Policy Adaptors and the Boost Iterator Adaptor Library

David Abrahams, www.boost.org
Jeremy Siek, Indiana University

Overview

- Motivation
- Design of the Library
- Implementation of the Library
- Metaprogramming Details
- Policy Adaptors
- Conclusion

Motivation

- Container of polymorphic objects
  - Use a container of pointers?

    ```
    vector<Base*>
    ```

  - Better: create a wrapper to make it look like a container of objects (not pointers)
  - To implement, need an iterator adaptor to dereference the pointer

Motivation

- Iterator adaptors are everywhere
  - VTL – M. Weiser & G. Powell
  - Date Iterator – Jeff Garland
  - `std::reverse_iterator`
  - `Checked_iter` – Bjarne Stroustrup
  - Smart Iterators – T. Becker
  - Compound Iterators – A. Alexandrescu
  - MTL & BGL iterators

Motivation

- Nonetheless, programmers often give up on the idea of creating a new iterator, instead resorting to rewriting algorithms. Why?
- Building iterators is
  - Tedious
  - Subtle

Motivation

- Building iterators is tedious
  - Redundant operators
    - Prefix and Postfix: `++` `--`
    - Dereferencing: `*` `->` `[]`
    - Comparison: `==` `!=` `<` `<=` `>` `>=`
  - Redundant type information
    - `reference == value_type&`
    - `pointer == value_type*`
  - Constant/mutable iterator interactions

Motivation

- Building iterator *adaptors* is tedious
  - Change one aspect of behavior
  - Leave other aspects the same
- Aspects:
  - Movement
  - Dereference
  - Equality Comparison
  - Dist Measurement
- Writing dispatching functions is boring

Motivation

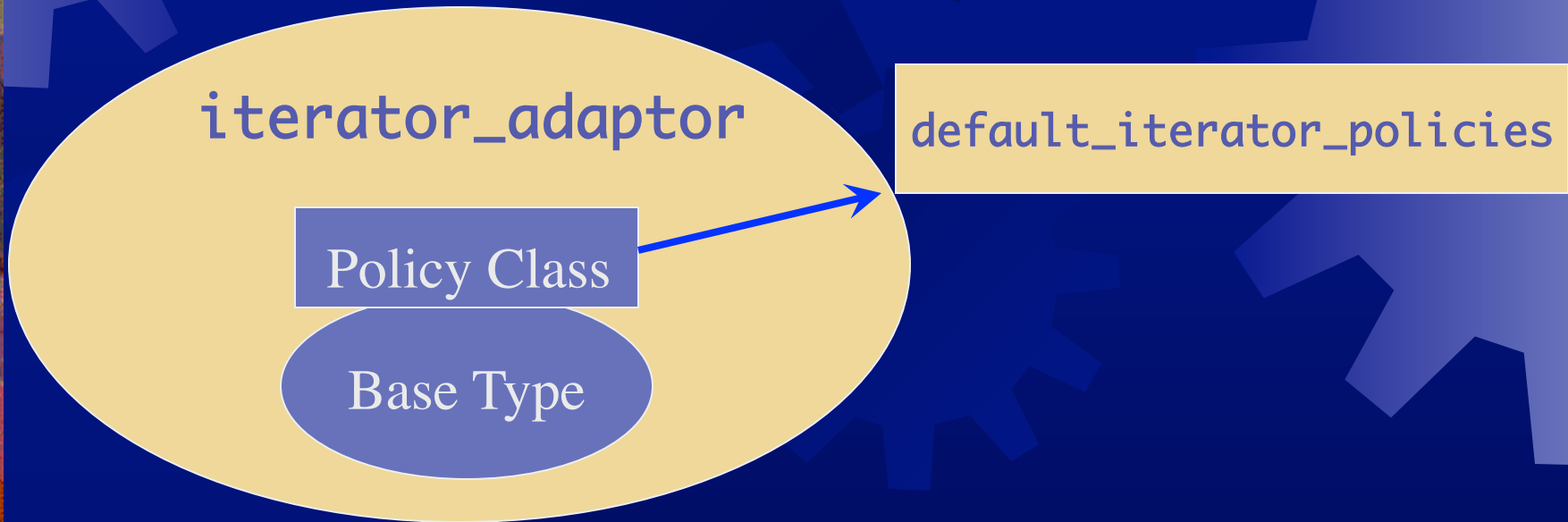
- Building iterators is hard to get right
  - `operator->` for input iterators
  - `operator[]` for adapted random access iterators

