

# Container Types, Stacks, and Queues

## Overview

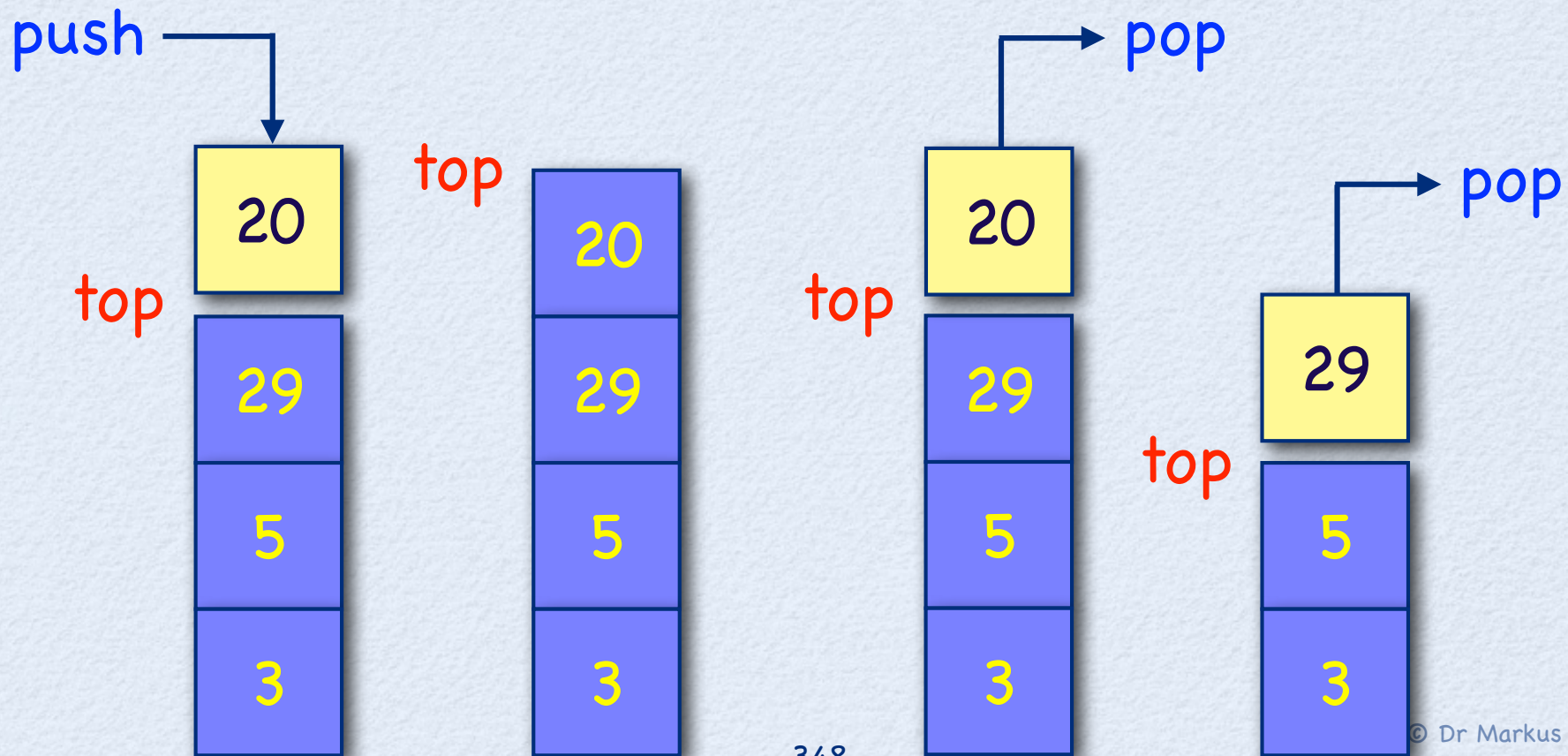
- Stacks
- Container types and references

## References

- Bruno R. Preiss: Data Structures and Algorithms with Object-Oriented Design Patterns in C++. John Wiley & Sons, Inc. (1999)
- Richard F. Gilberg and Behrouz A. Forouzan: Data Structures – A Pseudocode Approach with C. 2nd Edition. Thomson (2005)
- Nicolai M. Josuttis: The C++ Standard Library – A Tutorial and Reference. Addison-Wesley (1999)
- Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo: C++ Primer. 5th Edition. Addison-Wesley (2013)

