



Abstraction

New Sets of Values

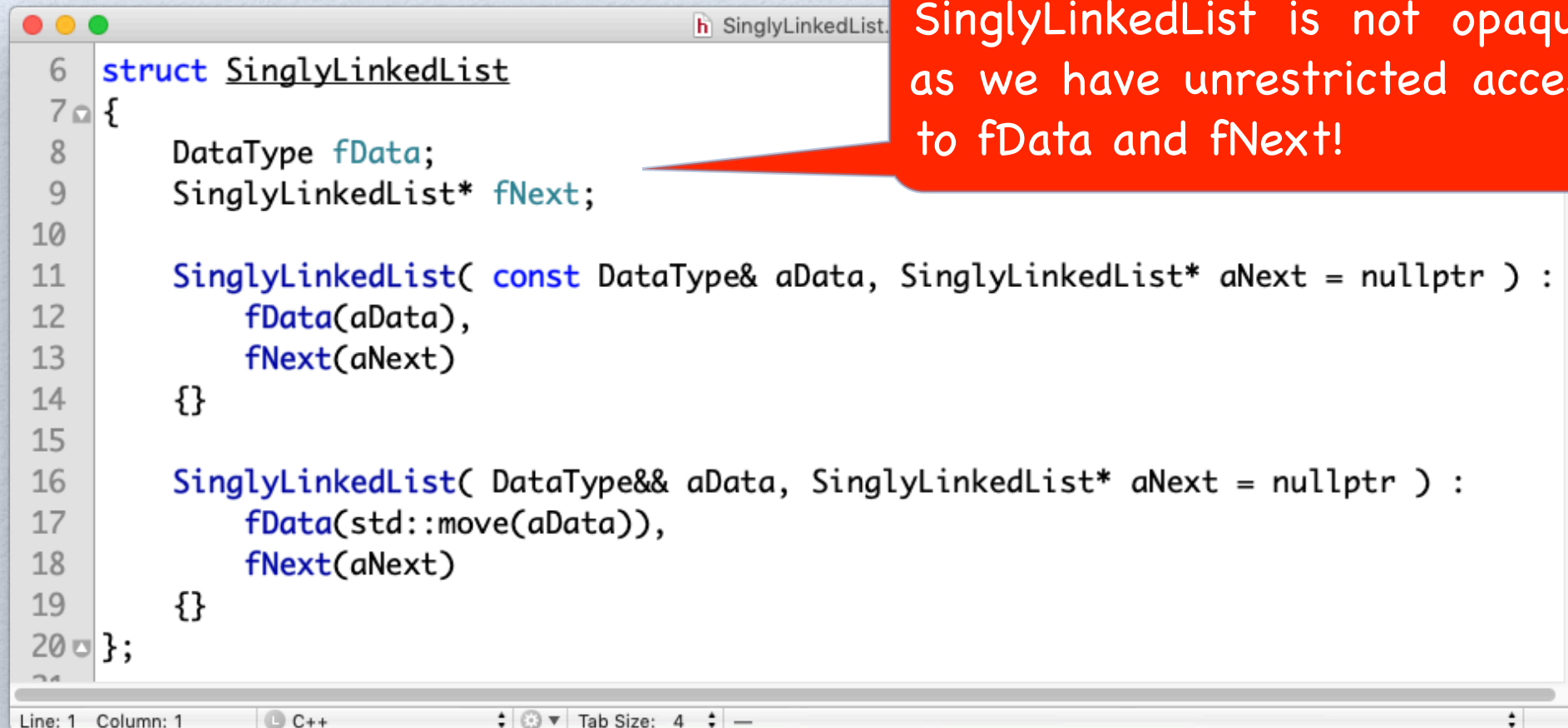
- The definition of a new data type (i.e., a new set of values) consists of two ingredients:
 - Some set, called the **interface**, that serves as representation of the newly define data type, and
 - Some set of procedures, called the **implementation**, that provides the operations, which can be used to manipulate the newly defined data type.

Representation Independence

- The representation of new data types can be often very complex.
- When working with new data types, we usually do not want to be concerned with their actual representation. In fact, program become more reliable and robust, if they do not depend on the actual representation of data type. Data types that do not expose their actual representation are called opaque. Otherwise, they are called transparent.
- Data types in C/C++ are in general transparent (e.g. the size of integers in C/C++ is platform dependent).
- Data types in Java are basically opaque (arrays are an exception, since they are represented by objects).

Opaque Representation

- A data type is **opaque** if there is no way to find out its representation, even by printing.



```
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 };
```

SinglyLinkedList is not opaque as we have unrestricted access to fData and fNext!

Object-Oriented Encapsulation

- Object-oriented encapsulation is a principle that provides the means to obtain an opaque data type:
 - All instance variables have private visibility.
 - All member functions have public visibility.
- The extent to which this scheme is used can differ!
- Opaque here does not necessarily mean that you cannot see the implementation, just that there are no pragmatic approaches to exploit this knowledge.

