Doubly-linked Lists

What is a 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 a primitive value, a composite value, or even another pointer.
- A singly-linked list is a recursive data structure whose nodes refers to nodes of the same type.

What is a template in C++?

```
template<typename T1, ..., typename Tn>
class AClassTemplate
{
    // class specification
};
```

- A template is a parameterized abstraction over a class.
- From the language-theoretical perspective, templates are 2nd order functions from types to classes/functions.
- To instantiate a class template we supply the desired types, as actual template parameters, so that the C++ compiler can synthesize a specialized class for the template.

Singly-Linked List Class Template

The typename parameter binds all occurrences of DataType in SinglyLinkedList

```
4  template <typename DataType>
5  struct SinglyLinkedList
6  {
7      DataType fData;
8      SinglyLinkedList* fNext;
9
10     SinglyLinkedList( const DataType& aData, SinglyLinkedList* aNext = nullptr ) :
11         fData(aData),
12         fNext(aNext)
13     {}
14
15     SinglyLinkedList( DataType&& aData, SinglyLinkedList* aNext = nullptr ) :
16         fData(std::move(aData)),
17         fNext(aNext)
18     {}
19 };
```

Line: 1 Column: 1 C++ Tab Size: 4

A Simple Test

Main.cpp

```
2  #include <iostream>
3  #include <string>
4
5  using namespace std;
6
7  #include "SinglyLinkedList.h"
8
9  int main()
10 {
11     SinglyLinkedList<int> One( 1 );
12     SinglyLinkedList<int> Two( 2, &One );
13     SinglyLinkedList<int> Three( 3, &Two );
14
15     SinglyLinkedList<int>* lTop = &Three;
16
17     for ( ; lTop != nullptr; lTop = lTop->fNext )
18     {
19         cout << "Value: " << lTop-> fData << endl;
20     }
21
22     return 0;
23 }
```

Line: 25 Column: 1 C++ Tab Size: 4 main

We instantiate the template SinglyLinkedList to SinglyLinkedList<int>.