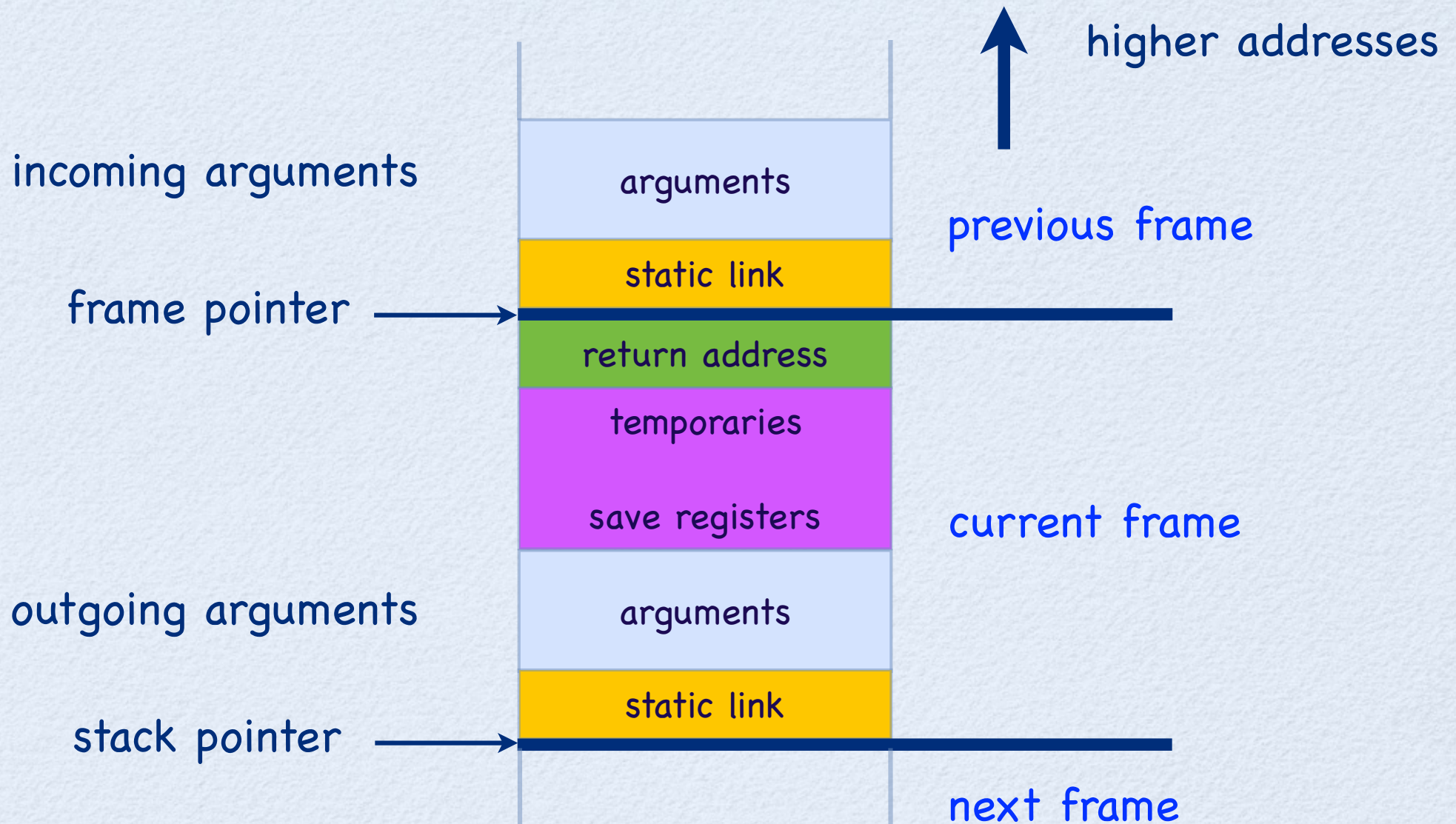
# Stacks

- A **stack** is a special version of a linear list where items are added and deleted at only one end called the **top**.





