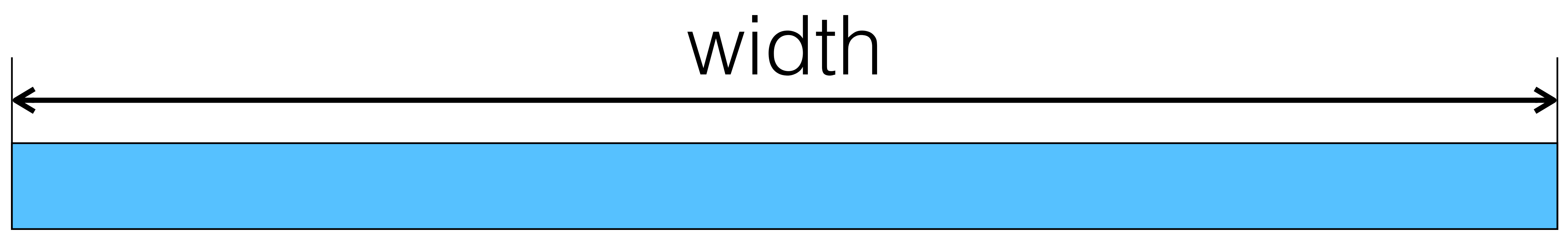
```
void counting_theorem( const int b,
                       const int e )
interface
  {
   extend_stability b, e;

   claim b <= e;

   claim implementation;

   auto i = b;
   while ( i != e )
     {
      claim i < e;
      ++i;
     }
  }
```

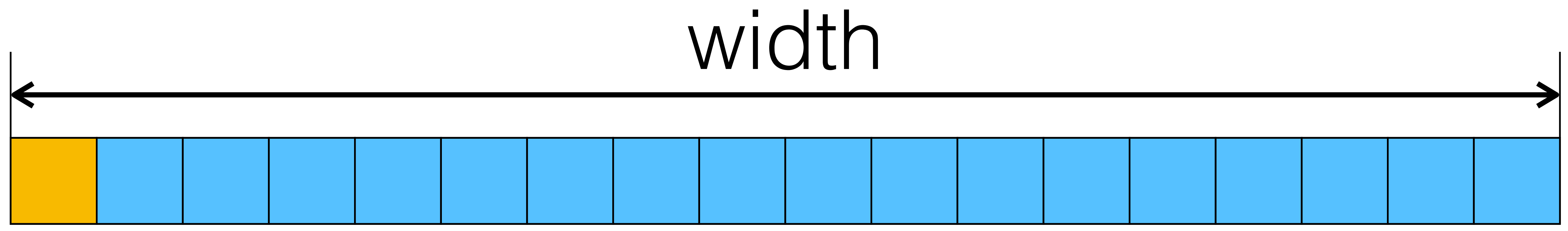
```
if ( b <= e )
  {
   counting_theorem( b, e );

   int i = b;
   while ( i != e )
      ++i;
  }
```

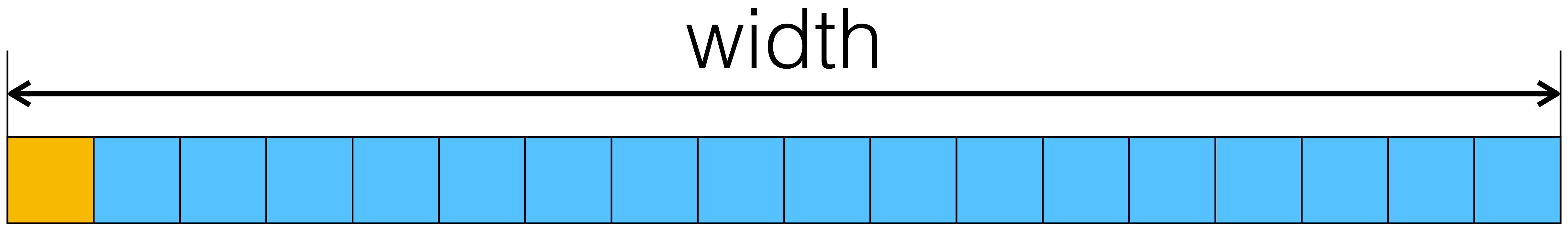
```
void counting_theorem( const int b,
                       const int e )
interface
  {
   extend_stability b, e;

   claim b <= e;

   claim implementation;

   auto i = b;
   while ( i != e )
     {
      claim i < e;
      ++i;
     }
  }
```

```
void counting_theorem( const int b,
                       const int e )
implementation
  {

        ?

  }
```

Familiar is not the same as trivial.