Overview

- Motivation
- Design of the Library
- Implementation of the Library
- Policy Adaptors
- Conclusion

Design

- `iterator_adaptor` class template
  - User supplies a policy class and Base type
  - `iterator_adaptor` generates a model of Random Access Iterator



Design

- Template parameters:
  - Base – the underlying adapted type
  - Policies – defines implementation of core behaviors
  - Associated Types: Value, Reference, Pointer, and Category
    - Sometimes deducible from Base (e.g. `reverse_iterator`)
    - But not always (e.g., `indirect_iterator`)

Design

- Default policies class forwards all operations to the Base type

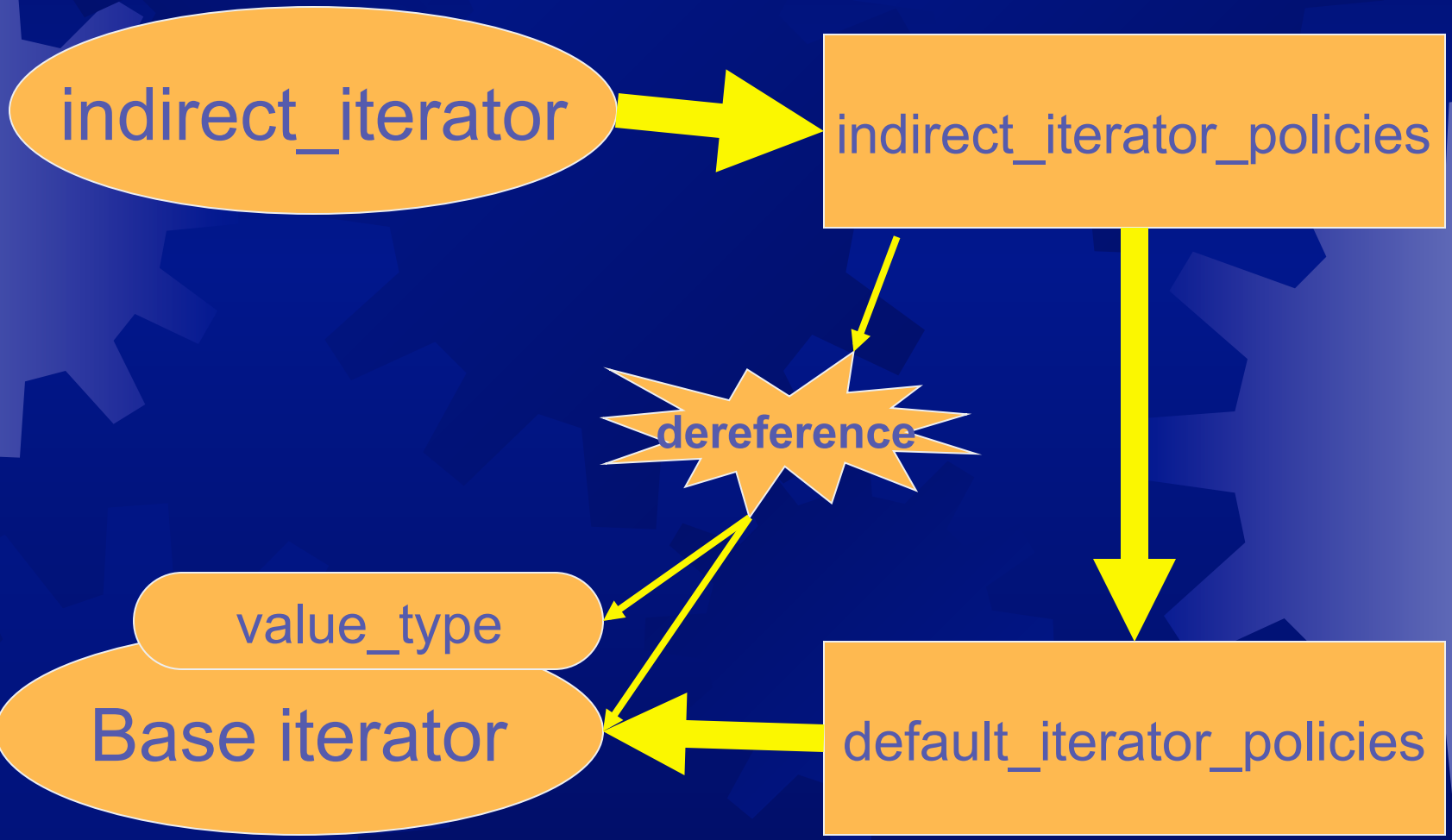
```
struct default_iterator_policies {
  template <class IterAdaptor>
  typename IterAdaptor::reference
  dereference(const IterAdaptor& x) const
      { return *x.base(); }
  // ...
};
```