# Stack Frames (PASCAL)



# Stack Behavior

- Stacks manage elements in last-in, first-out (LIFO) manner.
- A stack underflow happens when one tries to pop on an empty stack.
- A stack overflow happens when one tries to push onto a full stack.



# Applications of Stacks

- Reversal of input (like `push_front` for `List<T>`)
- Checking for matching parentheses (e.g., stack automata in compiler implementations)
- Backtracking (e.g., Prolog or graph analysis)
- State of program execution (e.g., storage for parameters and local variables of functions)
- Tree traversal

# Reverse Polish Notation

- Reverse Polish Notation (RPN) is a postfix notation wherein operands come before operators.

RPN:  $a \ b \ * \ c \ +$



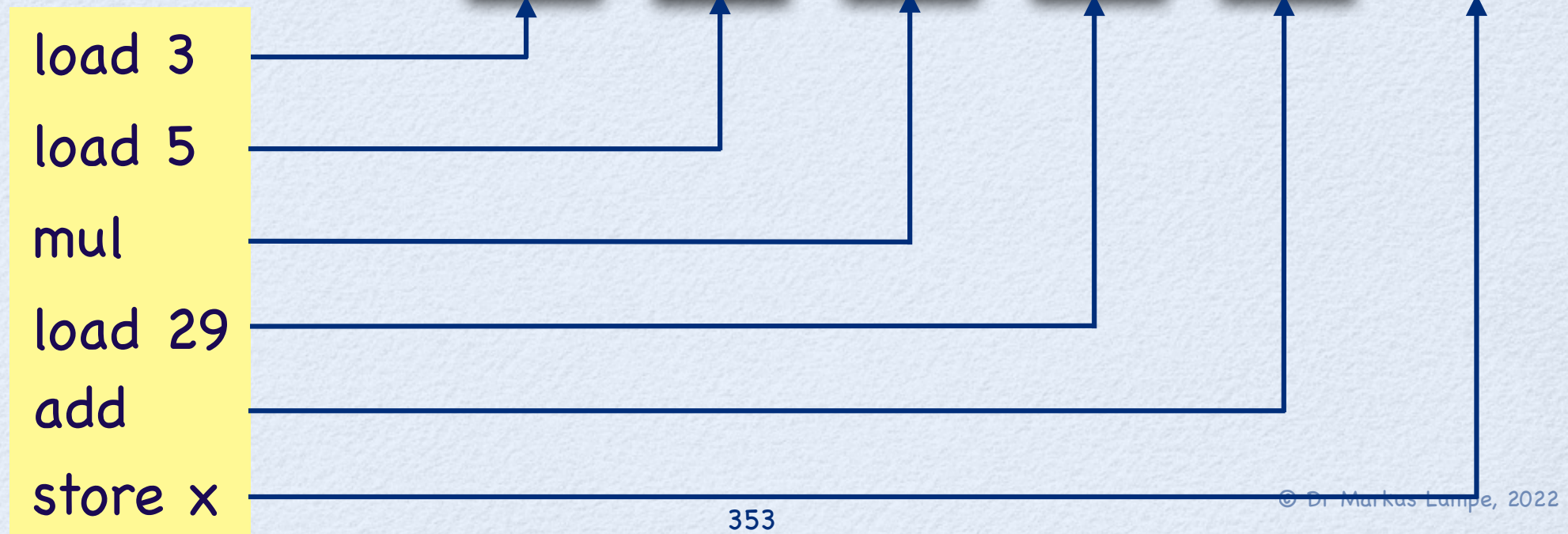
Infix:  $a \ * \ b \ + \ c$



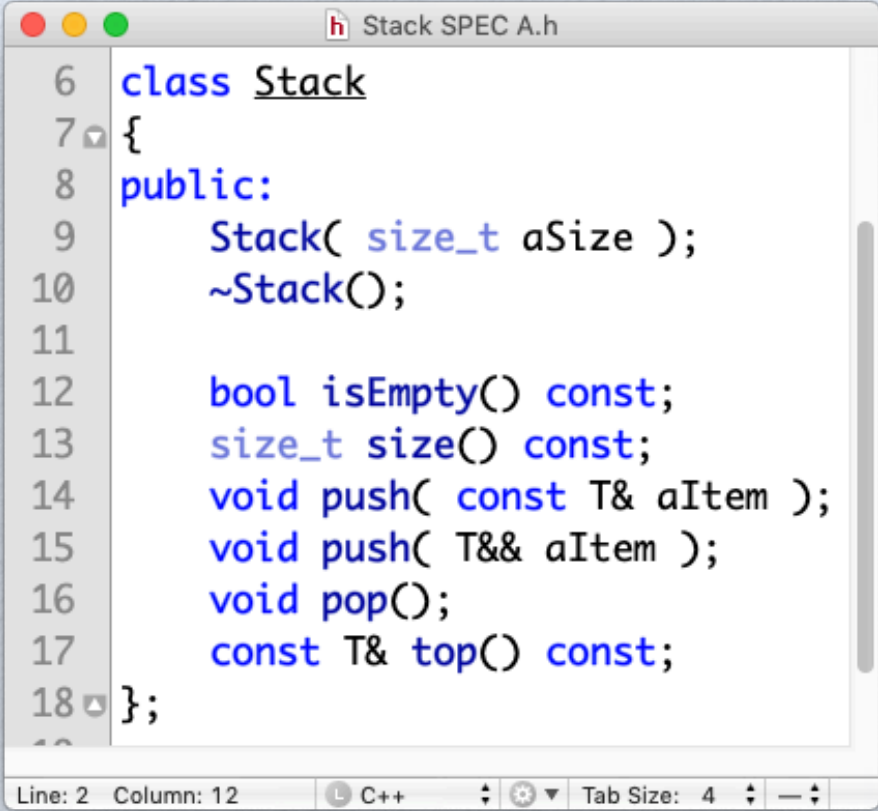
# RPN Calculation

$$\bullet x = 3 * 5 + 29$$