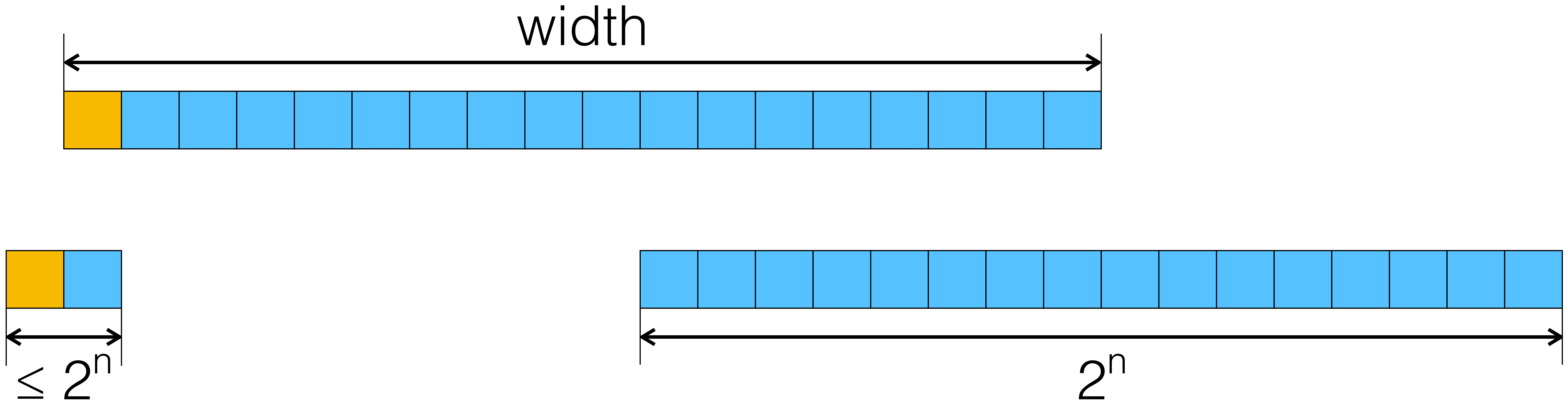

width

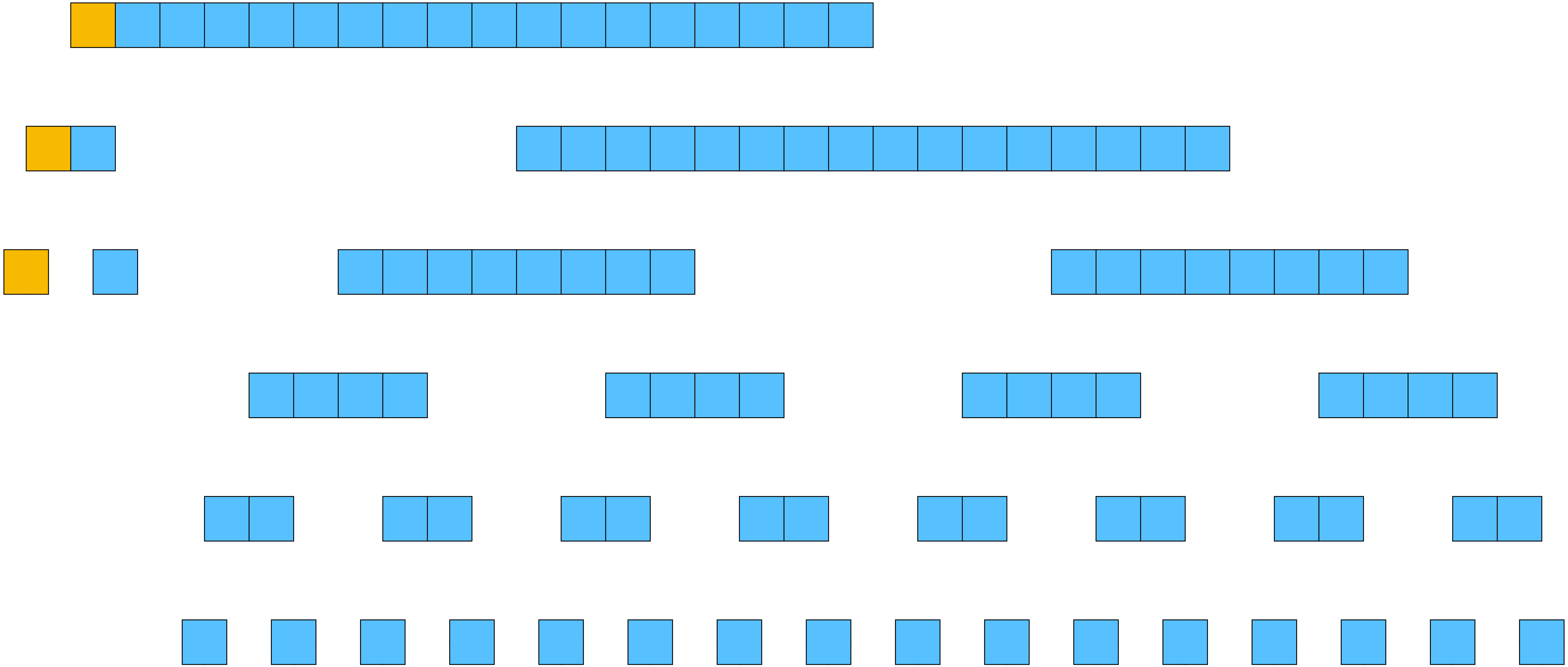

width

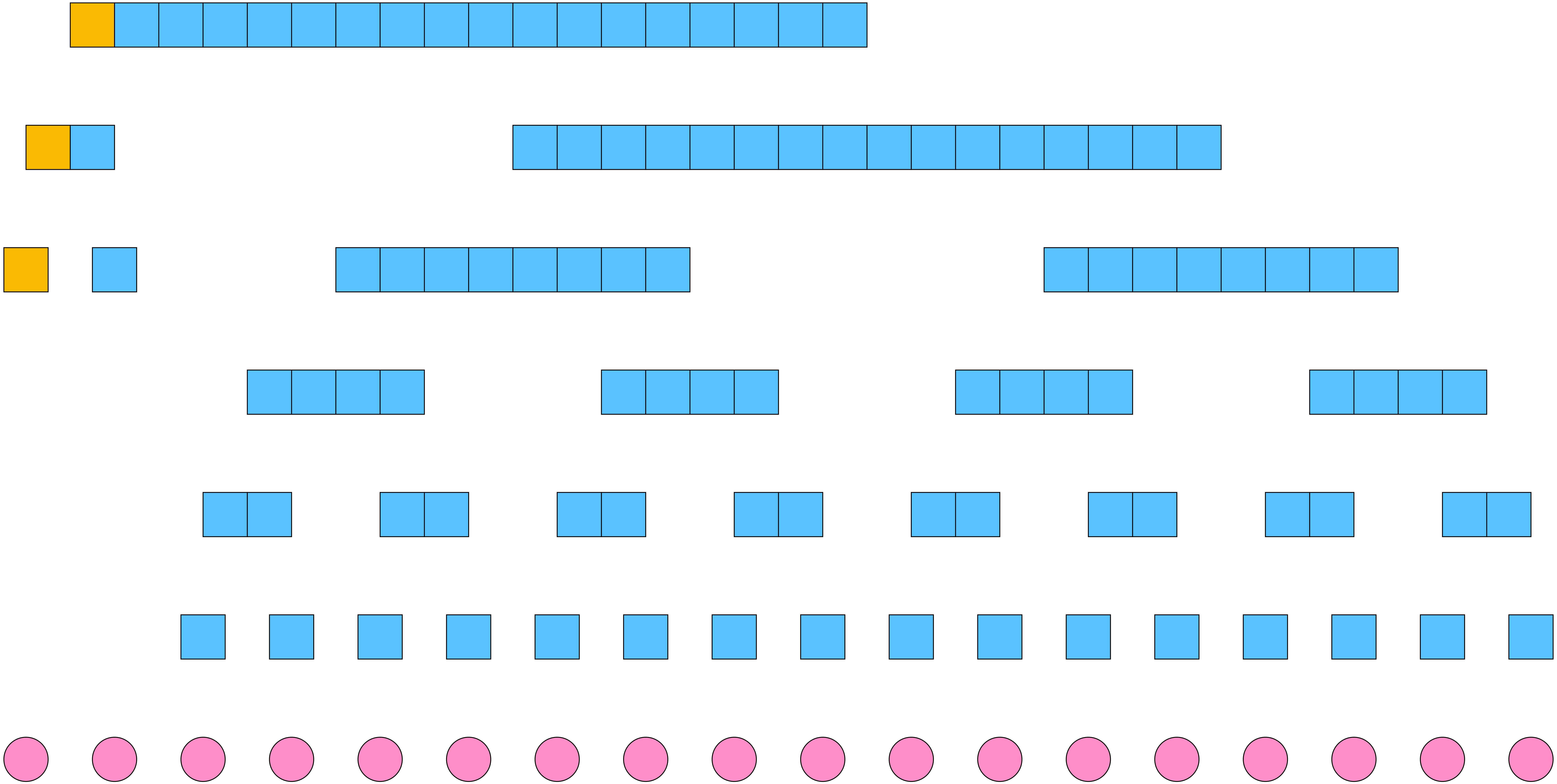


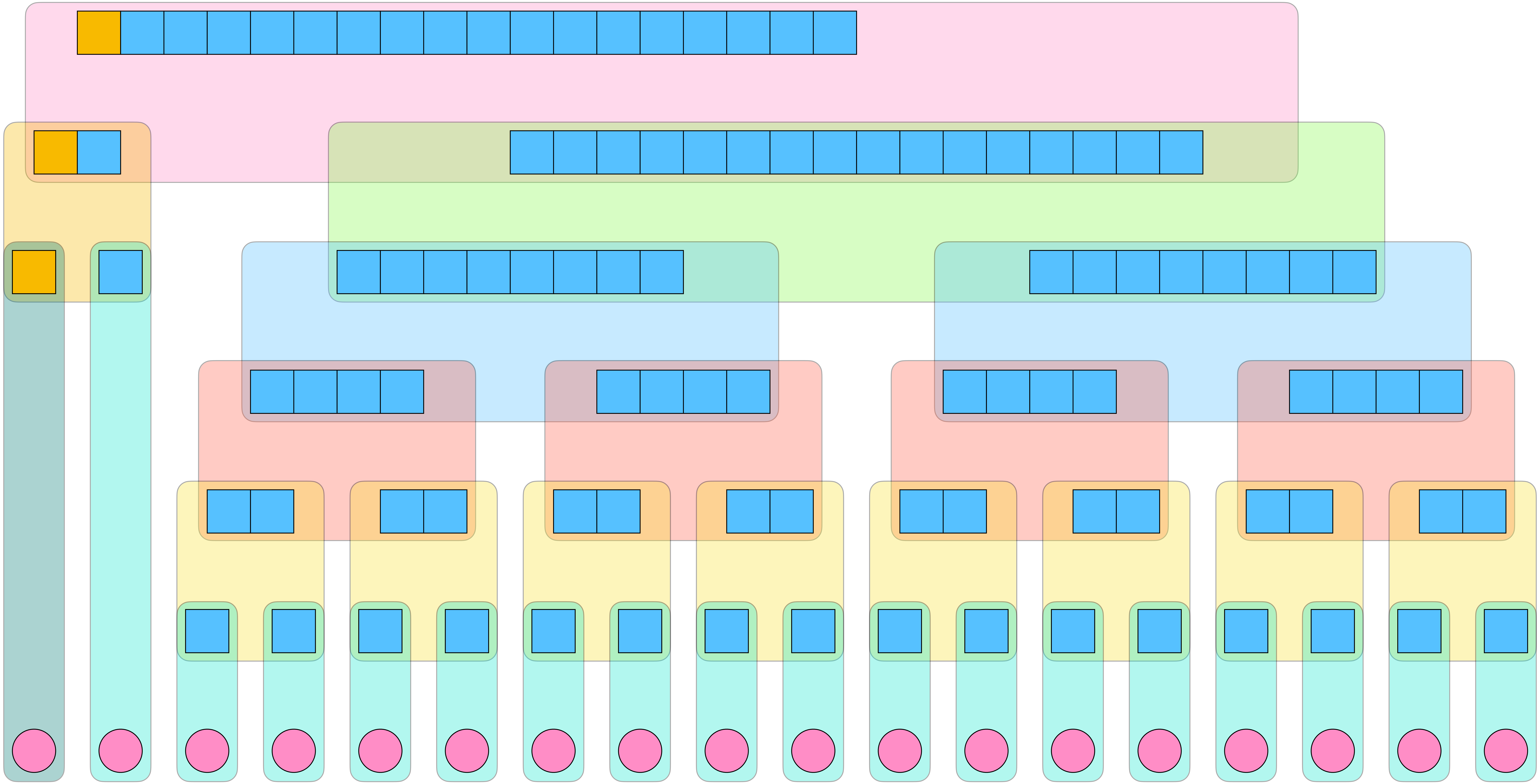

```
class integer_kind
  {
   // . . .
   constexpr bool          is_signed()  const;
   constexpr bit_size_t    width()      const;
  };
```

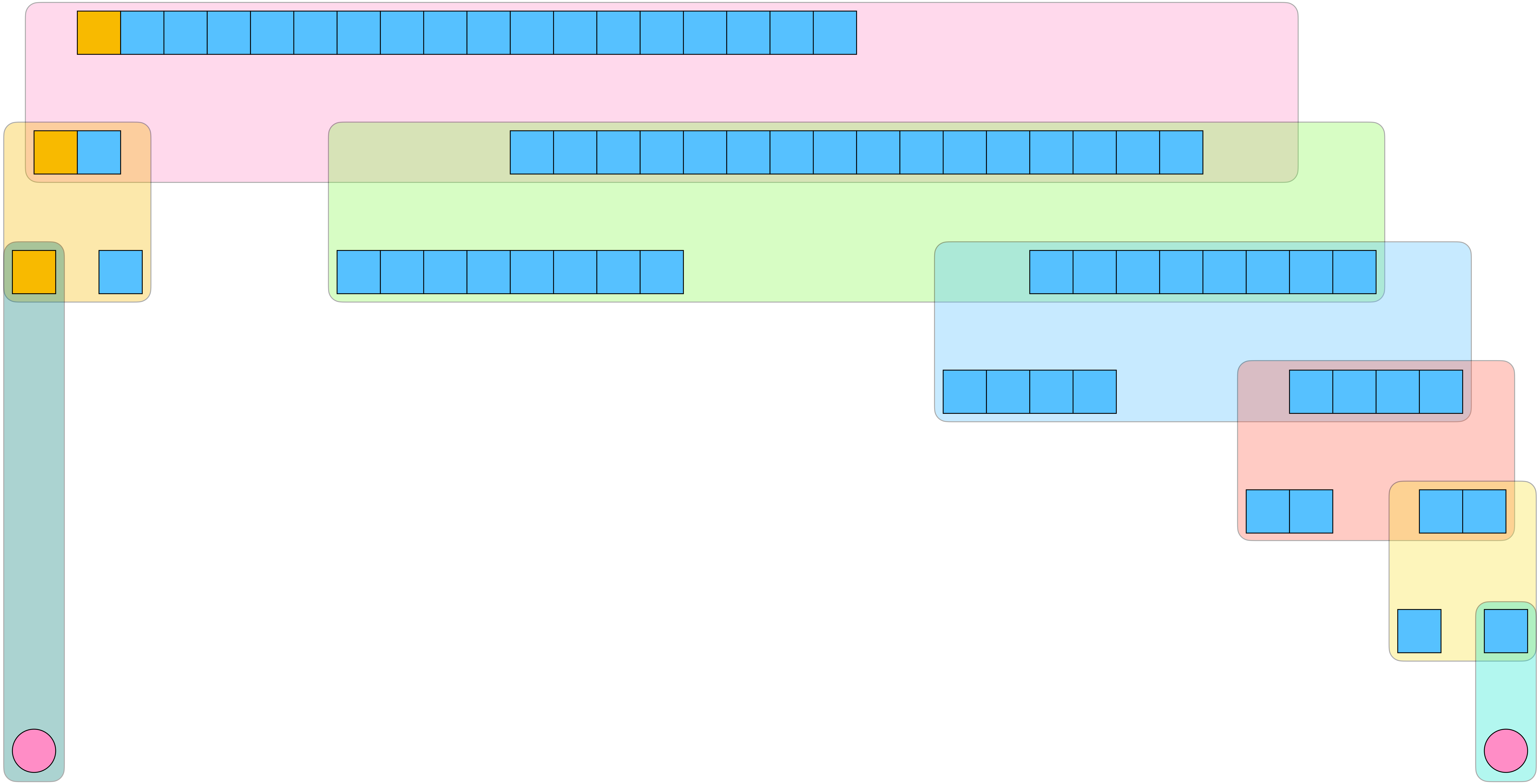
```
inline constexpr integer_kind        bitless_kind;   // no bits (unsigned)
inline constexpr integer_kind  unsigned_bit_kind;   // one bit, unsigned
inline constexpr integer_kind    signed_bit_kind;   // one bit, signed
```

