Practical Interfaces for Practical Functions

Lisa Lippincott

C++now, May 2019

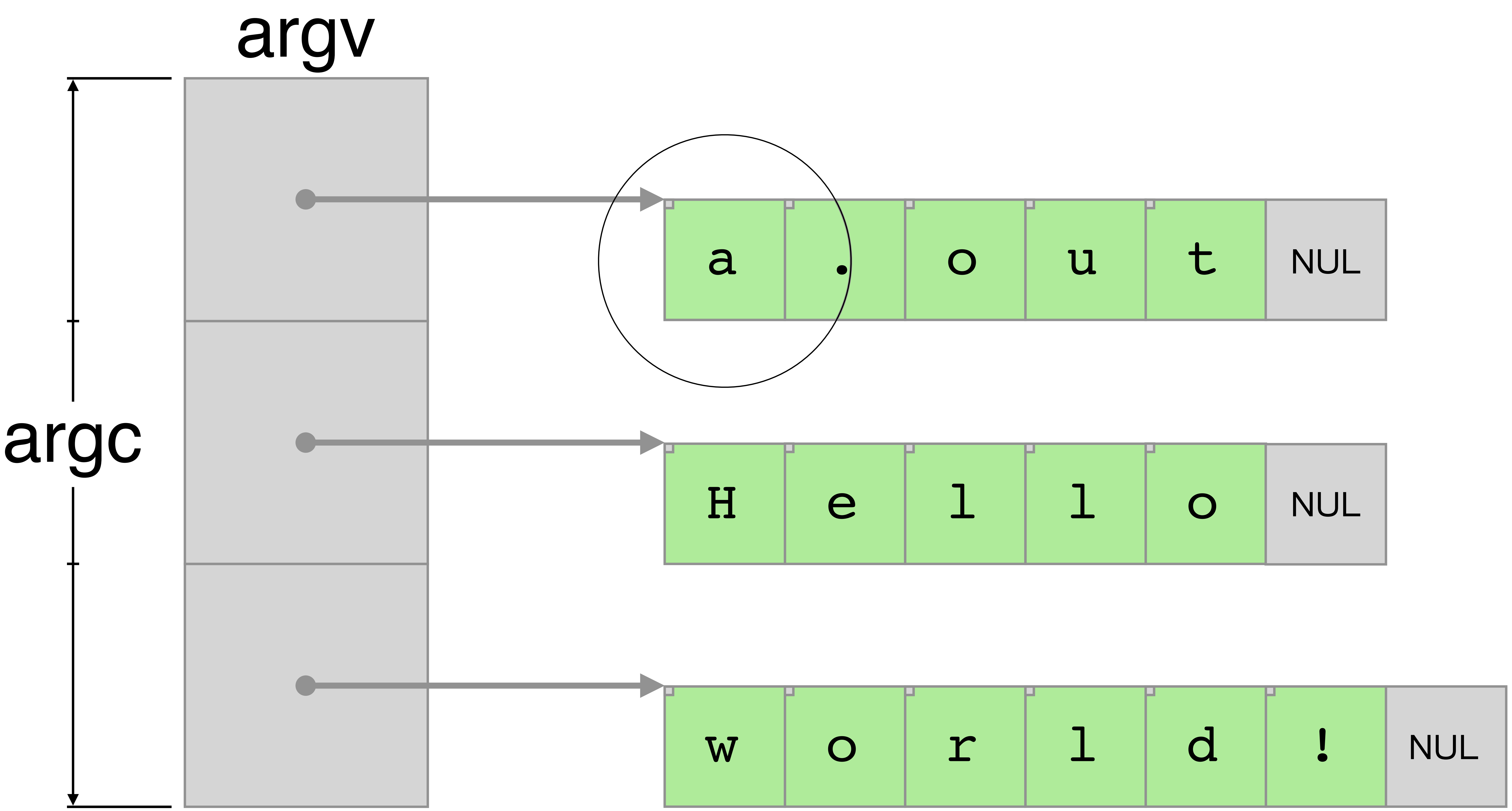
Why C++?

Why study the logic of C++?

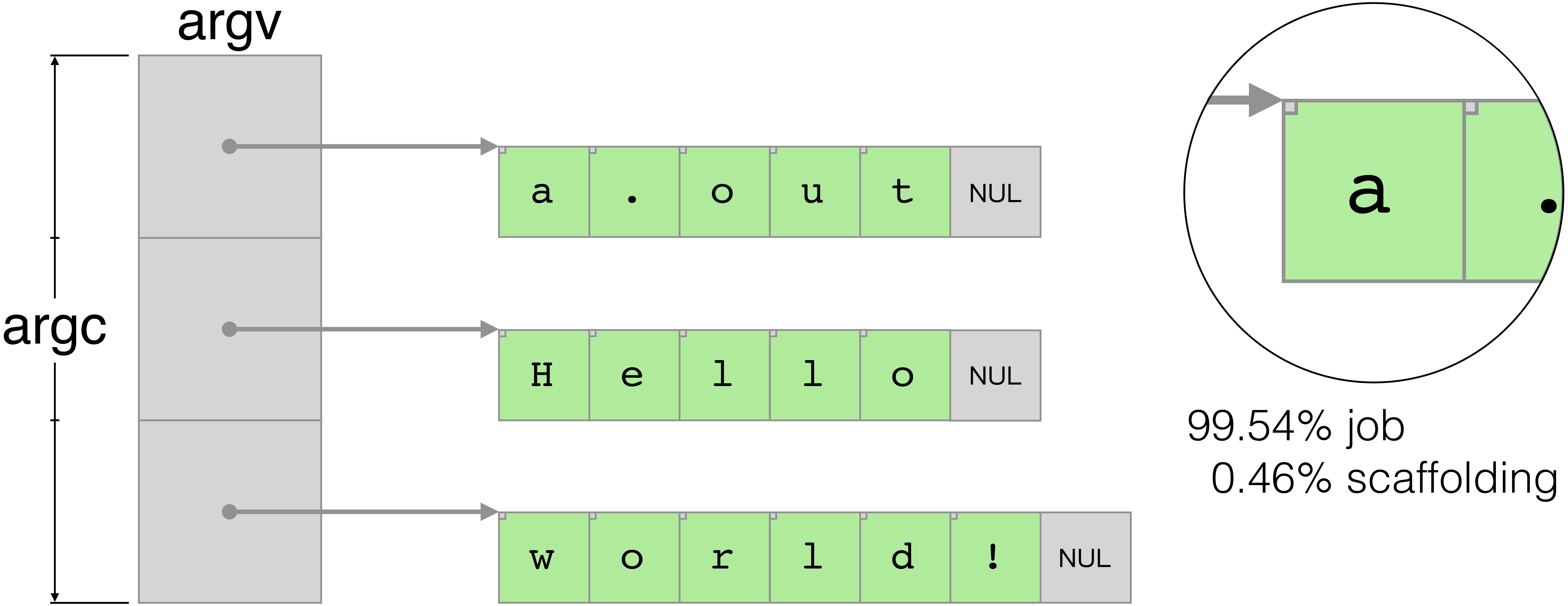
C++ is a demonstrably practical language which many programmers have reasoned about at large scale for many purposes over a long period with great success.

Worse things happen in C.

```
int main( int argc, char *argv[] )
```



99.54% job
0.46% scaffolding



99.54% job
0.46% scaffolding

```
int main( int argc, char *argv[] )
interface
  {

    // ...

    implementation;

    // ...

  }
```

```
int main( int argc, char *argv[] )
interface
  {
   for ( int i = 0; i != argc; ++i )
      for ( const char *p = argv[i]; *p != '\0'; ++p )
         claim usable( *p );


   implementation;


   // ...


  }
```

Direct input:

- the branches of the outer loop
  (equivalent to the value of argc)
- the branches of the inner loops
  (equivalent to strlen(argv[i]) for each i)
- the values of the characters *p

```
int main( int argc, char *argv[] )
interface
  {
   for ( int i = 0; i != argc; ++i )
      claim usable_NTBS( argv[i] );



   implementation;


   // ...


  }
```

```
inline claimable
usable_NTBS( const char *s )
  {
   for ( ; *s != '\0'; ++s )
      require usable( *s );
  }
```

Direct input:

- the branches of the outer loop
  (equivalent to the value of argc)
- the branches within usable_NTBS
  (equivalent to strlen(argv[i]) for each i)
- input asserted with require within usable_NTBS

```
int main( int argc, char *argv[] )
interface
  {
   for ( int i = 0; i != argc; ++i )
      claim usable_NTBS( argv[i] );


   implementation;


   // ...

   claim claim_usable( result );
  }
```

```
template < class T >
inline claimable
claim_usable( T& x )
  {
   claim usable( x );
  }
```

The result is *indirect* output: if **main** is called again with the same direct input, the result may differ.

```
int main( int argc, char *argv[] )
interface
  {
   for ( int i = 0; i != argc; ++i )
      claim usable_NTBS( argv[i] );



   implementation;

   for ( int i = 0; i != argc; ++i )
      claim usable_NTBS( argv[i] );

   claim claim_usable( result );
  }
```

When exiting a function, it's important to assert the capabilities the function is returning to its caller.

```
int main( const int argc, const char *const argv[] )
interface
  {
   for ( int i = 0; i != argc; ++i )
      claim usable_NTBS( argv[i] );



   implementation;

   for ( int i = 0; i != argc; ++i )
      claim usable_NTBS( argv[i] );

   claim claim_usable( result );
  }
```

```
int main( const int argc, char *const argv[] )
interface
  {
   for ( int i = 0; i != argc; ++i )
      claim usable_NTBS( argv[i] );



   implementation;

   for ( int i = 0; i != argc; ++i )
      claim usable_NTBS( argv[i] );

   claim claim_usable( result );
  }
```

