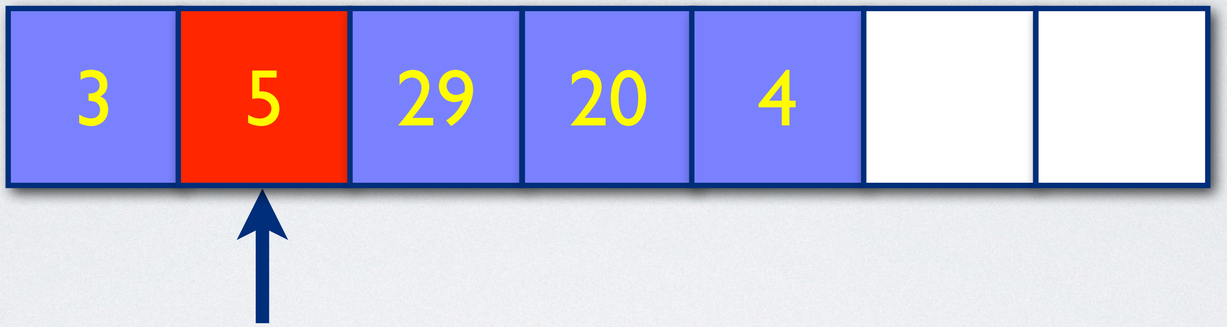
LINKED-LISTS

- **Overview**
  - Problems with Arrays
  - Linked Lists, Smart Pointers, List Iterators
- **References**
  - Richard F. Gilberg and Behrouz A. Forouzan: Data Structures - A Pseudocode Approach with C. 2nd Edition. Thomson (2005)
  - Russ Miller and Laurence Boxer: Algorithms Sequential & Parallel. 2nd Edition. Charles River Media Inc. (2005)
  - Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo: C++ Primer. 5th Edition. Addison-Wesley (2013)
  - Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, "Introduction to Algorithms", 4th Edition, The MIT Press (2022)
  - Scott Meyers: Effective Modern C++. O'Reilly (2014)
  - Anthony Williams: C++ Concurrency in Action - Practical Multithreading. Manning Publications Co. (2012)

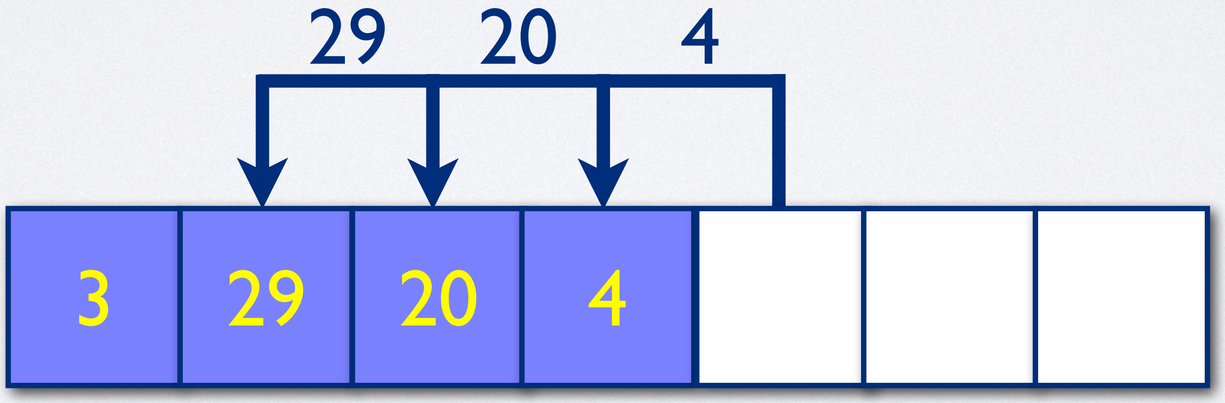
PROBLEMS WITH ARRAYS

- An array is a contiguous storage that provides insufficient abstractions for handling addition and deletion of elements.
- Addition and deletion require n/2 shifts on average.
- The computation time is O(n).
- Resizing affects performance.
- **Note 1**: Resizing might be an issue if we have to meet real-time deadlines, where any overrun is a failure.
- **Note 2**: Arrays exhibit several limitations but are better at exploiting memory locality. The elements in the array are contiguous in memory, which means lower latency when reading them. We should consider this when deciding which data structure to use and how to use it.

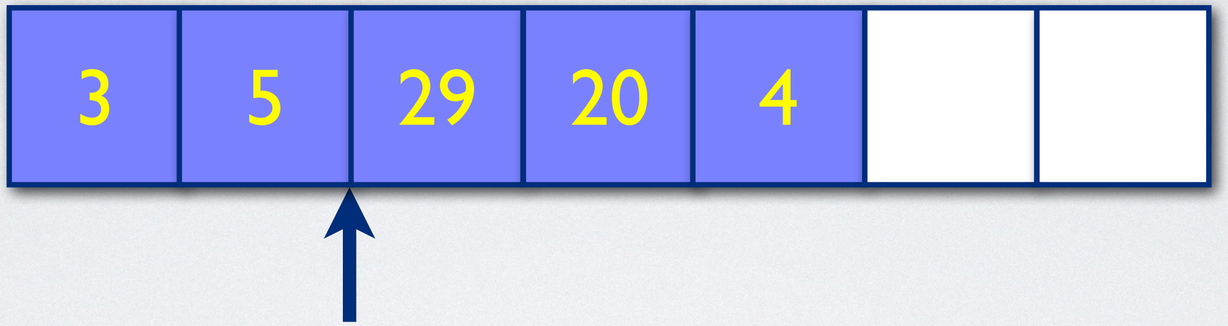
DELETION REQUIRES RELOCATION



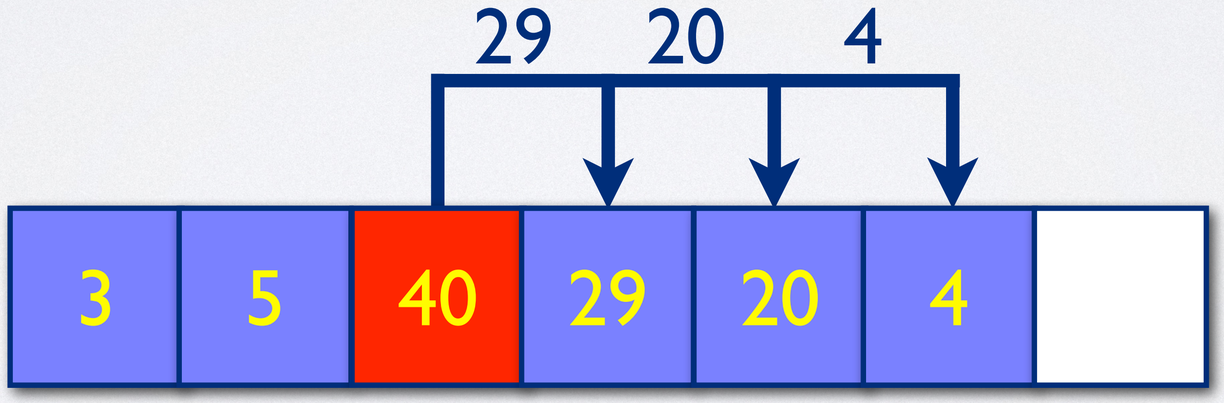
Delete 5



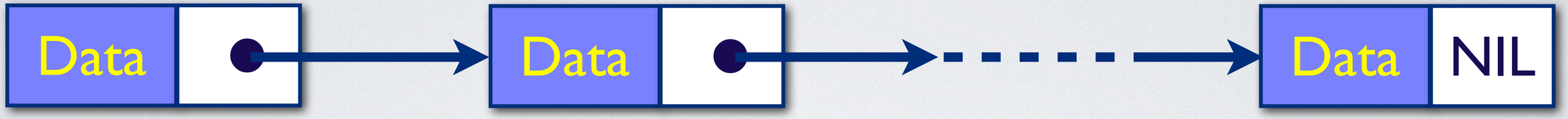
INSERTION REQUIRES RELOCATION



Insert 40 after 5



SINGLY LINKED LIST



- A singly linked list is a sequence of data items, each connected to the next by a pointer called next.
- A data item may be any object or even another pointer.
- A singly linked list is a recursive data structure whose nodes refer to nodes of the same type.

