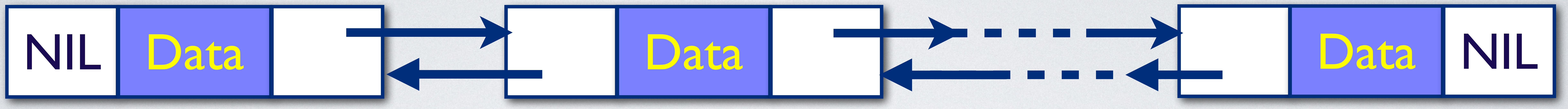
DOUBLY-LINKED LIST



- A doubly-linked list is a sequence of data items, each connected by two links called next and previous.

- A data item may be a primitive value, a composite value, or even another pointer.

- Traversal in a double-linked list is bidirectional.

- Deleting of a node at either end of a doubly-linked list is straight forward.

A DOUBLY-LINKED LIST REPRESENTATION WITH SHARED POINTERS

```
template<typename T>
class DoublyLinkedList
{
public:
    using Node = std::shared_ptr<DoublyLinkedList<T>>;

    T fData;
    std::shared_ptr<DoublyLinkedList<T>> fNext;
    std::weak_ptr<DoublyLinkedList<T>> fPrevious;

    DoublyLinkedList( const T& aData ) noexcept : fData(aData), fNext(), fPrevious()
    {}

    DoublyLinkedList( T&& aData ) noexcept : fData(std::move(aData)), fNext(), fPrevious()
    {}

    void isolate() noexcept;                    // unlink node

    // factory method for list nodes
    template<typename... Args>
    static Node makeNode( Args&&... args );
};
```

shared pointer

overloaded constructors

DOUBLY-LINKED LIST NODE ISOLATION

```
void isolate() noexcept
{
    if ( fNext )                          // Is there a next node?
    {
        fNext->fPrevious = fPrevious;
    }

    Node lNode = fPrevious.lock();        // lock std::weak_ptr

    if ( lNode )                          // Is there a previous node?
    {
        lNode->fNext = fNext;
    }

    fPrevious.reset();
    fNext.reset();
}
```