Specialization Using Type Alias

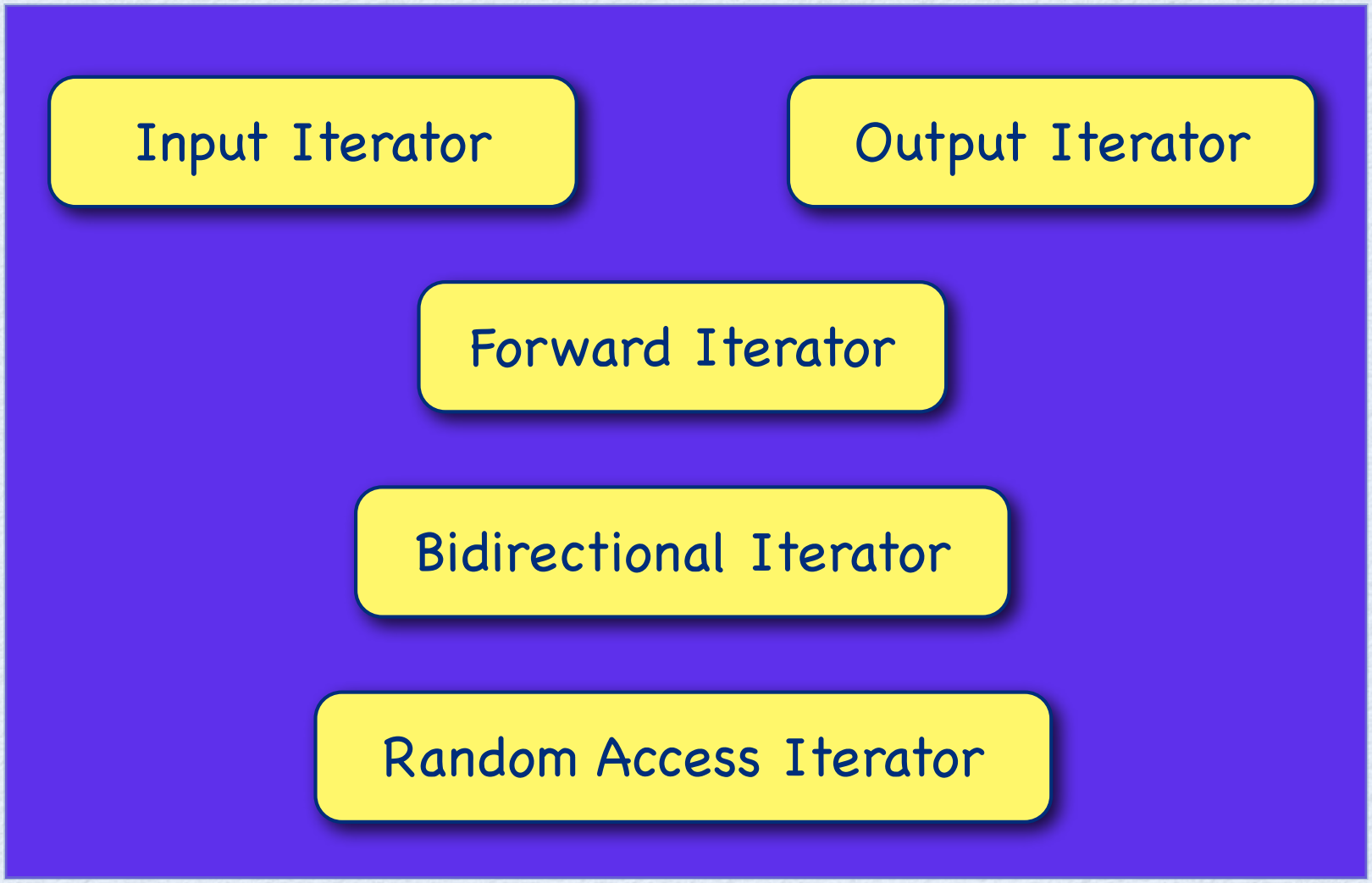
Main.cpp — Programs

```
1
2   #include <iostream>
3   #include <string>
4
5   using namespace std;
6
7   #include "SinglyLinkedListTemplate.h"
8
9   int main()
10  {
11      using StringList = SinglyLinkedList<string>;
12
13      string lA = "AAAA";
14      string lC = "CCCC";
15
16      StringList One( lA );
17      StringList Two( "BBBB", &One );
18      StringList Three( lC, &Two );
19
20      StringList* lTop = &Three;
21
22      for ( ; lTop != nullptr; lTop = lTop->fNext )
23      {
24          cout << "Value: " << lTop->fData << endl;
25      }
26
27      return 0;
28  }
29
```

Line: 37:19 C++ Tab Size: 4 main

We instantiate the template SinglyLinkedList to SinglyLinkedList<string>.

What are the iterator models supported by C++?



Forward Iterator

Expression	Effect
*iter	Provides read access to the actual element
iter->member	Provides read access to a member of the actual element
++iter	Steps forward (returns new position)
iter++	Steps forward (returns old position)
iter1 == iter2	Returns whether iter1 and iter2 are equal
iter1 != iter2	Returns whether iter1 and iter2 are not equal
iter1 = iter2	Assigns an iterator

How do we use iterators in C++?

```
for ( const auto& element : collection )
{
    // loop body using element
};
```

See lecture notes Using C++11's For-Each-Loop (Iterators)

- An iterator represents a certain position in a container, where the auxiliary methods begin() and end() return the position of the first element and the position after the last element, respectively.
- We can travers all elements of a data type via its corresponding iterator in a range loop. The proper functioning of the range loop depends on the proper definition of begin(), end(), and the iterator operators.