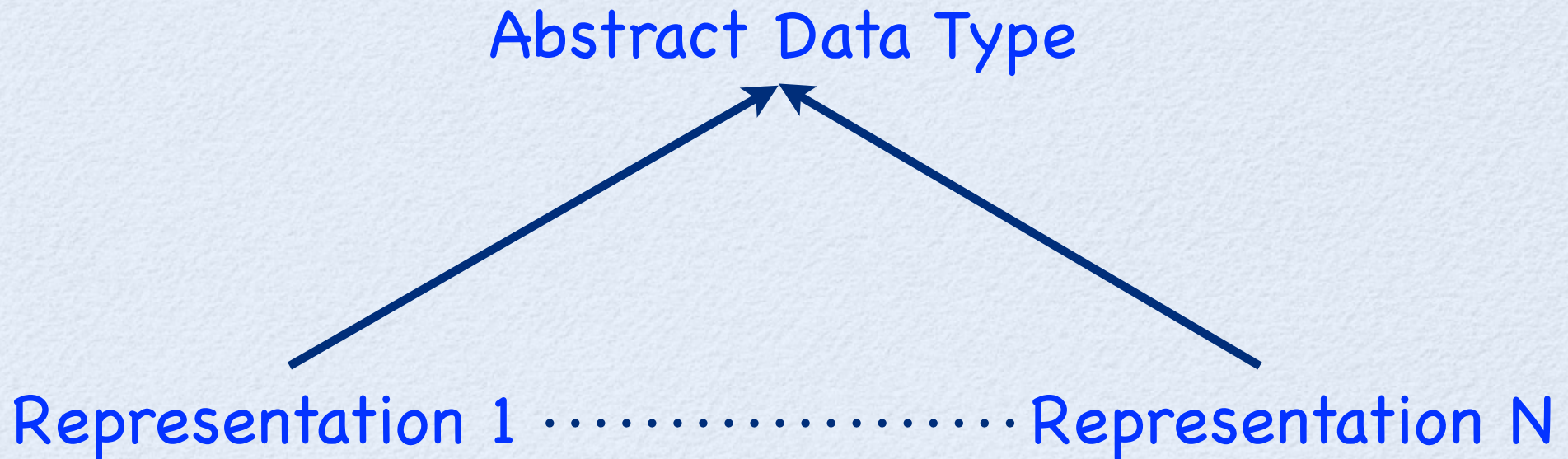
Pros & Cons

- Opaque data types enforce the use of defining procedure (i.e., a **constructor**).
- **Opaque data types are more secure.** Access to values of opaque data types is only possible by means of access procedures defined in an interface.
- **Transparent data types are easier to debug and to extend.**
- The fact that transparent data types expose their internal representation is also a disadvantage (limited security).

Abstract Data Type

- The technique used to define new data types independently of their actual representation is called **data abstraction**.
- A data type, which has been defined in this way is called **abstract data type**. A client (program) can use values of an abstract data type by means of the interface without knowing their actual representation (which can change over time).

Representation Strategies for ADTs



- Given an interface for a data type we can change the underlying representation if needed using different strategies.

Data Abstraction

- Data abstraction enforces representation independence.
- Data abstraction divides the data types in interfaces and implementations:
 - **Interfaces** are used to specify the set of values the data types represents, the operations, which are available for that data type, and properties these operations may be guaranteed to have.
 - **Implementations** provide a specific representation of the data and code for the operations.

Examples of Abstract Data Types

- Files
- Lists, hash tables, vectors, bags
- Strings, records, arrays
- Objects with private instance variables and public methods
- Standardized integers (e.g. in Java the type `int` is represented using 32 bits and big endian format, network byte order, on every platform)

Constructors and Access Procedures

- In order to create, manipulate, and verify that a given value is of the desired data type, we need the following ingredients:
 - **Constructors** that allow us to build values of a given data type,
 - A **predicate** that tests whether a given value is a representation of a particular data type, and
 - Some **access procedures** that allow us to extract a particular information from a given representation of a data type.