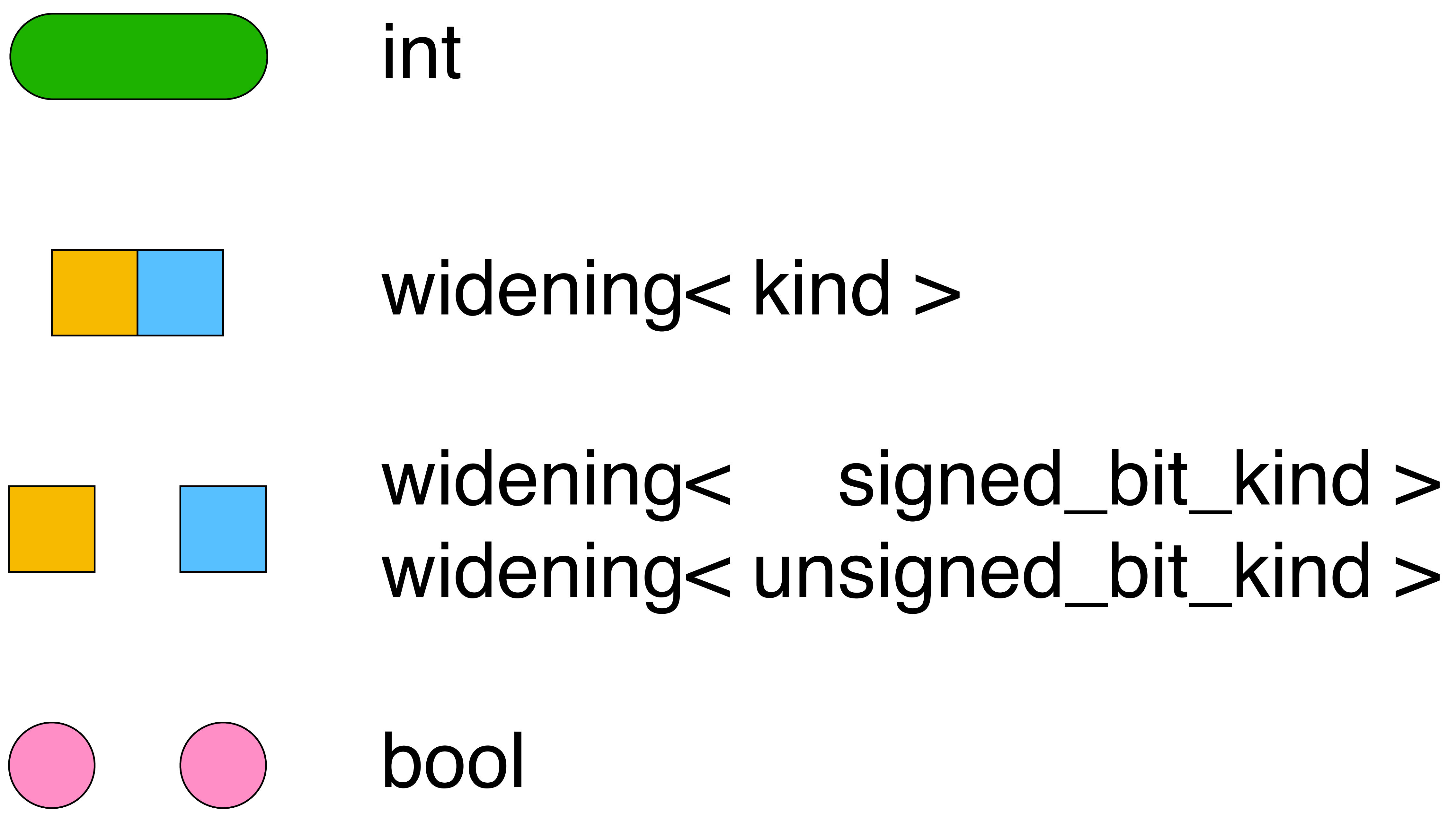

width
≤ 2^n
2^n



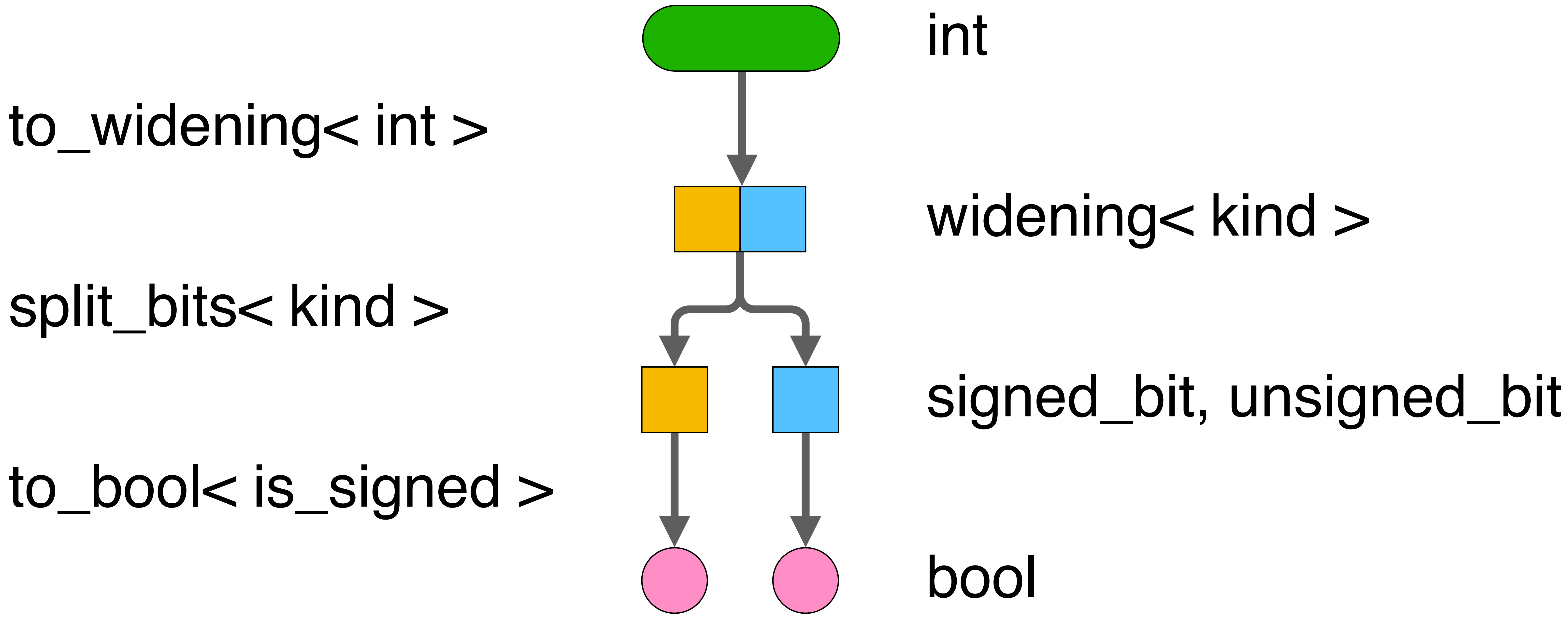




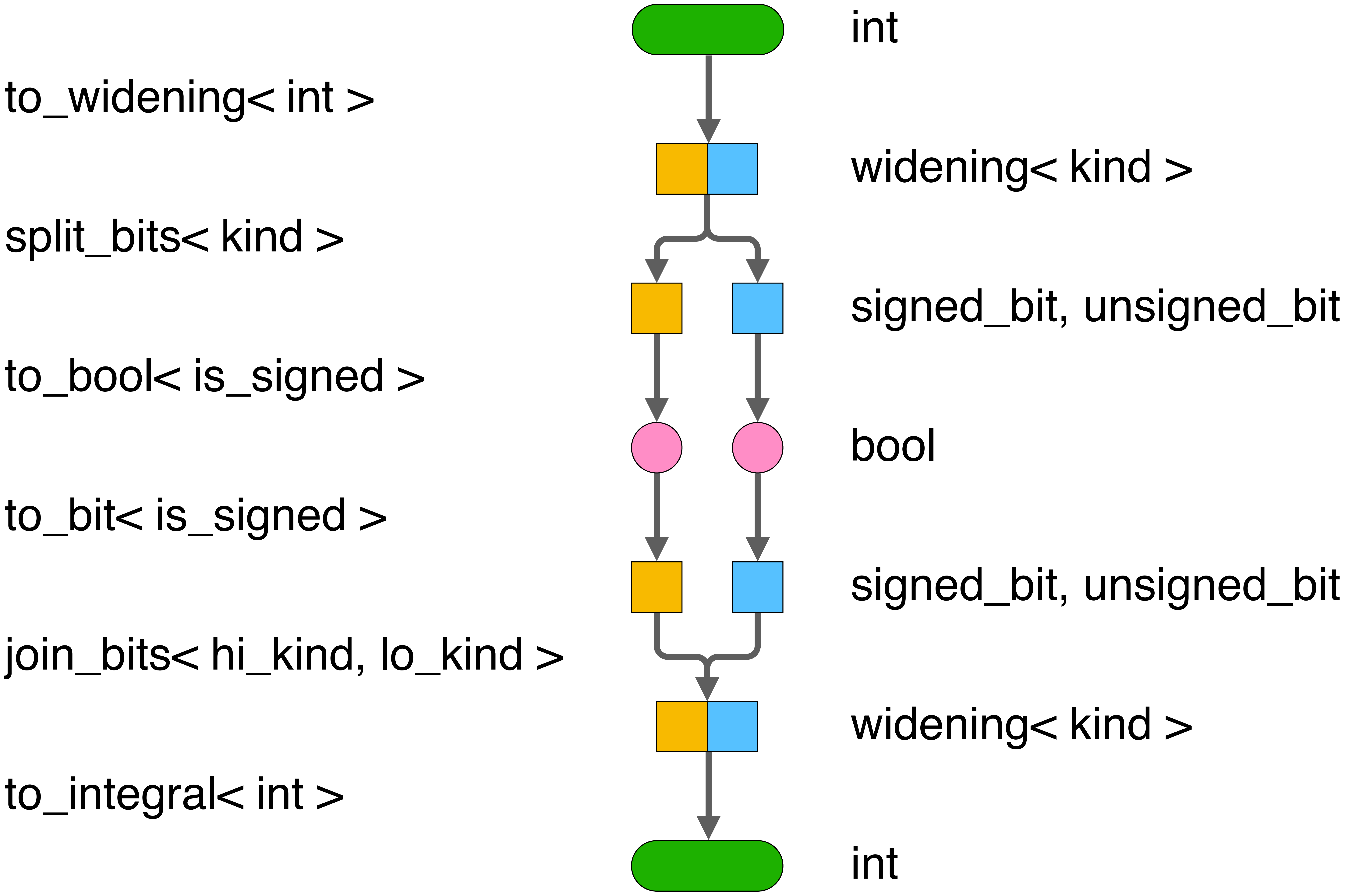


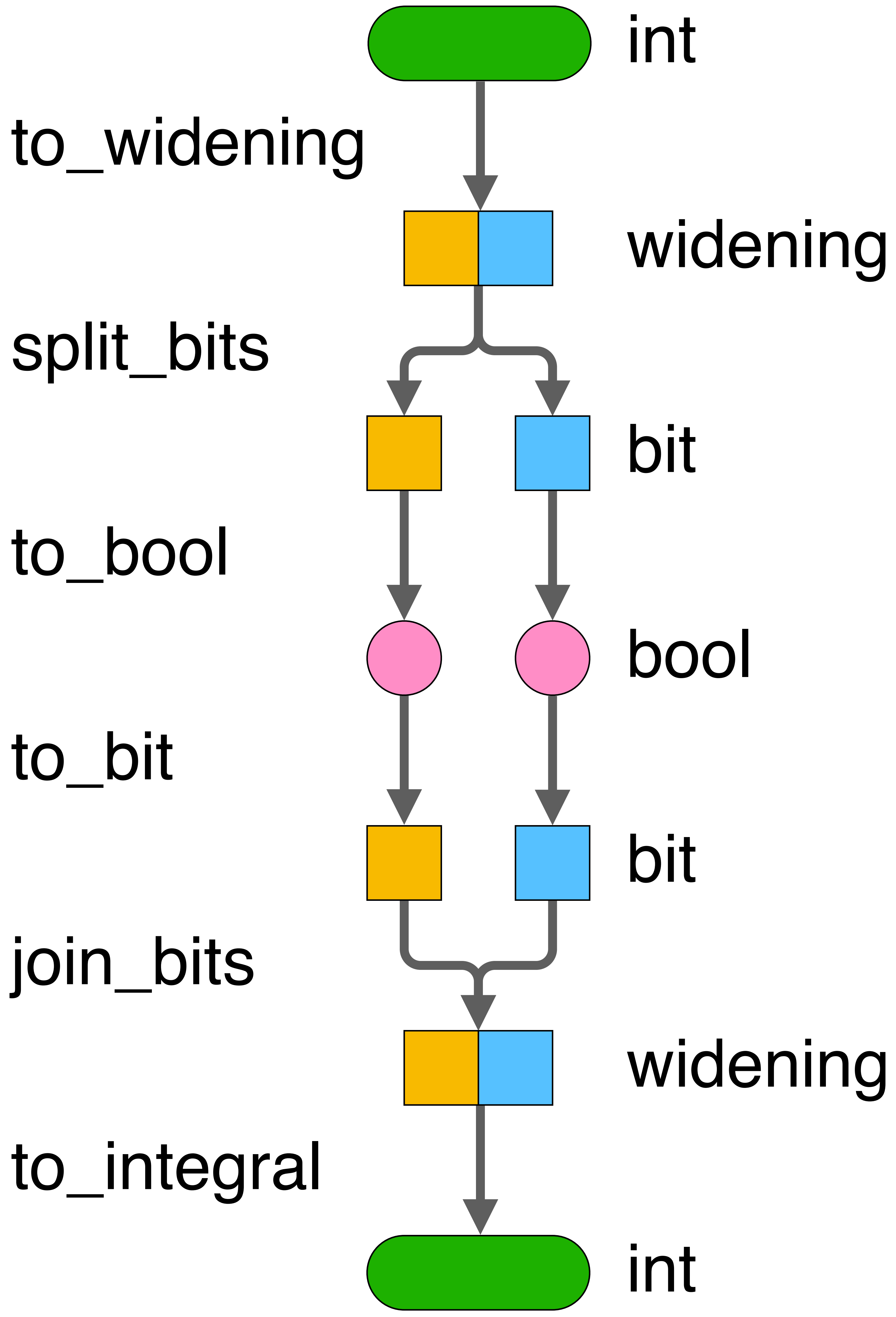

int
widening< kind >
widening< signed_bit_kind >
widening< unsigned_bit_kind >
bool


```
using          bitless = widening<       bitless_kind >;
using     unsigned_bit = widening< unsigned_bit_kind >;
using       signed_bit = widening<   signed_bit_kind >;
```




```
using          bitless = widening<      bitless_kind >;
using     unsigned_bit = widening< unsigned_bit_kind >;
using       signed_bit = widening<   signed_bit_kind >;
```


int
to_widening< int >
widening< kind >
split_bits< kind >
signed_bit, unsigned_bit
to_bool< is_signed >
bool
to_bit< is_signed >
signed_bit, unsigned_bit
join_bits< hi_kind, lo_kind >
widening< kind >
to_integral< int >
int