A SINGLY LINKED LIST REPRESENTATION

```
template<typename T>
struct SinglyLinkedList
{
    T data;
    SinglyLinkedList* next;

    SinglyLinkedList( const T& aData, SinglyLinkedList* aNext = nullptr ) noexcept :
        data(aData),
        next(aNext)
    {}

    SinglyLinkedList( T&& aData, SinglyLinkedList* aNext = nullptr ) noexcept :
        data(std::move(aData)),
        next(aNext)
    {}
};
```

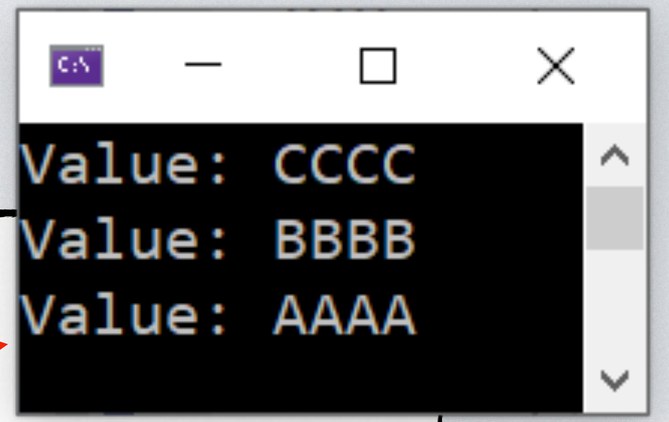
public field access

copy data

move data

A SINGLY LINKED LIST USE

```
using StringList = SinglyLinkedList<std::string>;

StringList One( "AAAA" );
StringList Two( "BBBB", &One );
StringList Three( "CCCC", &Two );

for ( StringList* lTop = &Three; lTop != nullptr; lTop = lTop->next )
{
    std::cout << "Value: " << lTop->data << std::endl;
}
```



LIMITATIONS OF CLASS TEMPLATE ARGUMENT DEDUCTION

```
SinglyLinkedList One( "AAAA" );
SinglyLinkedList Two( "BBBB", &One );
SinglyLinkedList Three( "CCCC", &Two );

for ( SinglyLinkedList* lTop = &Three; lTop != nullptr; lTop = lTop->next )
{
    std::cout << "Value: " << lTop->data << std::endl;
}
```

Does not compile!

cannot deduce class template arguments

- In the example above, we need to specify the class template argument for SinglyLinkedList. We can use a type alias specification for this purpose.

- Template arguments are deduced from function arguments. When we declare a raw pointer, we have no function arguments, hence, no template arguments can be deduced.

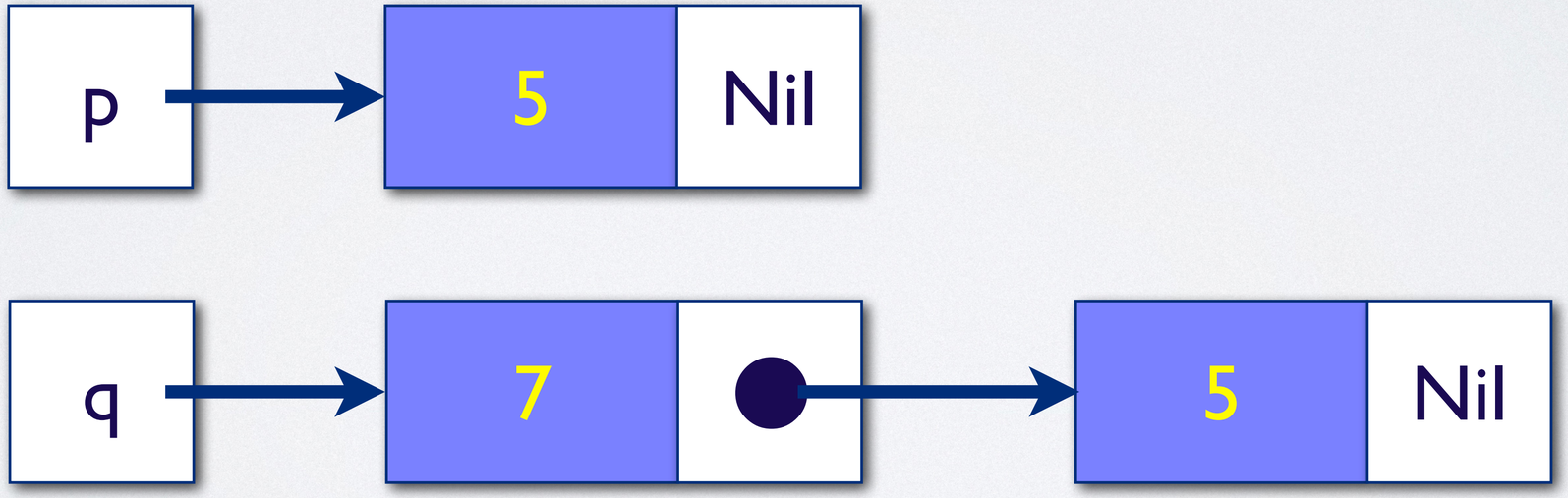
- Based on the constructor arguments, the deduced type is **char**[5]. We cannot use a c-string literal (i.e., a character array with five elements) to initialize the data member of SinglyLinkedList. C++ does not allow it. Hence, we explicitly specify the instantiation type to be std::string, a type that works with c-string literals.

LIST MANIPULATIONS

- A singly linked list is a dynamic data structure that uses forward pointers to the next element in the list.
- There is a dedicated pointer, called Nil, that terminates the list.
- Programmatically, we can use a **nullptr** to represent Nil.
- However, **nullptr** may not be the best option. Instead, we may use a sentinel object to denote the end of a list or any linked data structure.

