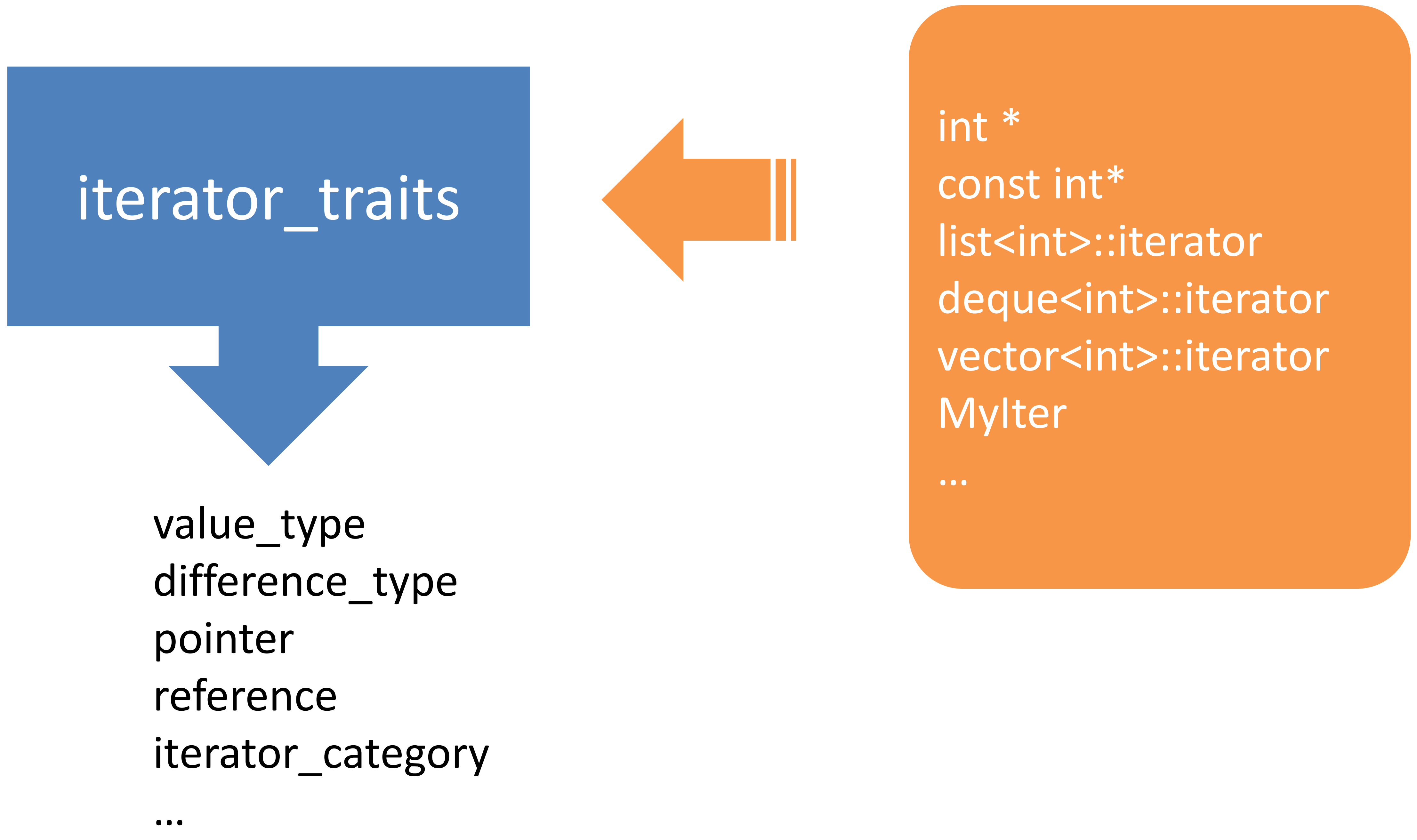
Standard traits in STL

The standard traits technique in STL:



Iterators

Iterator category (types):

- `InputIterator`
- `OutputIterator`
- `ForwardIterator`
- `BidirectionalIterator`
- `RandomAccessIterator`