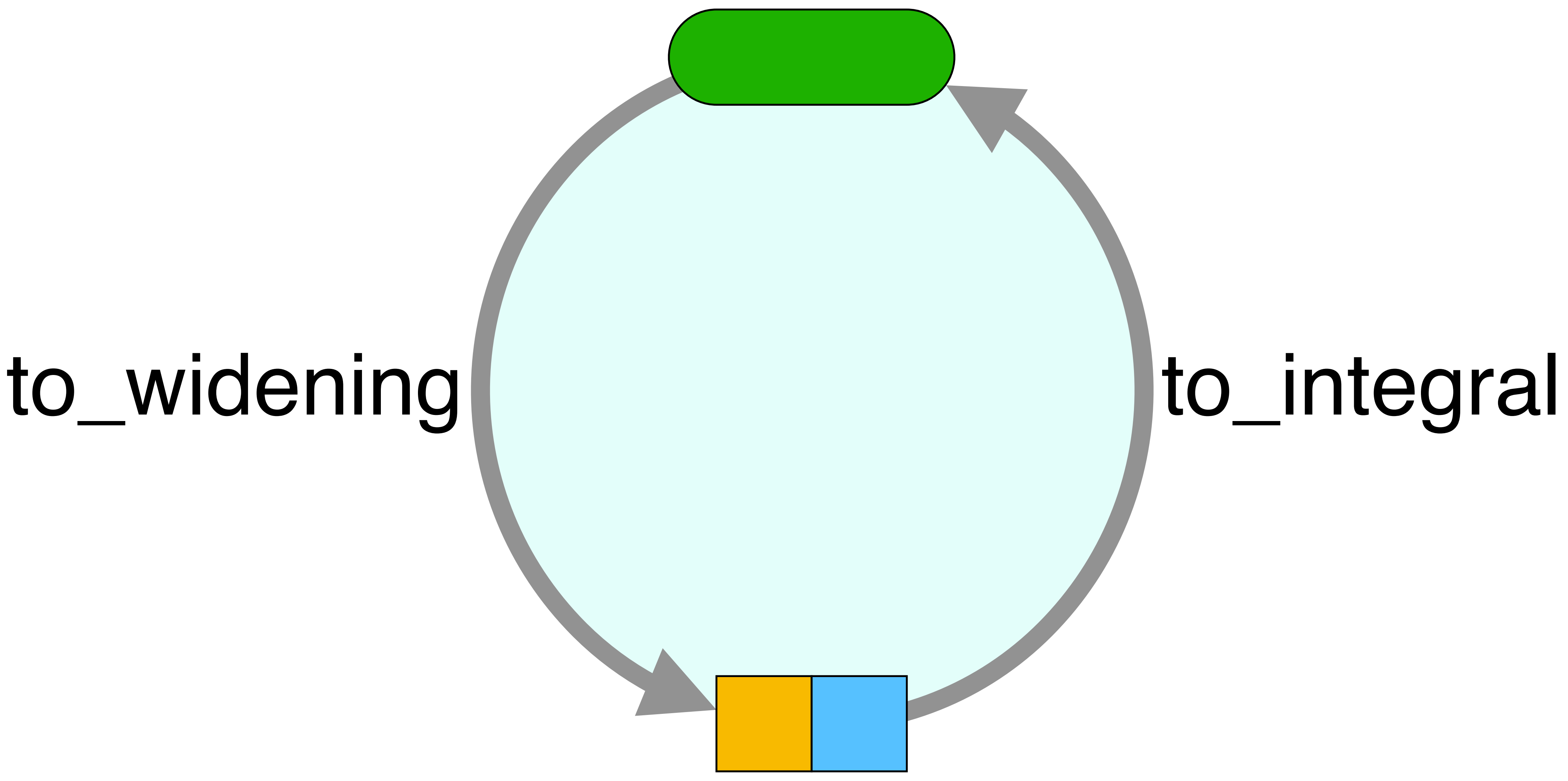
Principia Mathematica

The foundations of arithmetic in C++

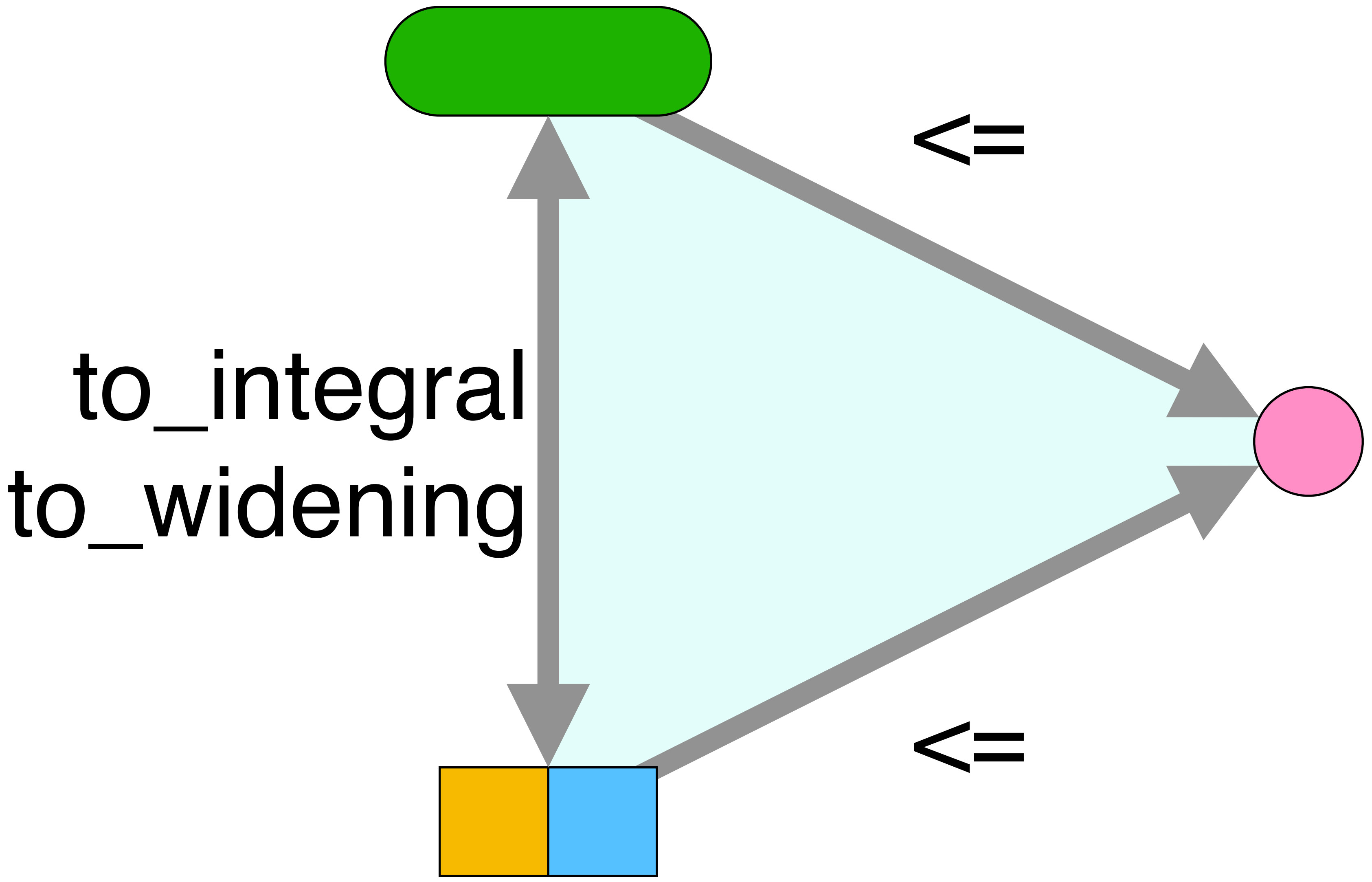
Lisa Lippincott

CppCon, September 2022

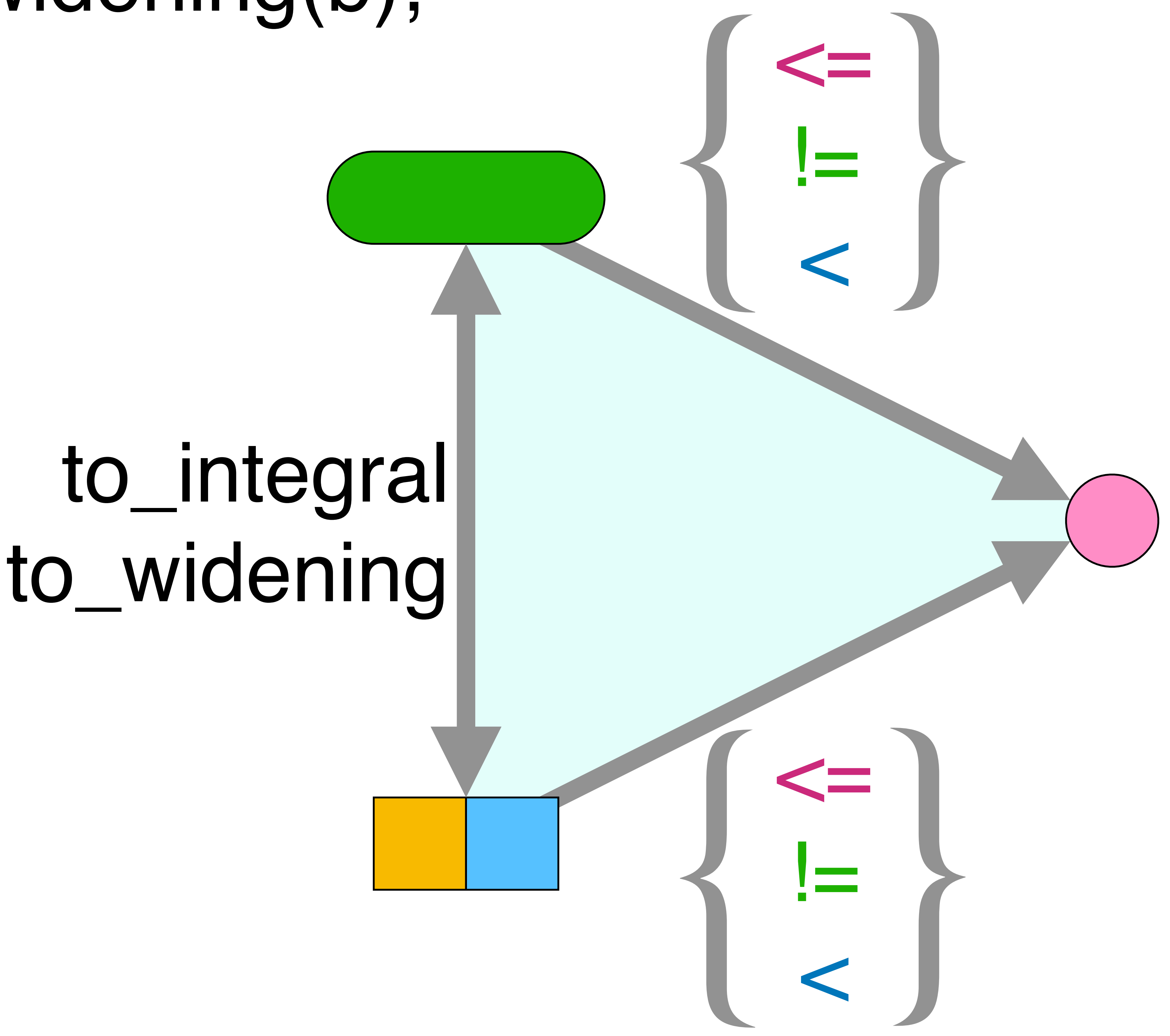
```
template < class T >
T to_integral( const widening< integer_kind_of<T> > a )
interface
  {
   primitive_interface = default;

   using_primitive to_widening;
   claim to_widening( result ) == a;
  }

widening< integer_kind_of<int> >
to_widening( const int a )
interface
  {
   primitive_interface = default;

   using_primitive to_integral;
   claim to_integral<int>( result ) == a;
  }
```



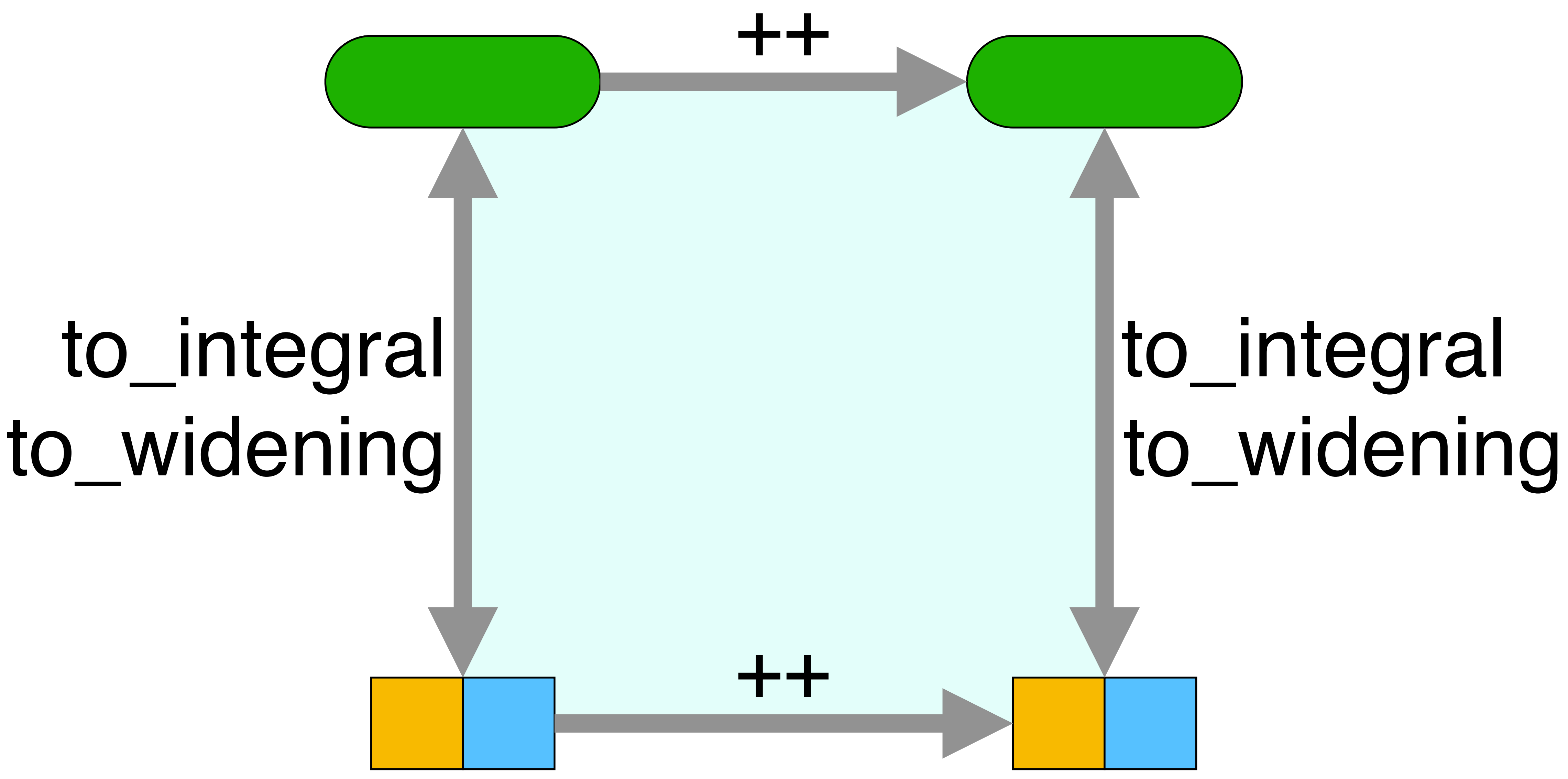
```
bool operator<=( const int a, const int b )

interface
  {
   const auto expected = to_widening(a) <= to_widening(b);

   primitive_interface = default;

   claim result == expected;

   // ...
  }
```



```
               <=
bool operator  !=  ( const int a, const int b )
               <
interface
  {
                                      <=
   const auto expected = to_widening(a)  !=  to_widening(b);
                                      <

   primitive_interface = default;

   claim result == expected;

   // ...
  }
```



```
int& operator++( int& b )
interface
  {
   auto wb = to_widening( b );
   ++wb;

   primitive_interface
     {
      // . . .
      implementation;
      // . . .
     }

   claim b == to_integral<int>( wb );
   // . . .
  }
```



```
void counting_theorem( const int b,
                       const int e )
interface
  {
   extend_stability b, e;

   claim b <= e;

   claim implementation;

   auto i = b;
   while ( i != e )
     {
      claim i < e;
      ++i;
     }
  }
```

```
void counting_theorem( const int b,
                       const int e )
implementation
  {

            ?

  }
```

```
void counting_theorem( const int b,
                       const int e )
interface
  {
   extend_stability b, e;

   claim b <= e;

   claim implementation;

   auto i = b;
   while ( i != e )
     {
      claim i < e;
      ++i;
     }
  }
```

```
void counting_theorem( const int b,
                       const int e )
implementation
  {
   counting_theorem( to_widening( b ),
                     to_widening( e ) );
  }
```

```
void counting_theorem( const int b,
                       const int e )
implementation
  {
   counting_theorem( to_widening( b ),
                     to_widening( e ) );
  }
```