Example Use

- `indirect_iterator` solves container of pointers problem

```
struct indirect_iterator_policies
  : public default_iterator_policies
{
  template <class IterAdaptor>
  typename IterAdaptor::reference
  dereference(const IterAdaptor& x) const
    { return **x.base(); }
};
```

Behavior Delegation



Overview

- Motivation
- Design of the Library
- Implementation of the Library
- Policy Adaptors
- Conclusion

Implementation

- Redundant public interface reduced to core policies

```
reference operator*() const {
    return policies().dereference(*this);
}

pointer operator->() const {
    return &policies().dereference(*this);
}
```

Implementation

- `default_iterator_policies` forwards behavior to the underlying iterator

```
template <class IterAdaptor>
typename IterAdaptor::reference
dereference(IterAdaptor& x) {
  return *x.base();
}
// more of the same ...
```

Implementation

- Constant/mutable iterator interactions should work:

```
iterator a, b;
const_iterator ca, cb

a == b;   a - b
ca == b;  ca - cb
a == cb;  a - cb;
ca == cb; ca - cb;
```

Implementation

- All four combinations implemented with one function template

```
template <class B1, class B2, class P, class V1,
  class V2, class R1, class R2, class P1,
  class P2, class Cat, class Dist>
bool operator==(
  iterator_adaptor<B1,Policies,V1,R1,P1,Cat,Dist> x,
  iterator_adaptor<B2,Policies,V2,R2,P2,Cat,Dist> y)
{
    return x.policies().equal(x.iter(), y.iter());
}
```

Implementation

- Redundant associated types handled via smart defaults
  - Value type, iterator category, and difference type obtained from `std::iterator_traits<Base>`
  - Reference and pointer type: if Value explicitly specified then Value&, Value*, otherwise use `std::iterator_traits<Base>`
  - `remove_const<Value>` → `value_type`

Implementation

- Input iterator dereference may return by-value
- `operator->` would return a pointer to a temporary!

```
pointer operator->() const {
  return &policies().dereference(*this);
}
```

- Instead return a proxy object that contains the value and has an `operator->`.
- Proxy has lifetime of the full expression

Implementation

- Naïve implementation of `operator[]`

```
reference operator[](difference_type n)
  { return *(*this+n); }
```

- Consider a disk-based iterator that reads in and caches the object.
- `(*this + n)` creates a temporary iterator
- Safe solution: return by value

Implementation

- May need additional state
- Example: `strided_iterator` needs to store stride
- Solution: store state in policies class; store instance of policies class in `iterator_adaptor`

Overview

- Motivation
- Design of the Library
- Implementation of the Library
- Policy Adaptors
- Conclusion

Policy Adaptors

- A Design Pattern
- Generates new models of a concept (or a family of concepts)
- Generates adaptors to selectively modify a behavioral aspect of any model of the concept.

Policy Adaptors

1. Identify core aspects of behavior.

   For iterators:

   - Traversal
   - Dereferencing
   - Equality comparison
   - Distance measurement
   - Associated type exposure