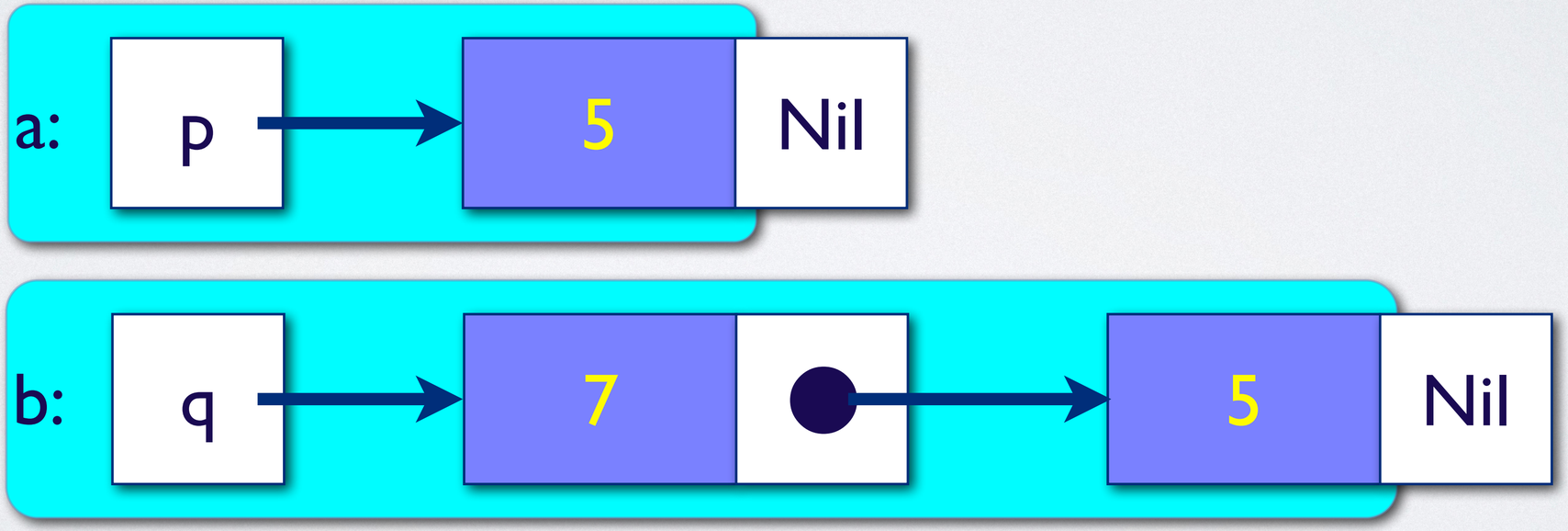
LIST NODE CONSTRUCTION

```
SinglyLinkedList<int> *p, *q;

p = new SinglyLinkedList( 5 );
q = new SinglyLinkedList( 7, p );
```



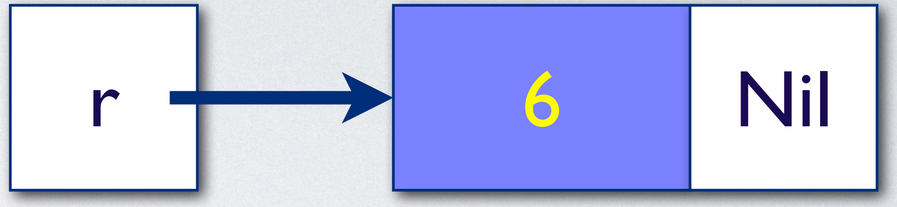
NODE ACCESS

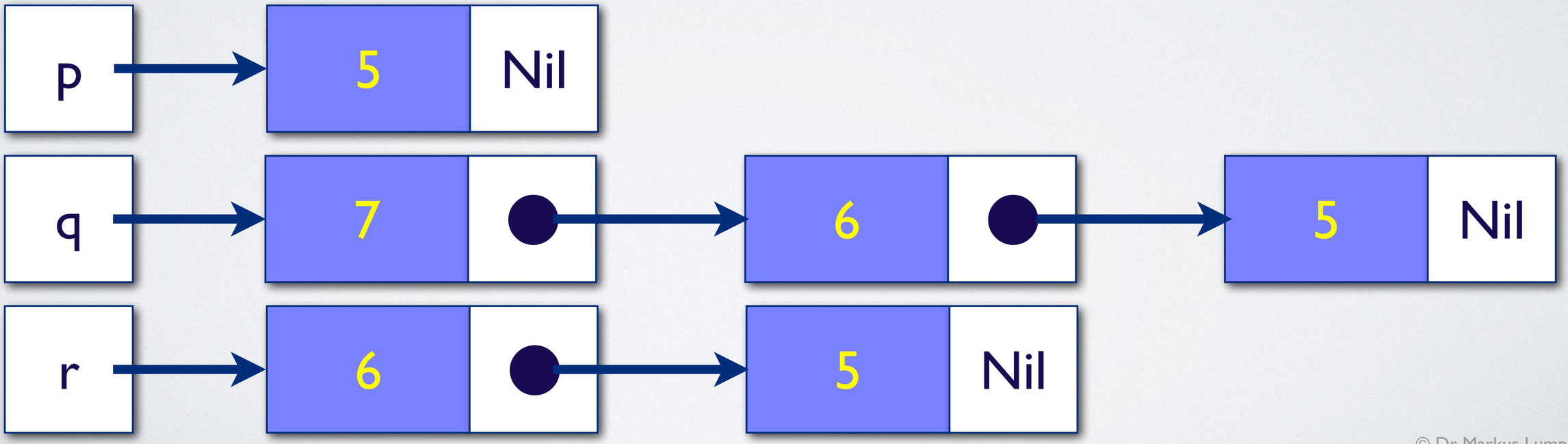
```
int a = p->data;
int b = q->next->data;
```



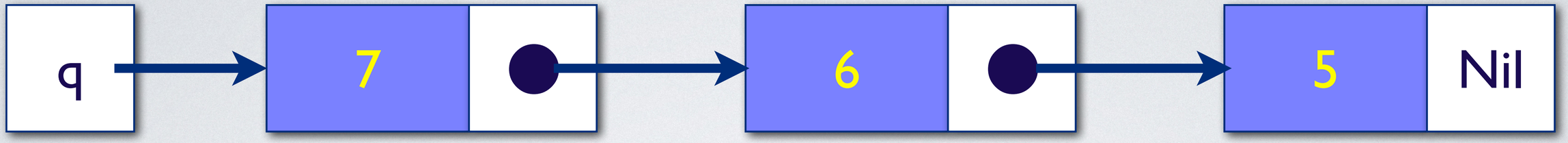
INSERT A NODE

```
SinglyLinkedList<int>* r = new SinglyLinkedList( 6 );

r->next = p;
q->next = r;
```