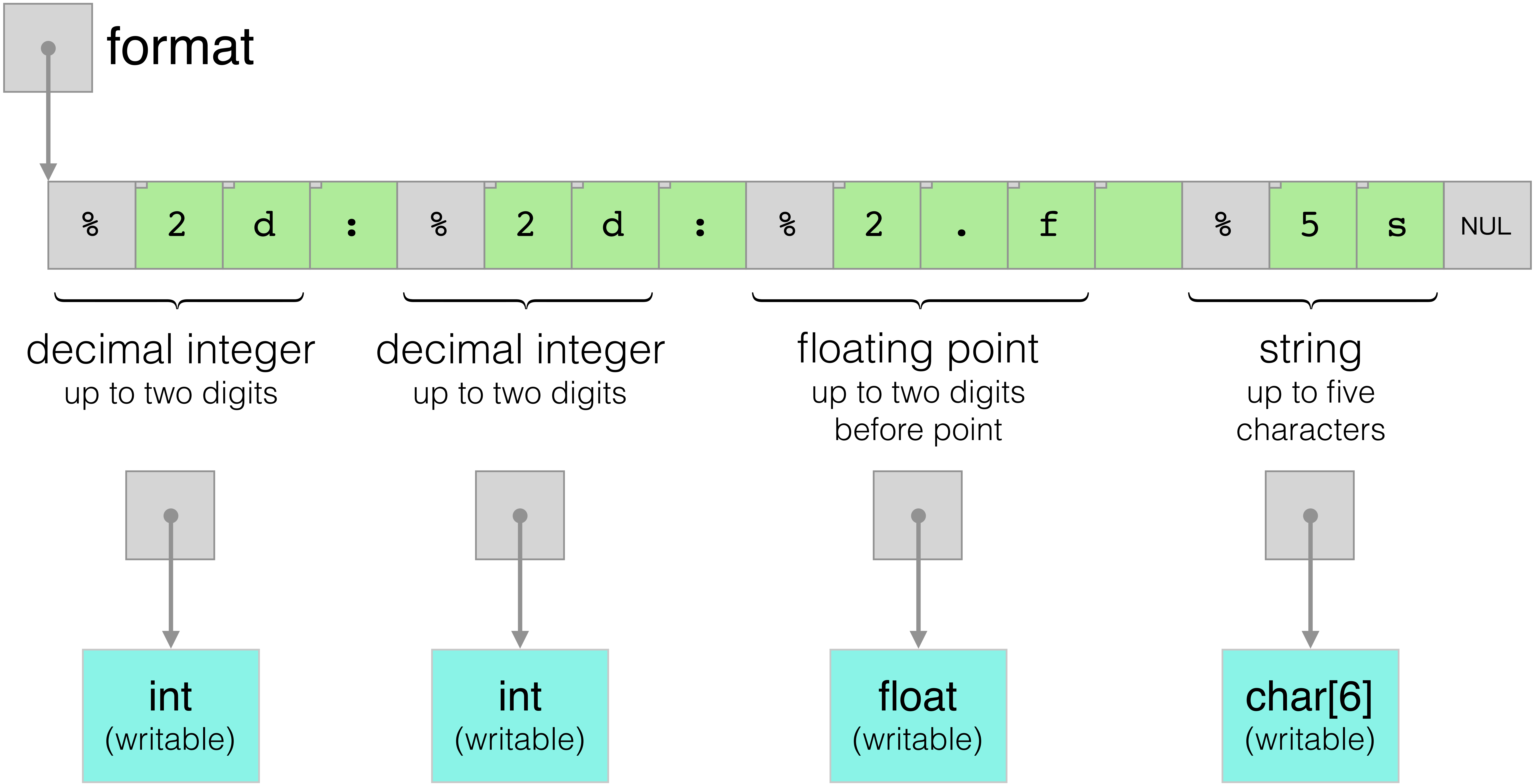
But the strings in **argv** aren't const. It's a long story…

```
inline
vector< span< char > >
argument_spans( int argc, char *const argv[] )
  {
   vector< span< char > > result;

   result.reserve( argc );

   for ( int i = 0; i != argc; ++i )
      result.push_back( span( argv[i], strlen(argv[i]) ) );

   return result;
  }
```

```
int main( const int argc, char *const argv[] )
interface
  {
   const auto arguments = argument_spans( argc, argv );


   claim usable( arguments );

   implementation;

   claim usable( arguments );


   claim claim_usable( result );
  }
```

```
using command_line = const span< const string_view >;

void main( command_line arguments )
interface
  {
   claim usable( arguments );

   implementation;

   claim usable( arguments );
  }
```

```
int scanf( const char *const format, ... )
```


format
% 2 d : % 2 d : % 2 . f % 5 s NUL
decimal integer
up to two digits
decimal integer
up to two digits
floating point
up to two digits
before point
string
up to five
characters
int
(writable)
int
(writable)
float
(writable)
char[6]
(writable)

```
int scanf( const char *const format, ... )
interface
  {



     // ...

     implementation;

     // ...


  }
```

```
int scanf( const char *const format, ... )
interface
  {
   claim usable_NTBS( format );





   // …


   implementation;


   // . . .



   claim usable_NTBS( format );
  }
```

```
int scanf( const char *const format, ... )
interface
  {
   claim usable_NTBS( format );

   va_list args;

   va_start( args );
   // ...
   va_end( args );

   implementation;

   va_start( args );
   // ...
   va_end( args );

   claim usable_NTBS( format );
  }
```

The type **va_list** is a sort of iterator, or maybe a container with a cursor.

The macro **va_start** starts an iteration over the unnamed arguments of the function in which it is expanded.

Each **va_start** requires a **va_end**, which must appear in the same scope.

```
int scanf( const char *const format, ... )
interface
  {
   claim usable_NTBS( format );


   va_list args;

   va_start( args );
   claim usable_scanf_args( format, &args, 0 );
   va_end( args );

   implementation;

   va_start( args );
   claim usable_scanf_args( format, &args, result );
   va_end( args );


   claim usable_NTBS( format );
  }
```

On the way in, none of the unnamed arguments need to point to readable objects.

On the way out, the **result** tells us how many arguments must now point to readable objects.

```
inline claimable
usable_scanf_args( const char *format,
                   va_list *args,
                   int written_count ) noexcept
  {
   // Strategy 1: Parse the format, and require each argument it specifies.
   try
     {
      require usable_args_for_scanf_format( format, args, written_count );
      return;
     }
   catch ( ... )
     {}

   // Strategy 2: Fail outright, because the format cannot be parsed.
   require false;
  }
```

```
inline claimable
usable_args_for_scanf_format( const char *format,
                              va_list *args,
                              int written_count )
  {
   // Iterate over the directives in the format, skipping those with no parameter.

   for ( auto d: format_directives( format ) )
      if ( d.has_parameter() )
        {
         require usable_arg_for_scanf_directive( d, args, written_count > 0 );
         --written_count;
        }
  }
```

```
inline claimable
usable_arg_for_scanf_directive( const format_directive& d,
                                va_list *args,
                                bool written )
  {
   // Dispatch to a function suitable for the type specified in the directive.

   using directive_type_code_literal::operator""_dtc;

   switch ( d.type_code() )
     {
      case "d"_dtc:   require usable_scanf_arg< "d"_dtc   >( d, args, written );   break;
      case "hd"_dtc:  require usable_scanf_arg< "hd"_dtc  >( d, args, written );   break;
      case "ld"_dtc:  require usable_scanf_arg< "ld"_dtc  >( d, args, written );   break;
      case "f"_dtc:   require usable_scanf_arg< "f"_dtc   >( d, args, written );   break;
      case "s"_dtc:   require usable_scanf_arg< "s"_dtc   >( d, args, written );   break;
      // ...
      default: throw unknown_type_code();
     }
  }
```

```
template < directive_type_code type_code >
inline claimable
usable_scanf_arg( const format_directive& d,
                  va_list *args,
                  bool written )
  {
   claim d.type_code() == type_code;

   using pointer_type = scanf_pointer_type_for_code< type_code >;
   pointer_type p = va_arg( *args, pointer_type );

   require !written || usable( *p );
   require writable( *p );
  }
```

```
template <>
inline claimable
usable_scanf_arg< "s"_dtc >( const format_directive& d,
                             va_list *args,
                             bool written )
  {
   claim d.type_code() == "s"_dtc;

   using pointer_type = char *;
   pointer_type p = va_arg( *args, pointer_type );

   require !written || usable_NTBS(p);

   // The field width is required so that we know how much space must be writable.
   require d.has_field_width();
   require writable_span( p, d.field_width()+1 );
  }
```

```
int scanf( const char *const format, ... )
interface
  {
   claim usable_NTBS( format );


   va_list args;

   va_start( args );
   claim usable_scanf_args( format, &args, 0 );
   va_end( args );

   implementation;

   va_start( args );
   claim usable_scanf_args( format, &args, result );
   va_end( args );


   claim usable_NTBS( format );
  }
```

```
int scanf( const char *const format, ... )
interface
  {
   claim usable_NTBS( format );
   claim usable_input_stream<char>( stdin );

   va_list args;

   va_start( args );
   claim usable_scanf_args( format, &args, 0 );
   va_end( args );

   implementation;

   va_start( args );
   claim usable_scanf_args( format, &args, result );
   va_end( args );

   claim usable_input_stream<char>( stdin );
   claim usable_NTBS( format );
  }
```

```
template < class... Types >
class scanf_functor
  {
   public:
      explicit constexpr scanf_functor( const char *f );

      tuple< Types... > operator()() const;
  };
```

```
template < class... Types >
class scanf_functor
  {
   public:
      explicit constexpr scanf_functor( const char *f )
      interface
        {
         claim scanf_format_matches< Types... >( f );

         implementation;

         claim usable( *this );
        }

      tuple< Types... > operator()() const;
  };
```

```
template < class... Types >
class scanf_functor
  {
   public:
      explicit constexpr scanf_functor( const char *f );

      tuple< Types... > operator()() const
      interface
        {
         claim usable_input_stream<char>( stdin );
         claim usable( *this );

         implementation;

         claim usable_input_stream<char>( stdin );
         claim usable( *this );
         claim usable( result );
        }
  };
```

```
template < class... Types >
class scanf_functor
  {
   public:
      explicit constexpr scanf_functor( const char *f );

      tuple< Types... > operator()() const;
      tuple< Types... > operator()( FILE *f ) const;
      tuple< Types... > operator()( const char *s ) const;
  };
```

```
template < class... Types >
class scanf_functor
  {
   public:
      explicit constexpr scanf_functor( const char *f );

      tuple< Types... > operator()() const;
      tuple< Types... > operator()( FILE *f ) const;
      tuple< Types... > operator()( const char *s ) const;
  };

template < const auto& format >
inline constexpr auto make_scanf_functor()
  {
   return scanf_functor< scanf_result_type<format> >( format );
  }
```

```
template < class... Types >
class scanf_functor
  {
   public:
       explicit constexpr scanf_functor( const char *f );

       tuple< Types... > operator()() const;
       tuple< Types... > operator()( FILE *f ) const;
       tuple< Types... > operator()( const char *s ) const;
  };

template < const auto& format >
inline constexpr auto make_scanf_functor();

template < const auto& format >
inline constexpr auto Scanf = make_scanf_functor< format >();
```

```
static constexpr char my_format[] = "%2d:%2d:%2.f %5s";

auto [ h, m, s, z ] = Scanf<my_format>();
```

constexpr object of type
scanf_functor< int, int, float, char[6] >

h, **m**, **s**, and **z** have types
int, **int**, **float**, and **char[6]**, respectively.