Stack:



# Stack Interface

A screenshot of a code editor window titled "Stack SPEC A.h". The editor shows the following C++ code:

```
6 class Stack
7 {
8 public:
9     Stack( size_t aSize );
10    ~Stack();
11
12    bool isEmpty() const;
13    size_t size() const;
14    void push( const T& aItem );
15    void push( T&& aItem );
16    void pop();
17    const T& top() const;
18 };
```

The code is color-coded: keywords like 'class', 'public', 'const', and 'void' are in blue, while identifiers and literals are in black. The editor has a line number margin on the left and a status bar at the bottom showing "Line: 2 Column: 12", "C++", and "Tab Size: 4".

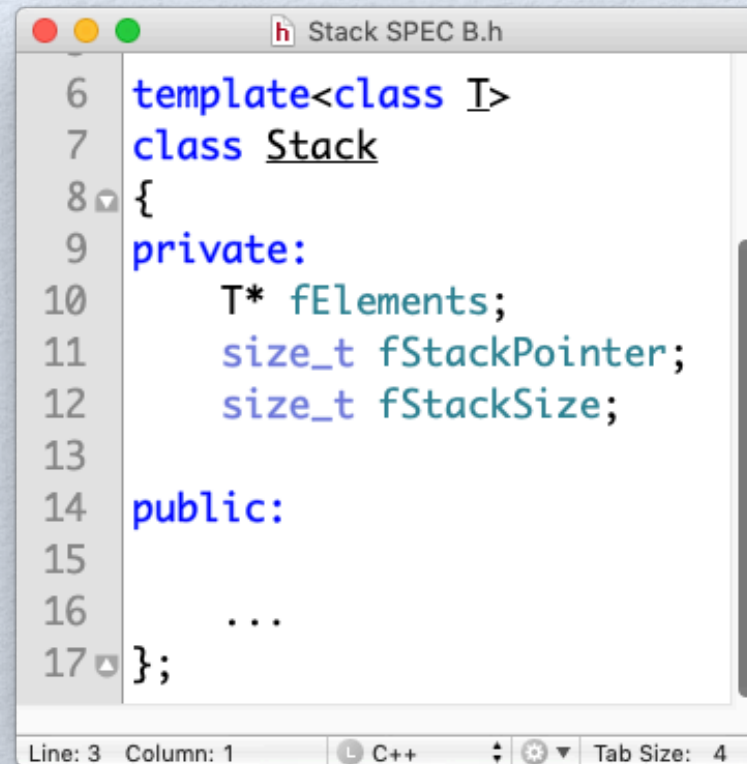
- When defining a **container type** we wish to minimize the number of value copies required for the objects stored in the container. In order to achieve this, we use references.