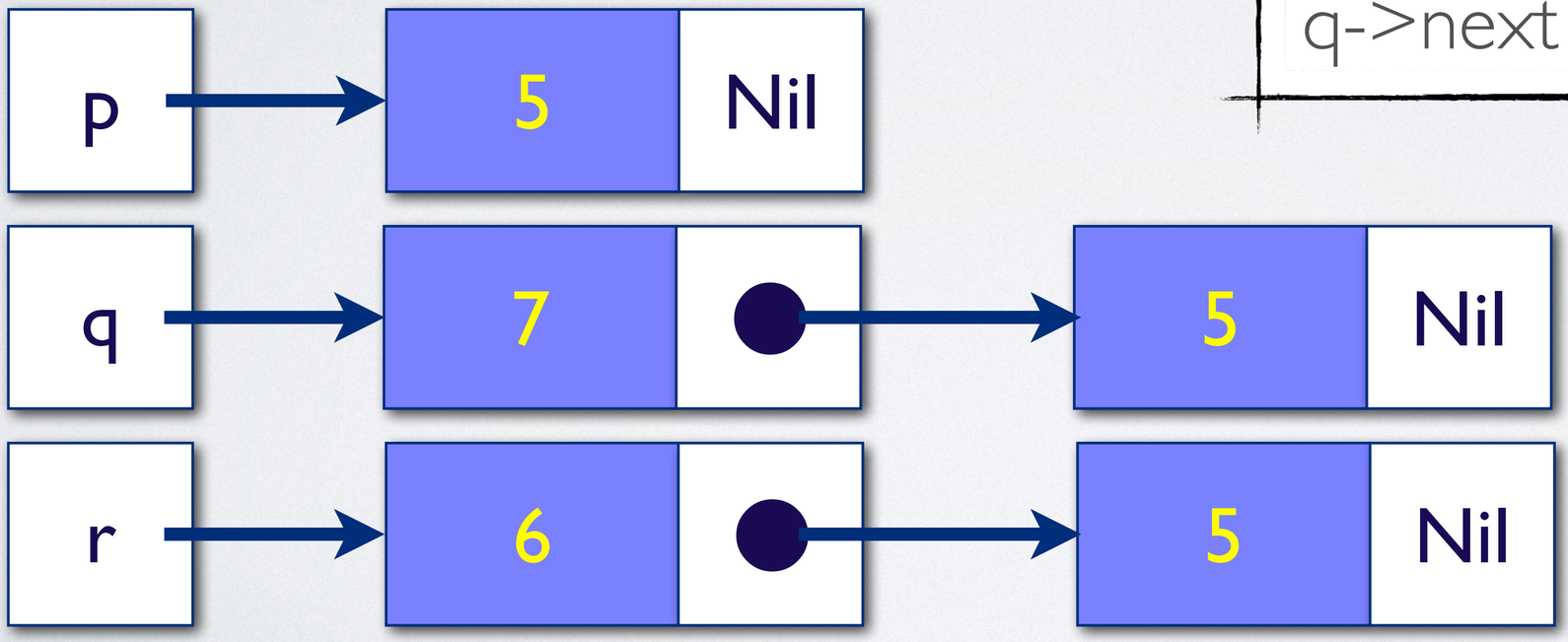




DELETE A NODE



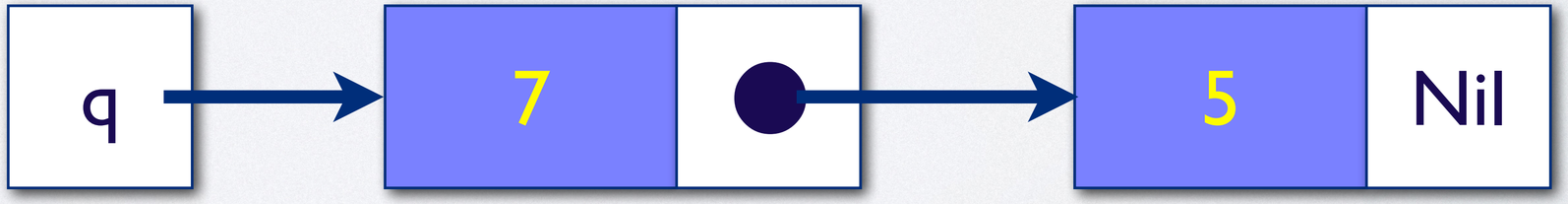

```
q->next = q->next->next;
```



INSERT A NODE AT THE TOP

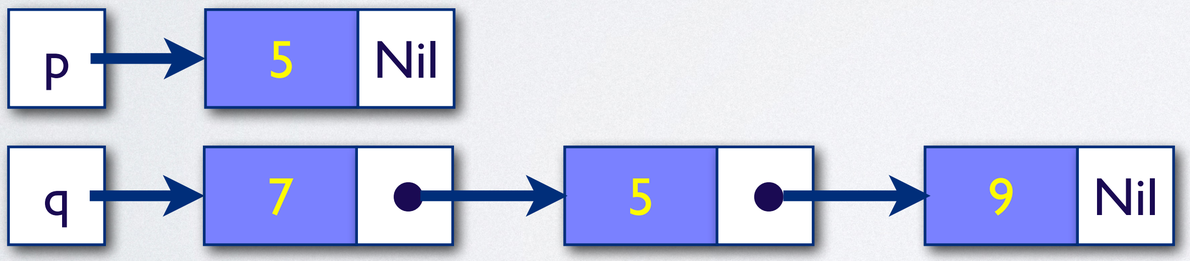
```
p = new SinglyLinkedList( 5 );
```


```
q = new SinglyLinkedList( 7, p );
```



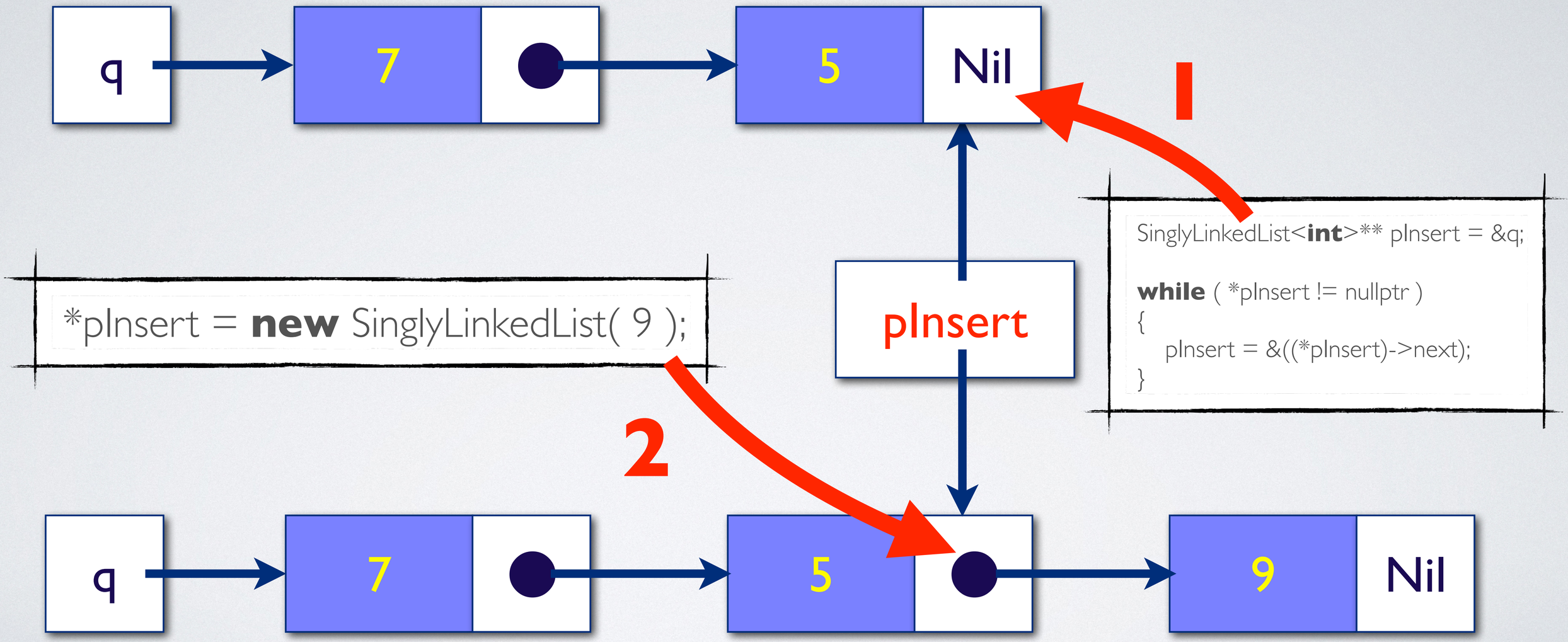
INSERT A NODE AT THE END

- To insert a new node at the end of a linked list, we need to search for the end:




```
SinglyLinkedList<int> *p, *q;

p = new SinglyLinkedList( 5 );
q = new SinglyLinkedList( 7, p );

SinglyLinkedList<int>** pInsert = &q;

while ( *pInsert != nullptr )
{
    pInsert = &((*pInsert)->next);
}

*pInsert = new SinglyLinkedList( 9 );
```

INSERT A NODE AT THE END: THE POINTERS



ELEMENT ORDER

- Insert at the end preserves the order of list elements while being added to the list.

INSERT A NODE AT THE END

- Rather than using a pointer to a pointer, we can record the last next pointer:

```
SinglyLinkedList<int>* pList = nullptr;
SinglyLinkedList<int>* pListLast = nullptr;

SinglyLinkedList<int>* pNewNode = new SinglyLinkedList( 5 );

if ( pList == nullptr )
    pList = pNewNode;
else
    pListLast->next = pNewNode;

pListLast = pNewNode;
```

CONSTRAINT OF SINGLY LIST LISTS

- To delete a node at the end of a list, we traverse it from the top to find the new last node.