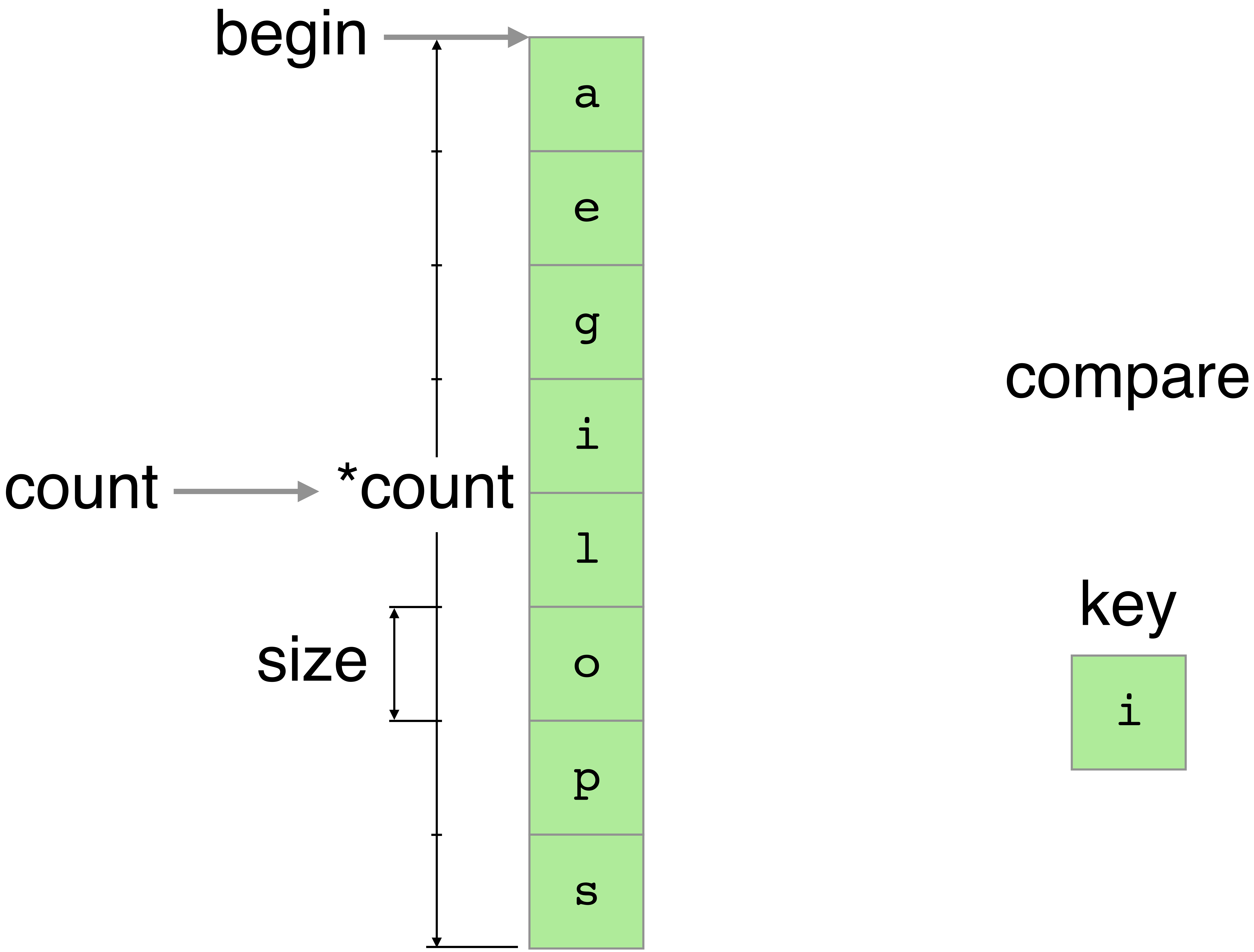
// ⚠️ Expected for C++20:

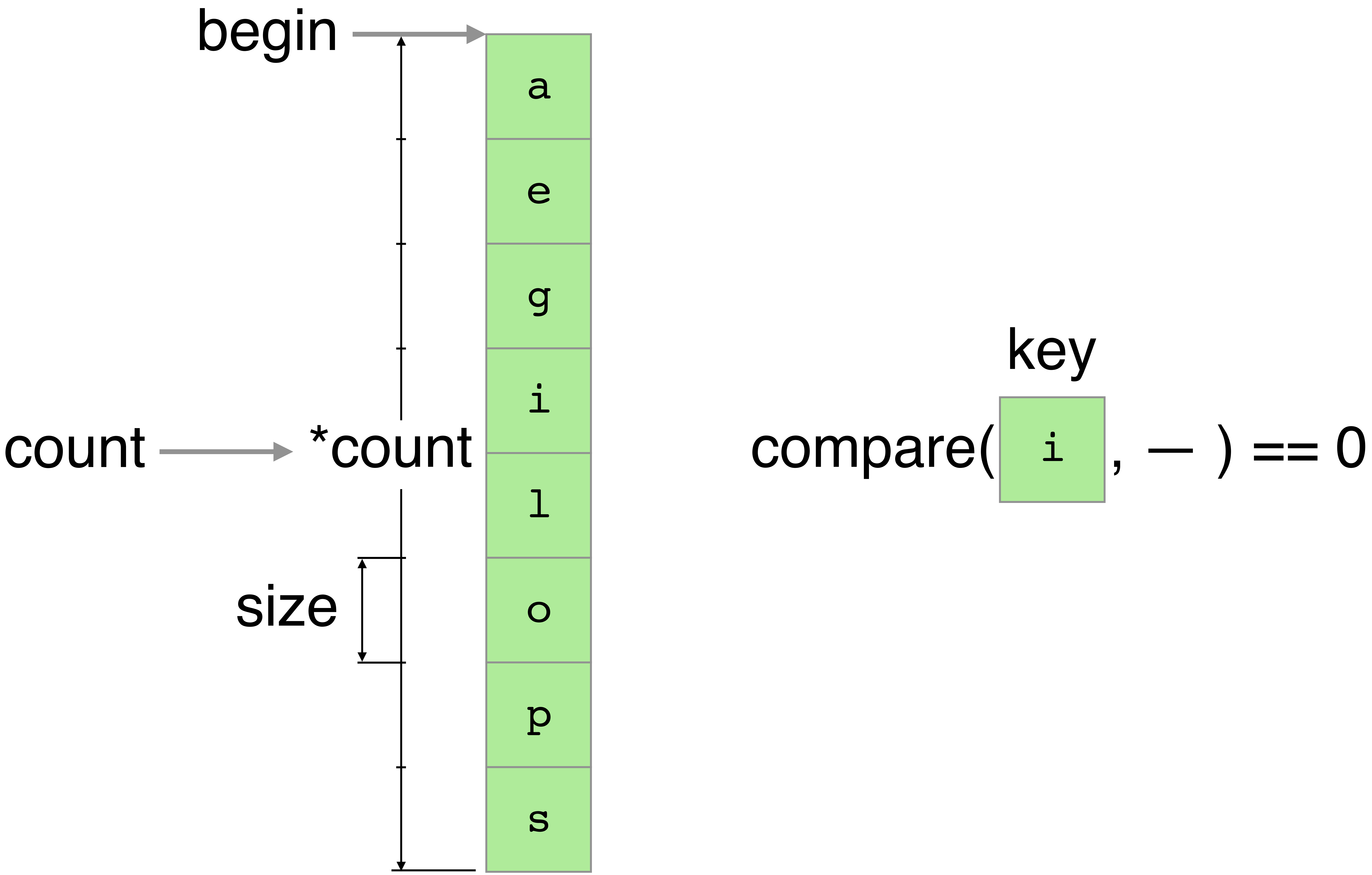
```
auto [ h, m, s, z ] = "%2d:%2d:%2.f %5s"_scanf();
```

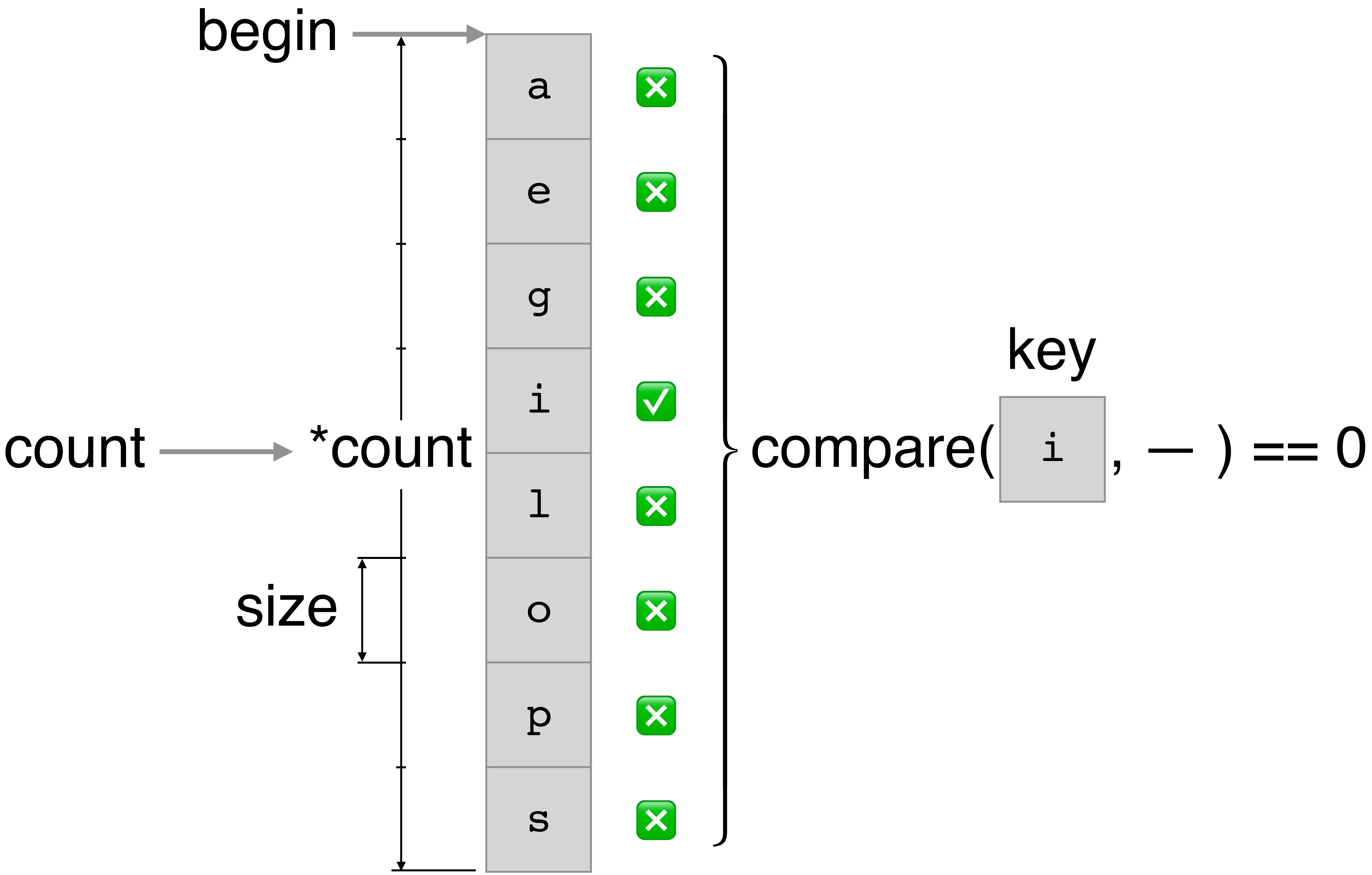
constexpr literal expression of type
scanf_functor< int, int, float, char[6] >