SinglyLinkedList Iterator Specification

SinglyLinkedListIteratorB SPEC.h — Programs

```
3   #include "SinglyLinkedListTemplate.h"
4
5   template <typename T>
6   class SinglyLinkedListIterator
7   {
8   private:
9
10      using ListNode = SinglyLinkedList<T>;
11
12      const ListNode& fList;
13      const ListNode* fIndex;
14
15  public:
16
17      using Iterator = SinglyLinkedListIterator<T>;
18
19      SinglyLinkedListIterator( const ListNode& aList );
20
21      const T& operator*() const;
22      Iterator& operator++();      // prefix
23      Iterator operator++(int);    // postfix
24      bool operator==( const Iterator& aRHS ) const;
25      bool operator!=( const Iterator& aRHS ) const;
26
27      Iterator begin();            // for-range feature
28      Iterator end();              // for-range feature
29  };
```

Line: 31 C++ Tab Size: 4 end

We maintain a read-only reference to list elements

SinglyLinkedList Iterator Test

Main.cpp — Programs

```
2  #include <iostream>
3  #include <string>
4
5  using namespace std;
6
7  #include "SinglyLinkedListIteratorB.h"
8
9  int main()
10 {
11     using StringList = SinglyLinkedList<string>;
12     using StringListIterator = SinglyLinkedListIterator<string>;
13
14     string lA = "AAAA";
15     string lC = "CCCC";
16
17     StringList One( lA );
18     StringList Two( "BBBB", &One );
19     StringList Three( lC, &Two );
20
21     for ( const string& i : StringListIterator( Three ) )
22     {
23         cout << "Value: " << i << endl;
24     }
25
26     return 0;
27 }
```

Line: 30 C++ Tab Size: 4 main

For iterator implementation see Canvas

Three passed as l-value reference

The deletion of a node at the end of a list requires a search from the top to find the new last node.

What is a 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.
- We can link the first and last element to create a loop. This allows for an effective forward and backwards traversal.

A Doubly-Linked List Template

DoublyLinkedList SPEC.h

```
4  #pragma once
5
6  template<typename T>
7  class DoublyLinkedList
8  {
9  private:
10
11     T fPayload;                                          // payload
12     DoublyLinkedList* fNext;                             // next element
13     DoublyLinkedList* fPrevious;                         // previous element
14
15 public:
16
17     explicit DoublyLinkedList( const T& aPayload );      // l-value constructor
18     explicit DoublyLinkedList( T&& aPayload );           // r-value constructor
19
20     DoublyLinkedList& push_front( DoublyLinkedList& aNode );   // aNode becomes previous of this
21     DoublyLinkedList& push_back( DoublyLinkedList& aNode );    // aNode becomes next of this
22     void isolate();                                      // removes this node
23
24     void swap( DoublyLinkedList& aNode );                // exchange payloads
25
26     const T& operator*() const;                          // dereference operator, payload
27     const T& getPayload() const;                         // returns constant reference to paylod
28     const DoublyLinkedList& getNext() const;             // returns constant reference to next
29     const DoublyLinkedList& getPrevious() const;         // returns constant reference to previous
30 };
```