REASONS RAW POINTERS ARE HARD TO LOVE

1. A pointer declaration does not indicate whether the pointer points to a single object or an array.

2. A pointer declaration reveals nothing about whether you should free the memory it points to when it goes out of scope. That is, whether the pointer owns the thing it points to.

3. If you have to free the memory a pointer points to, there is no way to tell how (i.e., delete or a different mechanism).

4. If delete is the right operation, reason one means it may be impossible to choose the correct form: single-object form, **delete**, or array form, **delete[]**.

5. If we know the pointer owns the thing and how to free the memory, it is still difficult to ensure that we free the memory exactly once along every path in our code.

6. There is typically no way to tell if a pointer dangles, that is, whether it points to memory already been freed.

RESOURCE ACQUISITION IS INITIALIZATION

- The Resource Acquisition Is Initialization (RAII) programming idiom ensures that resource acquisition occurs at the same time that the object is initialized.
- All resources for the object are created and made ready in one line of code.
- The main principle of RAII is to give ownership of any heap-allocated resource to a stack-allocated object whose destructor contains the code to delete or free the resource and also any associated cleanup code.
- The abstractions defined by the standard C++ library all adhere to RAII.

SMART POINTERS

- Smart pointers are defined in the std namespace in the <memory> header file. They are crucial to RAII.
  - std::unique_ptr<T>:

    Allows exactly one owner of the underlying pointer. Use as the default choice for plain old C++ objects (POCO). It can be moved to a new owner but not copied or shared.

  - std::shared_ptr<T>:

    Reference-counted smart pointer. Use it when you want to assign one raw pointer to multiple owners. The raw pointer is not deleted until all shared_ptr owners have gone out of scope or given up ownership otherwise.

  - std::weak_ptr<T>:

    A weak_ptr provides access to an object owned by one or more shared_ptr instances but does not participate in reference counting. Use when you want to observe an object but do not require it to remain alive. Breaks circular references between shared_ptr.

A SINGLY LINKED LIST REPRESENTATION WITH SHARED POINTERS

```
template<typename T>
struct SinglyLinkedListSP
{
    using node = std::shared_ptr<SinglyLinkedList>;

    T data;
    node next;

    SinglyLinkedListSP( const T& aData ) noexcept : data(aData), next(nullptr)
    {}

    SinglyLinkedListSP( const T& aData, node& aNext ) noexcept : data(aData), next(aNext)
    {}

    SinglyLinkedListSP( T&& aData ) noexcept : data(std::move(aData)), next(nullptr)
    {}

    SinglyLinkedListSP( T&& aData, node& aNext ) noexcept : data(std::move(aData)), next(aNext)
    {}
};
```

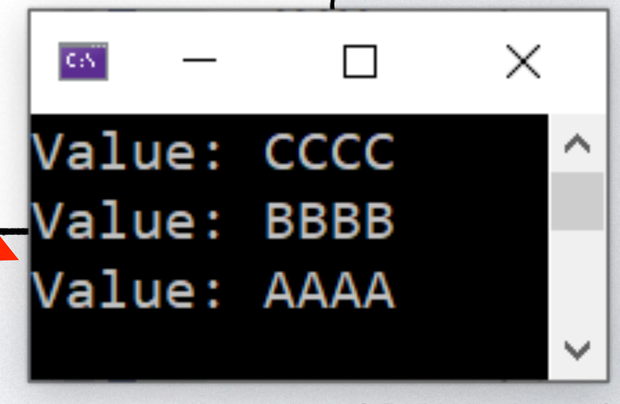
shared pointer

overloaded constructors

A SINGLY LINKED LIST WITH SHARED POINTERS USE

First attempt: It is quite verbose.

```
using StringList = SinglyLinkedListSP<std::string>;
using StringListNode = StringList::node;

StringListNode One = std::shared_ptr<StringList>( new StringList( "AAAA" ) );
StringListNode Two = std::shared_ptr<StringList>( new StringList( "BBBB", One ) );
StringListNode Three = std::shared_ptr<StringList>( new StringList( "CCCC", Two ) );

for ( StringListNode lTop = Three; lTop != nullptr; lTop = lTop->next )
{
    std::cout << "Value: " << lTop->data << std::endl;
}
```



STD::MAKE_UNIQUE AND STD::MAKE_SHARED

- To facilitate the construction of objects that are wrapped in an std::unique_ptr and std::shared_ptr, respectively, the standard library provides two auxiliary functions:
  - std::make_unique: Constructs an object of type T wrapped in a std::unique_ptr.
  - std::make_shared: Constructs an object of type T wrapped in a std::shared_ptr.
- Prefer std::make_unique and std::make_shared to the direct use of **new**. Both use perfect forwarding of their parameters to the constructor of the object being created.
- **Note**: C++17 only allows object creation. C++20 extends it to arrays also.

USING A FACTORY METHOD WITH PERFECT FORWARDING

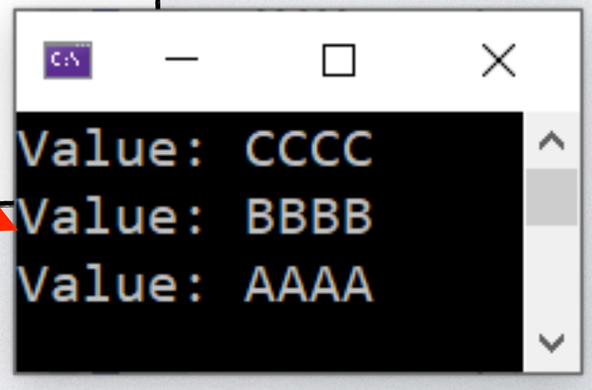
```
template<typename... Args>
static node makeNode( Args&&... args )
{
    // make_share<T, Args...>
   return std::make_shared<SinglyLinkedListSP>( std::forward<Args>(args)... );
}
```

- Factory Method is a creational design pattern that guides the design of concrete implementations of a common interface. It separates object creation from the code controlling it.
- Variadic template functions and perfect forwarding provide a convenient way to define factory methods.
- The static member function makeNode() takes a variable number of arguments. These arguments and their types are perfectly forwarded to std::make_shared, which calls the matching object constructor and wraps the object in an std::shared_ptr.

A SINGLY LINKED LIST WITH FACTORY METHOD

```
using StringList = SinglyLinkedListSP<std::string>;
using StringListNode = StringList::node;

StringListNode One = StringList::makeNode( "AAAA" );
StringListNode Two = StringList::makeNode( "BBBB", One );
StringListNode Three = StringList::makeNode( "CCCC", Two );

for ( StringListNode lTop = Three; lTop != nullptr; lTop = lTop->next )
{
    std::cout << "Value: " << lTop->data << std::endl;
}
```