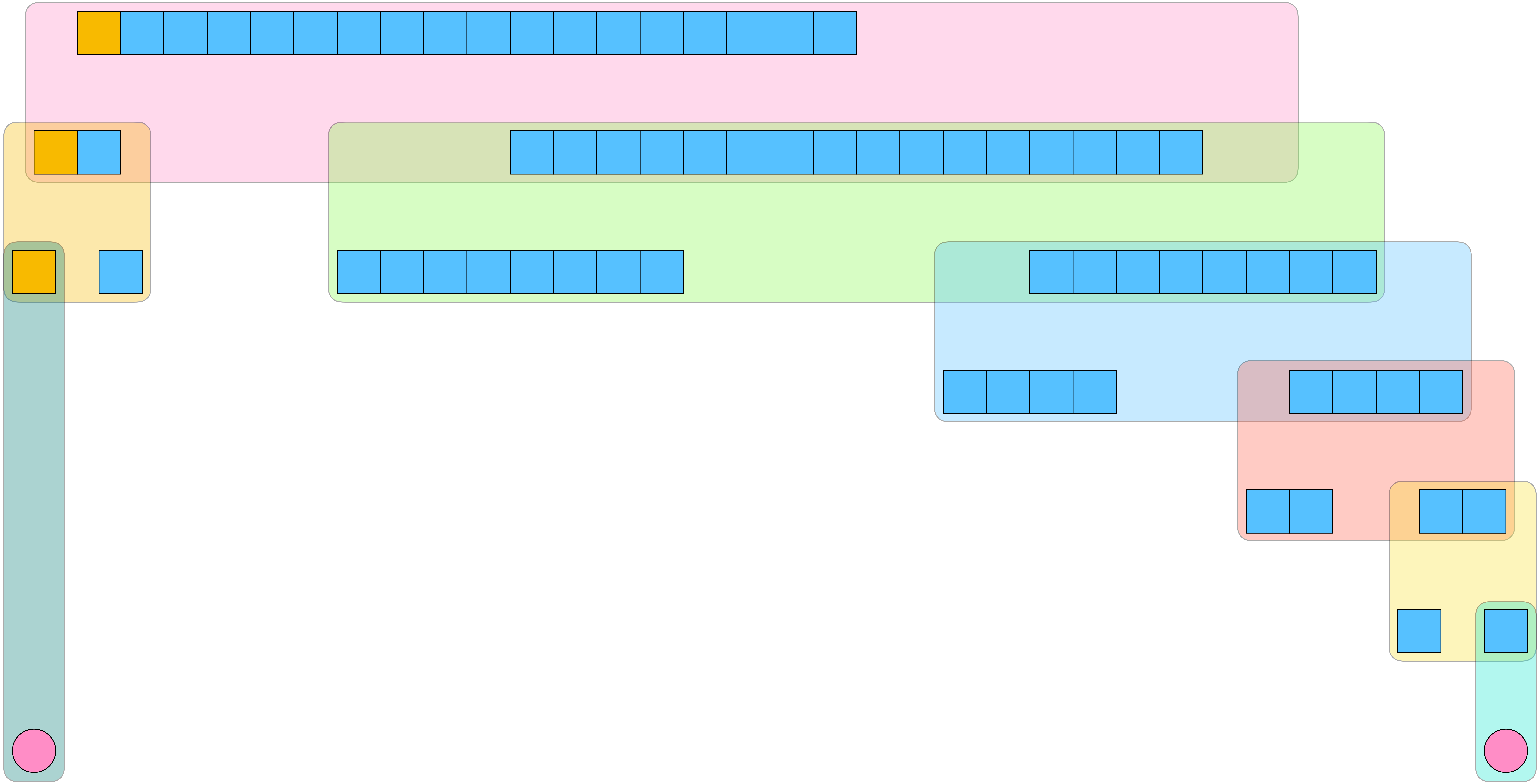
```
template < integer_kind k >
void counting_theorem( const widening<k>& ab,
                       const widening<k>& cd )
interface
  {
   extend_stability ab, cd;

   claim ab <= cd;

   claim implementation;

   auto xy = ab;
   while ( xy != cd )
     {
      claim xy < cd;
      ++xy;
     }
  }
```

```
void counting_theorem( const int b,
                       const int e )
interface
  {
   extend_stability b, e;

   claim b <= e;

   claim implementation;

   auto i = b;
   while ( i != e )
     {
      claim i < e;
      ++i;
     }
  }
```

```
template < integer_kind k >
void counting_theorem( const widening
                       const widening
interface
  {
   extend_stability ab, cd;

   claim ab <= cd;

   claim implementation;

   auto xy = ab;
   while ( xy != cd )
     {
      claim xy < cd;
      ++xy;
     }
  }
```

```
template < integer_kind k >
void counting_theorem( const widening<k>& ab,
                       const widening<k>& cd )
interface
  {
   extend_stability ab, cd;

   claim ab <= cd;

   claim implementation;

   auto xy = ab;
   while ( xy != cd )
     {
      claim xy < cd;
      ++xy;
     }
  }
```

```
template < integer_kind k >
void counting_theorem( const w
                       const w
implementation
  {

   ?

  }
```



```
template < integer_kind k >
void reference_less_or_equal_axiom( const widening<k> ab,
                                    const widening<k> cd )
interface
  {
   extend_stability ab, cd;

   const auto reference_result = reference_less_or_equal( ab, cd );

   posit implementation;

   using_primitive operator<=;

   claim ( ab <= cd ) == reference_result;
  }
```

```
template < integer_kind k >

     ⎧ reference_less_or_equal_axiom ⎫
void ⎨ reference_not_equal_axiom     ⎬ ( const widening<k>& ab,
     ⎩ reference_less_axiom          ⎭   const widening<k>& cd )

interface
  {
   extend_stability ab, cd;
                                     ⎧ reference_less_or_equal ⎫
   const auto reference_result =     ⎨ reference_not_equal     ⎬ ( ab, cd );
                                     ⎩ reference_less          ⎭
   posit implementation;
                              ⎧ <= ⎫
   using_primitive operator   ⎨ != ⎬ ;
                              ⎩ <  ⎭
                ⎧ <= ⎫
   claim ( ab   ⎨ != ⎬ cd ) == reference_result;
  }             ⎩ <  ⎭
```

```
template < integer_kind k >
inline bool reference_less_or_equal( const widening<k>& ab, const widening<k>& cd )
  {
   if        constexpr ( k == bitless_kind )

   else if constexpr ( k == unsigned_bit_kind )


   else if constexpr ( k == signed_bit_kind )


   else                              // multiple bits
     {



     }
  }
```

```
template < integer_kind k >
inline bool reference_less_or_equal( const widening<k>& ab, const widening<k>& cd )
  {
   if          constexpr ( k == bitless_kind )
      return                                                      true          ;
   else if constexpr ( k == unsigned_bit_kind )
      return to_bool(ab)  ? to_bool(cd) ?                         true   :   false
                          : to_bool(cd) ?                         true   :   true   ;
   else if constexpr ( k == signed_bit_kind )
      return to_bool(ab)  ? to_bool(cd) ?                         true   :   true
                          : to_bool(cd) ?                         false  :   true   ;
   else                     // multiple bits
     {




     }
  }
```

```
template < integer_kind k >
inline bool reference_less_or_equal( const widening<k>& ab, const widening<k>& cd )
  {
   if        constexpr ( k == bitless_kind )
      return                                                  true          ;
   else if constexpr ( k == unsigned_bit_kind )
      return to_bool(ab) ? to_bool(cd) ?                      true    :   false
                         : to_bool(cd) ?                      true    :   true    ;
   else if constexpr ( k == signed_bit_kind )
      return to_bool(ab) ? to_bool(cd) ?                      true    :   true
                         : to_bool(cd) ?                      false   :   true    ;
   else                  // multiple bits
     {
      const auto [ a, b ] = split_bits( ab );
      const auto [ c, d ] = split_bits( cd );

      return                                                  ( a != c ) ? ( a <= c )
                                                                         : ( b <= d ) ;
     }
  }
```

not_equal

false

false	:	true
true	:	false

false	:	true
true	:	false

```
( a != c ) ? ( a != c )
           : ( b != d )
```

less

false

false	:	false
true	:	false

false	:	true
false	:	false

```
( a != c ) ? ( a < c )
           : ( b < d )
```

less_or_equal

true

true	:	false
true	:	true

true	:	true
false	:	true

```
( a != c ) ? ( a <= c )
           : ( b <= d )
```

```
template < integer_kind k >
void reference_increment_axiom( const widening<k>& ab )
interface
  {
   extend_stability ab;

   auto reference = ab;
   reference_increment( reference );

   posit implementation;

   using_primitive operator++;

   auto actual = ab;
   ++ab;

   claim actual == reference;
  }
```

```
template < integer_kind k >
inline widening<k>& reference_increment( widening<k>& ab )
  {


   if         constexpr ( k == bitless_kind )

   else if constexpr ( k == unsigned_bit_kind )

   else if constexpr ( k == signed_bit_kind )

   else                              // multiple bits
     {




     }
  }
```

```
template < integer_kind k >
inline widening<k>& reference_increment( widening<k>& ab )
  {
   claim ab != ab.max_value;

   if         constexpr ( k == bitless_kind )

   else if constexpr ( k == unsigned_bit_kind )

   else if constexpr ( k == signed_bit_kind )

   else                              // multiple bits
     {


     }
  }
```

```
template < integer_kind k >
inline widening<k>& reference_increment( widening<k>& ab )
  {
   claim ab != ab.max_value;

   if         constexpr ( k == bitless_kind )
      return ab =                                std::unreachable()  ;
   else if constexpr ( k == unsigned_bit_kind )
      return ab =                                1_bit               ;
   else if constexpr ( k == signed_bit_kind )
      return ab =                                0_signed_bit        ;
   else                       // multiple bits
     {



     }
  }
```

```
template < integer_kind k >
inline widening<k>& reference_increment( widening<k>& ab )
  {
   claim ab != ab.max_value;

   if          constexpr ( k == bitless_kind )
      return ab =                                      std::unreachable()   ;
   else if constexpr ( k == unsigned_bit_kind )
      return ab =                                            1_bit          ;
   else if constexpr ( k == signed_bit_kind )
      return ab =                                        0_signed_bit       ;
   else                             // multiple bits
     {
      const auto [ a, b ] = split_bits( ab );

      return ab =                 ( b != b.max_value ) ? join_bits(    a,          ++b )
                                                       : join_bits( ++a, b.min_value ) ;
     }
  }
```

```
template < integer_kind k >
void counting_theorem( const widening<k>& ab,
                       const widening<k>& cd )
interface
  {
   extend_stability ab, cd;

   claim ab <= cd;

   claim implementation;

   auto xy = ab;
   while ( xy != cd )
     {
      claim xy < cd;
      ++xy;
     }
  }
```

```
template < integer_kind k >
void counting_theorem( const w
                       const w
implementation
  {

        ?

  }
```

```
template < integer_kind k >
void counting_theorem( const widening<k>& ab,
                       const widening<k>& cd )
implementation
  {
   if         constexpr ( k == bitless_kind )
     { /* ... */}

   else if constexpr ( k == unsigned_bit_kind )
     { /* ... */}

   else if constexpr ( k == signed_bit_kind )
     { /* ... */}

   else                          // multiple bits
     { /* ... */}
  }
```

```
if          constexpr ( k == bitless_kind )
  {


  }
```

not_equal

```
if        constexpr ( k == bitless_kind )
  {
   reference_not_equal_axiom( ab, cd );

  }
```

false		

false	:	true
true	:	false

false	:	true
true	:	false

```
( a != c )  ?  ( a != c )
            :  ( b != d )
```

```
if          constexpr ( k == bitless_kind )
   {
    reference_not_equal_axiom( ab, cd );

   }
```

not_equal

false

```
if         constexpr ( k == bitless_kind )
  {
   reference_not_equal_axiom( ab, cd );
   claim (ab != cd) == false;
  }
```

not_equal
false

```
if          constexpr ( k == bitless_kind )
  {
   reference_not_equal_axiom( ab, cd );
   claim (ab != cd) == false;
  }
```

```
void counting_theorem(

interface
  {
   extend_stability ab, cd;

   claim ab <= cd;

   claim implementation;

   auto xy = ab;
   while ( xy != cd )
     {
      claim xy < cd;
      ++xy;
     }
  }
```

```
else if constexpr ( k == unsigned_bit_kind )
  {



  }
```

```
else if constexpr ( k == unsigned_bit_kind )
  {
   if ( ab != cd )
     {




















     }
  }
```

```
else if constexpr ( k == unsigned_bit_kind )
  {
   if ( ab != cd )
     {
      reference_less_or_equal_axiom( ab, cd );




     }
  }
```

less_or_equal

true	:	false
true	:	true

```
else if constexpr ( k == unsigned_bit_kind )
  {
   if ( ab != cd )
     {
      reference_less_or_equal_axiom( ab, cd );



      claim ab == 0_bit  ||  cd == 1_bit;



     }
  }
```

less_or_equal

true	:	
true	:	true

```
else if constexpr ( k == unsigned_bit_kind )
  {
   if ( ab != cd )
     {
      reference_less_or_equal_axiom( ab, cd );
      reference_not_equal_axiom      ( ab, cd );



      claim ab == 0_bit  ||  cd == 1_bit;





     }
  }
```

less_or_equal

true	:	
true	:	true

not_equal

false	:	true
true	:	false

```
else if constexpr ( k == unsigned_bit_kind )
  {
   if ( ab != cd )
     {
      reference_less_or_equal_axiom( ab, cd );
      reference_not_equal_axiom      ( ab, cd );



      claim ab == 0_bit  ||  cd == 1_bit;







     }
  }
```