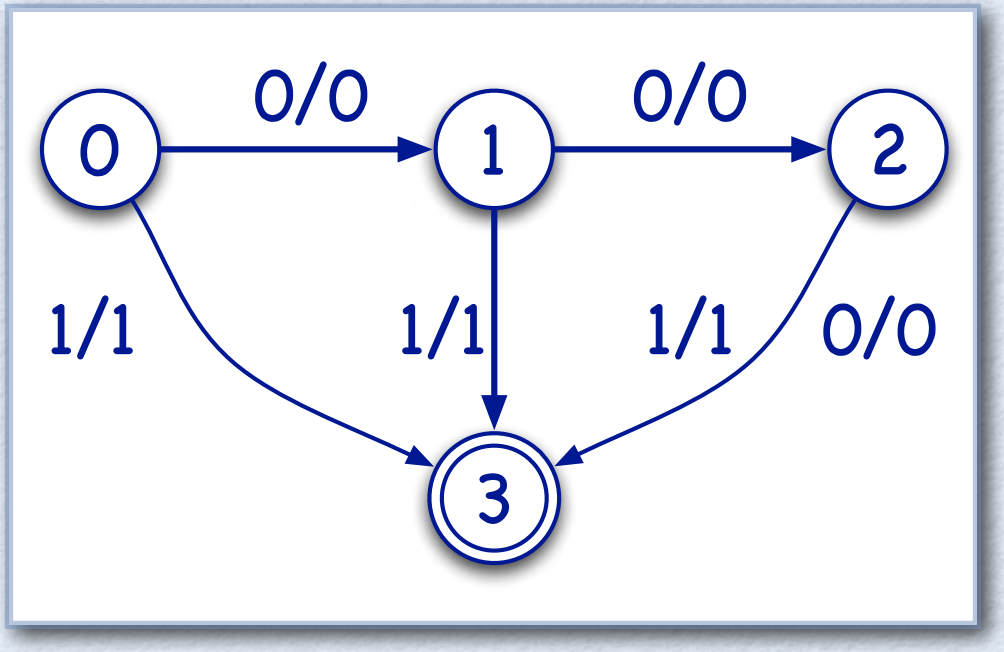
Line: 2 Column: 7 C++ Tab Size: 4

An iterator for a doubly-linked list requires a state machine.

What is a state machine?

A state machine is a piece of software that explicitly maintains states to control the behavior of the associated program:

	0	1
0	(1,0)	(3,1)
1	(2,0)	(3,1)
2	(3,0)	(3,1)
3	(3,0)	(3,1)



Both specifications describe the same state machine, which stops in state 3 - the final state. Some state machines may not have a final state - they can continue ad infinitum. Example, state machine for doubly-linked list iterator.

Bidirectional Iterator

Expression	Effect
*iter	Provides read access to the actual element
iter->member	Provides read access to a member of the actual element
++iter	Steps forward (returns new position)
iter++	Steps forward (returns old position)
--iter	Steps backward (returns new position)
iter--	Steps backward (returns old position)
iter1 == iter2	Returns whether iter1 and iter2 are equal
iter1 != iter2	Returns whether iter1 and iter2 are not equal
iter1 = iter2	Assigns an iterator