Use of factory method



SINGLY LINKED LIST ITERATOR

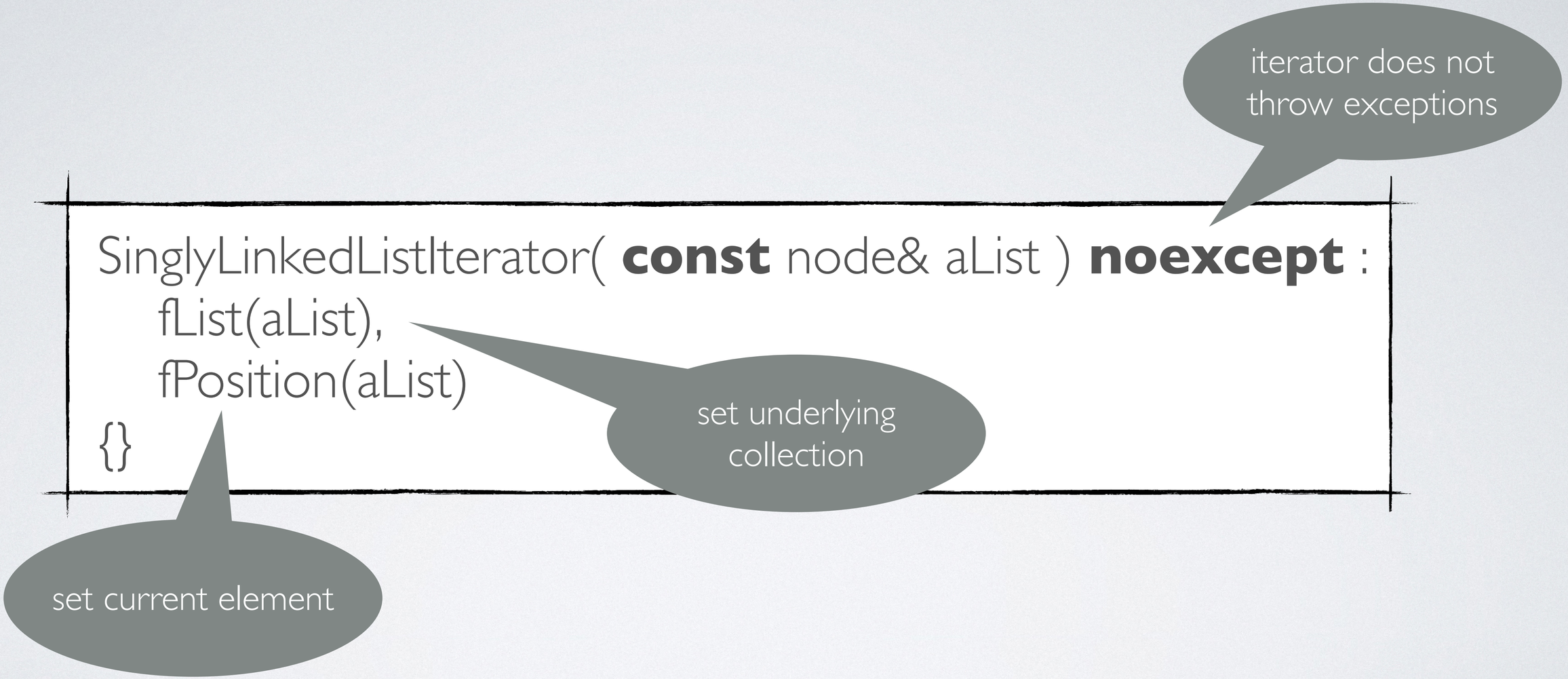
SINGLY LINKED LIST ITERATOR SPECIFICATION

```
template<typename T>
class SinglyLinkedListIterator
{
public:
    using node = std::shared_ptr<SinglyLinkedListSP<T>>;          // node is a dependent member type name
    using iterator = SinglyLinkedListIterator<T>;
    using difference_type = std::ptrdiff_t;                       // to satisfy concept weakly_incrementable
    using value_type = T;                                         // to satisfy concept indirectly_readable

    SinglyLinkedListIterator() noexcept {}
    SinglyLinkedListIterator( const node& aList ) noexcept;

    const T& operator*() const noexcept;
    iterator& operator++() noexcept;                              // prefix increment
    iterator operator++(int) noexcept;                            // postfix increment
    bool operator==( const iterator& aOther ) const noexcept;

    iterator begin() const noexcept;
    iterator end() const noexcept;

private:
    node fList;
    node fPosition;
};
```

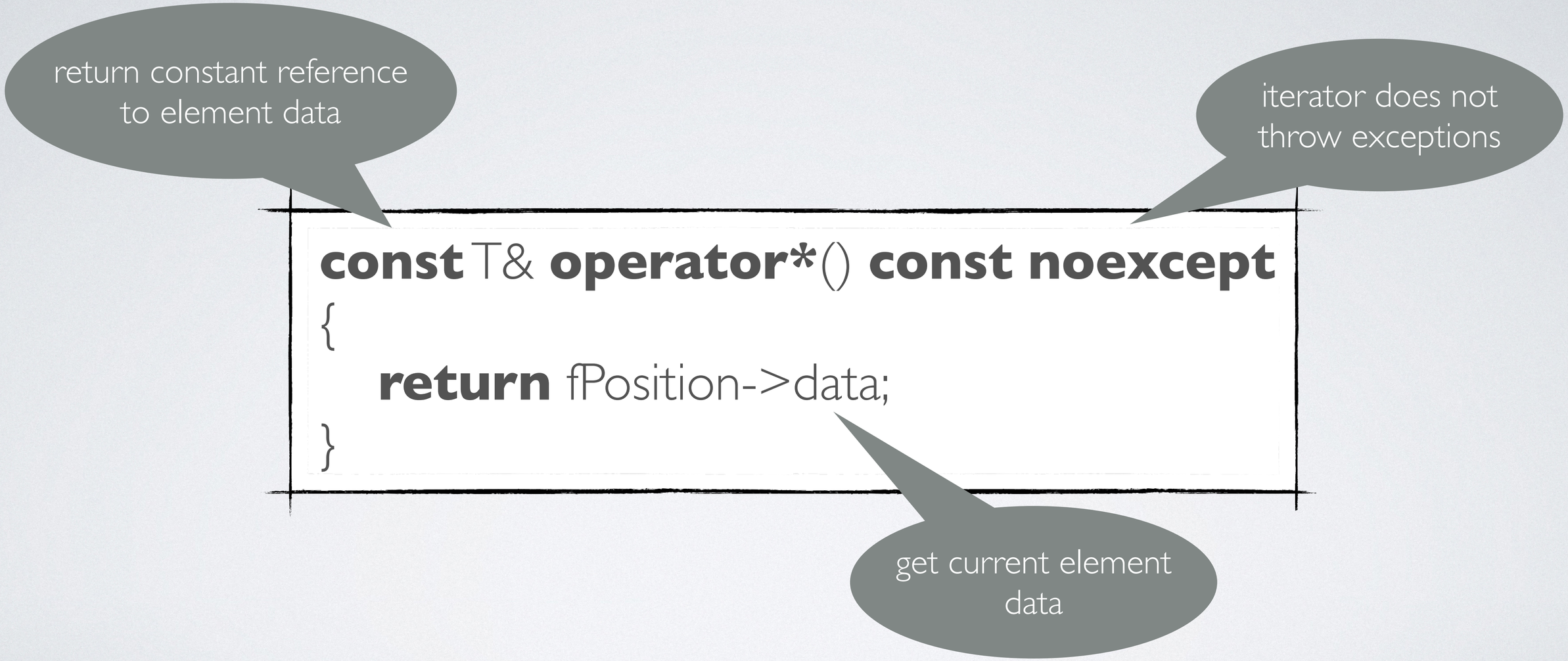
Forward iterator

```
static_assert( BasicBidirectionalIterator<SinglyLinkedListIterator<int>>);
```

SINGLY LINKED LIST ITERATOR CONSTRUCTOR



SINGLY LINKED LIST ITERATOR DEREFERENCE



SINGLY LINKED LIST ITERATOR EQUIVALENCE

```
iterator& operator++() noexcept     // prefix
{
    fPosition = fPosition->next;

    return *this;
}

iterator operator++(int) noexcept // postfix
{
    iterator old = *this;

    ++(*this);

    return old;
}
```

advance pointer

reuse implementation

SINGLY LINKED LIST ITERATOR INCREMENT

```
bool operator==( const Iterator& aOther ) const noexcept
{
    return
        fList == aOther.fList &&
        fPosition == aOther.fPosition;
}

bool operator!=( const Iterator& aOther ) const noexcept
{
    return !(*this == aOther);
}
```

same index

same underlying collection

reuse implementation

SINGLY LINKED LIST ITERATOR RANGE-LOOP METHODS

```
iterator begin() const noexcept
{
    iterator copy = *this;

    copy.fCurrent = copy.fList;

    return copy;
}

iterator end() const noexcept
{
    iterator copy = *this;

    copy.fCurrent = nullptr;

    return copy;
}
```