clear smart pointer references

DOUBLY-LINKED LIST ITERATOR SPECIFICATION

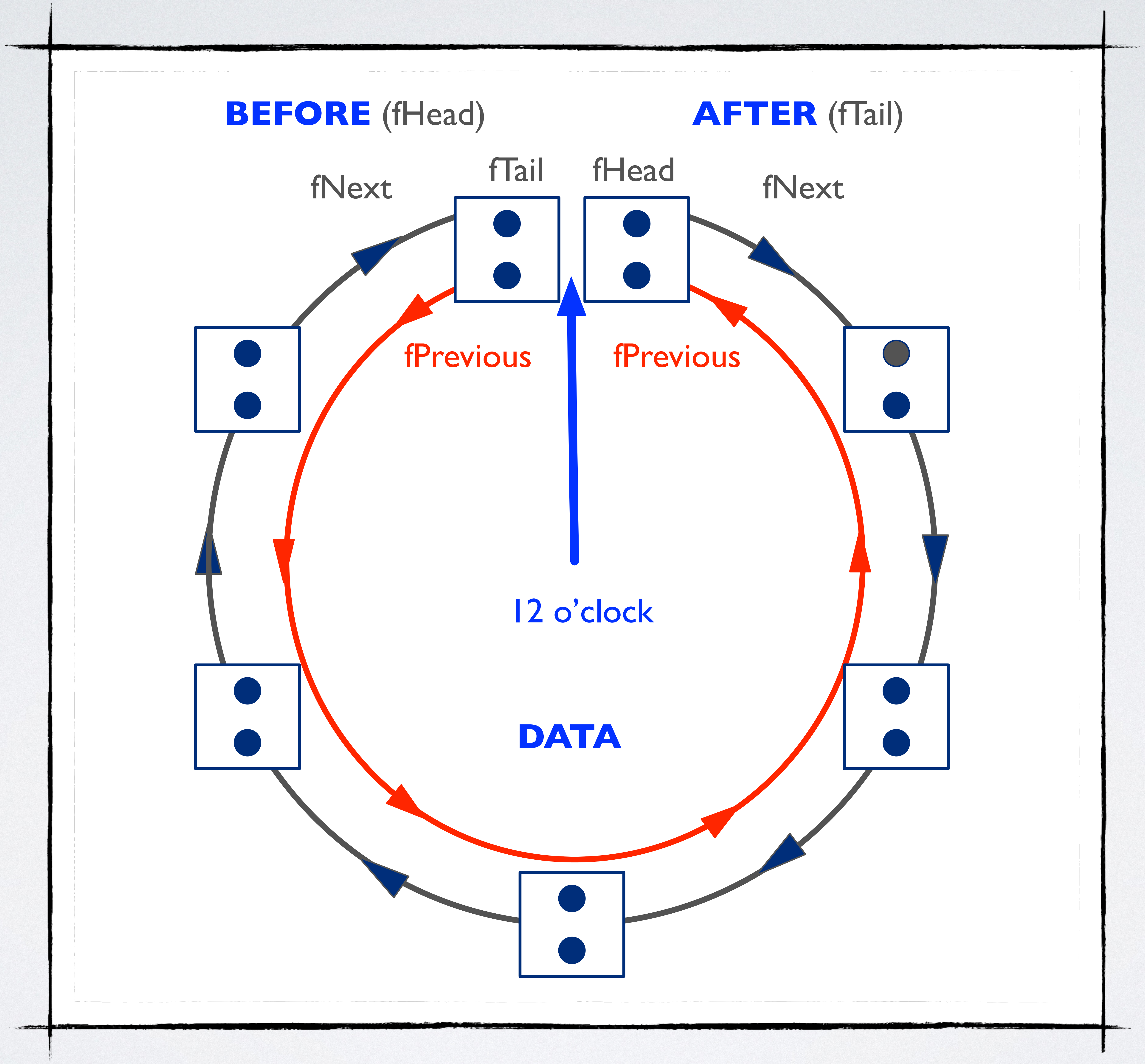
```
template<typename T>
class DoublyLinkedListIterator
{
public:
    using Iterator = DoublyLinkedListIterator<T>;
    using Node = typename DoublyLinkedList<T>::Node;

    enum class States { BEFORE, DATA, AFTER };

    DoublyLinkedListIterator( const Node& aHead, const Node& aTail ) noexcept;

    const T& operator*() const noexcept;
    Iterator& operator++() noexcept;                  // prefix
    Iterator operator++(int) noexcept;                // postfix
    Iterator& operator--() noexcept;                  // prefix
    Iterator operator--(int) noexcept;                // postfix

    bool operator==( const Iterator& aOther ) const noexcept;
    bool operator!=( const Iterator& aOther ) const noexcept;

    Iterator begin() const noexcept;
    Iterator end() const noexcept;
    Iterator rbegin() const noexcept;
    Iterator rend() const noexcept;

private:
    Node fHead;
    Node fTail;
    Node fCurrent;
    States fState;
};
```