less_or_equal

true	:	
true	:	true

not_equal

false	:	
true	:	false

```
else if constexpr ( k == unsigned_bit_kind )
  {
   if ( ab != cd )
     {
      reference_less_or_equal_axiom( ab, cd );
      reference_not_equal_axiom       ( ab, cd );



      claim ab == 0_bit  &&  cd == 1_bit;





     }
  }
```

less_or_equal

:
true :

not_equal

:
true :

```
else if constexpr ( k == unsigned_bit_kind )
  {
   if ( ab != cd )
     {
      reference_less_or_equal_axiom( ab, cd );
      reference_not_equal_axiom      ( ab, cd );
      reference_less_axiom               ( ab, cd );


      claim ab == 0_bit  &&  cd == 1_bit;




     }
  }
```

less_or_equal

	:	
true	:	

not_equal

	:	
true	:	

less

false	:	false
true	:	false

```
else if constexpr ( k == unsigned_bit_kind )
  {
   if ( ab != cd )
     {
      reference_less_or_equal_axiom( ab, cd );
      reference_not_equal_axiom    ( ab, cd );
      reference_less_axiom         ( ab, cd );


      claim ab == 0_bit  &&  cd == 1_bit;
      claim ab < cd;




     }
  }
```

less_or_equal

	:
true	:

not_equal

	:
true	:

less

	:
true	:

```
else if constexpr ( k == unsigned_bit_kind )
  {
   if ( ab != cd )
     {
      reference_less_or_equal_axiom( ab, cd );
      reference_not_equal_axiom    ( ab, cd );
      reference_less_axiom         ( ab, cd );
      reference_increment_axiom    ( ab );

      claim ab == 0_bit  &&  cd == 1_bit;
      claim ab < cd;

      auto xy = ab;
      ++xy;

      claim (xy != cd) == false;
     }
  }
```

less_or_equal

	:
true	:

not_equal

	:
true	:

less

	:
true	:

increment

1_bit

```
else if constexpr ( k == unsigned_bit_kind )
  {
   if ( ab != cd )
     {
      reference_less_or_equal_axiom( ab, cd );
      reference_not_equal_axiom      ( ab, cd );
      reference_less_axiom               ( ab, cd );
      reference_increment_axiom      ( ab );

      claim ab == 0_bit  &&  cd == 1_bit;
      claim ab < cd;

      auto xy = ab;
      ++xy;

      claim (xy != cd) == false;
     }
  }
```

```
interface
  {
   extend_stability ab, cd;

   claim ab <= cd;

   claim implementation;

   auto xy = ab;
   while ( xy != cd )
     {
      claim xy < cd;
      ++xy;
     }
  }
```

```
else if constexpr ( k == unsigned_bit_kind )
  {
   if ( ab != cd )
     {
      reference_less_or_equal_axiom( ab, cd );
      reference_not_equal_axiom     ( ab, cd );
      reference_less_axiom          ( ab, cd );
      reference_increment_axiom     ( ab );

      claim ab == 0_bit  &&  cd == 1_bit;
      claim ab < cd;

      auto xy = ab;
      ++xy;

      claim (xy != cd) == false;
     }
  }
```

```
interface
  {
   extend_stability ab, cd;

   claim ab <= cd;

   claim implementation;

   auto xy = ab;
   while ( xy != cd )
     {
      claim xy < cd;
      ++xy;
     }
  }
```

```
else if constexpr ( k == signed_bit_kind )
  {
   if ( ab != cd )
     {
      reference_less_or_equal_axiom( ab, cd );
      reference_not_equal_axiom      ( ab, cd );
      reference_less_axiom               ( ab, cd );
      reference_increment_axiom      ( ab );

      claim ab == -1_signed_bit  &&  cd == 0_signed_bit;
      claim ab < cd;

      auto xy = ab;
      ++xy;

      claim (xy != cd) == false;
     }
  }
```

```
else if constexpr ( k == signed_bit_kind )
  {
   if ( ab != cd )
     {
      reference_less_or_equal_axiom( ab, cd );
      reference_not_equal_axiom    ( ab, cd );
      reference_less_axiom         ( ab, cd );
      reference_increment_axiom    ( ab );

      claim ab == -1_signed_bit  &&  cd == 0_signed_bit;
      claim ab < cd;

      auto xy = ab;
      ++xy;

      claim (xy != cd) == false;
     }
  }
```

less_or_equal

true	:	true
false	:	true

not_equal

false	:	true
true	:	false

less

true	:	true
false	:	true

increment

0_signed_bit

```
else if constexpr ( k == signed_bit_kind )
  {
   if ( ab != cd )
     {
      reference_less_or_equal_axiom( ab, cd );
      reference_not_equal_axiom    ( ab, cd );
      reference_less_axiom         ( ab, cd );
      reference_increment_axiom    ( ab );

      claim ab == -1_signed_bit  &&  cd == 0_signed_bit;
      claim ab < cd;

      auto xy = ab;
      ++xy;

      claim (xy != cd) == false;
     }
  }
```

less_or_equal

:	true
:	

not_equal

:	true
:	

less

:	true
:	

increment

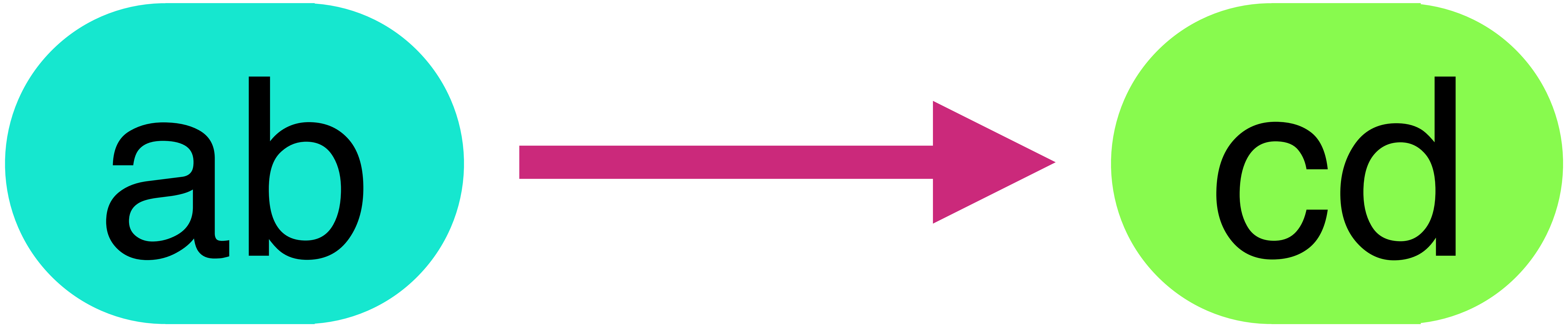
0_signed_bit

```
else if constexpr ( k == signed_bit_kind )
  {
   if ( ab != cd )
     {
      reference_less_or_equal_axiom( ab, cd );
      reference_not_equal_axiom    ( ab, cd );
      reference_less_axiom         ( ab, cd );
      reference_increment_axiom    ( ab );

      claim ab == -1_signed_bit  &&  cd == 0_signed_bit;
      claim ab < cd;

      auto xy = ab;
      ++xy;

      claim (xy != cd) == false;
     }
  }
```

```
interface
  {
   extend_stability ab, cd;

   claim ab <= cd;

   claim implementation;

   auto xy = ab;
   while ( xy != cd )
     {
      claim xy < cd;
      ++xy;
     }
  }
```

else
{

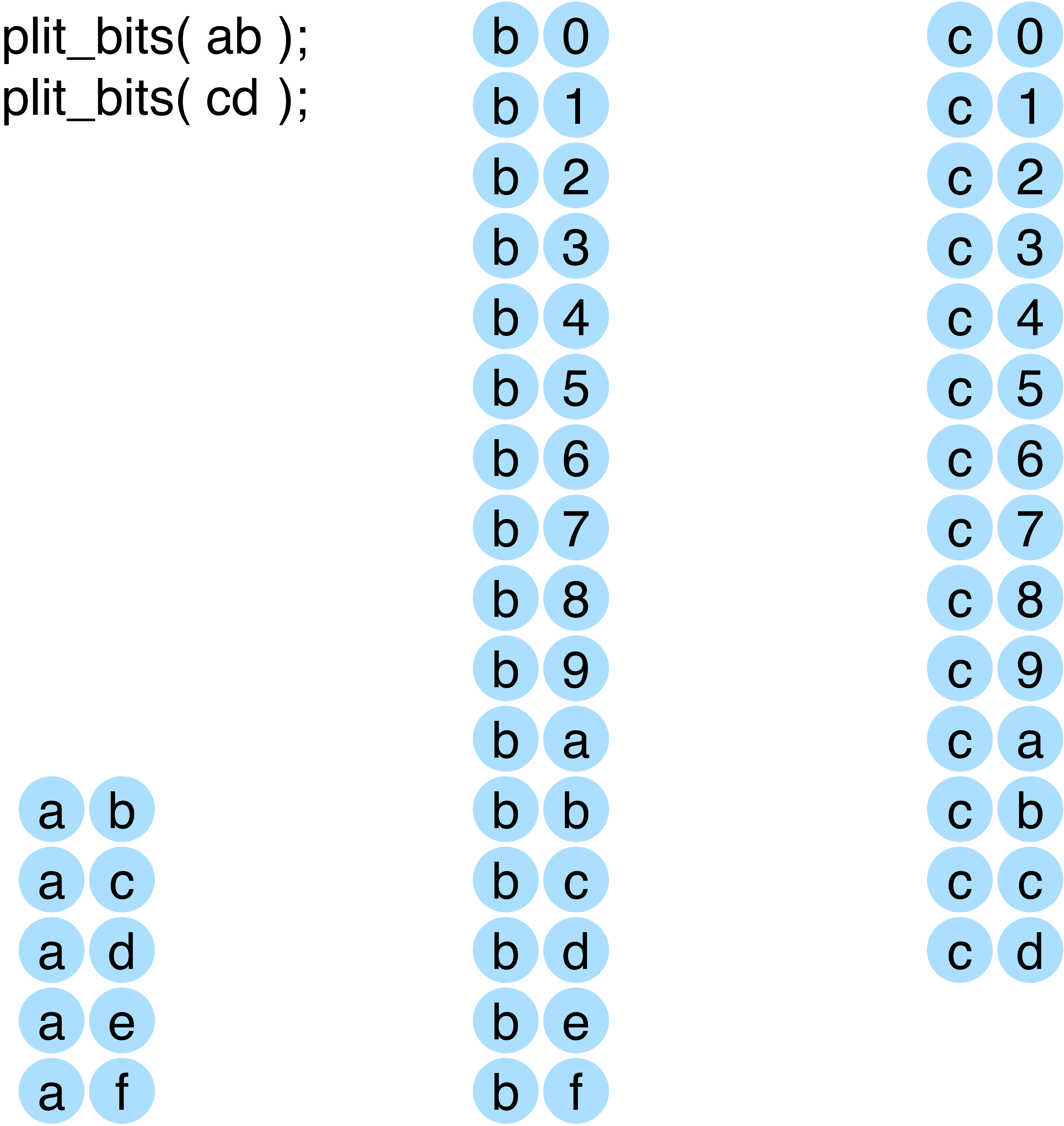
// multiple bits



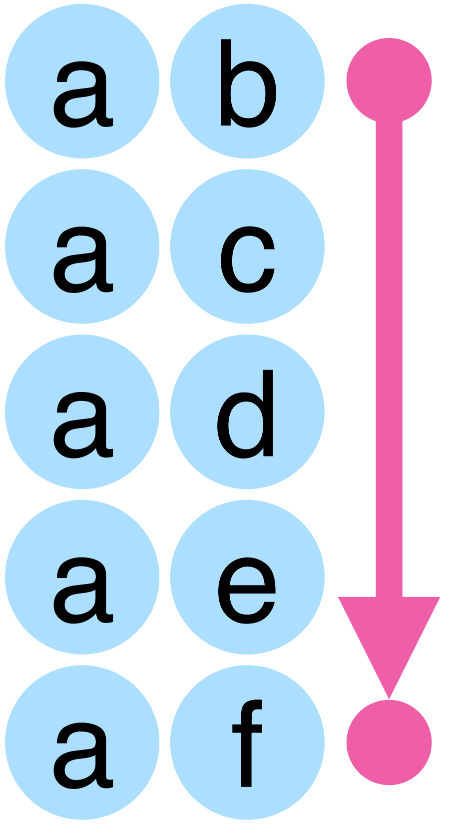
}

```
const auto [ a, b ] = split_bits( ab );
const auto [ c, d ] = split_bits( cd );

auto x = a;
auto y = b;
```

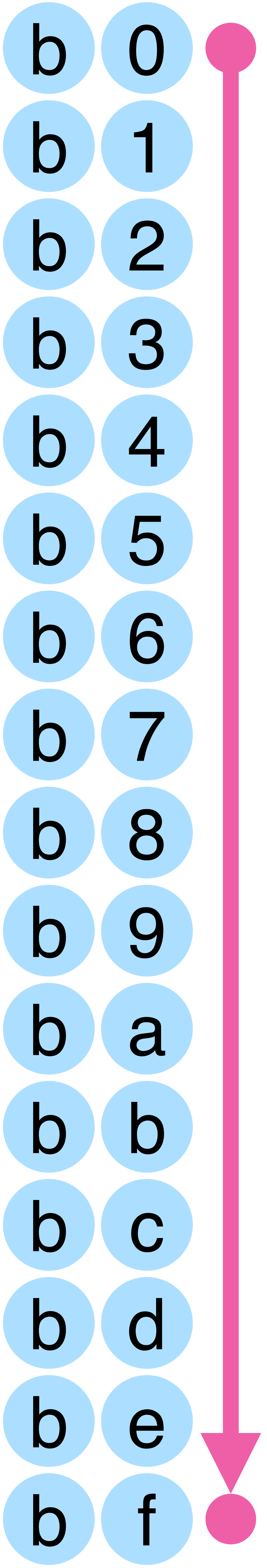
```
const auto [ a, b ] = split_bits( ab );
const auto [ c, d ] = split_bits( cd );

auto x = a;
auto y = b;
```