Iterators

Iterator category (types):

$$\begin{tikzcd}
\texttt{InputIterator} \arrow[dr] & & \texttt{OutputIterator} \arrow[dl] \\
& \texttt{ForwardIterator} \arrow[d] & \\
& \texttt{BidirectionalIterator} \arrow[d] & \\
& \texttt{RandomAccessIterator} &
\end{tikzcd}$$

Iterators

Iterator methods: `advance()`

```
template <class InputIterator, class Distance>
void advance_II(InputIterator &i, Distance n)
{
  while (n--) ++i;
}
```

Iterators

Iterator methods: `advance()`

```
template <class BidirectionalIterator, class Distance>
void advance_BI(BidirectionalIterator &i, Distance n)
{
  if (n >= 0)
    while (n--) ++i;
  else
    while (n++) --i;
}
```

Iterators

Iterator methods: `advance()`

```
template <class RandomAccessIterator, class Distance>
void advance_RAI(RandomAccessIterator &i, Distance n)
{
   i += n;
}
```

Iterators

Iterator methods: `advance()`

But how to call them according to iterator types?

Iterators

Use iterator category information:

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag : public
  input_iterator_tag {};
struct bidirectional_iterator_tag : public
  forward_iterator_tag {};
struct random_access_iterator_tag : public
  bidirectional_iterator_tag {};
```

Iterators

Iterator methods: `advance()`

```
template <class InputIterator, class Distance>
inline void __advance(InputIterator &i, Distance n,
                      input_iterator_tag)
{
  while (n--) ++i;
}
```

Iterators

Iterator methods: `advance()`

```
template <class BidirectionalIterator, class Distance>
inline void __advance(BidirectionalIterator &i,
                      Distance n,
                      bidirectional_iterator_tag)
{
 if (n >= 0)
    while (n--) ++i;
  else
    while (n++) --i;
}
```

Iterators

Iterator methods: `advance()`

```
template <class RandomAccessIterator, class Distance>
inline void __advance(RandomAccessIterator &i,
                      Distance n,
                      random_access_iterator_tag)
{
  i += n;
}
```

Iterators

Use traits again!

```
template <class Iterator, class Distance>
inline void advance(Iterator &i, Distance n)
{
   __advance(i, n,
     iterator_traits<Iterator>::iterator_category());
}
```

Iterators

Use traits again!

```
template <class Iterator, class Distance>
inline void advance(Iterator &i, Distance n)
{
  __advance(i, n,
    iterator_traits<Iterator>::iterator_category());
}
```

Temporary object

Iterators

Partial specialization for raw pointers

```
template <class I>
struct iterator_traits {
  …
  typedef typename I::iterator_category iterator_category;
};

template <class T>
struct iterator_traits<T*> {
  …
  typedef random_access_iterator_tag iterator_category;
};
```

Iterators

Pure transfer can be removed due to inheritance

```
template <class ForwardIterator, class Distance>
inline void __advance(ForwardIterator &i, Distance n,
                      forward_iterator_tag)
{                                           ↓   Implicit conversion
  __advance(i, n, input_iterator_tag());
}
```

Iterators

Iterator methods: `distance()`

```
template <class InputIterator>
inline iterator_traits<InputIterator>::difference_type
__distance(InputIterator first, InputIterator last,
           input_iterator_tag)
{
  iterator_traits<InputIterator>::difference_type n=0;
  while (first != last) {
    ++first; ++n;
  }
  return n;
}
```

Iterators

Iterator methods: `distance()`

```
template <class RandomAccessIterator>
inline iterator_traits<InputIterator >::difference_type
__distance(RandomAccessIterator first,
           RandomAccessIterator last,
           random_access_iterator_tag)
{
  return last - first;
}
```

Iterators

Iterator methods: `distance()`

```
template <class Iterator>
inline iterator_traits<Iterator>::difference_type
distance(Iterator first, Iterator last)
{
  return __distance(first, last,
    iterator_traits<Iterator>::iterator_category());
}
```

Iterators

- Container knows how to design its own iterator.
- Traits trick extracts type information embedded in different iterators, including raw pointers.
- Algorithms are independent to containers through the design philosophy of iterators.