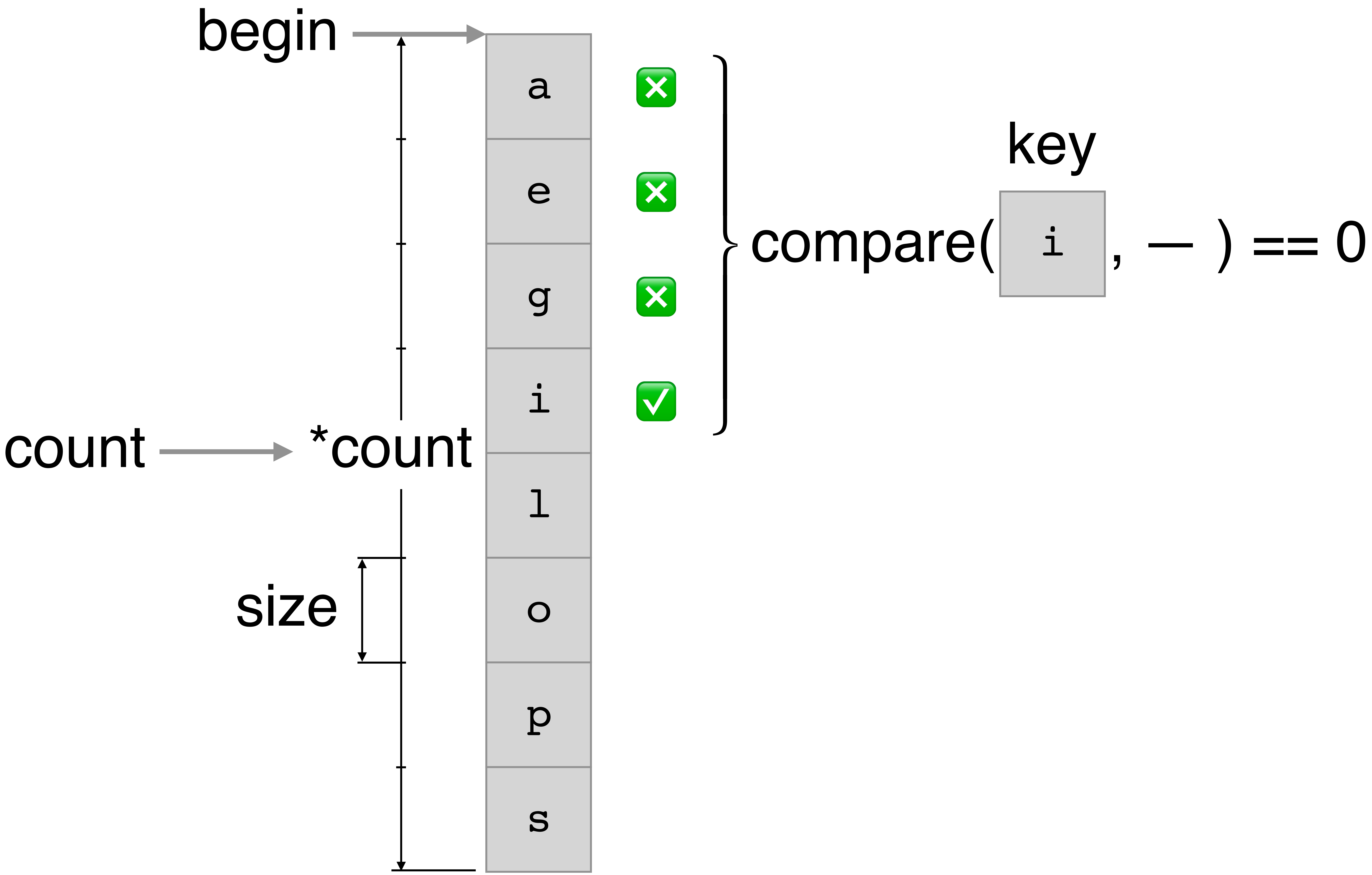
h, **m**, **s**, and **z** have types
int, **int**, **float**, and **char[6]**, respectively.

```
void *lfind( const void *key,
             const void *begin,
             size_t *count,
             size_t size,
             int (*compare)( const void *, const void * ) )
```


begin
a
e
g
i
l
o
p
s
count
*count
size
compare
key
i


begin
a
e
g
i
l
o
p
s
count
*count
size
key
compare(i , —) == 0


begin
a
e
g
i
l
o
p
s
count
*count
size
key
compare(i , —) == 0


begin
a
e
g
i
l
o
p
s
key
compare(i , —) == 0
count
*count
size

```
void *lfind( const void *key,
             const void *begin,
             size_t *count,
             size_t size,
             int (*compare)( const void *, const void * ) )
 {


   // …


   implementation;

   // …

 }
```

```
inline const void *end_pointer( const void *p,
                                 size_t size,
                                 size_t count )
  {
   return static_cast< const byte * >( p ) + count * size;
  }


inline const void *next_pointer( const void *p,
                                  size_t size )
  {
   return static_cast< const byte * >( p ) + size;
  }
```

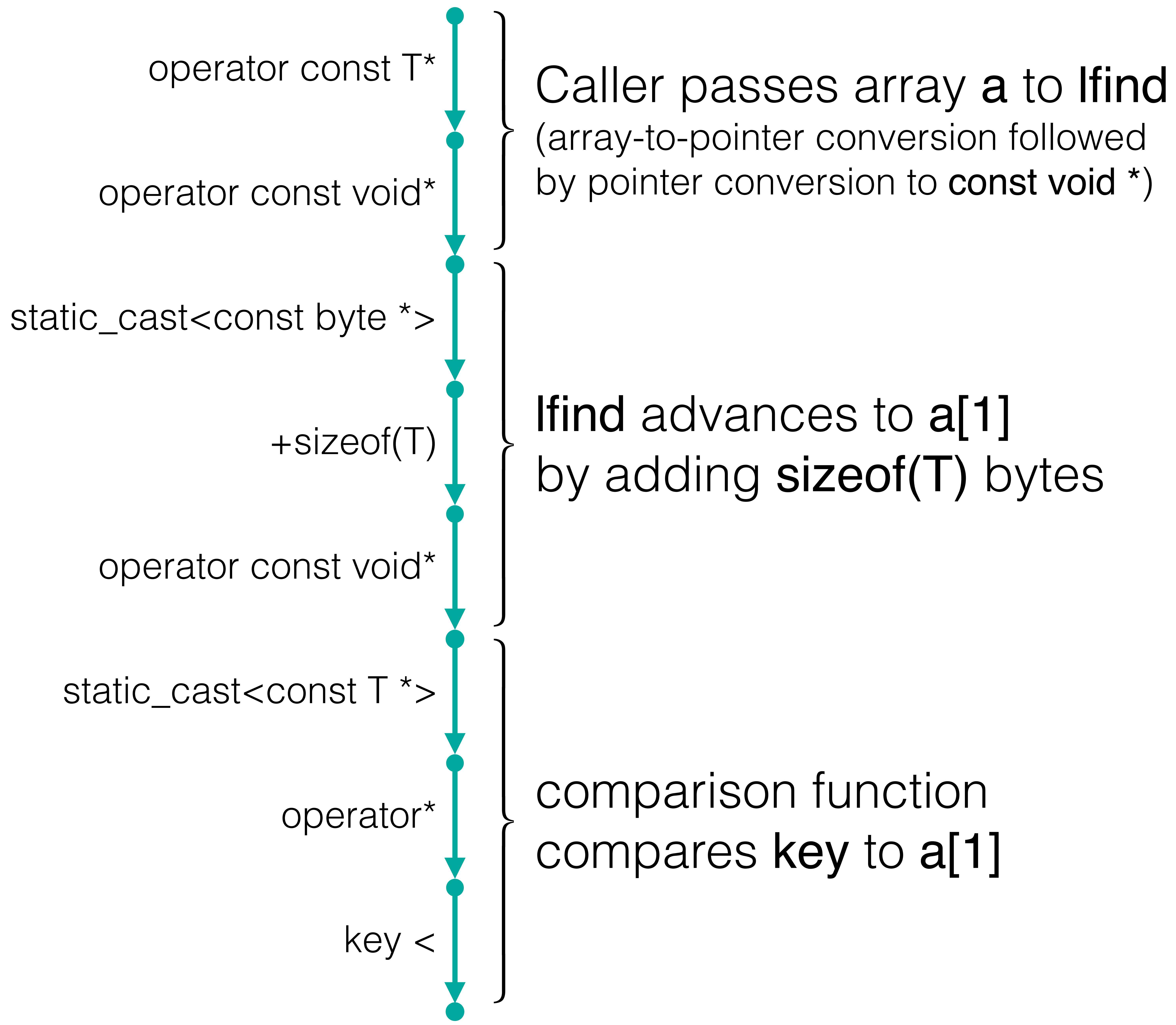
```
void *lfind( const void *key,
             const void *begin,
             size_t *count,
             size_t size,
             int (*compare)( const void *, const void * ) )
  {
   const auto end = end_pointer( begin, size, *count );

   auto p = begin;
   while (  p != end  &&  compare( key, p ) != 0  )
       p = next_pointer( p, *size );

   implementation;

   // . . .

  }
```

