Doubly-linked Lists

Tutorial 5

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

- A template is a parameterized abstraction over a class.
- 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.

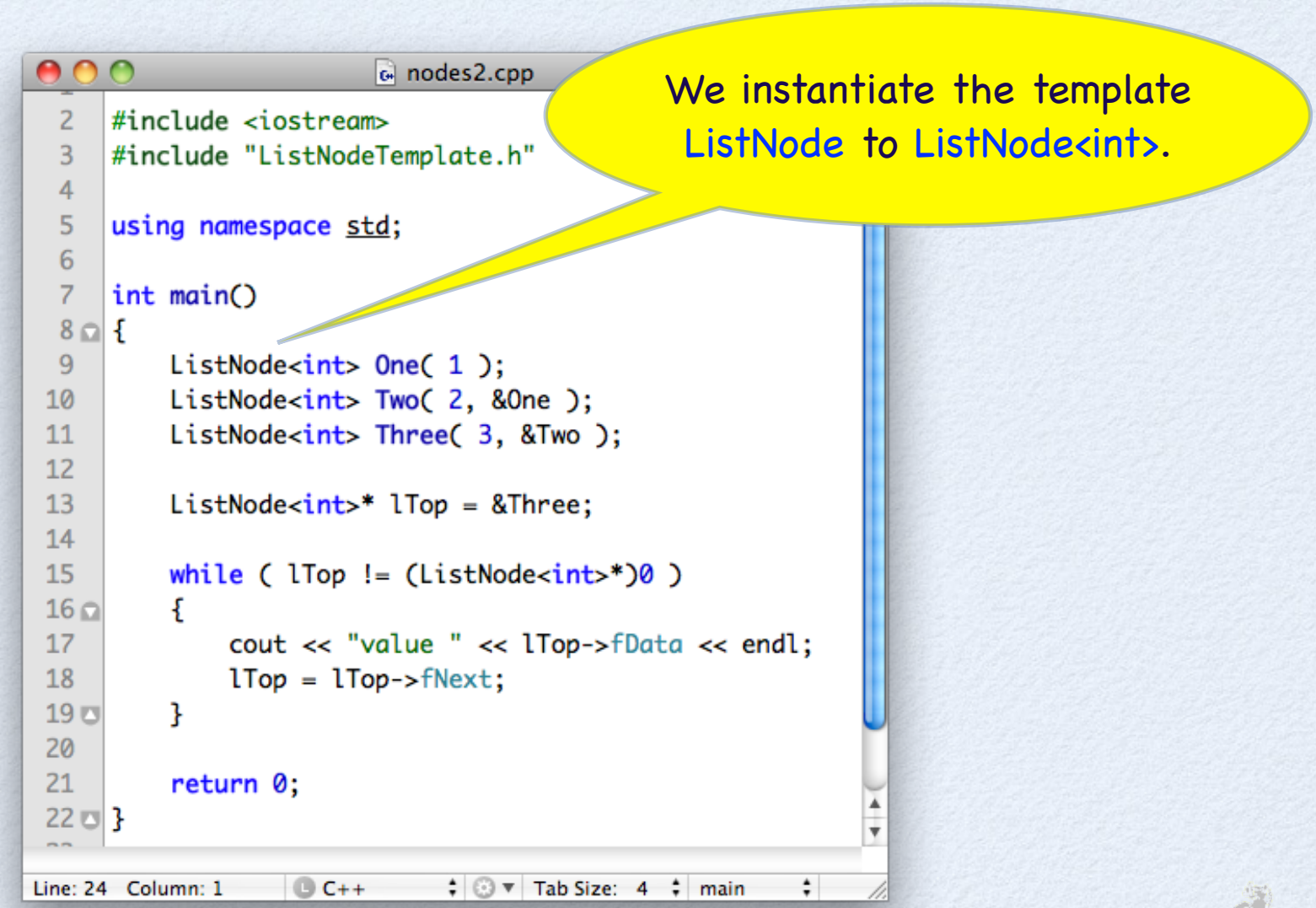
Node Class Template

ListNodeTemplate.h

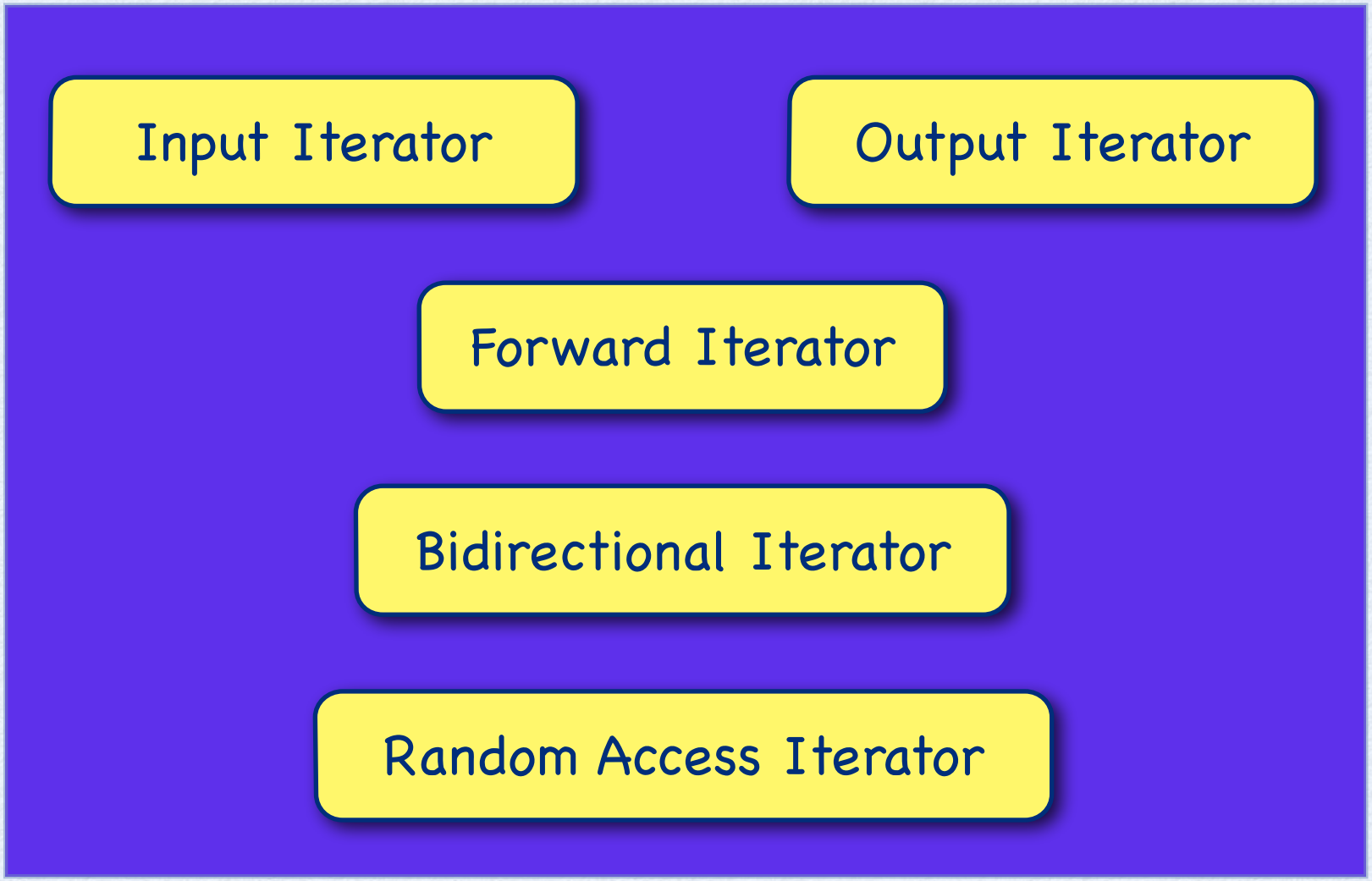
```
2   #pragma once
3
4   template <class DataType>
5   class ListNode
6   {
7   public:
8       DataType fData;
9       ListNode* fNext;
10
11  public:
12      ListNode( const DataType& aData, ListNode* aNext = (ListNode*)0 )
13      {
14          fData = aData;
15          fNext = aNext;
16      }
17  };
```

Line: 19 Column: 1 C++ Tab Size: 4 ListNode

A Simple Test



What are the iterator models supported by C++?



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

How do we use iterators in C++?

```
container<T>::iterator pos;
for ( pos = coll.begin(); pos != coll.end(); pos++ )
{
    // access elements through *pos
};
```

- An iterator is an object that allows one to navigate (sequentially) through a data container like vectors or lists.
- 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.

NodeIterator

ListNodeIterator.h

```
#include "ListNodeTemplate.h"

template<class T>
class ListNodeIterator
{
private:
    ListNode<T>* fNode;

public:
    typedef ListNodeIterator<T> Iterator;    // Iterator type definition

    ListNodeIterator( ListNode<T>* aNode );

    const T& operator*() const;
    Iterator& operator++();                  // prefix
    Iterator operator++( int );              // postix (extra unused argument)
    bool operator==( const Iterator& aOther ) const;
    bool operator!=( const Iterator& aOther) const;

    Iterator end();
};
```

Line: 27 Column: 14 C++ Tab Size: 4 T

NodeIterator Test

ListNodeIteratorTest.cpp

```
2  #include <iostream>
3
4  #include "ListNodeIterator.h"
5
6  using namespace std;
7
8  int main()
9  {
10     typedef ListNode<int> IntegerNode;
11
12     IntegerNode One( 1 );
13     IntegerNode Two( 2, &One );
14     IntegerNode Three( 3, &Two );
15
16     for ( ListNodeIterator<int> iter( &Three ); iter != iter.end(); ++iter )
17     {
18         cout << "value " << *iter << endl;
19     }
20
21     return 0;
22 }
```

Line: 24 Column: 1 C++ Tab Size: 4 main

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 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.