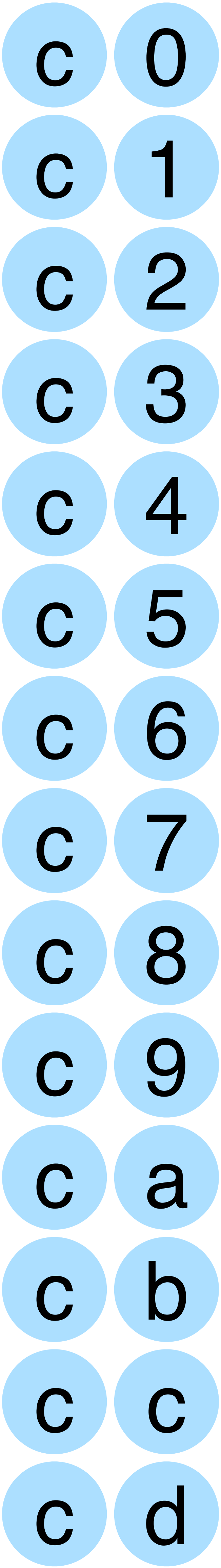
```
const auto [ a, b ] = split_bits( ab );
const auto [ c, d ] = split_bits( cd );

auto x = a;
auto y = b;
```



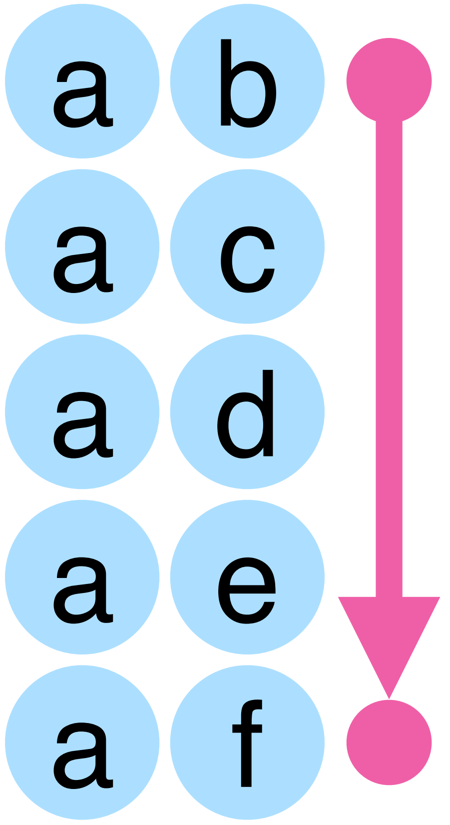

```
while ( y != y.max_value )
    ++y;
```



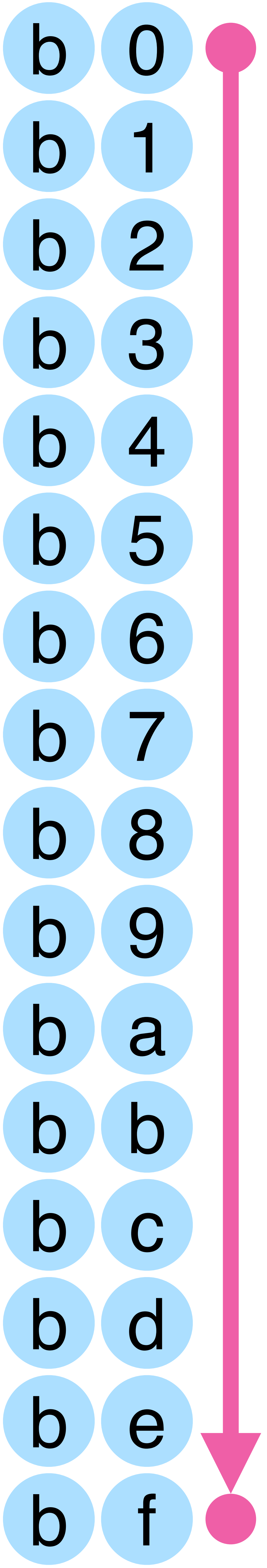

```
while ( y != y.max_value )
    ++y;
```



```
const auto [ a, b ] = split_bits( ab );
const auto [ c, d ] = split_bits( cd );

auto x = a;
auto y = b;
```

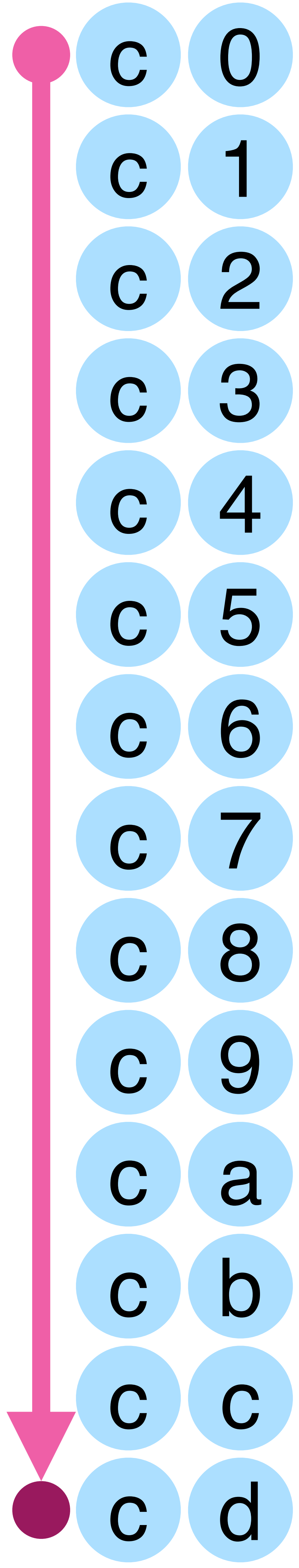


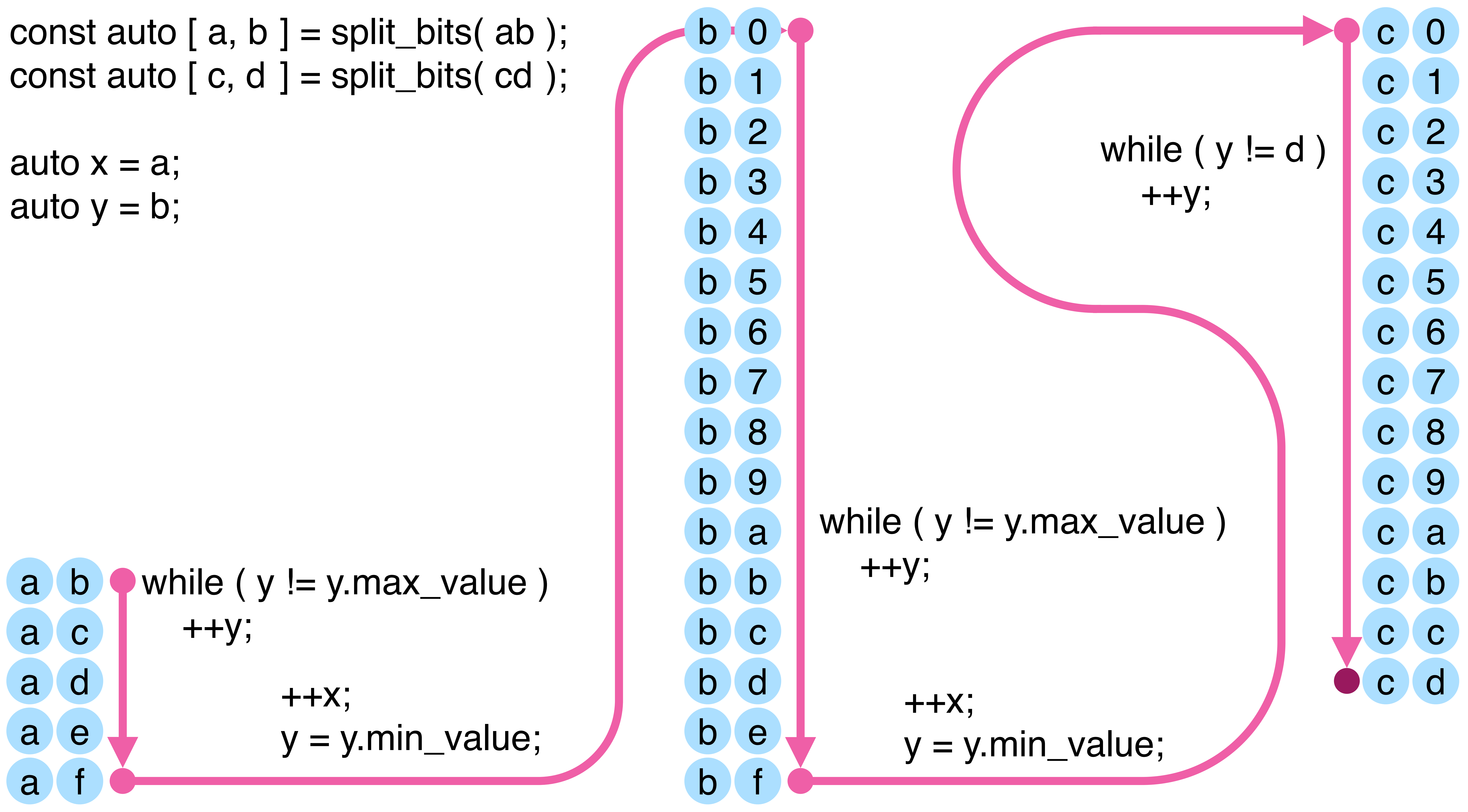

```
while ( y != y.max_value )
    ++y;
```




```
while ( y != y.max_value )
    ++y;
```