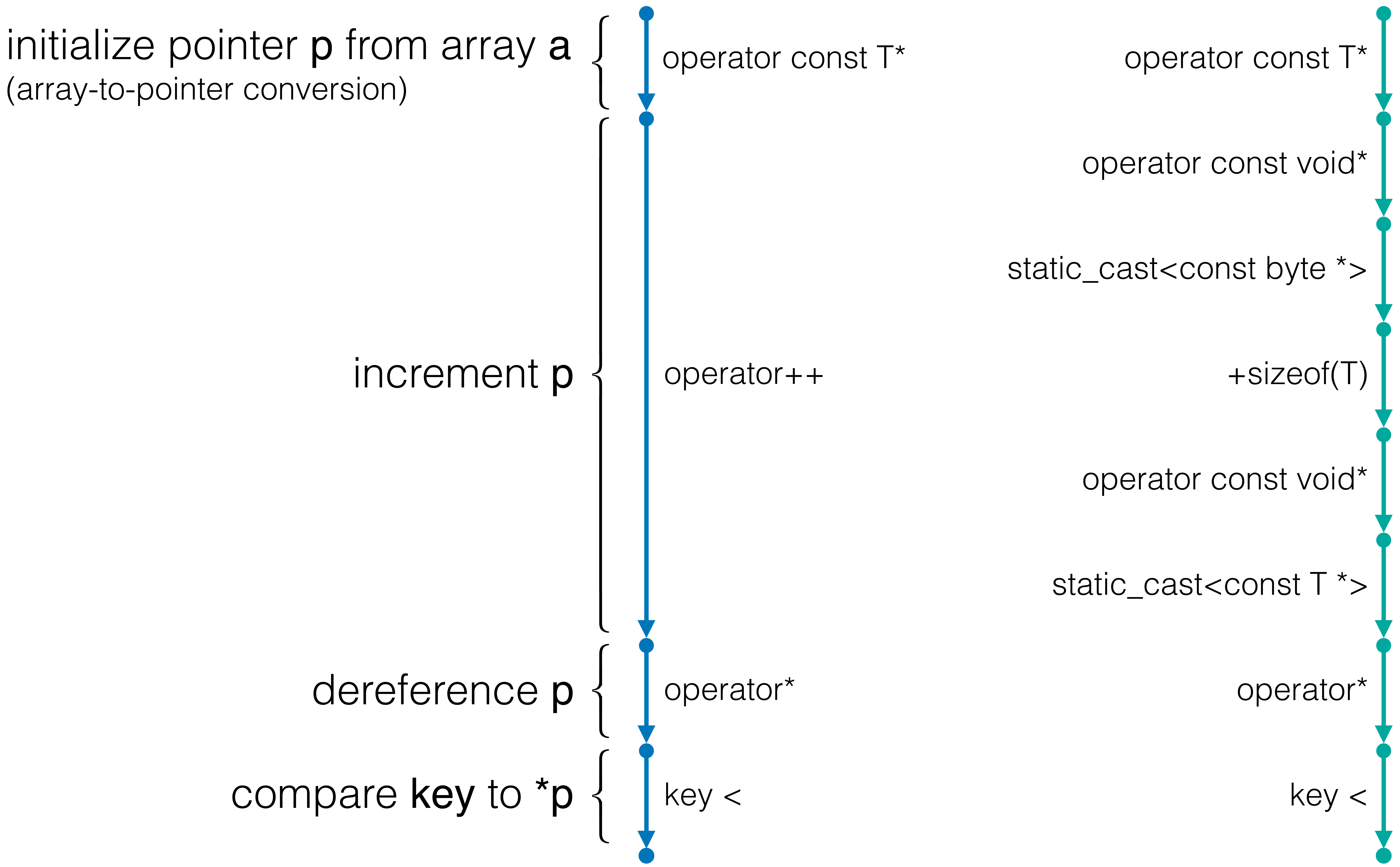
```
void *lfind( const void *key,
             const void *begin,
             size_t *count,
             size_t size,
             int (*compare)( const void *, const void * ) )
  {
   const auto end = end_pointer( begin, size, *count );

   auto p = begin;
   while (  p != end  &&  compare( key, p ) != 0  )
      p = next_pointer( p, *size );

   implementation;

   auto expected_result = ( p == end ) ? nullptr : p;
   claim substitutable_and_equal( result, expected_result );
  }
```

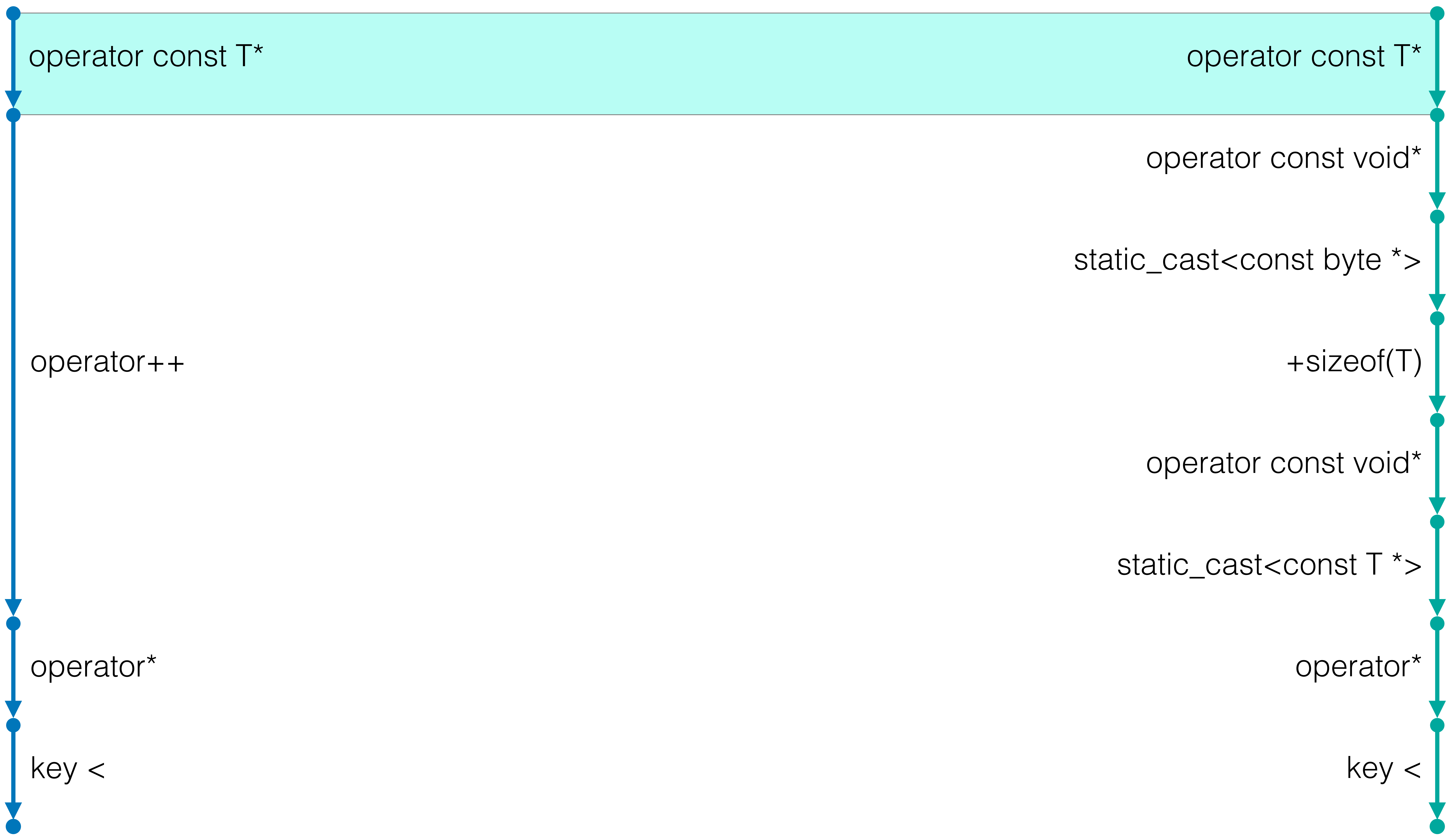
```
template < class T >
inline claimable substitutable_and_equal( T& a, T& b )
  {
   require substitutable( a, b );
   require a == b;
  }
```

```
void bad_function()
  {
   int a;
   int b;
   if ( &a + 1 == &b )                          // These pointers might be equal,
🔥     claim substitutable( &a+1, &b );    // but they are not substitutable.
  }
```

```
void *lfind( const void *key,
             const void *begin,
             size_t *count,
             size_t size,
             int (*compare)( const void *, const void * ) )
  {
   const auto end = end_pointer( begin, size, *count );

   auto p = begin;
   while (  p != end  &&  compare( key, p ) != 0  )
      p = next_pointer( p, *size );

   implementation;

   auto expected_result = ( p == end ) ? nullptr : p;
   claim substitutable_and_equal( result, expected_result );
  }
```


operator const T*
operator const void*
static_cast<const byte *>
+sizeof(T)
operator const void*
static_cast<const T *>
operator*
key <
Caller passes array **a** to **lfind**
(array-to-pointer conversion followed
by pointer conversion to **const void** *)
lfind advances to **a[1]**
by adding **sizeof(T)** bytes
comparison function
compares **key** to **a[1]**