# Container Types

- Stacks belong to a special group of data types called container types.
- The *de facto* standard approach for the definition of container types in C++ is to use *value-based semantics*.
- Other examples of container types are Lists, Queues, Hash Tables, Maps, Arrays, or Trees.

# The Stack's Private Interface

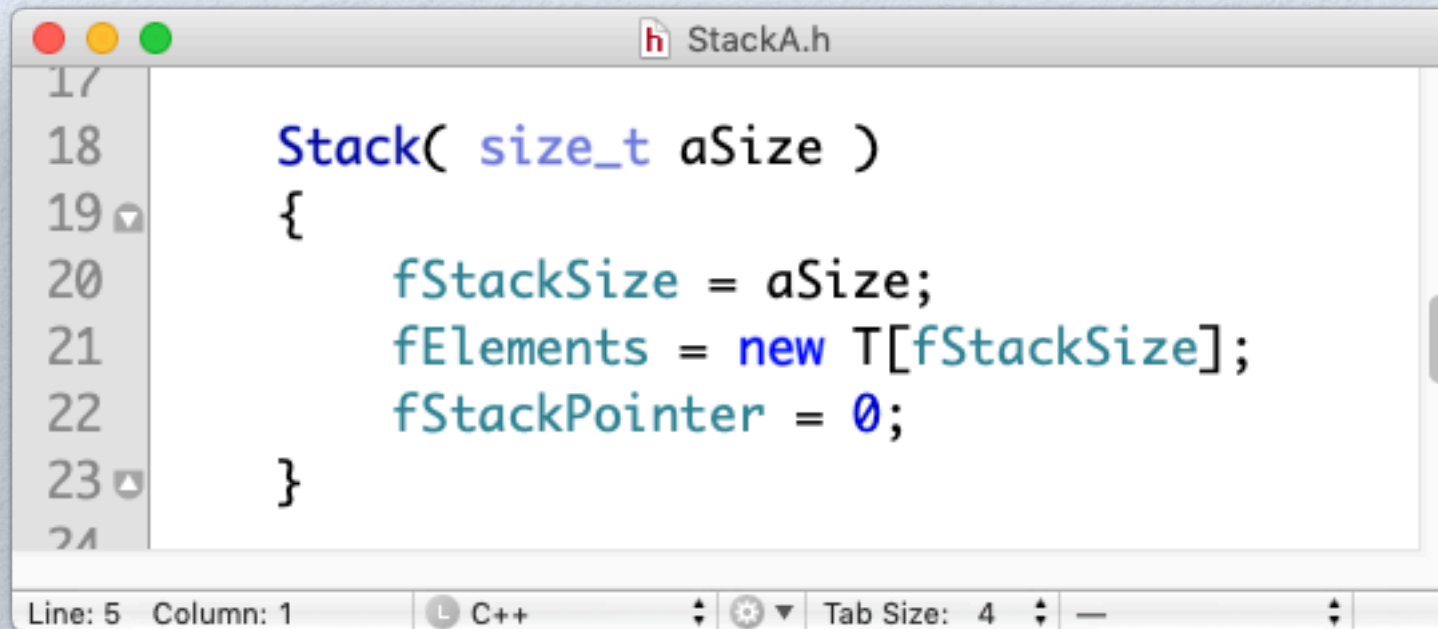


```
Stack SPEC B.h
6  template<class T>
7  class Stack
8  {
9  private:
10     T* fElements;
11     size_t fStackPointer;
12     size_t fStackSize;
13
14 public:
15
16     ...
17 };
Line: 3 Column: 1 C++ Tab Size: 4
```

- Inside Stack we need to be able to store objects of type T. Hence we need to dynamically allocate memory (i.e, an array of type T) and store the address of the first element in a matching pointer variable.



