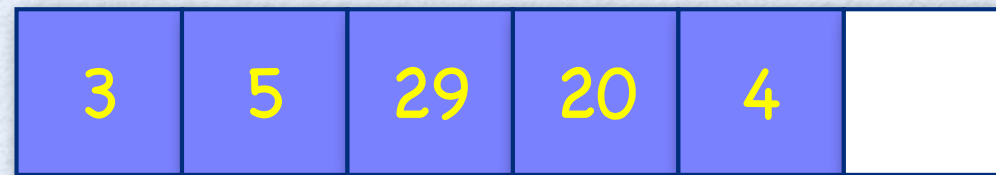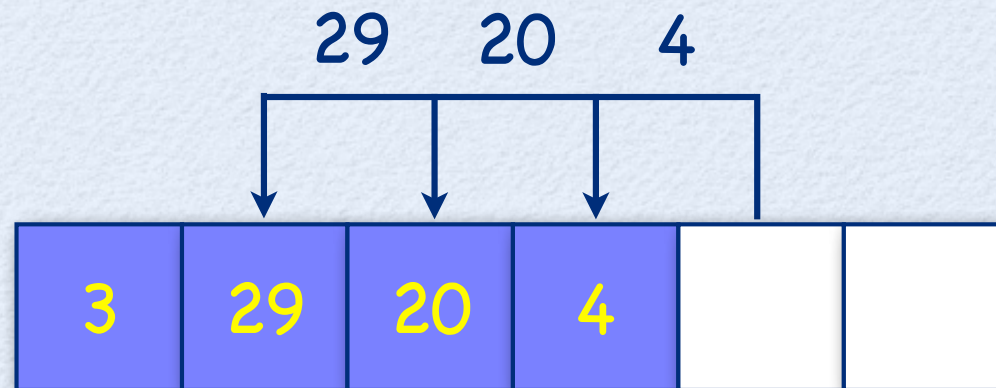# Problems with Arrays

- 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).

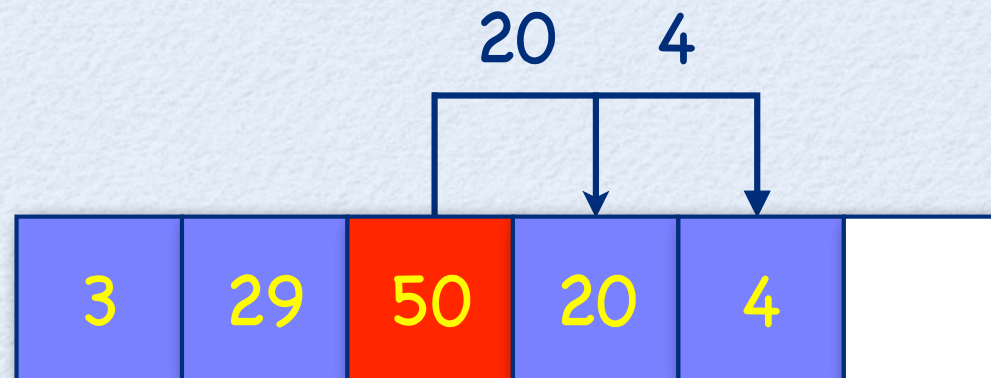- Resizing affects performance.

Deletion Requires Relocation

Insertion Requires Relocation

3 | 29 | 20 | 4 | | |

Insert 50 after 29

20   4

3 | 29 | 50 | 20 | 4 | |

Singly-Linked Lists

- 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.
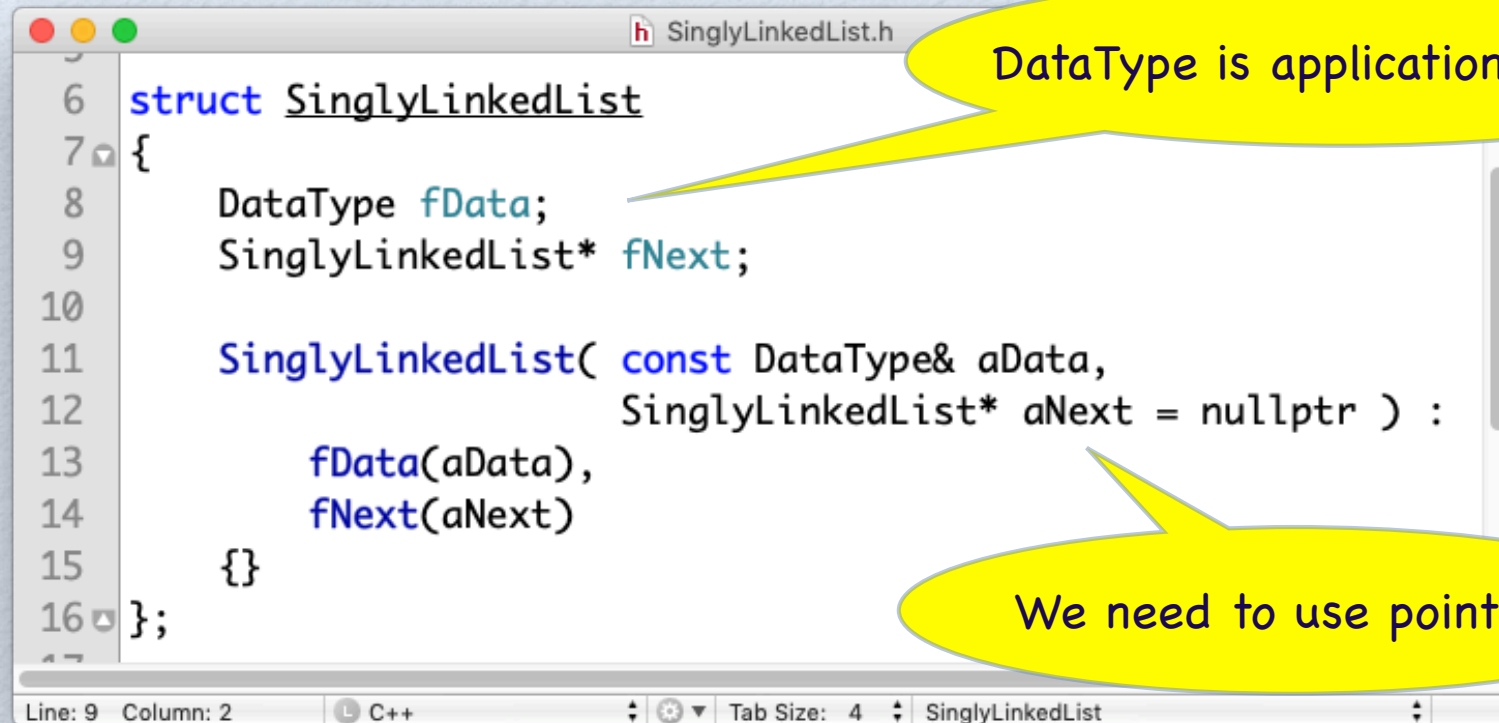
Impossible Singly-Linked List Structure in C++:

```cpp
// impossible singly-linked list

struct SinglyLinkedList
{
    DataType fData;
    SinglyLinkedList fNext;

    SinglyLinkedList( const DataType& aData,
                      const SinglyLinkedList& aNext ) :
        fData(aData),
        fNext(aNext)
    {}
};
```

$$X = X \to X \to X \to X \to \ldots$$
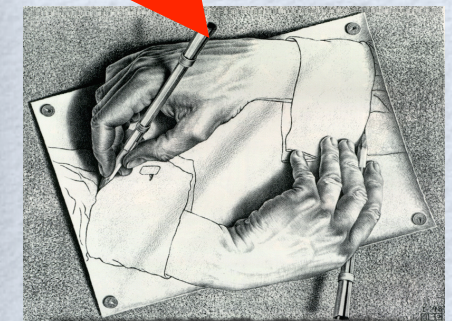
Field fNext has incomplete type.

Singly-Linked List Using Pointers

```cpp
struct SinglyLinkedList
{
    DataType fData;
    SinglyLinkedList* fNext;

    SinglyLinkedList( const DataType& aData,
                      SinglyLinkedList* aNext = nullptr ) :
        fData(aData),
        fNext(aNext)
    {}
};
```

DataType is application-specific

We need to use pointers

- A list manages a collection of elements.

- The class SinglyLinkedList defines a value-based sequence container for values of type DataType.

- To break infinite recursion, we have to use pointers to the next elements. This way, the compiler can deduce the size of a singly-linked list element and compile the definition.