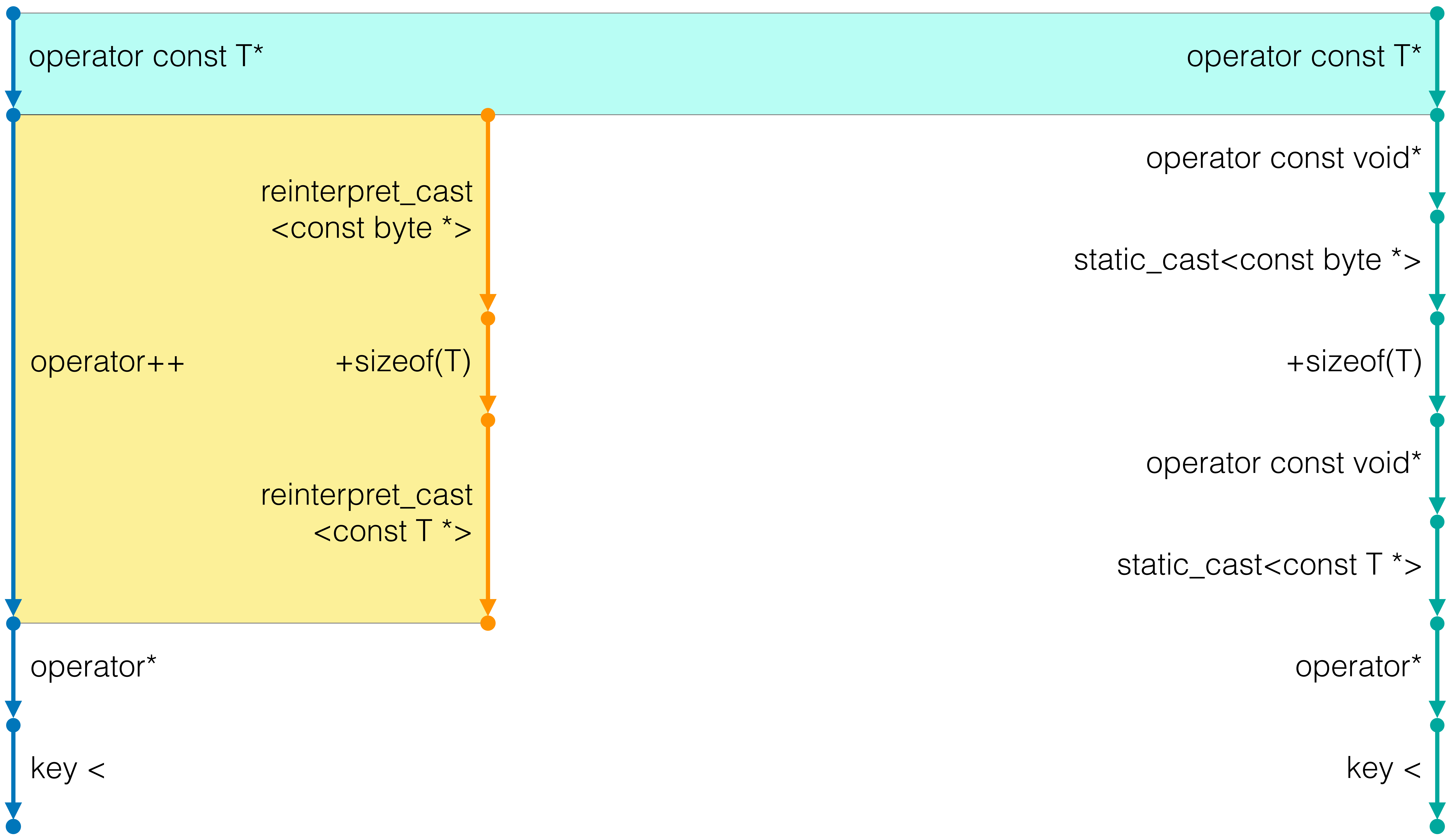

initialize pointer **p** from array **a**
(array-to-pointer conversion)
increment **p**
dereference **p**
compare **key** to ***p**
operator const T*
operator++
operator*
key <
operator const T*
operator const void*
static_cast<const byte *>
+sizeof(T)
operator const void*
static_cast<const T *>
operator*
key <


operator const T*
operator const T*
operator const void*
static_cast<const byte *>
operator++
+sizeof(T)
operator const void*
static_cast<const T *>
operator*
operator*
key <
key <

```
template < class T >
T *& operator++( T*& pointer )
  {
   const auto next_byte = reinterpret_cast< const byte * >( pointer ) + sizeof(T);

   // . . .

   implementation;

   // . . .

   claim substitutable_and_equal( result, reinterpret_cast< T * >( next_byte ) );
  }
```

An object of array type contains a **contiguously** allocated non-empty set of `N` subobjects of type `T`.

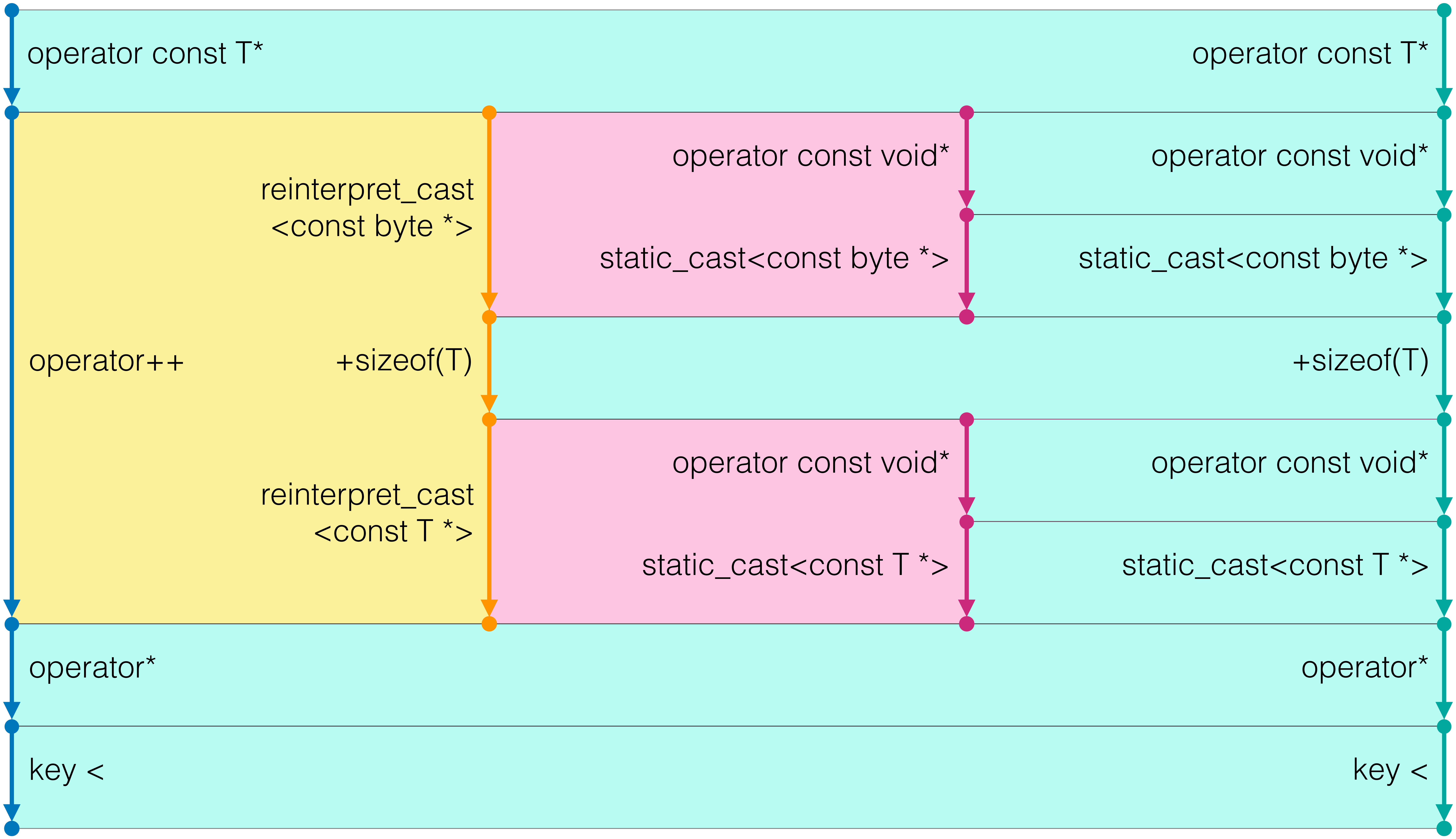
9.2.3.4 [dcl.array] ¶1


operator const T*
operator const T*
operator const void*
reinterpret_cast
<const byte *>
static_cast<const byte *>
operator++
+sizeof(T)
+sizeof(T)
operator const void*
reinterpret_cast
<const T *>
static_cast<const T *>
operator*
operator*
key <
key <

```
template < class To, class From >
To *reinterpret_cast( From *const original )
  {

   // ...

   implementation;

   // ...

   using cvv_ptr = common_type_t< void *, To * >;
   claim substitutable_and_equal( result,
                                  static_cast< To * >( static_cast< cvv_ptr >( original ) );
  }
```

When a prvalue `v` of object pointer type is converted to the object pointer type “pointer to *cv* `T`”, the result is `static_cast<`*cv* `T*>(static_cast<`*cv* `void*>(v))`.

7.6.1.9 [expr.reinterpret.cast] ¶7

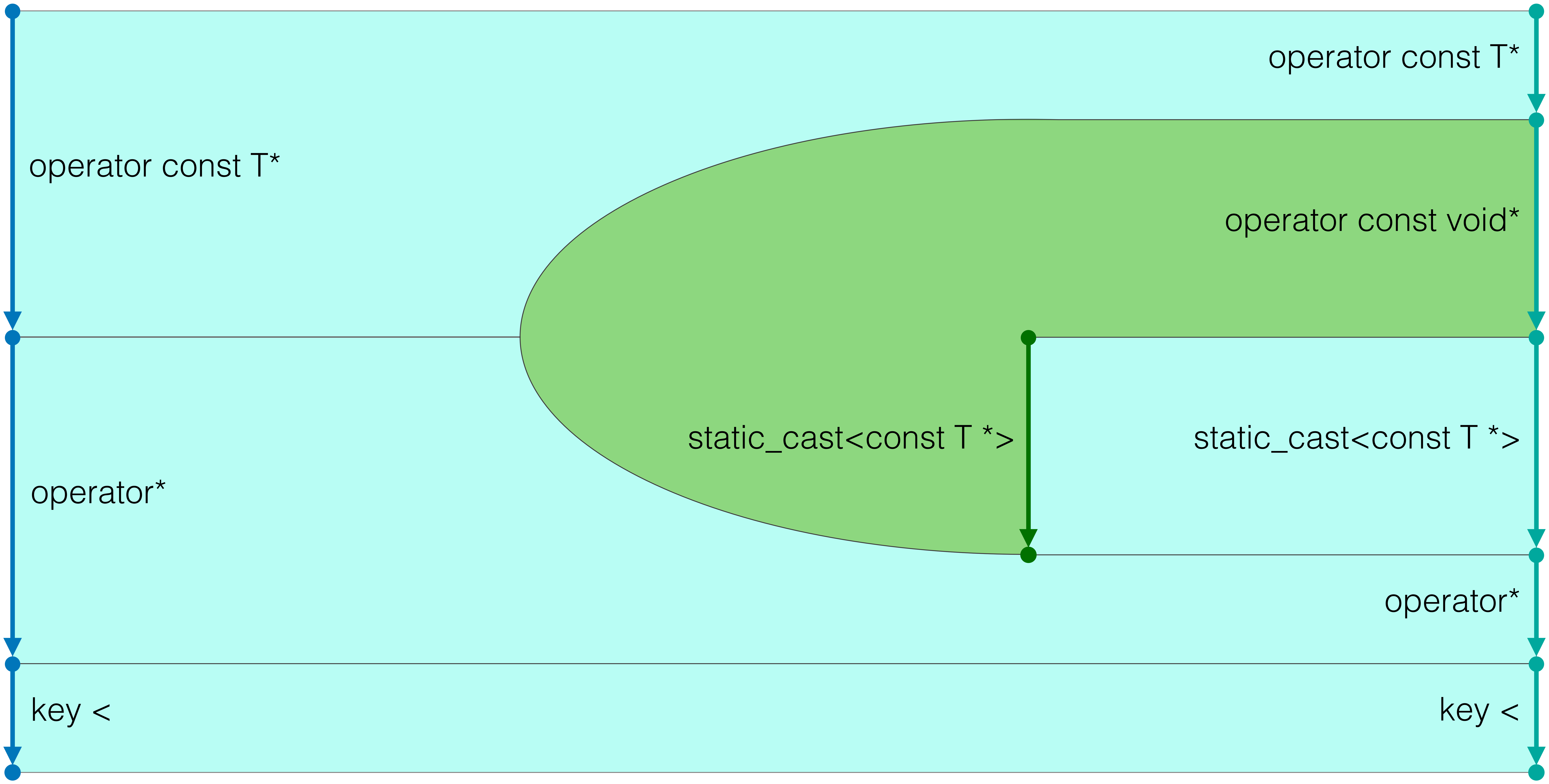

operator const T*
operator const T*
operator const void*
operator const void*
reinterpret_cast
<const byte *>
static_cast<const byte *>
static_cast<const byte *>
operator++
+sizeof(T)
+sizeof(T)
operator const void*
operator const void*
reinterpret_cast
<const T *>
static_cast<const T *>
static_cast<const T *>
operator*
operator*
key <
key <


operator const T*
operator*
key <
operator const T*
operator const void*
static_cast<const T *>
operator*
key <

```
template < class T >
operator void *( T *const original )
  {

   // ...

   implementation;

   // ...

   claim substitutable_and_equal( original, static_cast< T* >( result ) );
  }
```

The inverse of any standard conversion sequence (7.3) not containing an [...] conversion, can be performed explicitly using `static_cast`.