Sentinel Node: NIL

- A sentinel node is a programming idiom used to replace null-pointers with a special token denoting "no value" or nil.
- Sentinel nodes behave like null-pointers. However, unlike null-pointers, which refer to nothing, sentinels denote proper, yet empty, values.

A Doubly-Linked List Node

```
#pragma once

template<class DataType>
class DoublyLinkedNode
{
public:
    typedef DoublyLinkedNode<DataType> Node; // nominal equivalence

private:
    DataType fValue;    // stored datum
    Node* fNext;        // forward pointer to next element
    Node* fPrevious;    // backward pointer to previous element

    // private default constructor for sentinel
    DoublyLinkedNode()
    {
        fValue = DataType();    // initialize fValue with default for DataType
        fNext = &NIL;           // set forward pointer to NIL
        fPrevious = &NIL;       // set backward pointer to NIL
    }

public:
    static Node NIL;                              // sentinel declaration

    DoublyLinkedNode( const DataType& aValue ); // constructor (unlinked node)

    void prepend( Node& aNode );                  // aNode becomes left node of this
    void append( Node& aNode );                   // aNode becomes right node of this
    void remove();                                // this node is removed

    // getter functions
    const DataType& getValue() const;             // return constant reference to datum
    const Node& getNext() const;                  // return constant reference to next node
    const Node& getPrevious() const;              // return constant reference to previous node
};

// sentinel implementation
template<class DataType>
DoublyLinkedNode<DataType> DoublyLinkedNode<DataType>:: NIL;
```

Template Implementation: Variant A

```
#pragma once

template<class DataType>
class DoublyLinkedNode
{
private:

   …

public:

   …

    void prepend( Node& aNode )           // aNode becomes left node of this
    {
        aNode.fNext = this;               // make this the forward pointer of aNode

        if ( fPrevious != &NIL )          // make this's backward pointer aNode's
        {                                 // backward pointer and make previous'
            aNode.fPrevious = fPrevious;  // forward pointer aNode
            fPrevious->fNext = &aNode;
        }

        fPrevious = &aNode;               // this' backward pointer becomes aNode
    }

   …
};
```

Implementation within template class specification

Template Implementation: Variant B

```
#pragma once

template<class DataType>
class DoublyLinkedNode
{
   …
};

template<class DataType>
void DoublyLinkedNode<DataType>::prepend( Node& aNode )
{
    aNode.fNext = this;                  // make this the forward pointer of aNode

    if ( fPrevious != &NIL )             // make this's backward pointer aNode's
    {                                    // backward pointer and make previous'
        aNode.fPrevious = fPrevious;     // forward pointer aNode
        fPrevious->fNext = &aNode;
    }

    fPrevious = &aNode;                  // this' backward pointer becomes aNode
}
```

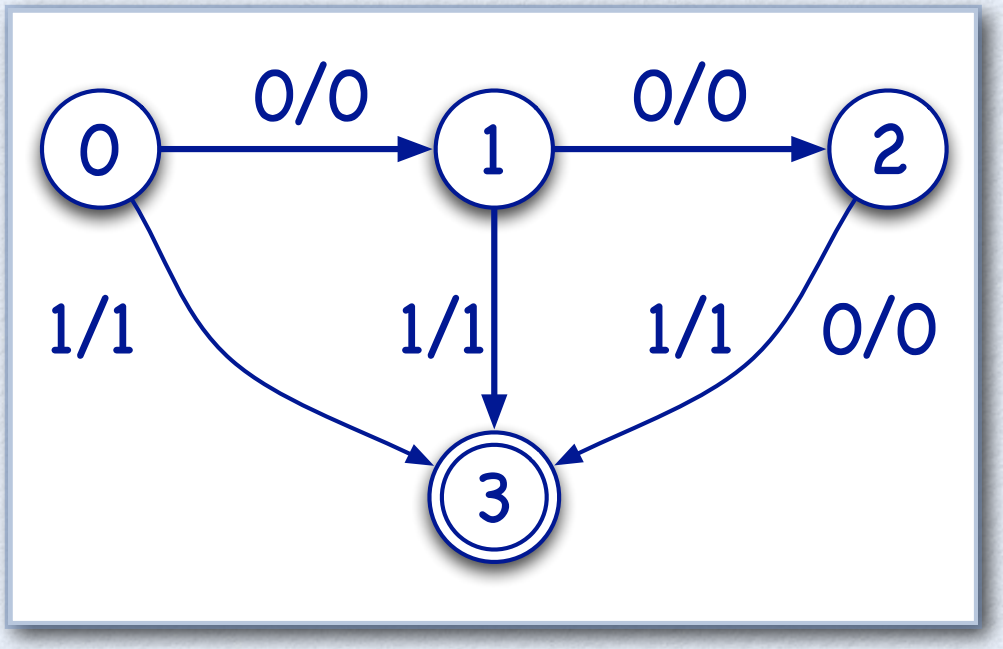
Implementation outside template class specification, but within same header file

An iterator for a doubly-linked list may require 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.