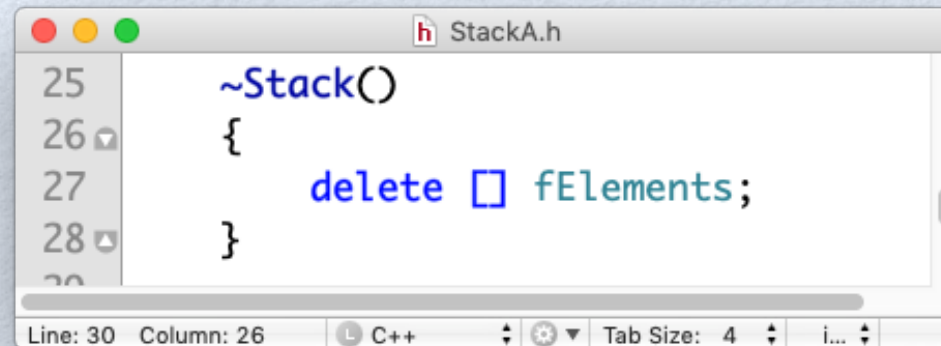
# Stack Constructor



```
17
18 Stack( size_t aSize )
19 {
20     fStackSize = aSize;
21     fElements = new T[fStackSize];
22     fStackPointer = 0;
23 }
24
```

Line: 5 Column: 1 C++ Tab Size: 4

# Stack Destructor

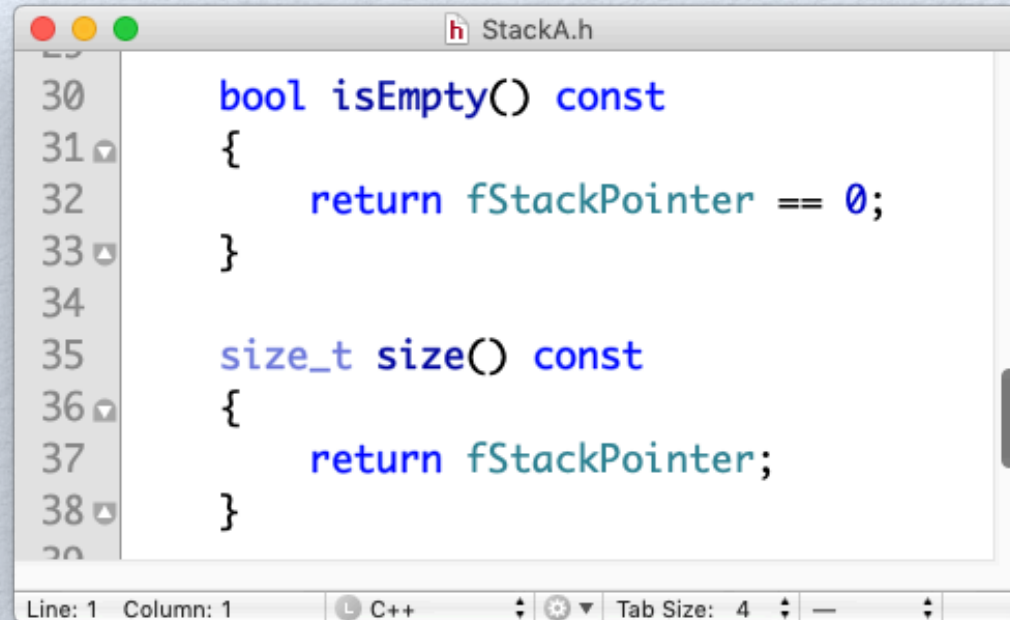


```
25 ~Stack()  
26 {  
27     delete [] fElements;  
28 }  
29  
Line: 30 Column: 26 C++ Tab Size: 4
```

- There are two forms of delete:
  - `delete ptr` - release the memory associated with pointer `ptr`.
  - `delete [] ptr` - release the memory associated with all elements of array `ptr` and the array `ptr` itself.
- Whenever you allocate memory for an array of elements of a generic type, say `T* arr = new T[10]`, you must use the array form of delete, `delete []`, to guarantee that all array cells are released before the array itself is freed.