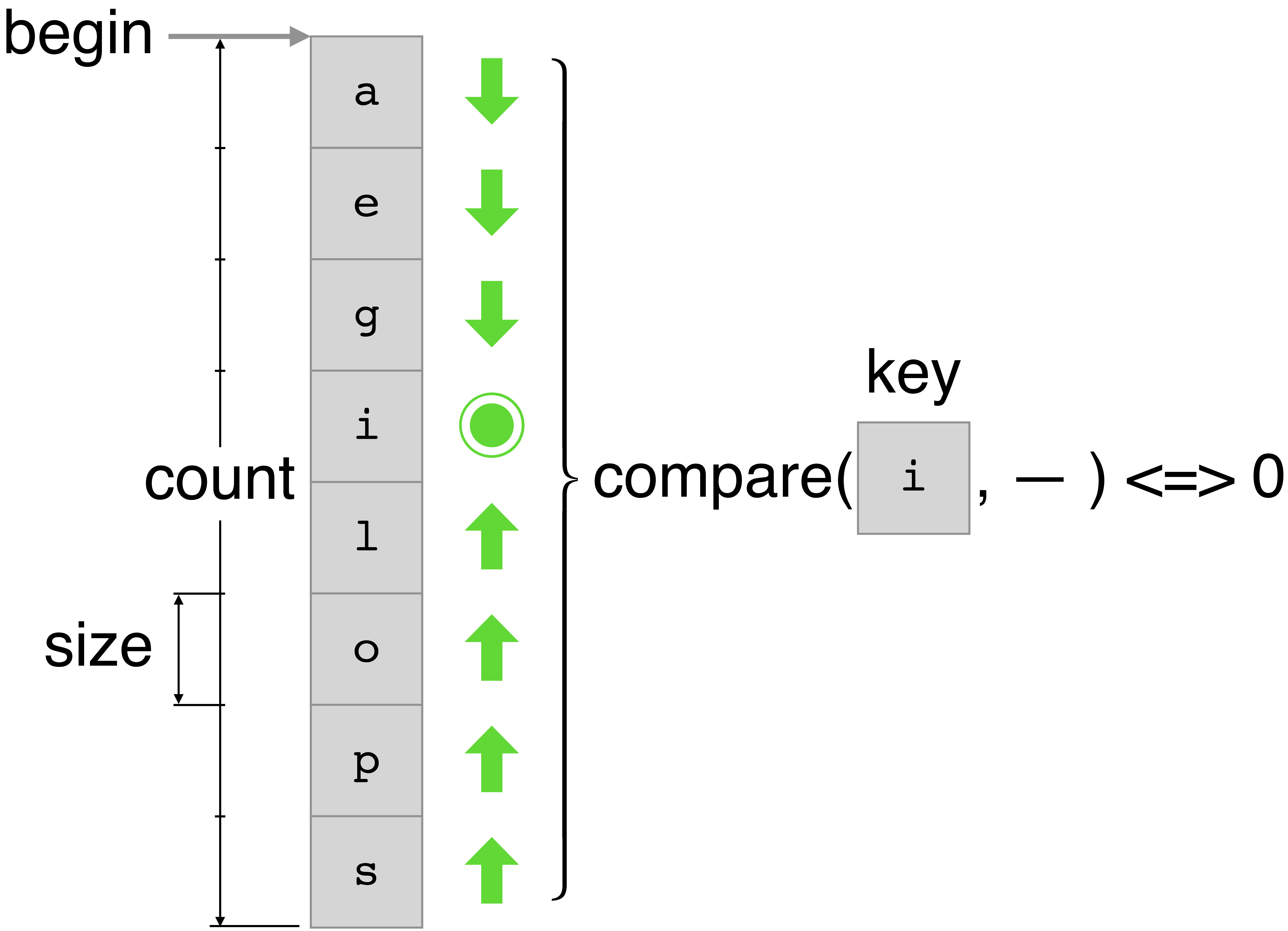
7.6.1.8 [expr.static.cast] ¶7



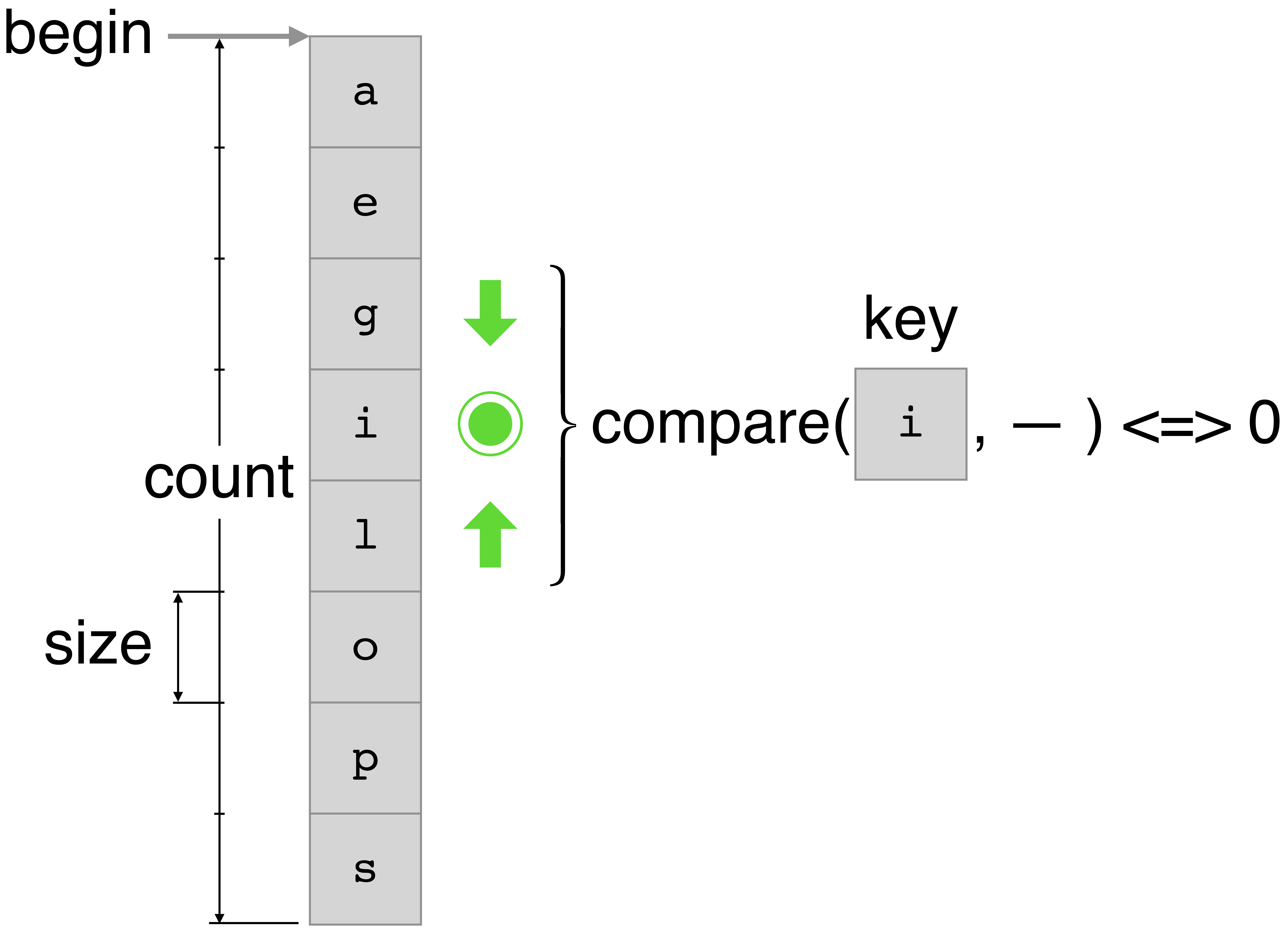
operator const T*
operator const T*
operator const void*
static_cast<const T *>
static_cast<const T *>
operator*
operator*
key <
key <

```
template < class T, class F >
T *lfind( const span<T> elements, const F is_target )
  {
   const auto reference_implementation = [&]()
     {
      for ( auto& e: elements )
         if ( is_target( e ) )
            return &e;

      return nullptr;
     };

   const auto expected_result = reference_implementation();

   implementation;

   claim substitutable_and_equal( result, expected_result );
  }
```

```
void *bsearch( const void *key,
               const void *begin,
               size_t count,
               size_t size,
               int (*compare)( const void *, const void * ) )
```



begin
a
e
g
i
l
o
p
s
count
size
key
compare(i , —) <=> 0


begin
a
e
g
i
l
o
p
s
count
size
key
compare(i , —) <=> 0

```
void *bsearch( const void *key,
               const void *begin,
               size_t count,
               size_t size,
               int (*compare)( const void *, const void * ) )
{







    // ...

    implementation;

    // ...
}
```

```
void *bsearch( const void *key,
               const void *begin,
               size_t count,
               size_t size,
               int (*compare)( const void *, const void * ) )
 {
  const auto compare_to_key = [compare, key]( const void *p )
    {
     return compare( key, p );
    };



  // ...

  implementation;

  // ...
 }
```