240

Singly-Linked List with R-Values

```cpp
6    struct SinglyLinkedList
7    {
8        DataType fData;
9        SinglyLinkedList* fNext;
10
11       SinglyLinkedList( const DataType& aData, SinglyLinkedList* aNext = nullptr ) :
12           fData(aData),
13           fNext(aNext)
14       {}
15
16       SinglyLinkedList( DataType&& aData, SinglyLinkedList* aNext = nullptr ) :
17           fData(std::move(aData)),
18           fNext(aNext)
19       {}
20   };
```
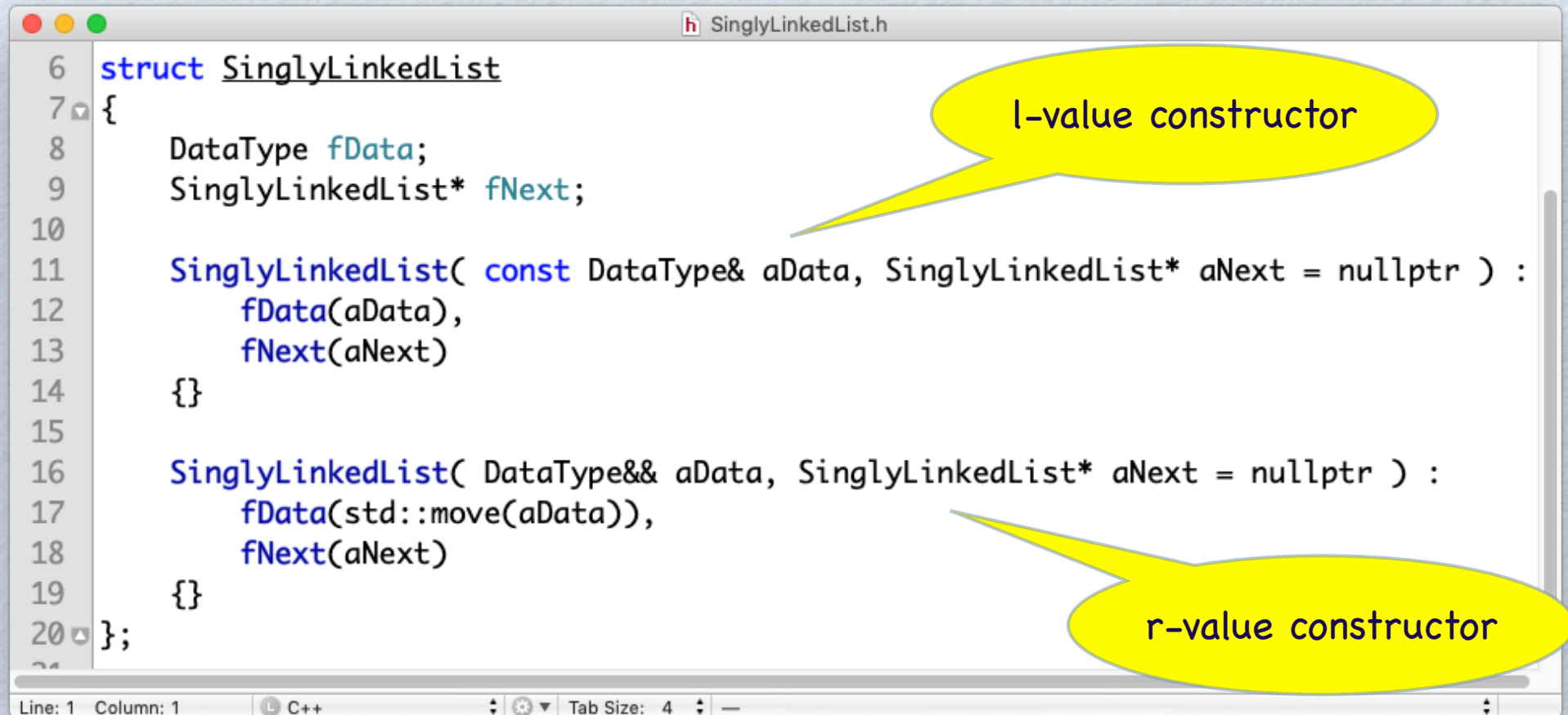
l-value constructor

r-value constructor

Window title: h SinglyLinkedList.h

Status bar: Line: 1   Column: 1   C++   Tab Size: 4   —

- The r-value constructor can "steal" the memory of the argument aData to initialize the payload fData. To use this constructor, the parameter aData must be a temporary of literal value.

R-Value References (C++-11)

L-Values and L-Value References &

- The references that we have seen so far are l-value references, that is, references to l-values.

- The term l-value refers to a thing that can occur on the left side of an assignment, named objects that have a defined storage location (i.e., an address). L-value references can only be bound to l-values (exception: we can bind an r-value to a const l-value reference):

```
int var = 42;            // l-value: initialized variable declaration

int& ref = var;          // l-value reference to an l-value

int& ref2 = 42;          // error: non-const l-value reference cannot bind to temporary

const int& ref3 = 42;    // const l-value reference to an r-value
```

R-Values and R-Value References &&

- The term r-value refers to things that can occur on the right side of an assignment, literals and temporaries that do **not** have a defined storage locations.
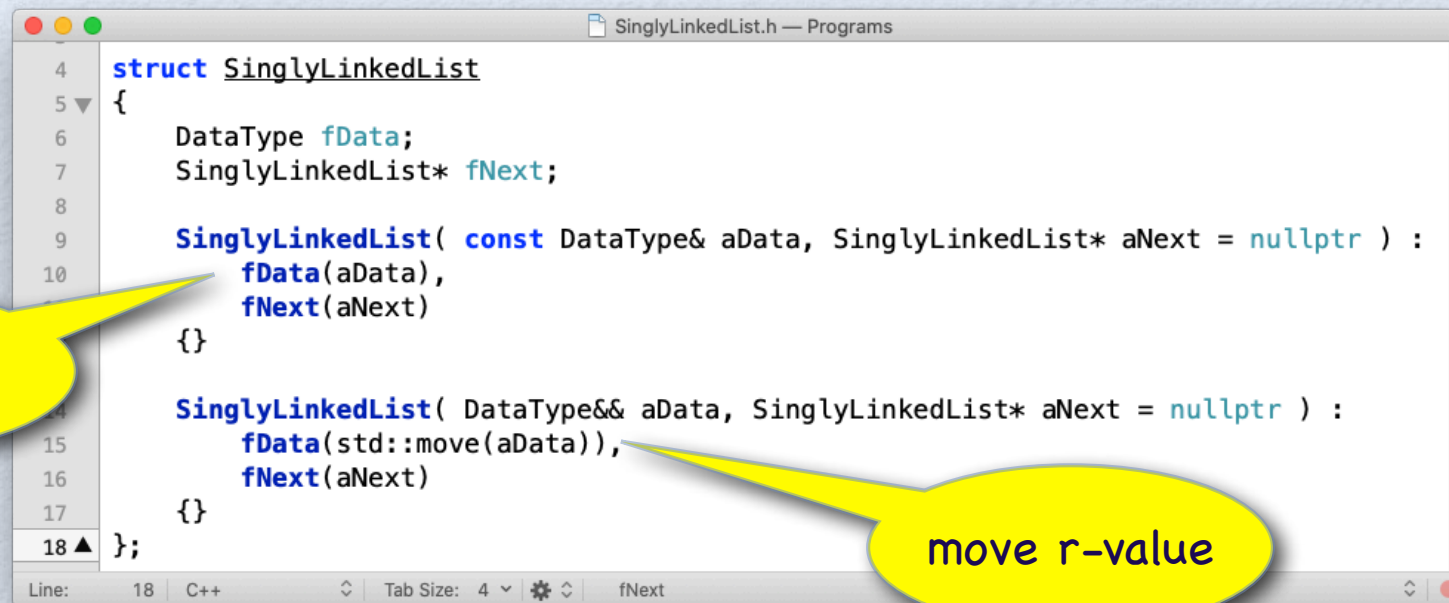
- R-value references only bind to r-values:

```
int&& ref = 42;        // r-value reference to an r-value

int val = 42;          // l-value: initialized variable declaration

int&& ref2 = val;      // error: r-value reference cannot bind to l-value
```

Move Semantics (std::move)

- R-values are typically temporary and so can be freely modified: if you know that your function parameter is an r-value, you can use it as temporary storage, or "steal" its contents without affecting program correctness.