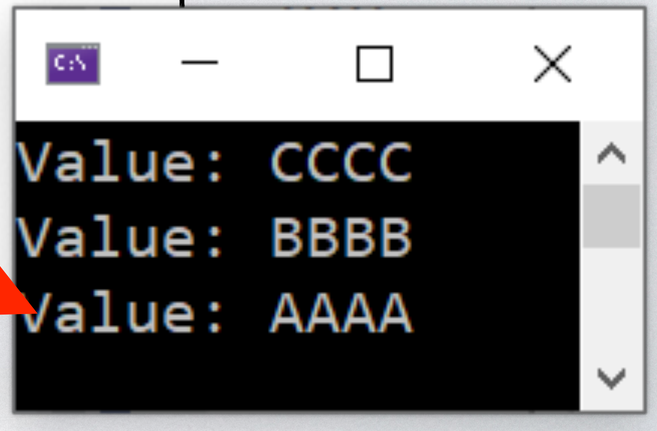
reset to start

set to Nil

A SINGLY LINKED LIST ITERATOR RANGE LOOP

```
using StringList = SinglyLinkedListSP<std::string>;
using StringListNode = StringList::node;

StringListNode One = StringList::makeNode( "AAAA" );
StringListNode Two = StringList::makeNode( "BBBB", One );
StringListNode Three = StringList::makeNode( "CCCC", Two );

for ( auto& value : SinglyLinkedListIterator( Three ) )
{
    std::cout << "Value: " << value << std::endl;
}
```

range loop



ISSUES WITH SMART POINTERS

- The use of smart pointers is not without risks:
  - In a cyclic data structure, the reference count of an std::shared_ptr object may never reach zero. In this case, the managed object (i.e., heap memory) is never destroyed, resulting in a memory leak (i.e., memory becomes unavailable for the application).
  - To break an existing cycle, at least one smart pointer must be a std::weak_ptr holding a non-owning reference to an object managed by an std::shared_ptr.
  - When accessing std::weak_ptr references, we lock the smart pointer first to obtain a usable std::shared_ptr.

DOUBLY LINKED LIST

- A doubly linked list is a sequence of data items, each connected by two links 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 a node at either end of a doubly linked list is straightforward.

A DOUBLY LINKED LIST REPRESENTATION WITH SHARED POINTERS

```
template<typename T>
struct DoublyLinkedList
{
   using node = std::shared_ptr<DoublyLinkedList>;
    using node_w = std::weak_ptr<DoublyLinkedList>;

   T data;
    node next;
    node_w previous;

    DoublyLinkedList( const T& aData ) noexcept : data(aData) {}

    DoublyLinkedList( T&& aData ) noexcept : data(std::move(aData)) {}

    void link( node& aPrevious, node& aNext ) noexcept;     // link adjacent nodes
    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 ( next )                          // Is there a next node?
    {
        next->previous = previous;
    }

    Node INode = previous.lock();        // lock std::weak_ptr

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

    previous.reset();
    next.reset();
    // INode goes out of scope
}
```