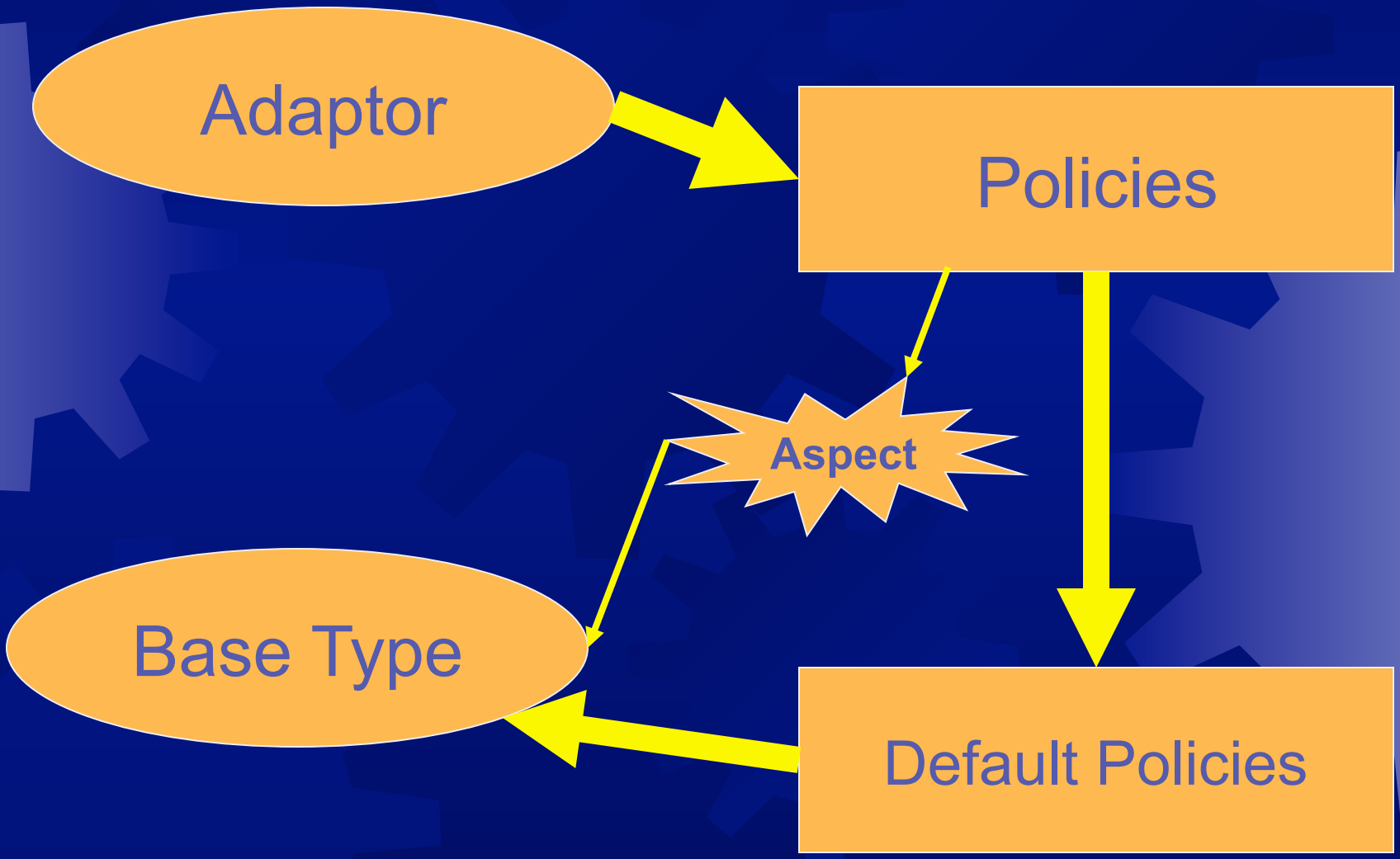
2. Define policy interface that encapsulates the core aspects

Policy Adaptors

3. Write default policies class that dispatches to concept's public interface

4. Build the adaptor class
   - Generalized model of concept
   - Parameterized on Policies class and Base type
   - Public interface delegates work to Policies class
   - Stores Policies instance as a data member

Behavior Delegation



Policy Adaptors

- Policy adaptor design pattern applicable when:
  - Concept has multiple orthogonal aspects
  - Concepts with “rich” interface (redundant functions for ease of use)
  - Concept is popular, i.e., it is a common task to create new models of the concept

Overview

- Motivation
- Design of the Library
- Implementation of the Library
- Policy Adaptors
- Conclusion

Conclusion

- Boost Iterator Adaptor Library automates creation of iterators
  - Factors out functional redundancy
  - Change one aspect while reusing others
- Makes simple ideas easy to implement
- Policy adaptors are a powerful design pattern for creating models of rich concepts