- This means that rather than copying the contents (expensive) of an r-value, you can move the contents (cheap).

```cpp
struct SinglyLinkedList
{
    DataType fData;
    SinglyLinkedList* fNext;

    SinglyLinkedList( const DataType& aData, SinglyLinkedList* aNext = nullptr ) :
        fData(aData),
        fNext(aNext)
    {}

    SinglyLinkedList( DataType&& aData, SinglyLinkedList* aNext = nullptr ) :
        fData(std::move(aData)),
        fNext(aNext)
    {}
};
```

copy
l-value

move r-value

std::move is a function that performs a type cast of its argument to an r-value.

Using L-Values and R-Values

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

Line: 18 | C++ | Tab Size: 4 | fNext

- Using the different constructors, we can write:

```cpp
string lValue = "COS30008";

SinglyLinkedList lNodeWithCopy( lValue );

SinglyLinkedList lNodeWithMove( "COS30008" );
```

copy
l-value

move
r-value

© Dr Markus Lumpe, 2022

247

Using L-Values and R-Values: No Move

```
SinglyLinkedList.h — Programs

4   struct SinglyLinkedListCopyOnly
5   {
6       DataType fData;
7       SinglyLinkedListCopyOnly* fNext;
8
9       SinglyLinkedListCopyOnly( const DataType& aData, SinglyLinkedList* aNext = nullptr ) :
10          fData(aData),
11          fNext(aNext)
12      {}
13  };

Line:   15:19   C++                  Tab Size:  4 ∨   ⚙ ↕   SinglyLinkedList
```

- If a class does not support move semantics, then the compiler will use copy semantics. That is, an r-value decays to an l-value.

```
string lValue = "COS30008";

SinglyLinkedListCopyOnly lNodeWithLValue( lValue );

SinglyLinkedListCopyOnly lNodeWithRValue( "COS30008" );
```

> copy l-value

> copy
> r-value

We discuss more details later when we study memory management.

A Simple List of Integers

```cpp
#include <iostream>
#include <string>

using namespace std;

using DataType = string;

#include "SinglyLinkedList.h"

int main()
{
    string lA = "AAAA";
    string lC = "CCCC";

    SinglyLinkedList One( lA );
    SinglyLinkedList Two( "BBBB", &One );
    SinglyLinkedList Three( lC, &Two );

    SinglyLinkedList* lTop = &Three;

    for ( ; lTop != nullptr; lTop = lTop->fNext )
    {
        cout << "Value: " << lTop->fData << endl;
    }

    return 0;
}
```

define DataType a synonym for string

Microsoft Visual Studio Deb...

```
Value: CCCC
Value: BBBB
Value: AAAA
```

Using Declaration — C++11 Type Aliases

- A type alias is a name that refers to a previously defined type.

> **using** identifier = type;

- Type aliases are commonly used for three purposes:

  - To hide the implementation of a given type.

  - To streamline complex type definitions making them easier to understand, and

  - To allow a single type to be used in different contexts under different names.

- Type aliases establish a nominal equivalence between types.

- Type aliases are similar to **typedef**. However, type aliases are better suited when creating alias templates.

Can we do better?

Templates – C++'s Generic Types
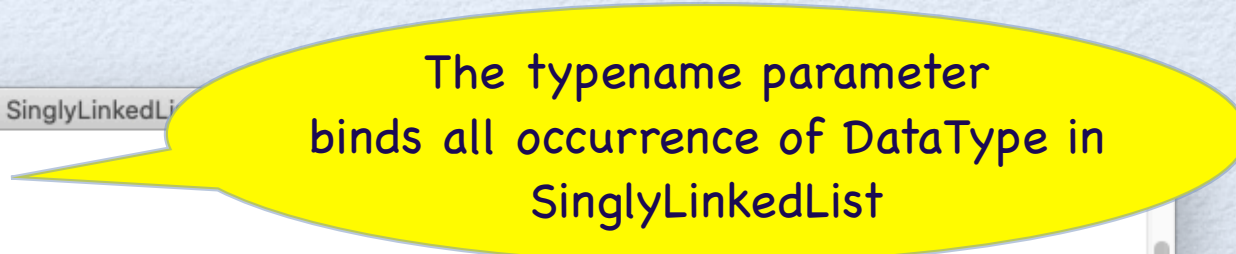
- Templates are blueprints from which classes and/or functions are automatically generated by the compiler based on a set of parameters.

- Note, every time a given template is being instantiated with type parameters that have not been used before, a new version of the class or function is generated.

- A new version of a class or function is called specialization of the template. Specializations are not mutually compatible.

Class Template

```
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

```cpp
template <typename DataType>
struct SinglyLinkedList
{
    DataType fData;
    SinglyLinkedList* fNext;

    SinglyLinkedList( const DataType& aData, SinglyLinkedList* aNext = nullptr ) :
        fData(aData),
        fNext(aNext)
    {}

    SinglyLinkedList( DataType&& aData, SinglyLinkedList* aNext = nullptr ) :
        fData(std::move(aData)),
        fNext(aNext)
    {}
};
```

The typename parameter binds all occurrence of DataType in SinglyLinkedList

The New Main


```cpp
#include <iostream>
#include <string>

using namespace std;

#include "SinglyLinkedListTemplate.h"

int main()
{
    using StringList = SinglyLinkedList<string>;

    string lA = "AAAA";
    string lC = "CCCC";

    StringList One( lA );
    StringList Two( "BBBB", &One );
    StringList Three( lC, &Two );

    StringList* lTop = &Three;

    for ( ; lTop != nullptr; lTop = lTop->fNext )
    {
        cout << "Value: " << lTop->fData << endl;
    }

    return 0;
}
```

We instantiate the template SinglyLinkedList to SinglyLinkedList<string>.

256

© Dr Markus Lumpe, 2022

Class Template Instantiation

using IntegerList = SinglyLinkedList<int>;

using ListOfIntegerLists = SinglyLinkedList<IntegerList>;

- Types used as arguments cannot be classes with local scope.

- Once instantiated, a class template can be used as any other class.

List Iterator Template

SinglyLinkedList Iterator Specification

```cpp
#pragma once

#include "SinglyLinkedListTemplate.h"

template <typename T>
class SinglyLinkedListIterator
{
private:

    using ListNode = SinglyLinkedList<T>;

    const ListNode* fList;
    const ListNode* fIndex;

public:

    using Iterator = SinglyLinkedListIterator<T>;

    SinglyLinkedListIterator( const ListNode* aList );

    const T& operator*() const;
    Iterator& operator++();        // prefix
    Iterator operator++(int);      // postfix
    bool operator==( const Iterator& aRHS ) const;
    bool operator!=( const Iterator& aRHS ) const;