**Can we represent lists as
abstract data type?**

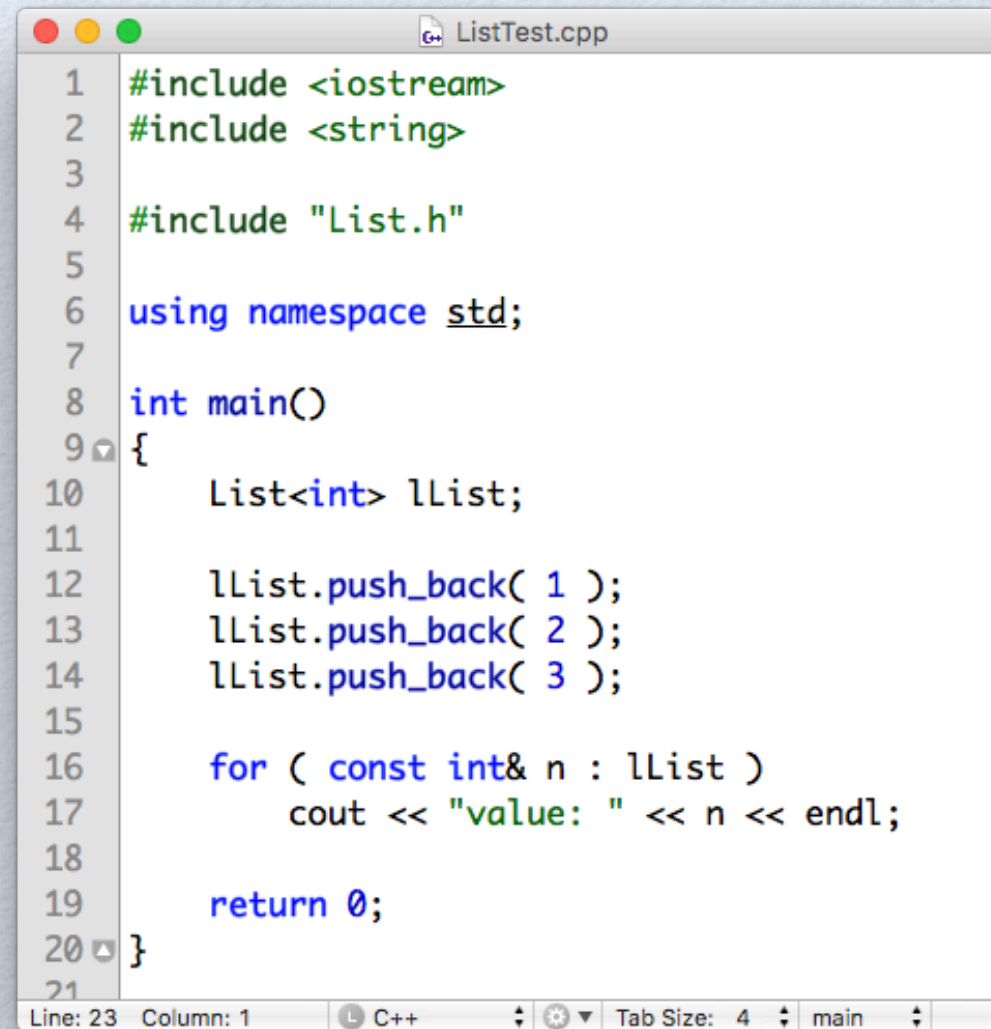

```

6 #include "DoublyLinkedList.h"
7 #include "DoublyLinkedListIterator.h"
8
9 #include <stdexcept>
10
11 template<class T>
12 class List
13 {
14 private:
15     // auxiliary definition to simplify node usage
16     using Node = DoublyLinkedList<T>;
17
18     Node* fRoot;    // the first element in the list
19     int fCount;     // number of elements in the list
20
21 public:
22     // auxiliary definition to simplify iterator usage
23     using Iterator = DoublyLinkedListIterator<T>;
24
25     List();           // default constructor - creates empty list
26     List( const List& aOtherList ); // copy constructor
27     List( List&& aOtherList );      // move constructor
28     List& operator=( const List& aOtherList ); // assignment operator
29     List& operator=( List&& aOtherList );      // move assignment operator
30     ~List();           // destructor - frees all nodes
31
32     bool isEmpty() const;           // Is list empty?
33     int size() const;              // list size
34
35     void push_front( const T& aElement ); // adds aElement at front (copy)
36     void push_front( T&& aElement );      // adds aElement at front (move)
37     void push_back( const T& aElement );  // adds aElement at back (copy)
38     void push_back( T&& aElement );       // adds aElement at back (move)
39     void remove( const T& aElement );     // remove first match from list
40
41     const T& operator[]( size_t aIndex ) const; // list indexer
42
43     Iterator begin() const;           // return a forward iterator
44     Iterator end() const;            // return a forward end iterator
45     Iterator rbegin() const;         // return a backwards iterator
46     Iterator rend() const;           // return a backwards end iterator
47 };

```

r-value variant

List Test



```
1  #include <iostream>
2  #include <string>
3
4  #include "List.h"
5
6  using namespace std;
7
8  int main()
9  {
10     List<int> lList;
11
12     lList.push_back( 1 );
13     lList.push_back( 2 );
14     lList.push_back( 3 );
15
16     for ( const int& n : lList )
17         cout << "value: " << n << endl;
18
19     return 0;
20 }
21
```

Line: 23 Column: 1 C++ Tab Size: 4 main

The type `list<T>` is part of the standard template library.