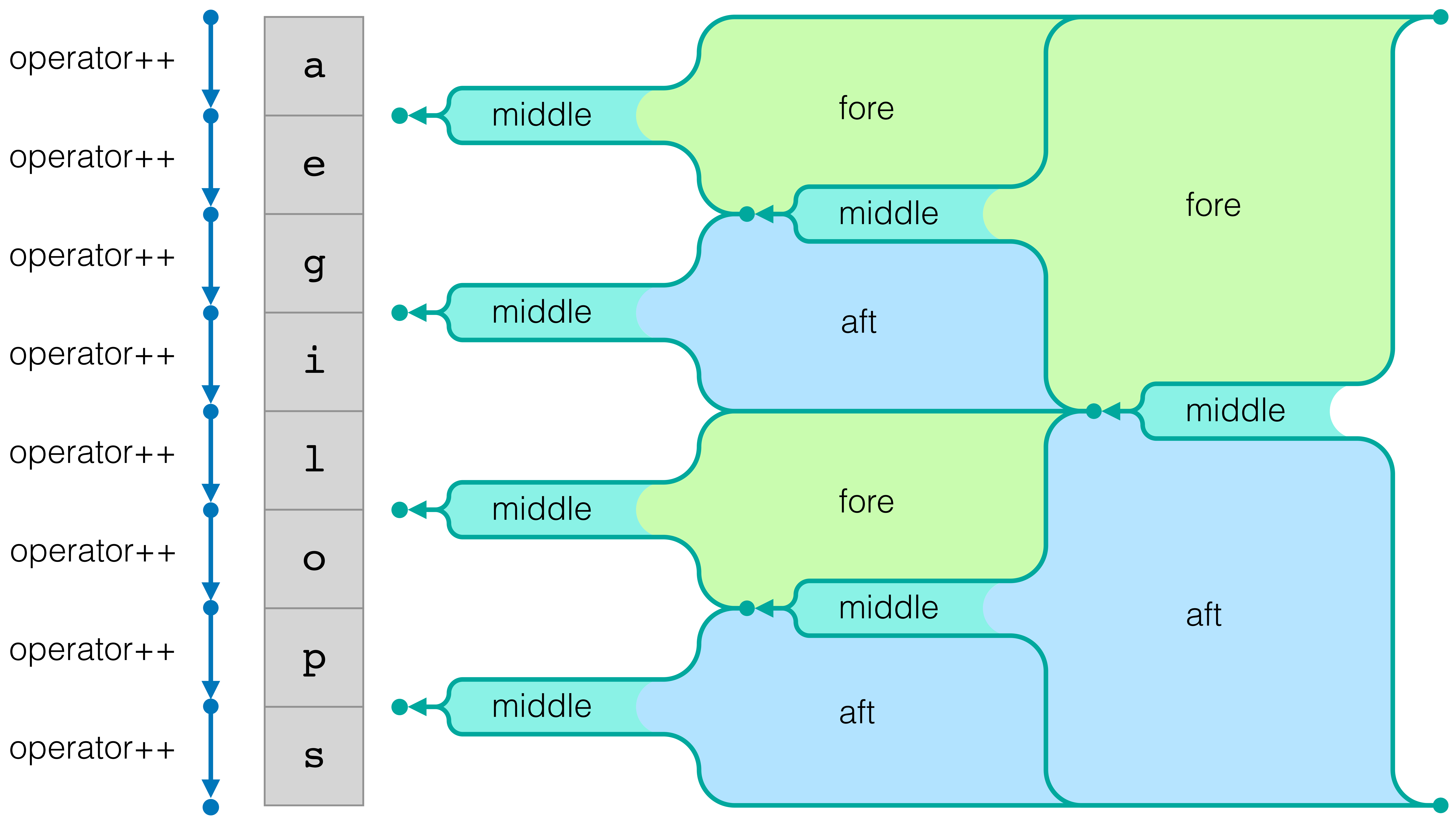
```
struct bsearch_view
  {
   const byte *begin;
   size_t count;
   size_t size;

   bool empty() const                  { return count == 0u; }

   const byte *middle() const          { return begin + (count/2u) * size; }

   bsearch_view fore() const           { return { begin,            count/2u,              size }; }
   bsearch_view aft() const            { return { middle()+size, count-(count/2u+1u), size }; }
  };
```

```
void *bsearch( const void *key,
               const void *begin,
               size_t count,
               size_t size,
               int (*compare)( const void *, const void * ) )
  {
   const auto compare_to_key = [compare, key]( const void *p )
      {
       return compare( key, p );
      };

   const bsearch_view view{ static_cast< const byte * >( begin ), count, size };

   // ...

   implementation;

   // ...
  }
```

```
void *bsearch( const void *key,
               const void *begin,
               size_t count,
               size_t size,
               int (*compare)( const void *, const void * ) )
  {
   const auto compare_to_key = [compare, key]( const void *p )
     {
      return compare( key, p );
     };

   const bsearch_view view{ static_cast< const byte * >( begin ), count, size };

   const auto expected = bsearch_reference_implementation( view, compare_to_key );

   implementation;

   claim substitutable_and_equal( result, expected );
  }
```

```
template < class Comparison >
inline const void *
bsearch_reference_implementation( bsearch_view view,
                                  const Comparison compare_to_key )
  {
   while ( !view.empty() )
     {
      const auto c = compare_to_key( view.middle() );

      if        ( c > 0 )  view = view.fore();
      else if   ( c < 0 )  view = view.aft();
      else                 return view.middle();
     }

   return nullptr;
  }
```

```
template < class T, class Comparison >
T *
bsearch( tree_view<T> view,
         const Comparison compare_to_key )
  {
   const auto expected = bsearch_reference_implementation( view, compare_to_key );

   implementation;

   claim substitutable_and_equal( result, expected );
  }
```


operator++
operator++
operator++
operator++
operator++
operator++
operator++
operator++
a
e
g
i
l
o
p
s
middle
fore
middle
fore
middle
aft
middle
middle
fore
middle
aft
middle
aft

```
template < class T >
inline claimable
tree_matches_span(   tree_view<T> t,   span<T> s   )
```

```
template < class T >
inline claimable
tree_matches_span(   tree_view<T> t,   span<T> s   )
  {
   const auto tree_size_bound = s.size();

   can_halve_until_zero( tree_size_bound );
      //    postcondition:     for (   auto i = tree_size_bound;   i != 0;   i = half(i)   )   {}

   require tree_matches_subspan( t,                       // a tree
                                 s,                       // a span
                                 0,                       // starting index of a subspan
                                 s.size(),                // length of the subspan
                                 tree_size_bound );       // bound on the tree size
  }
```

```
template < class T >
void can_halve_until_zero( const T n )
interface
  {
   static_assert( is_integral_v<T> );

   claim usable( n );

   claim implementation;

   for (   auto i = n;   i != 0;   i = half( i )   )
     {}
  }
```

```
template < class T >
void can_halve_until_zero( const T n )
implementation
  {
   can_halve_tilde_zero_until_zero< T >();

   auto b = ~( T{0} );
   claim n & b == n;

   while (  n != T{0}  &&  b != T{0}  )
     {
      n = half( n );
      bound = half( b );
      claim n & b == n;
     }

   claim n & T{0} == T{0};
  }
```

Notable postconditions

```
for (  auto i = ~0;  i != 0;  i = half(i)  )  {}
```
(a fundamental axiom)

`n & ~0 == n`
(a postcondition of operator&)

`half( n & b ) == half( n ) & half( b )`
(a postcondition of operator&)

`n & 0 == 0`
(a postcondition of operator&)

```
template < class T >
inline claimable
tree_matches_span(   tree_view<T> t,   span<T> s   )
  {
   const auto tree_size_bound = s.size();

   can_halve_until_zero( tree_size_bound );
      //    postcondition:     for (   auto i = tree_size_bound;   i != 0;   i = half(i)   )   {}

   require tree_matches_subspan( t,                         // a tree
                                 s,                         // a span
                                 0,                         // starting index of a subspan
                                 s.size(),                  // length of the subspan
                                 tree_size_bound );         // bound on the tree size
  }
```

```
template < class T >
inline claimable
tree_matches_subspan(   tree_view<T> t,    span<T> s,
                        size_t start,    size_t count,    size_t bound   )
  {
   claim count <= bound;
   require t.empty() == ( count == 0u );

   if ( count != 0u )
     {
      const auto [ fc, ac ] = fore_and_aft_counts( count, bound );
      const auto m          = start + fc;
      const auto hb         = half( bound );

      require tree_matches_subspan(   t.fore(),              s, start,   fc,  hb );
      require equal_and_substitutable( t.middle(), &s[m] );
      require tree_matches_subspan(   t.aft(),               s, m+1u, ac, hb );
     }
  }
```

```
std::tuple< size_t, size_t >
fore_and_aft_counts( const size_t count, const size_t bound )
interface
  {
   claim usable( count, bound );
   claim count != 0u;
   claim count <= bound;

   implementation;
   const auto [ fc, ac ] = result;

   claim fc + 1u + ac == count;

   const auto hb = half( bound );
   claim fc <= hb;
   claim ac <= hb;
   claim usable( count, bound, result );
  }
```

implementation	Notable postconditions
`std::tuple< size_t, size_t >`	
`fore_and_aft_counts( const size_t count, const size_t bound )`	
`implementation`	
`{`	
`const auto r    = count % 2u;`	r <= 1
`const auto fc  = half( count );`	fc + fc + r == count
`const auto ac = count - ( fc+1 );`	fc + 1 + ac == count
`const auto hb = half( bound );`	
`associative_add( fc, 1u, ac );`	fc + (1 + ac) == count
`claim fc+r == count-fc && count-fc == ac+1u;`	fc + r == ac + 1
`associative_add( ac, 1u-r, r );`	ac + (1-r) == (ac+1) - r
`claim ac + (1-r) == fc;`	ac <= fc
`ordered_halves( count, bound );`	fc <= hb
`transitively_ordered( ac, fc, hb );`	ac <= hb
`return { fc, ac };`	
`}`	

```
void associative_add( size_t a,
                      size_t b,
                      size_t c )
interface
  {
   const auto sum = (a + b) + c;
   claim implementation;
   claim sum == a + (b + c);
  }
```

```
void ordered_halves( size_t a, size_t b )
interface
  {
   claim a <= b;
   claim implementation;
   claim half( a ) <= half( b );
  }
```

```
void transitively_ordered( size_t a,
                           size_t b,
                           size_t c )
interface
  {
   claim  a <= b  &&  b <= c;
   claim implementation;
   claim a <= c;
   claim ( a < c ) == ( a < b || b < c );
  }
```

Interfaces operate through repetition.

The prologue repeats expressions from the calling neighborhood, so that they may be repeated again in the implementation neighborhood.

The epilogue repeats expressions from the implementation neighborhood, so that they may be repeated again in the calling neighborhood.

The complexity of interfaces matters.

The complexity of an interface should be no more than that of its implementation.*

*Complexity measured by counting locally atomic operations.

The complexity of interfaces matters.

The complexity of an interface should be **the same as** that of its implementation.*

*Complexity measured by counting locally atomic operations.

Questions?