    Iterator begin();              // for-range feature
    Iterator end();                // for-range feature
};
```

We maintain a pointer to read-only list elements

SinglyLinkedListIterator SPEC.h — Programs

Line: 32   C++   Tab Size: 4   end

Notes on Templates

- When using templates, the C++ compiler must have access not only the specification of a template class but also to all method implementations in order to properly instantiate the template.

- Templates are not to be confused with library classes. Templates are blueprint that have to be instantiated for each separate type application.

- Think of templates as special forms of macros.

- When defining a template class, we need to implement, like in Java, all methods within the class definition, usually in a header file.

Templates have no .cpp file!

Constructor & Deference
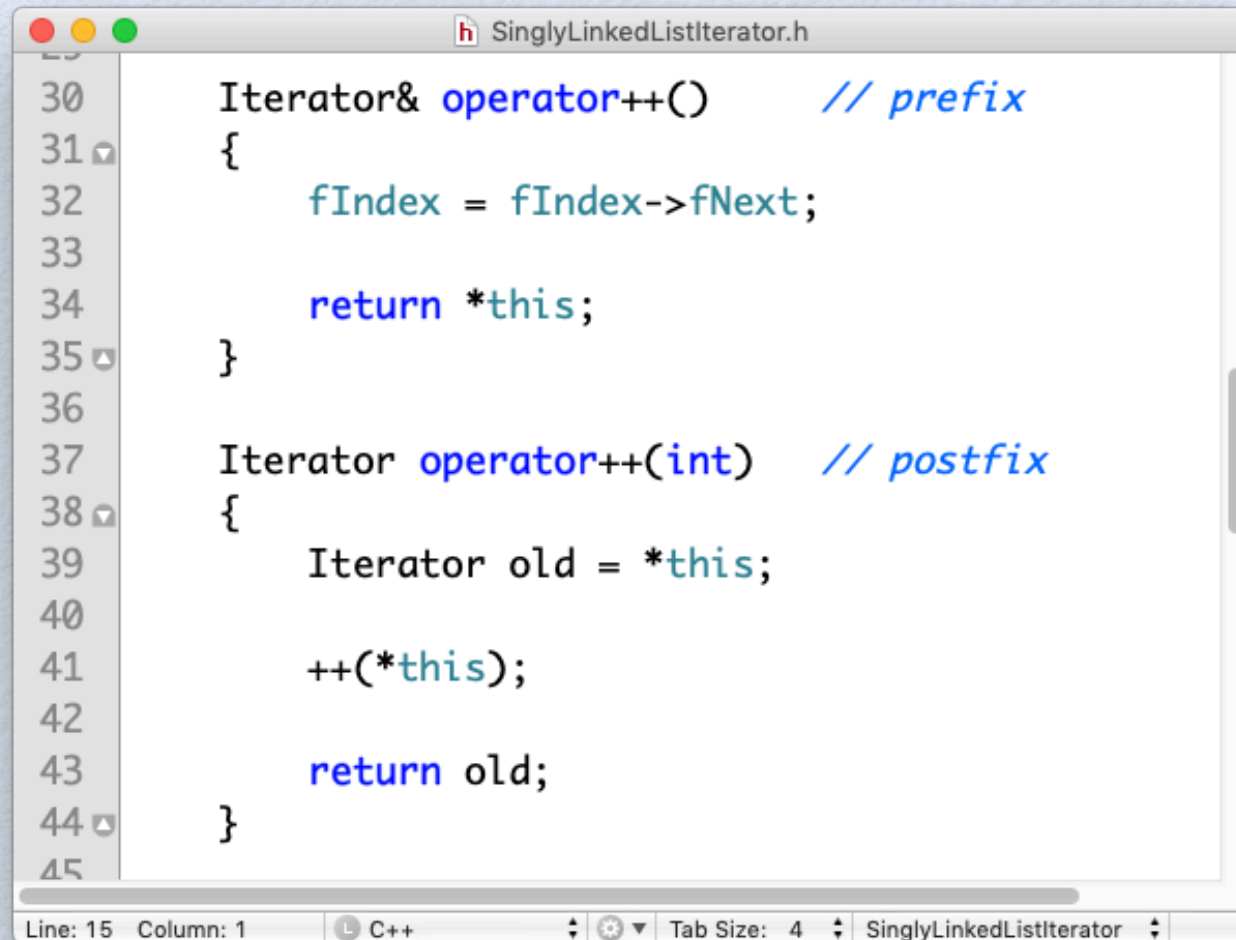
```cpp
17
18   using Iterator = SinglyLinkedListIterator<T>;
19
20   SinglyLinkedListIterator( const ListNode* aList ) :
21       fList(aList),
22       fIndex(aList)
23   {}
24
25   const T& operator*() const
26   {
27       return fIndex->fData;
28   }
29
```

Line: 15   Column: 1   C++   Tab Size: 4   SinglyLink

For iterator implementation in header file.
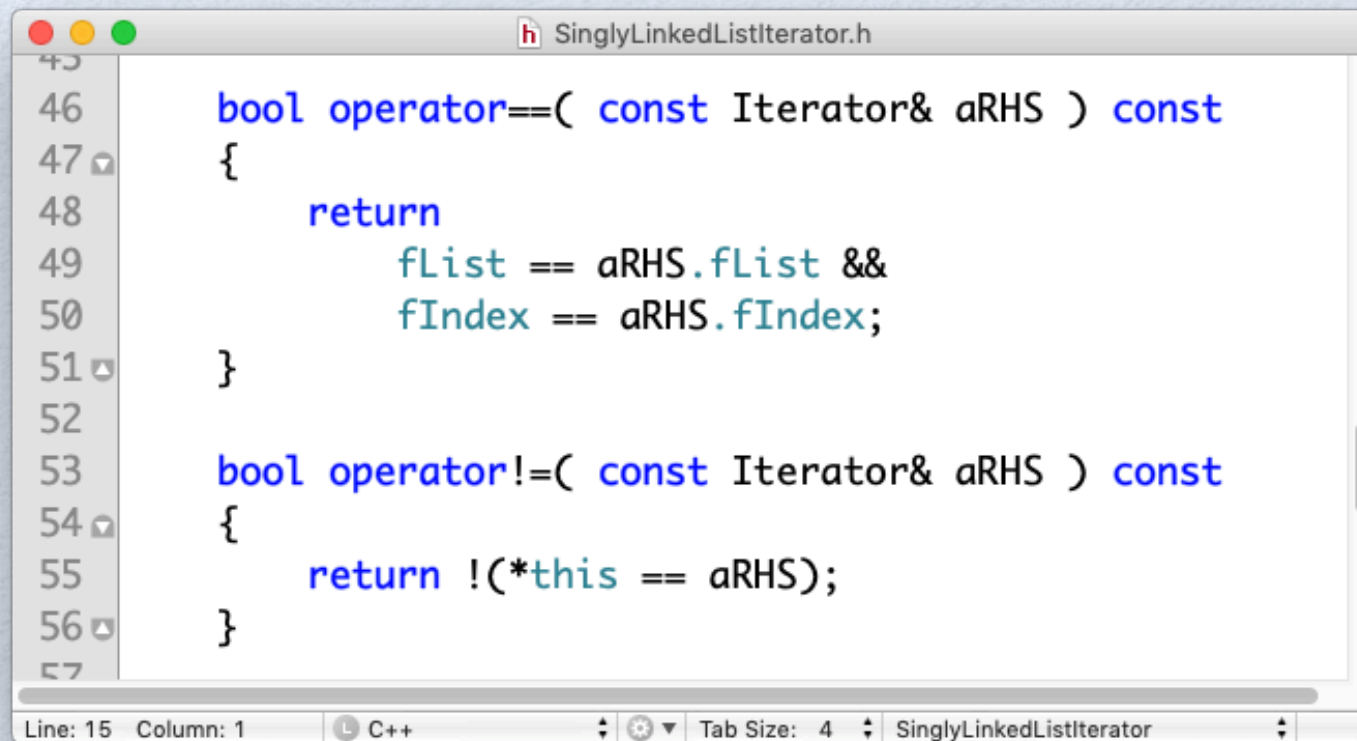
262

Increments

```cpp
Iterator& operator++()        // prefix
{
    fIndex = fIndex->fNext;

    return *this;
}


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

    ++(*this);

    return old;
}
```

Line: 15   Column: 1   C++   Tab Size: 4   SinglyLinkedListIterator

Equivalence


```cpp
bool operator==( const Iterator& aRHS ) const
{
    return
        fList == aRHS.fList &&
        fIndex == aRHS.fIndex;
}