clear smart pointer references

DOUBLY LINKED LIST ITERATOR SPECIFICATION

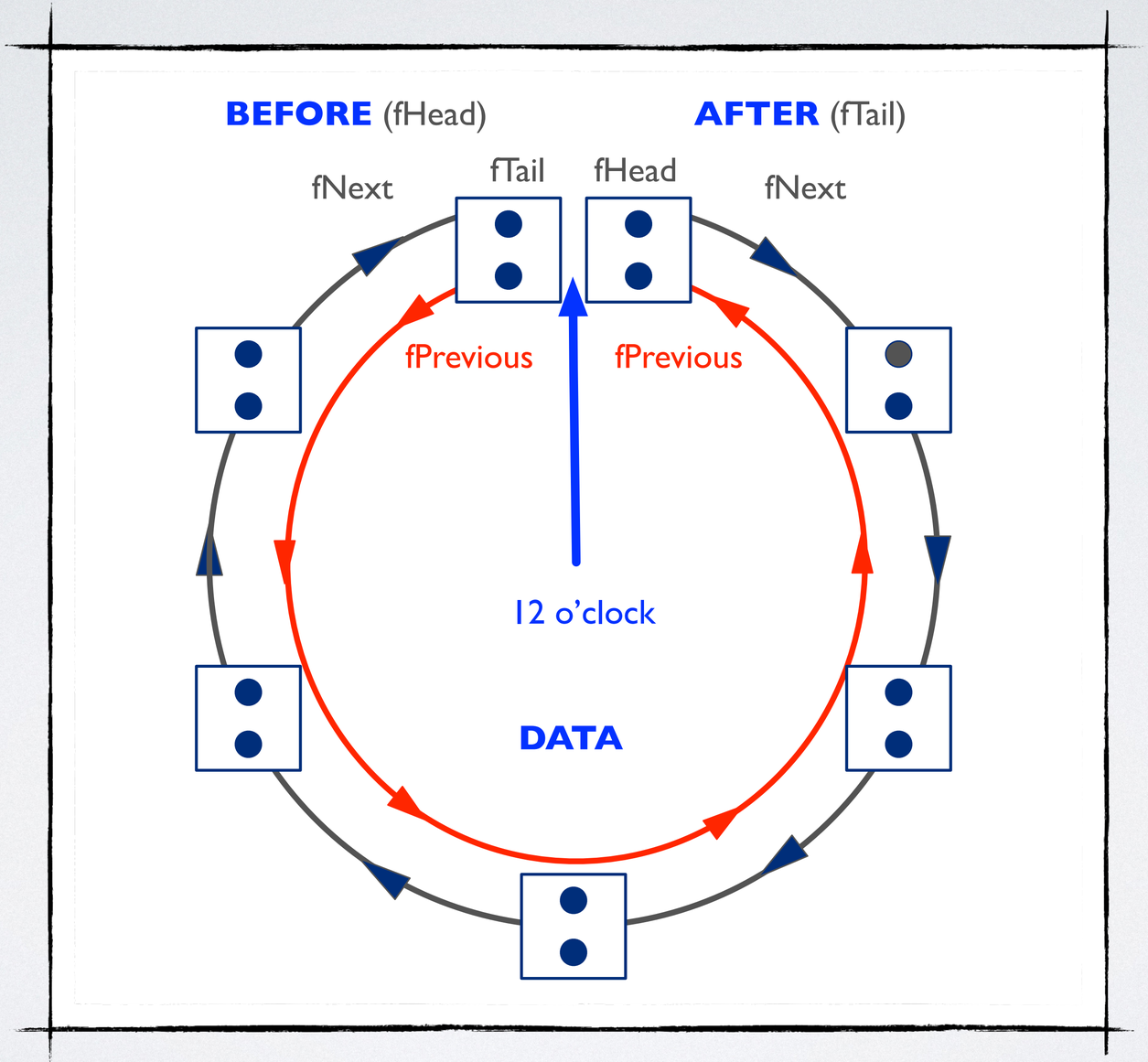
```
template<typename T>
class DoublyLinkedListIterator
{
public:
    using Iterator = DoublyLinkedListIterator<T>;
    using Node = typename DoublyLinkedList<T>::Node;

    enum class States { BEFORE, DATA, AFTER };

    DoublyLinkedListIterator() noexcept : fState(States::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;

    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



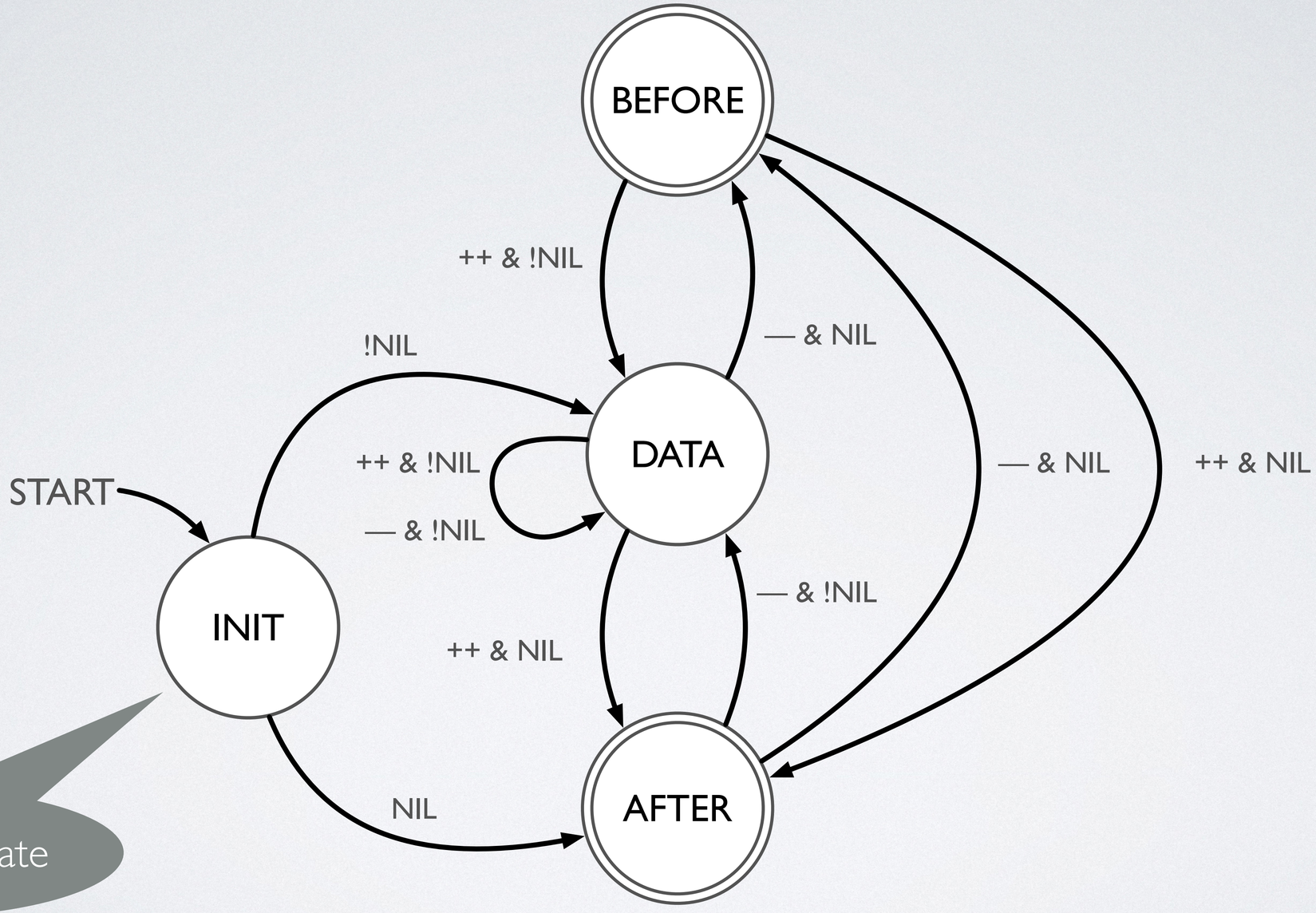
DETERMINISTIC FINITE AUTOMATA

- 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, a DFA must 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 several actions cause transitions from q to p, then the transition diagram can have one arc, labeled by the list of these actions.
  - An arrow to the start state q_0 indicates the start state of the automaton. This arrow does not originate at any node.
  - Nodes marked by a double circle denote accepting states (those in F). 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 inspect the current position first and perform a state transition to the next state second.
- 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.
- In C++, the **break** statement is optional; the compiler does not report an error if it is missing. The resulting fall-throughs are a source of hard-to-find defects. Always end a case with a **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:
    fPosition = fPosition->next;

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

    break;
```

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

LIST ADT

```
template<std::equality_comparable T>
class List
{
private:
    using node = typename DoublyLinkedList<T>::node;

    node fHead;
    node fTail;
    size_t fSize;

public:
    using iterator = DoublyLinkedListIterator<T>;
    using value_type = T;

    List() noexcept;
    ~List() noexcept;

    List( const List& aOther );
    List& operator=( const List& aOther );

    List( List&& aOther ) noexcept;
    List& operator=( List&& aOther ) noexcept;

    void swap( List& aOther ) noexcept;
    size_t size() const noexcept;
    template<typename U> void push_front( U&& aData );
    template<typename U> void push_back( U&& aData );
    void remove( const T& aElement ) noexcept;
    const T& operator[]( size_t aIndex ) const;

    iterator begin() const noexcept;
    iterator end() const noexcept;
    iterator rbegin() const noexcept;
    iterator rend() const noexcept;
};
```

constraint: T must implement **operator==**

list endpoints

copy semantics

move semantics

iterator methods

list operations

WHY DOES THE CLASS LIST NEED AN EXPLICIT DESTRUCTOR?

- All member variables except fSize in class List are of class type.
- The member variables fHead and fTail are shared pointers that will be destroyed when a List object goes out of scope.
- Unfortunately, the destruction recursively traverses the next links in the list. If the next link is not **nullptr**, first, the destructor on next is called, and, second, the list node itself is destroyed. This is the default behavior when we rely on the synthesized destructor for class List.
- This works for small lists but can produce a stack overflow on larger lists.
- To avoid this error, we must explicitly define a suitable List destructor that traverses the list elements backward, starting from fTail.

LIST TEST

```
using StringList = List<std::string>;

StringList IList;

IList.push_back( "AAAA" ); IList.push_front( "BBBB" ); IList.push_back( "CCCC" );
IList.push_front( "DDDD" ); IList.push_back( "EEEE" ); IList.push_front( "FFFF" );

std::cout << "List size: " << IList.size() << std::endl;
std::cout << "5th element: " << IList[4] << std::endl;

IList.remove( "CCCC" );

std::cout << "Remove 5th element." << std::endl;
std::cout << "New 5th element: " << IList[4] << std::endl;
std::cout << "List size: " << IList.size() << std::endl;

std::cout << "Forward iteration:" << std::endl;
for ( auto& item : IList ) { std::cout << item << std::endl; }

std::cout << "Backward iteration:" << std::endl;
for ( auto iter = IList.rbegin(); iter != iter.rend(); iter-- ) { std::cout << *iter << std::endl; }

StringList ICopy = IList;

std::cout << "Copied list iteration:" << std::endl;
for ( auto& item : ICopy ) { std::cout << item << std::endl; }

StringList IMoveCopy = std::move(IList);

std::cout << "Moved list iteration (source):" << std::endl;
for ( auto& item : IList ) { std::cout << item << std::endl; }

std::cout << "Moved list iteration (target):" << std::endl;
for ( auto& item : IMoveCopy ) { std::cout << item << std::endl; }
```