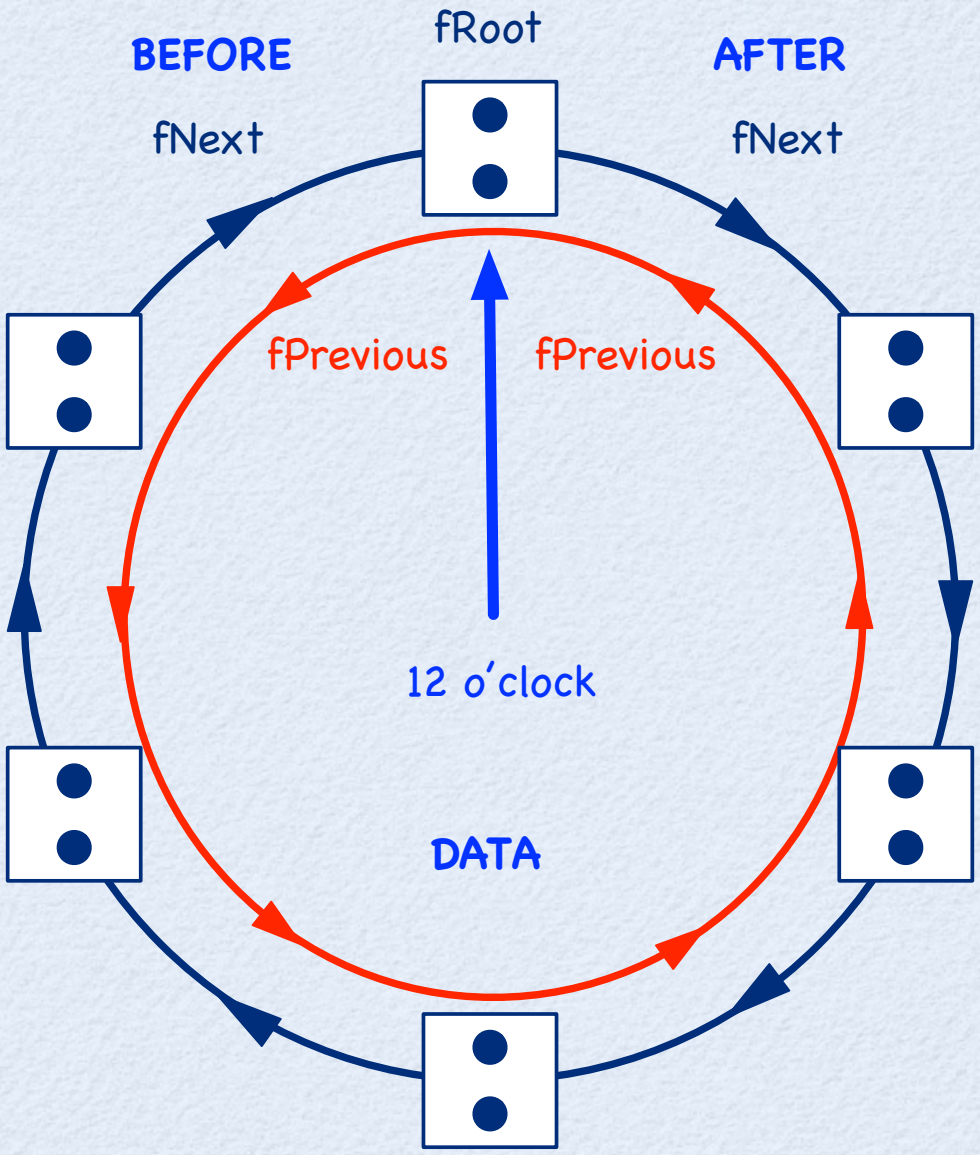
A Doubly-Linked List Iterator Template

DoublyLinkedListIterator SPEC.h

```
4  #pragma once
5
6  #include "DoublyLinkedList.h"
7
8  template<typename T>
9  class DoublyLinkedListIterator
10 {
11 private:
12     enum class States { BEFORE, DATA , AFTER };              // iterator states
13
14     using Node = DoublyLinkedList<T>;
15
16     const Node* fRoot;                                       // doubly-linked list
17
18     States fState;                                           // iterator state
19     const Node* fCurrent;                                    // iterator position
20
21 public:
22
23     using Iterator = DoublyLinkedListIterator<T>;
24
25     DoublyLinkedListIterator( const Node* aRoot );           // constructor
26
27     const T& operator*() const;                              // dereference
28     Iterator& operator++();                                  // prefix increment
29     Iterator operator++(int);                                // postfix increment
30     Iterator& operator--();                                  // prefix decrement
31     Iterator operator--(int);                                // postfix decrement
32     bool operator==( const Iterator& aOtherIter ) const;     // equivalence
33     bool operator!=( const Iterator& aOtherIter ) const;     // not equal
34
35     Iterator begin() const;                                  // first element forward
36     Iterator end() const;                                    // after last element forward
37     Iterator rbegin() const;                                 // first element backwards
38     Iterator rend() const;                                   // before first element backwards
39 };
```

Line: 2 Column: 9 C++ Tab Size: 4

Iterator Moving Around The Clock



Doubly-linked List Iterator State Machine

- Think of fRoot as 12 o'clock and the iterator moving around the clock.

	++	–
BEFORE	fCurrent = fRoot; fCurrent == null/AFTER fCurrent != null/DATA	NOP
DATA	fCurrent == previous of fRoot: fCurrent == null/AFTER fCurrent = next of fCurrent	fCurrent == fRoot: fCurrent == null/BEFORE fCurrent = previous of fCurrent
AFTER	NOP	fCurrent = fRoot; fCurrent == null/BEFORE fCurrent = previous of fCurrent /DATA