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

Auxiliaries (For-Range)
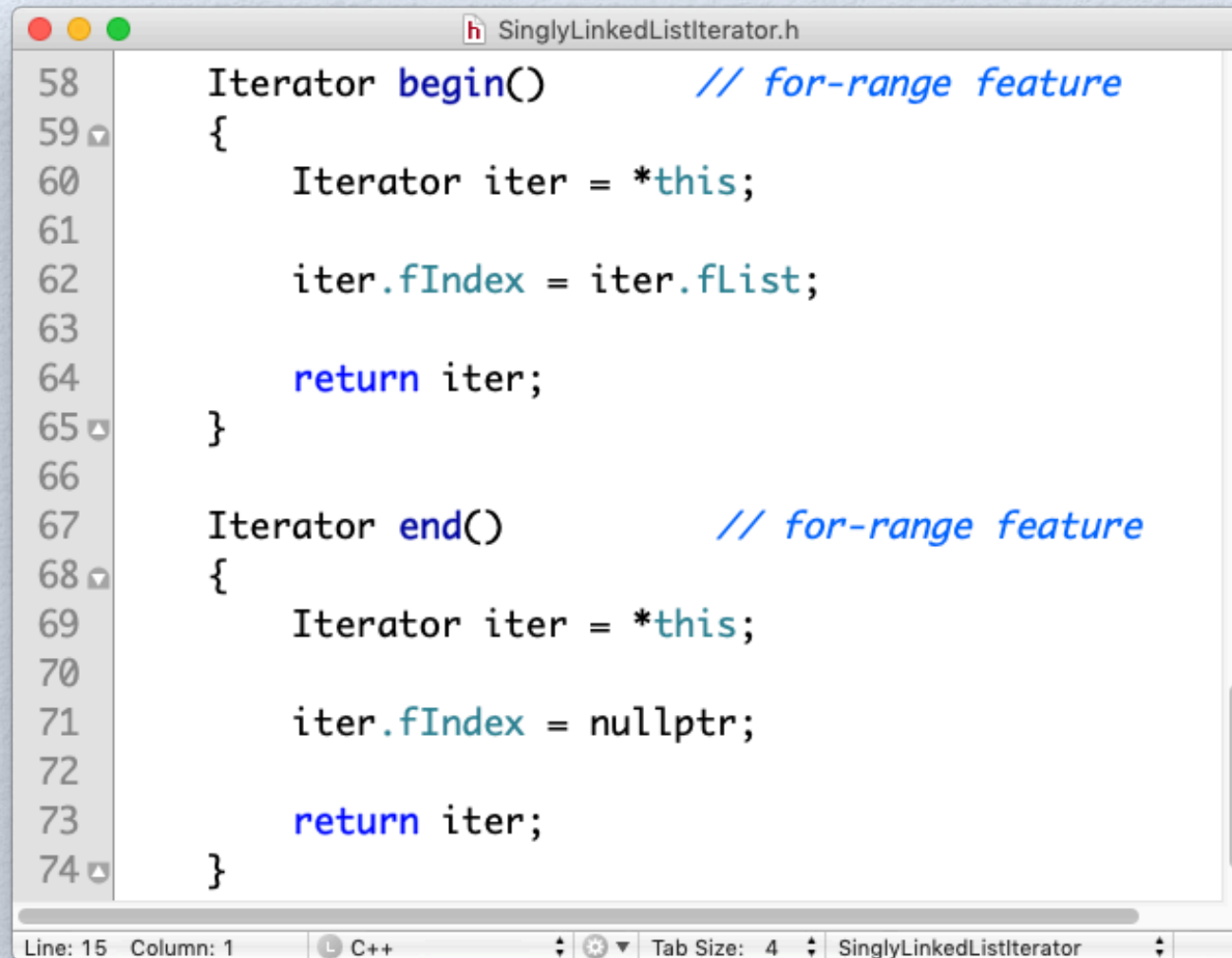
```cpp
58     Iterator begin()        // for-range feature
59     {
60         Iterator iter = *this;
61
62         iter.fIndex = iter.fList;
63
64         return iter;
65     }
66
67     Iterator end()          // for-range feature
68     {
69         Iterator iter = *this;
70
71         iter.fIndex = nullptr;
72
73         return iter;
74     }
```

SinglyLinkedListIterator.h

Line: 15   Column: 1   C++   Tab Size: 4   SinglyLinkedListIterator

SinglyLinkedList Iterator Test

```cpp
#include <iostream>
#include <string>

using namespace std;

#include "SinglyLinkedListIterator.h"

int main()
{
    using StringList = SinglyLinkedList<string>;
    using StringListIterator = SinglyLinkedListIterator<string>;

    string lA = "AAAA";
    string lC = "CCCC";

    StringList One( lA );
    StringList Two( "BBBB", &One );
    StringList Three( lC, &Two );

    for ( const string& i : StringListIterator( &Three ) )
    {
        cout << "Value: " << i << endl;
    }

    return 0;
}
```

For iterator implementation see Canvas

address of Three

SinglyLinkedList Iterator: Specification B

```cpp
#include "SinglyLinkedListTemplate.h"

template <typename T>
class SinglyLinkedListIterator
{
private:

    using ListNode = SinglyLinkedList<T>;

    const ListNode& fList;
    const ListNode* fIndex;

public:

    using Iterator = SinglyLinkedListIterator<T>;

    SinglyLinkedListIterator( const ListNode& aList );

    const T& operator*() const;
    Iterator& operator++();        // prefix
    Iterator operator++(int);      // postfix
    bool operator==( const Iterator& aRHS ) const;
    bool operator!=( const Iterator& aRHS ) const;

    Iterator begin();              // for-range feature
    Iterator end();                // for-range feature
};
```

We maintain a read-only reference to list elements

267
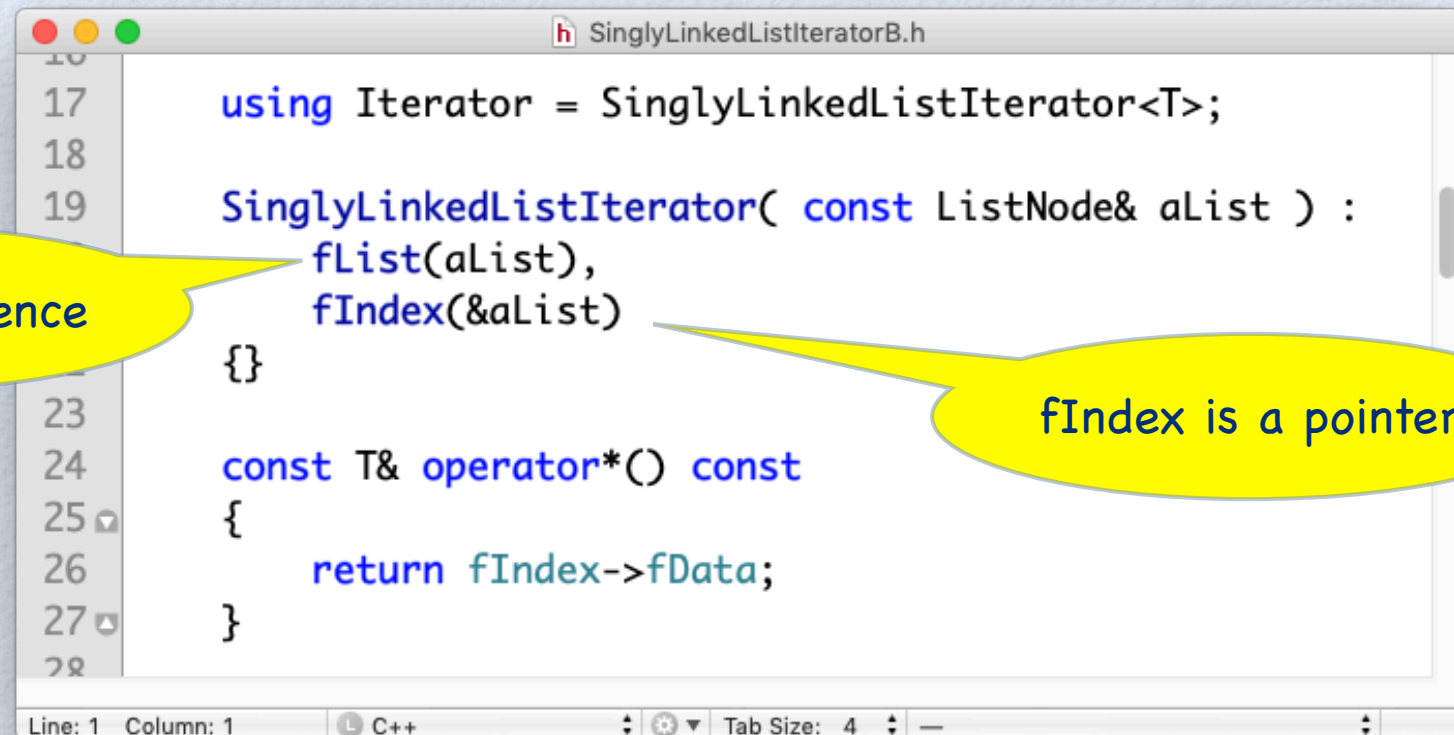
Reference Data Members

```
class ClassWithRefMember
{
private:
    SomeType& fRef;
public:
    ClassWithRefMember( SomeType& aRef ) : fRef(aRef)
    { ... }
};
```

> Reference data members must be initialized before the contructor body is entered!

- Reference member variables store references to data outside an object. These references are established upon object creation via a member initializer.

- In case of iterators this might be an attractive option to avoid coping the underlying collection.

- Important: Reference data members require a constructor initializer.