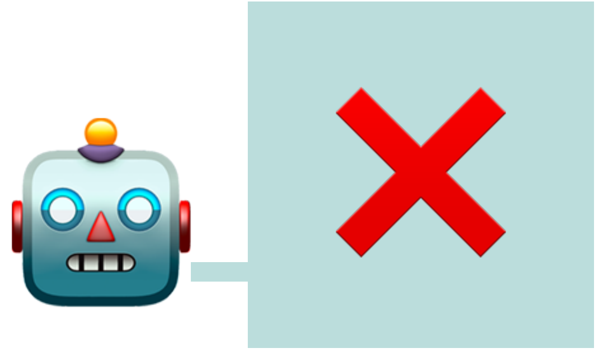
```
while ( y != d )
    ++y;
```

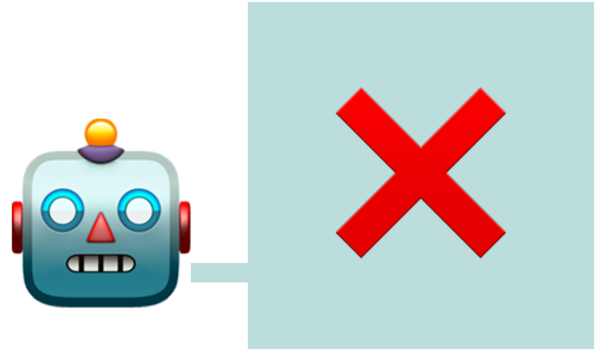


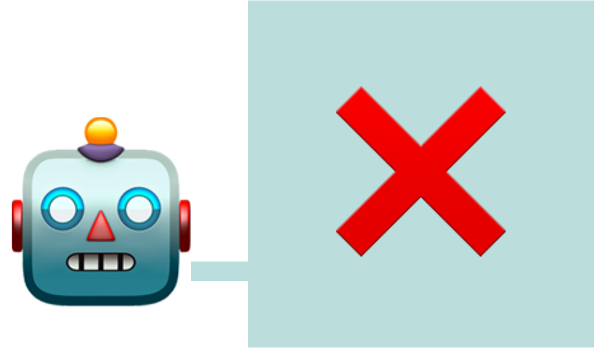

const auto [a, b] = split_bits(ab);
const auto [c, d] = split_bits(cd);
auto x = a;
auto y = b;
while (y != y.max_value)
++y;
++x;
y = y.min_value;
while (y != y.max_value)
++y;
++x;
y = y.min_value;
while (y != d)
++y;
a b
a c
a d
a e
a f
b 0
b 1
b 2
b 3
b 4
b 5
b 6
b 7
b 8
b 9
b a
b b
b c
b d
b e
b f
c 0
c 1
c 2
c 3
c 4
c 5
c 6
c 7
c 8
c 9
c a
c b
c c
c d

```
const auto [ a, b ] = split_bits( ab );
const auto [ c, d ] = split_bits( cd );

auto x = a;
auto y = b;
```

```
while ( x != c )
  {
    while ( y != y.max_value )
        ++y;

    ++x;
    y = y.min_value;
  }

while ( y != d )
    ++y;
```







```
while ( x != c )
  {
   while ( y != y.max_value )
       ++y;

   ++x;
   y = y.min_value;
  }

while ( y != d )
   ++y;
```

```
counting_theorem( x, c );
while ( x != c )
  {
   while ( y != y.max_value )
      ++y;

   ++x;
   y = y.min_value;
  }

while ( y != d )
   ++y;
```

```
template < integer_kind k
void counting_theorem(

interface
  {
   extend_stability ab, cd;

   claim ab <= cd;

   claim implementation;

   auto xy = ab;
   while ( xy != cd )
     {
      claim xy < cd;
      ++xy;
     }
  }
```

```
counting_theorem( x, c );
while ( x != c )
  {
   while ( y != y.max_value )
      ++y;

   ++x;
   y = y.min_value;
  }

while ( y != d )
   ++y;
```

```
template < integer_kind k
void counting_theorem(

interface
  {
   extend_stability ab, cd;

   claim ab <= cd;

   claim implementation;

   auto xy = ab;
   while ( xy != cd )
     {
      claim xy < cd;
      ++xy;
     }
  }
```

```
counting_theorem( x, c );
while ( x != c )
  {
   while ( y != y.max_value )
      ++y;

   ++x;
   y = y.min_value;
  }

while ( y != d )
   ++y;
```

```
template < integer_kind k
void counting_theorem(

interface
  {
   extend_stability ab, cd;

   claim ab <= cd;

   claim implementation;

   auto xy = ab;
   while ( xy != cd )
     {
      claim xy < cd;
      ++xy;
     }
  }
```

```
const auto [ a, b ] = split_bits( ab );
const auto [ c, d ] = split_bits( cd );

reference_less_or_equal_axiom( ab, cd );
claim ( a != c ) ? ( a <= c )
                 : ( b <= d );

auto x = a;
auto y = b;
```

```
counting_theorem( x, c );
while ( x != c )
  {
   while ( y != y.max_value )
      ++y;

   ++x;
   y = y.min_value;
  }

while ( y != d )
   ++y;
```

```
counting_theorem( x, c );
while ( x != c )
  {
   while ( y != y.max_value )
      ++y;

   ++x;
   y = y.min_value;
  }

while ( y != d )
   ++y;
```

```
template < integer_kind k
void counting_theorem(

interface
  {
   extend_stability ab, cd;

   claim ab <= cd;

   claim implementation;

   auto xy = ab;
   while ( xy != cd )
     {
      claim xy < cd;
      ++xy;
     }
  }
```

```
counting_theorem( x, c );
while ( x != c )
  {
   counting_theorem( y, y.max_value );
   while ( y != y.max_value )
      ++y;

   ++x;
   y = y.min_value;
  }

while ( y != d )
   ++y;
```

```
template < integer_kind k
void counting_theorem(

interface
  {
   extend_stability ab, cd;

   claim ab <= cd;

   claim implementation;

   auto xy = ab;
   while ( xy != cd )
     {
      claim xy < cd;
      ++xy;
     }
  }
```

```
counting_theorem( x, c );
while ( x != c )
  {
   counting_theorem( y, y.max_value );
   while ( y != y.max_value )
      ++y;

   ++x;
   y = y.min_value;
  }

while ( y != d )
   ++y;
```

🤖 operator<= tells me
y <= y.max_value.

🤖 ✅

```
template < integer_kind k
void counting_theorem(

interface
  {
   extend_stability ab, cd;

   claim ab <= cd;

   claim implementation;

   auto xy = ab;
   while ( xy != cd )
     {
      claim xy < cd;
      ++xy;
     }
  }
```

```
counting_theorem( x, c );
while ( x != c )
  {
   counting_theorem( y, y.max_value );
   while ( y != y.max_value )
      ++y;

   ++x;
   y = y.min_value;
  }

counting_theorem( y, d );
while ( y != d )
   ++y;
```

```
template < integer_kind k
void counting_theorem(

interface
  {
   extend_stability ab, cd;

   claim ab <= cd;

   claim implementation;

   auto xy = ab;
   while ( xy != cd )
     {
      claim xy < cd;
      ++xy;
     }
  }
```

```
counting_theorem( x, c );
while ( x != c )
  {
   counting_theorem( y, y.max_value );
   while ( y != y.max_value )
      ++y;

   ++x;
   y = y.min_value;
  }

counting_theorem( y, d );
while ( y != d )
   ++y;
```

If a == c, y is b and b <= d.
Otherwise, y is d.min_value.

```
template < integer_kind k
void counting_theorem(

interface
  {
   extend_stability ab, cd;

   claim ab <= cd;

   claim implementation;

   auto xy = ab;
   while ( xy != cd )
     {
      claim xy < cd;
      ++xy;
     }
  }
```

✅

```
const auto [ a, b ] = split_bits( ab );
const auto [ c, d  ] = split_bits( cd );

reference_less_or_equal_axiom( ab, cd );
claim ( a != c ) ? ( a <= c )
                 : ( b <= d );

auto x = a;
auto y = b;
```

```
counting_theorem( x, c );
while ( x != c )
  {
   counting_theorem( y, y.max_value );
   while ( y != y.max_value )
      ++y;

   ++x;
   y = y.min_value;
  }

counting_theorem( y, d );
while ( y != d )
   ++y;
```

```
counting_theorem( x, c );
while ( x != c )
  {
   counting_theorem( y, y.max_value );
   while ( y != y.max_value )
       ++y;

   ++x;
   y = y.min_value;
  }

counting_theorem( y, d );
while ( y != d )
   ++y;
```

```
{
 extend_stability ab, cd;

 claim ab <= cd;

 claim implementation;

 auto xy = ab;
 while ( xy != cd )
   {
    claim xy < cd;
    ++xy;
   }
}
```

```
    ++y;

++x;
y = y.min_value;



++y;
```

```
counting_theorem( x, c );
while ( x != c )
  {
    counting_theorem( y, y.max_value );
    while ( y != y.max_value )
        ++y;

    ++x;
    y = y.min_value;
  }

counting_theorem( y, d );
while ( y != d )
    ++y;
```

```
const auto advance_y =
  [&]() -> void
  {
   ++y;
  };

const auto advance_x =
  [&]() -> void
  {
   ++x;
   y = y.min_value;
  };
```

```
counting_theorem( x, c );
while ( x != c )
  {
    counting_theorem( y, y.max_value );
    while ( y != y.max_value )
        ++y;

    ++x;
    y = y.min_value;
  }

counting_theorem( y, d );
while ( y != d )
    ++y;
```

```
const auto advance_y =
  [&]() -> void
  {
   ++y;
  };

const auto advance_x =
  [&]() -> void
  {
   ++x;
   y = y.min_value;
  };
```

```
counting_theorem( x, c );
while ( x != c )
  {
   counting_theorem( y, y.max_value );
   while ( y != y.max_value )
      advance_y();


   advance_x();
  }

counting_theorem( y, d );
while ( y != d )
   advance_y();
```

```
const auto advance_y =
  [&]() -> void
  {
   claim y != y.max_value;
   ++y;
  };

const auto advance_x =
  [&]() -> void
  {
   claim y == y.max_value;
   ++x;
   y = y.min_value;
  };
```

```
counting_theorem( x, c );
while ( x != c )
  {
   counting_theorem( y, y.max_value );
   while ( y != y.max_value )
      advance_y();


   advance_x();
  }

counting_theorem( y, d );
while ( y != d )
   advance_y();
```

y != y.max_value

y != d

```
counting_theorem( x, c );
while ( x != c )
  {
   counting_theorem( y, y.max_value );
   while ( y != y.max_value )
      advance_y();


   advance_x();
  }

counting_theorem( y, d );
while ( y != d )
   advance_y();
```

```
const auto y_is_not_max =
  [&]() -> bool
  {
   return y != y.max_value;
  };

const auto y_is_not_d =
  [&]() -> bool
  {
   return y != d;
  };
```

```
counting_theorem( x, c );
while ( x != c )
  {
   counting_theorem( y, y.max_value );
   while ( y_is_not_max() )
      advance_y();


   advance_x();
  }

counting_theorem( y, d );
while ( y_is_not_d() )
   advance_y();
```

```
const auto y_is_not_max =
  [&]() -> bool
  {
   claim x != c;
   return y != y.max_value;
  };

const auto y_is_not_d =
  [&]() -> bool
  {
   claim x == c;
   return y != d;
  };
```

```
counting_theorem( x, c );
while ( x != c )
  {
   counting_theorem( y, y.max_value );
   while ( y_is_not_max() )
      advance_y();


   advance_x();
  }

counting_theorem( y, d );
while ( y_is_not_d() )
   advance_y();
```

```
const auto [ a, b ] = split_bits( ab );
const auto [ c, d ] = split_bits( cd );

reference_less_or_equal_axiom( ab, cd );
claim ( a != c ) ? ( a <= c )
                 : ( b <= d );

auto x = a;
auto y = b;

auto xy = ab;
claim xy == join_bits( x, y );
```

```
const auto advance_y =
  [&]() -> void
  {
   claim y != y.max_value;
   ++y;
  };

const auto advance_x =
  [&]() -> void
  {
   claim y == y.max_value;
   ++x;
   y = y.min_value;
  };
```

```
const auto advance_y =
  [&]() -> void
  {
   claim y != y.max_value;

   ++y;


   reference_increment_axiom( xy );
   ++xy;

   claim xy == join_bits( x, y );
  };
```

```
const auto advance_x =
  [&]() -> void
  {
   claim y == y.max_value;

   ++x;
   y = y.min_value;

   reference_increment_axiom( xy );
   ++xy;

   claim xy == join_bits( x, y );
  };
```

```
const auto y_is_not_max =
  [&]() -> bool
  {
   claim x != c;

   reference_not_equal_axiom( xy, cd );
   claim xy != cd;

   return y != y.max_value;
  };
```

```
const auto y_is_not_d =
  [&]() -> bool
  {
   claim x == c;

   reference_not_equal_axiom( xy, cd );
   claim (y != d) == (xy != cd);

   return y != d;
  };
```

```
counting_theorem( x, c );
while ( x != c )
  {
   counting_theorem( y, y.max_value );
   while ( y_is_not_max() )
      advance_y();

   advance_x();
  }

counting_theorem( y, d );
while ( y_is_not_d() )
   advance_y();
```

```
{
 extend_stability ab, cd;

 claim ab <= cd;

 claim implementation;

 auto xy = ab;
 while ( xy != cd )
   {
    claim xy < cd;
    ++xy;
   }
}
```

```
counting_theorem( x, c );
while ( x != c )
  {
   counting_theorem( y, y.max_value );
   while ( y_is_not_max() )
     {
      claim x < c;
      advance_y();
     }
   claim x < c;
   advance_x();
  }

counting_theorem( y, d );
while ( y_is_not_d() )
  {
   claim y < d;
   advance_y();
  }
```

```
{
 extend_stability ab, cd;

 claim ab <= cd;

 claim implementation;

 auto xy = ab;
 while ( xy != cd )
   {
    claim xy < cd;
    ++xy;
   }
}
```

```
const auto x_is_below_c =
  [&]() -> bool
  {
   return x < c;
  };


const auto y_is_below_d =
  [&]() -> bool
  {
   return y < d;
  };
```

```
counting_theorem( x, c );
while ( x != c )
  {
   counting_theorem( y, y.max_value );
   while ( y_is_not_max() )
     {
      claim x_is_below_c();
      advance_y();
     }
   claim x_is_below_c();
   advance_x();
  }

counting_theorem( y, d );
while ( y_is_not_d() )
  {
   claim y_is_below_d();
   advance_y();
  }
```

```
const auto x_is_below_c =
  [&]() -> bool
  {
   claim x != c;
   return x < c;
  };


const auto y_is_below_d =
  [&]() -> bool
  {
   claim x == c;
   return y < d;
  };
```

```
counting_theorem( x, c );
while ( x != c )
  {
   counting_theorem( y, y.max_value );
   while ( y_is_not_max() )
     {
      claim x_is_below_c();
      advance_y();
     }
   claim x_is_below_c();
   advance_x();
  }

counting_theorem( y, d );
while ( y_is_not_d() )
  {
   claim y_is_below_d();
   advance_y();
  }
```

```
const auto x_is_below_c =
  [&]() -> bool
  {
   claim x != c;

   reference_less_axiom( xy, cd );
   claim (x < c) == (xy < cd);

   return x < c;
  };

const auto y_is_below_d =
  [&]() -> bool
  {
   claim x == c;

   reference_less_axiom( xy, cd );
   claim (y < d) == (xy < cd);

   return y < d;
  };
```

```
counting_theorem( x, c );
while ( x != c )
  {
   counting_theorem( y, y.max_value );
   while ( y_is_not_max() )
     {
      claim x_is_below_c();
      advance_y();
     }
   claim x_is_below_c();
   advance_x();
  }

counting_theorem( y, d );
while ( y_is_not_d() )
  {
   claim y_is_below_d();
   advance_y();
  }
```

```
counting_theorem( x, c );
while ( x != c )
  {
   counting_theorem( y, y.max_value );
   while ( y_is_not_max() )
     {
      claim x_is_below_c();
      advance_y();
     }
   claim x_is_below_c();
   advance_x();
  }

counting_theorem( y, d );
while ( y_is_not_d() )
  {
   claim y_is_below_d();
   advance_y();
  }
```

```
{
 extend_stability ab, cd;

 claim ab <= cd;

 claim implementation;

 auto xy = ab;
 while ( xy != cd )
   {
    claim xy < cd;
    ++xy;
   }
}
```

Will this loop ever end?

Yes! The loop repeats a sequence of local events that happened before the loop started.

Each iteration consumes some events, and the events before the loop are eventually exhausted.

I appreciate the locality of this reason!

I was looking for repetition anyway, so checking this is easy!

This is a practical way to express why a loop ends!

Loops that end

Loops that don’t end

Loops that are required to end

Loops that are not required to end

Loops that are required to end

- while
- for
- goto

Loops that are not required to end

- while_unbounded
- for_unbounded
- goto_unbounded

Loops that are required to end and have a clearly explained local reason to end

while
for
goto

Loops that are required to end but have **no** clearly explained local reason to end

🚫

Loops that are not required to end

while_unbounded
for_unbounded
goto_unbounded

Thank you for listening.

Questions?