iterator states

bidirectional iterator

iterator auxiliaries

ITERATOR MOVING AROUND THE CLOCK



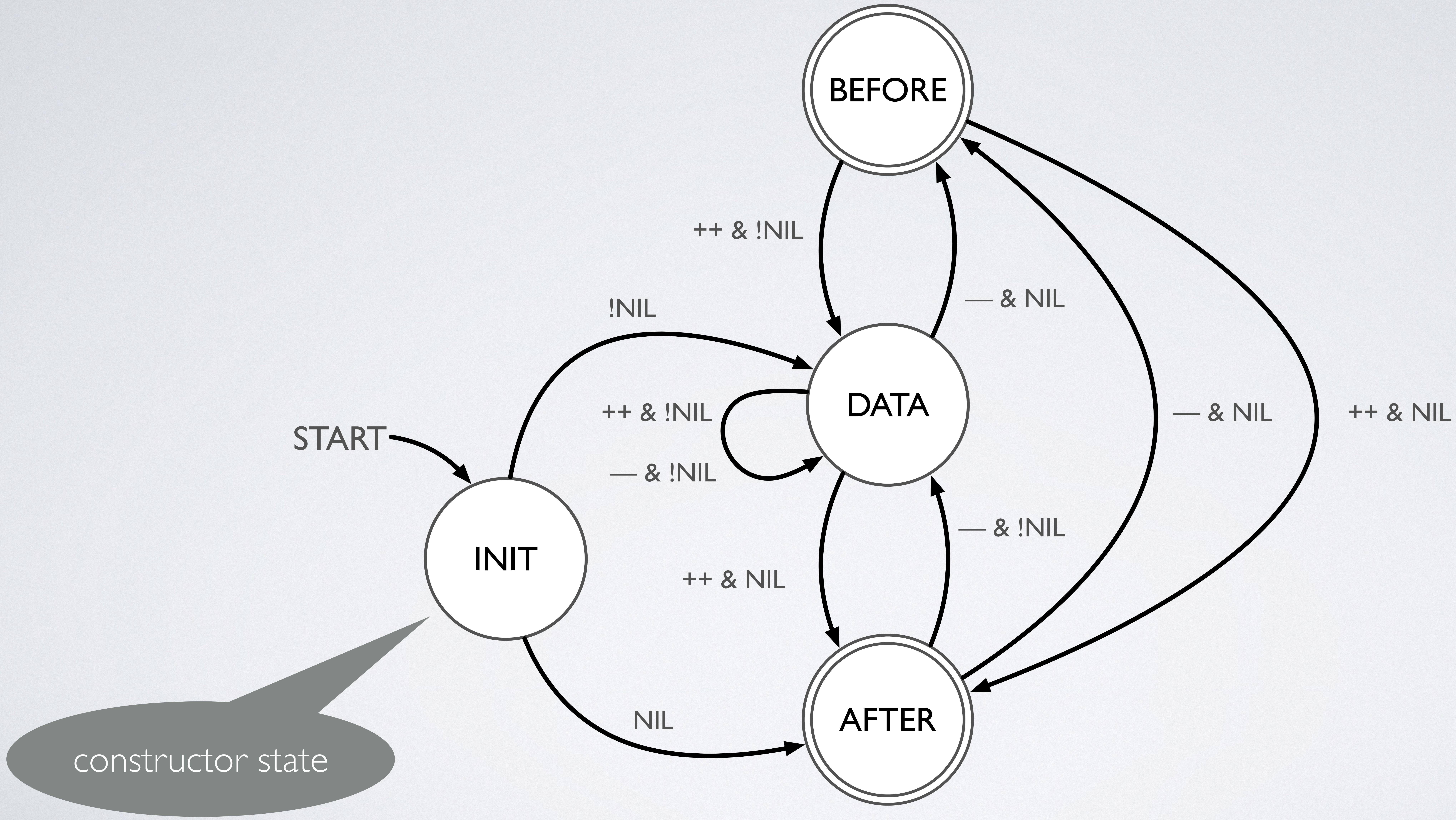
DETERMINISTIC FINITE AUTOMATA

- A deterministic finite automaton is a quintuple $(\Sigma, Q, q_0, \sigma, F)$ with:
  - a finite, non-empty set Σ of actions (input alphabet),
  - a non-empty set $Q = \{q_0, q_1, \ldots\}$ of states,
  - a subset $F \subseteq Q$, called the accepting states,
  - a function $\sigma = Q \times \Sigma \times Q$, called the transition relation,
  - a designated initial state q_0.
- A transition $(q, a, q') \in \sigma$ is usually written $q \xrightarrow{a} q'$
- Note, we require a DFA to be enabled on all actions in all states, that is, the transition relation σ is defined for all pairs $(a, q) \in \Sigma \times Q$.

TRANSITION DIAGRAMS

- A transition diagram for a DFA $A = (\Sigma, Q, q_0, \sigma, F)$ is a graph defined as follows:
  - For each state in Q there is a node.
  - For each state $q \in Q$ and each action $a \in \Sigma$, let $\sigma(q, a) = p$. Then the transition diagram has an arc from node q to node p, labeled a. If there are several actions that cause transitions from q to p, then the transition diagram can have one arc, labeled by the list of these actions.
  - There is an arrow to the start state q_0, labeled Start. This arrow does not originate at any node.
  - Nodes corresponding to accepting states (those in F) are marked by a double circle. States not in F have a single circle.

LIST ITERATOR STATE TRANSITION DIAGRAM



STATE TRANSITIONS AS SWITCH STATEMENT

```
switch ( fState )
{
   case States::BEFORE:

      // BEFORE logic

      break;

   case States::DATA:

      // DATA logic

      break;

   case States::AFTER:

      // AFTER logic

      break;
}
```

exhaustive case analysis

- In every state, we first have to inspect the current position of the iterator and second perform a state transition to the next state.
- The transition logic can be empty, if the iterator is already in an end position.
- The iterator can "hop onto" a list from either end using the corresponding endpoints passed to the iterator.
- In C++, the **break** statement is optional, that is, the compiler does not report an error if it is missing. The resulting fall though are a source of hard-to-find defects. Always end a case with **break**, if no fall through is permitted.
- A switch statement usually requires a **default** case unless you perform an exhaustive case analysis.

ITERATOR ADVANCING FORWARD IN STATE DATA

```
case States::DATA:
    fCurrent = fCurrent->fNext;

    if ( !fCurrent )
    {
        fState = States::AFTER;
    }

    break;
```

- We advance the iterator forward along the next link.
- If the next node is NIL (i.e., the smart pointer fCurrent evaluates to false in a Boolean context), then the next state is AFTER. Otherwise, the iterator remains in state DATA.