Constructor & Deference B
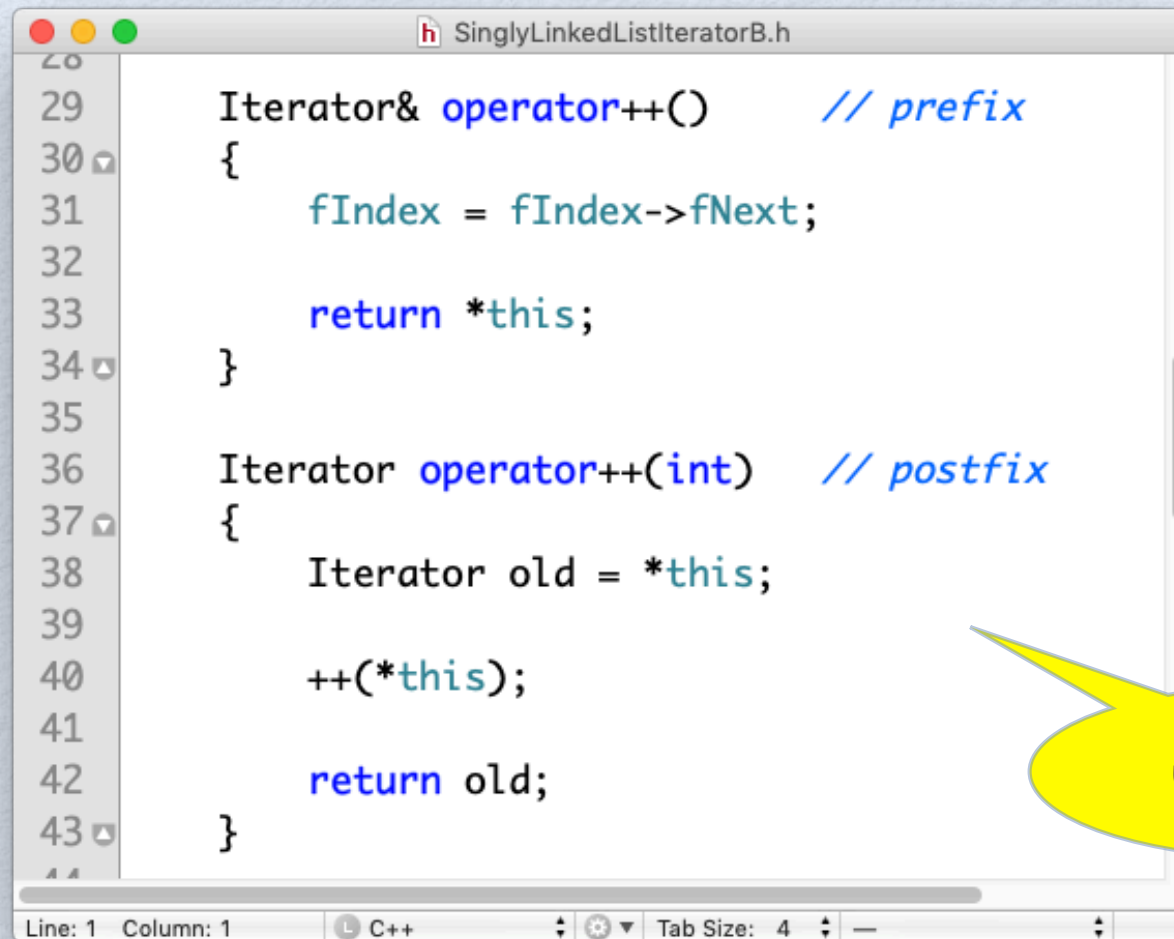

```cpp
using Iterator = SinglyLinkedListIterator<T>;

SinglyLinkedListIterator( const ListNode& aList ) :
    fList(aList),
    fIndex(&aList)
{}


const T& operator*() const
{
    return fIndex->fData;
}
```

Establish reference

fIndex is a pointer

© Dr Markus Lumpe, 2022

269

Increments B

```cpp
Iterator& operator++()        // prefix
{
    fIndex = fIndex->fNext;

    return *this;
}

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

    ++(*this);

    return old;
}
```

unchanged

Equivalence B


```cpp
bool operator==( const Iterator& aRHS ) const
{
    return
        &fList == &aRHS.fList &&
        fIndex == aRHS.fIndex;
}


bool operator!=( const Iterator& aRHS ) const
{
    return !(*this == aRHS);
}
```
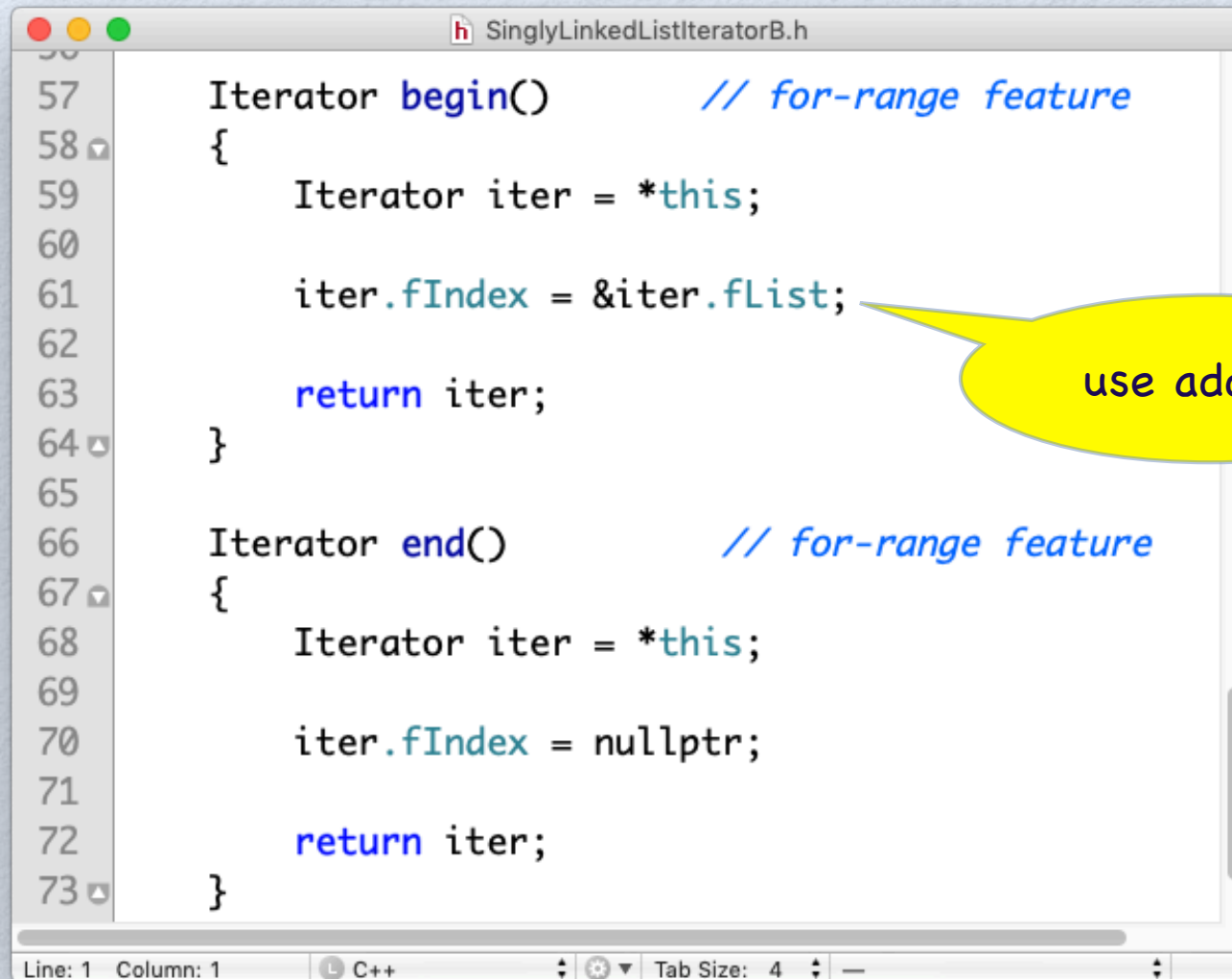
compare addresses

271

Auxiliaries (For-Range) B

```cpp
57     Iterator begin()            // for-range feature
58     {
59         Iterator iter = *this;
60
61         iter.fIndex = &iter.fList;
62
63         return iter;
64     }
65
66     Iterator end()              // for-range feature
67     {
68         Iterator iter = *this;
69
70         iter.fIndex = nullptr;
71
72         return iter;
73     }
```

use address

272

SinglyLinkedList Iterator Test B

```cpp
#include <iostream>
#include <string>

using namespace std;

#include "SinglyLinkedListIteratorB.h"

int main()
{
    using StringList = SinglyLinkedList<string>;
    using StringListIterator = SinglyLinkedListIterator<string>;

    string lA = "AAAA";
    string lC = "CCCC";

    StringList One( lA );
    StringList Two( "BBBB", &One );
    StringList Three( lC, &Two );

    for ( const string& i : StringListIterator( Three ) )
    {
        cout << "Value: " << i << endl;
    }

    return 0;
}
```

For iterator implementation see Canvas

Three passed as l-value reference

Main.cpp — Programs

Line: 30   C++   Tab Size: 4   main

273

Pointers

The Need for Pointers

- A linked-list is a dynamic data structure with a varying number of nodes.

- Access to a linked-list is through a pointer variable in which the base type is the same as the node type:

    IntegerList<int>* pListOfIntegers = &Three;

    IntegerList<int>* Nil = nullptr;

  Nil means "empty list."

Node Construction

IntegerList *p, *q;

p = new IntegerList(5);

q = new IntegerList(7, p);

Node Access

int a = p->fData;

int b = q->fNext->fData;
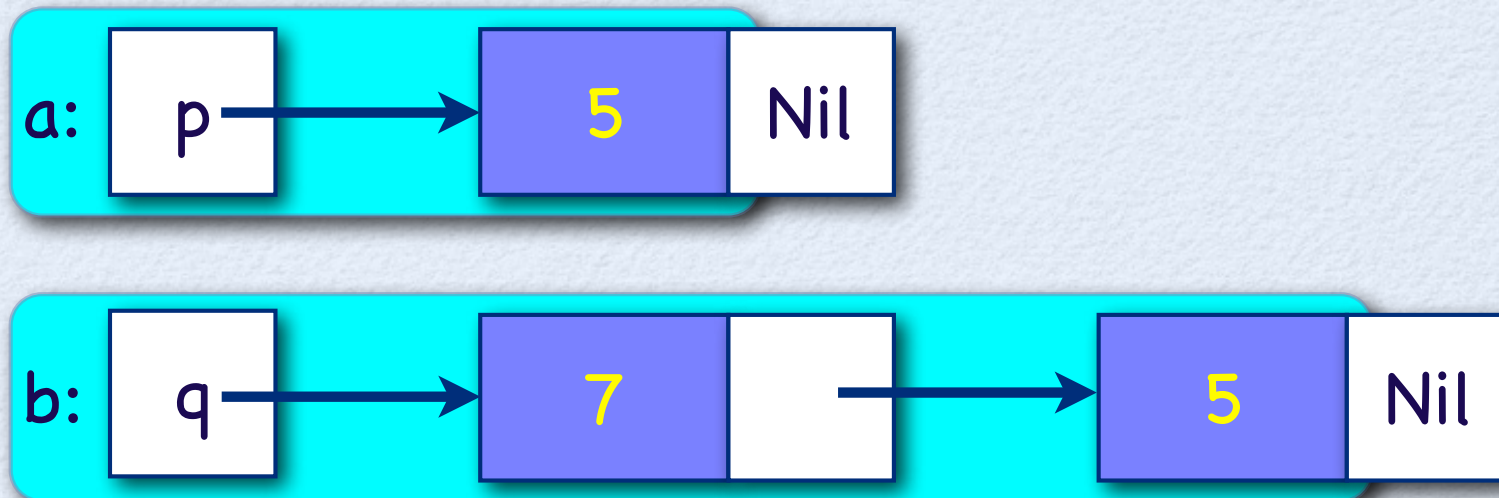
Inserting a Node
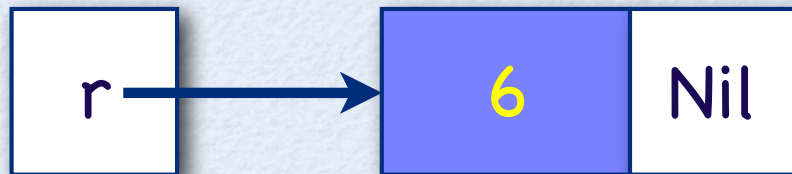

```
IntegerList *r;

r = new IntegerList( 6 );
```
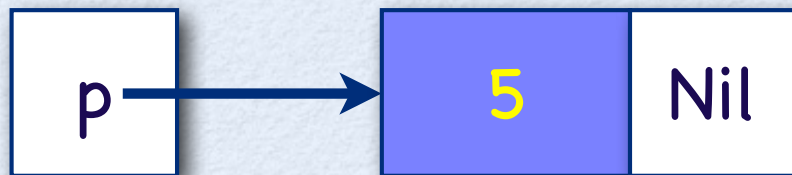
```
r->fNext = p;

q->fNext = r;
```
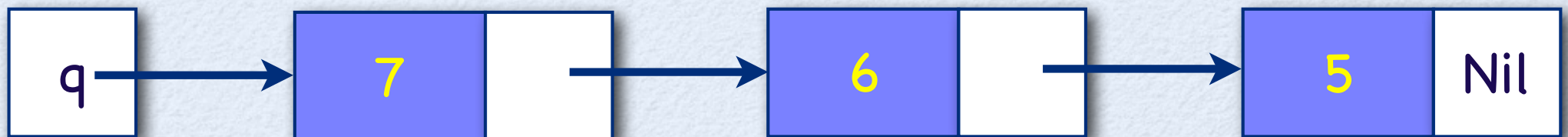
Deleting a Node
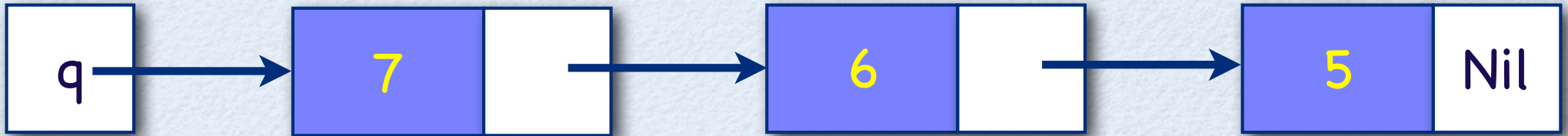
q->fNext = q->fNext->fNext;
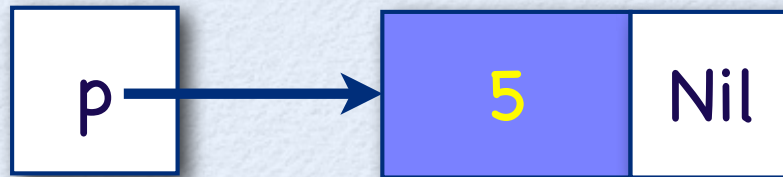
Insert at the Top

IntegerList *p = nullptr;

p = new IntegerList(5, p);

```
p  →  5 | Nil
```
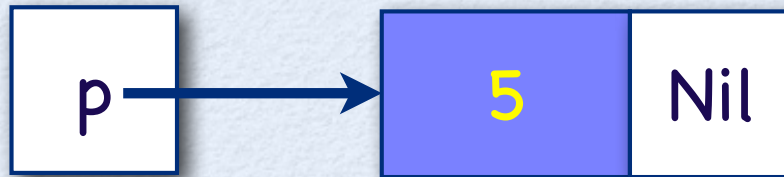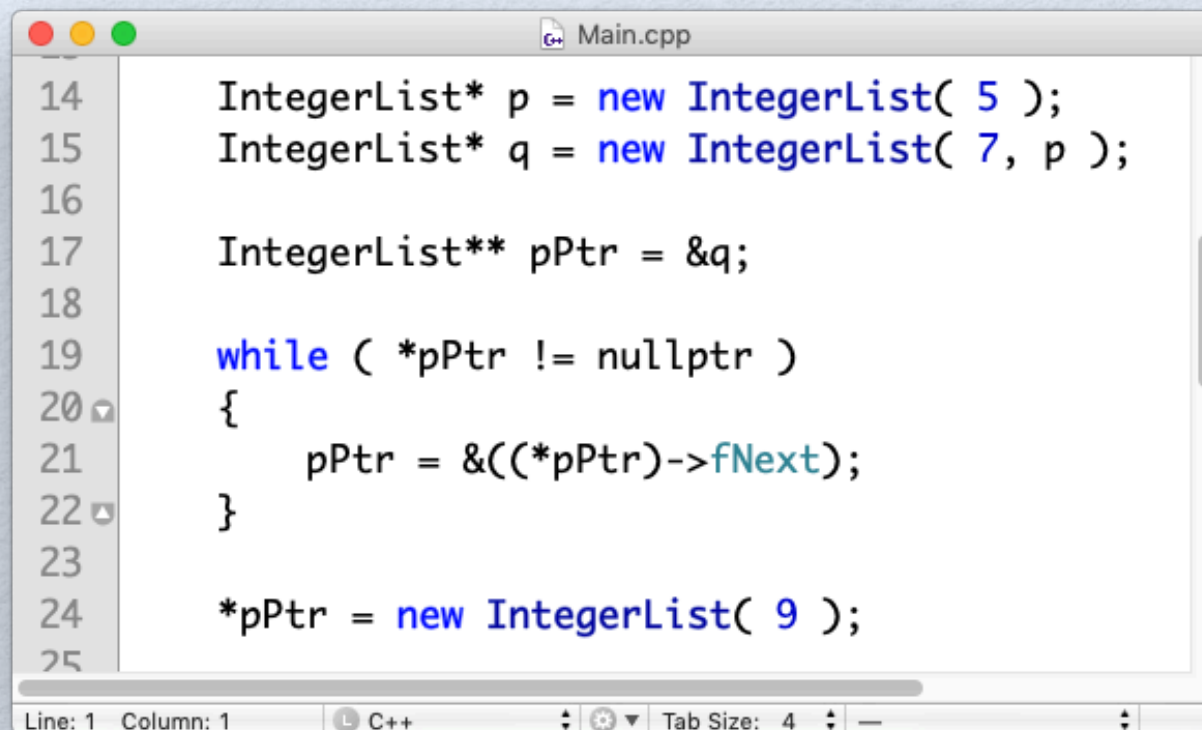
p = new IntegerList(7, p);

```
p  →  7 |  →  5 | Nil
```

Insert at the End

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

```cpp
IntegerList* p = new IntegerList( 5 );
IntegerList* q = new IntegerList( 7, p );

IntegerList** pPtr = &q;

while ( *pPtr != nullptr )
{
    pPtr = &((*pPtr)->fNext);
}

*pPtr = new IntegerList( 9 );
```
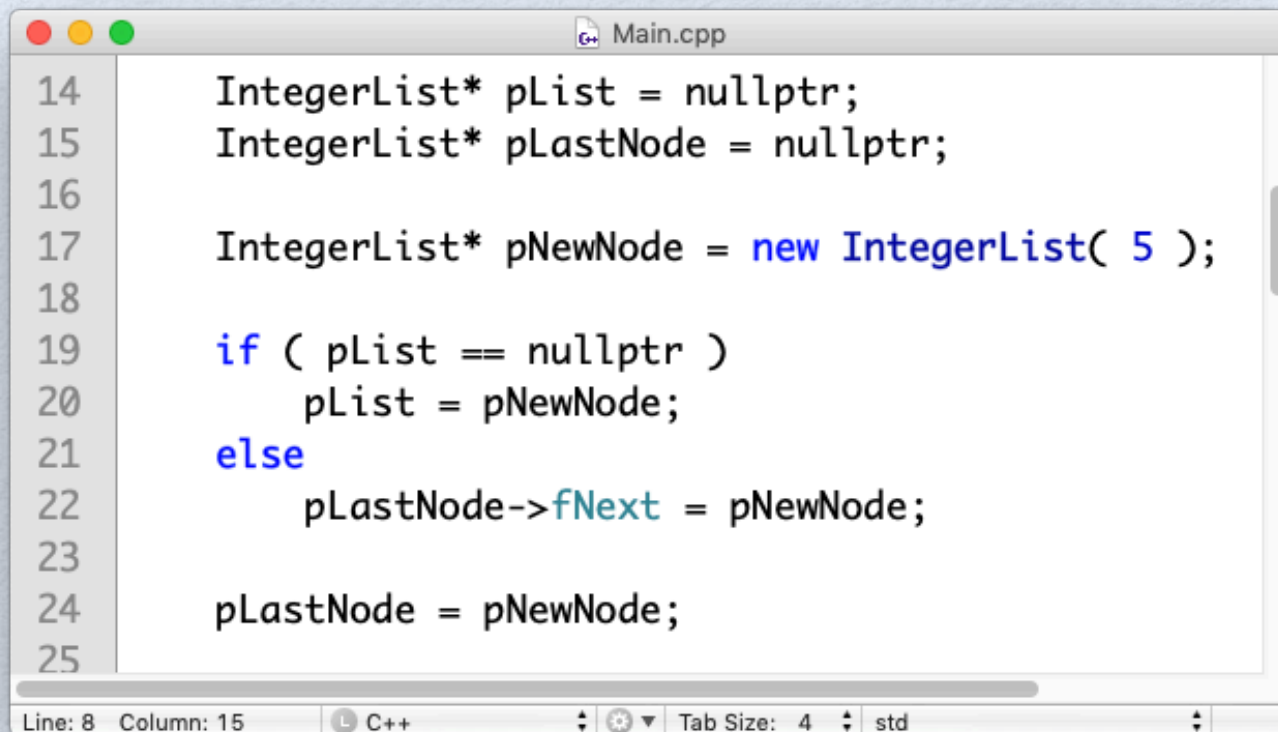
Insert at the End: The Pointers
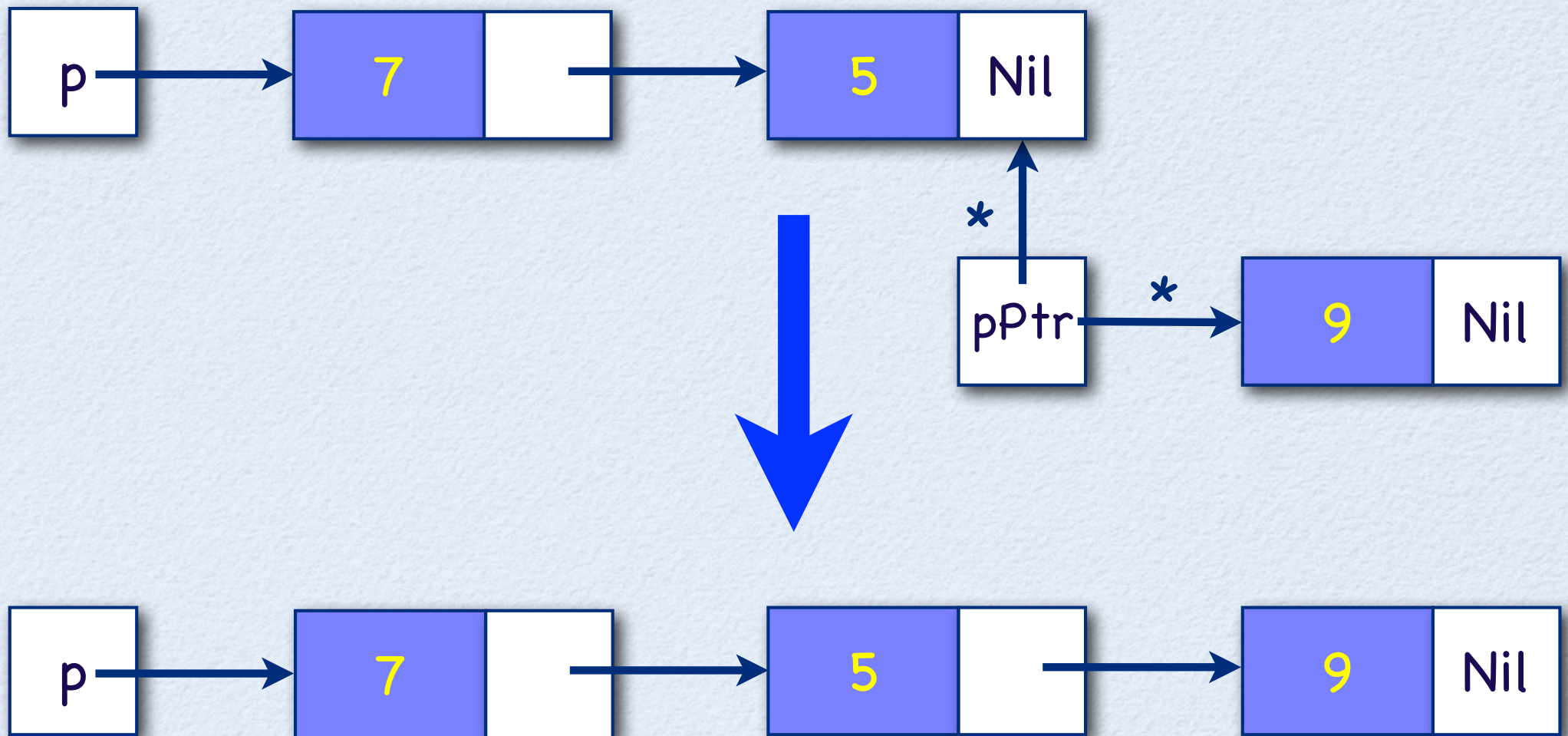
Insert at the end preserves the order of list nodes.

Insert at the End with Aliasing

- Rather than using a Pointer-to-Pointer we can just record the last next pointer.

```cpp
      IntegerList* pList = nullptr;
      IntegerList* pLastNode = nullptr;

      IntegerList* pNewNode = new IntegerList( 5 );

      if ( pList == nullptr )
          pList = pNewNode;
      else
          pLastNode->fNext = pNewNode;

      pLastNode = pNewNode;
```

Complications with Singly-Linked Lists

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