# Stack Auxiliaries

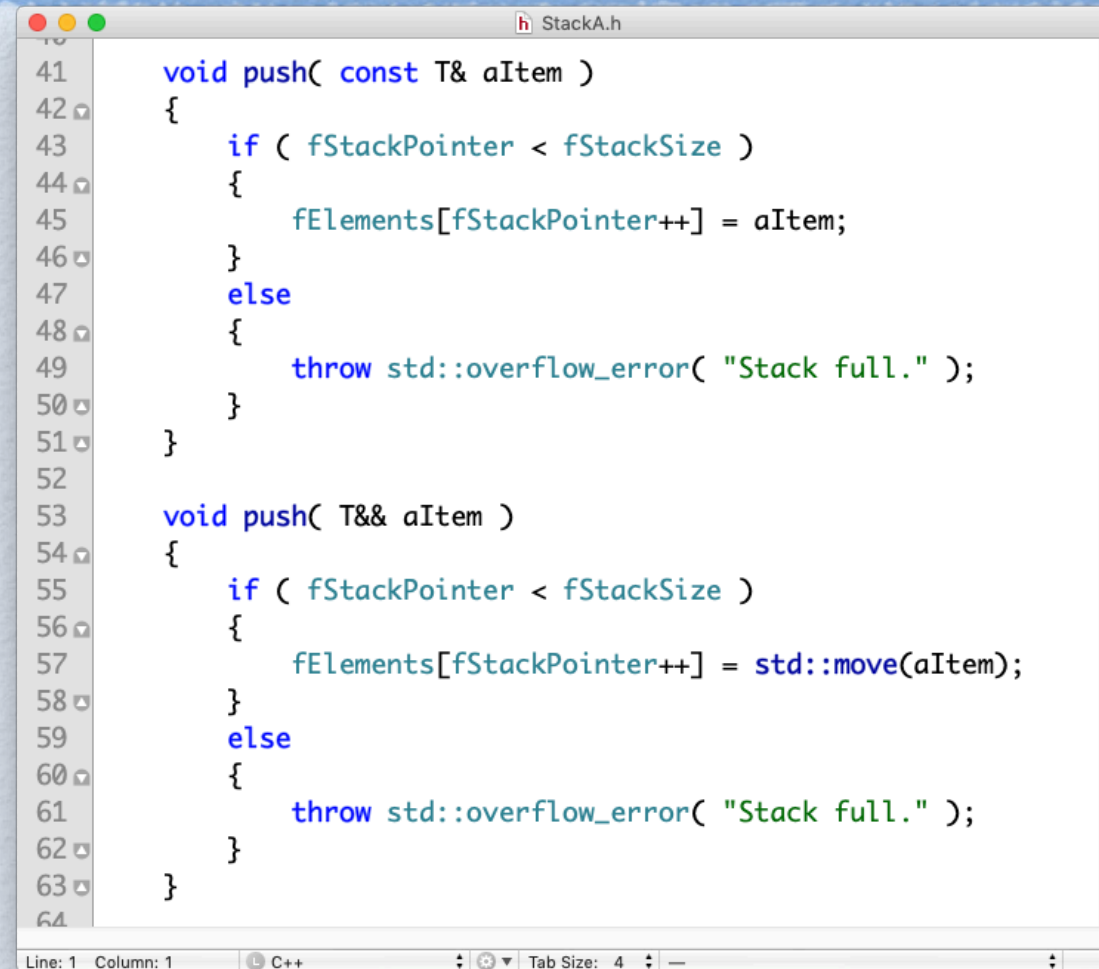


```
30     bool isEmpty() const
31     {
32         return fStackPointer == 0;
33     }
34
35     size_t size() const
36     {
37         return fStackPointer;
38     }
```

The screenshot shows a code editor window with the title 'StackA.h'. The code is written in C++ and defines two functions: `isEmpty()` and `size()`. The `isEmpty()` function returns `fStackPointer == 0`, and the `size()` function returns `fStackPointer`. The editor has a line number column on the left and a status bar at the bottom showing 'Line: 1 Column: 1', 'C++', and 'Tab Size: 4'.

- `isEmpty()`: Boolean predicate to indicate whether there are elements on the stack.
- `size()`: returns the actual stack size.

# Push

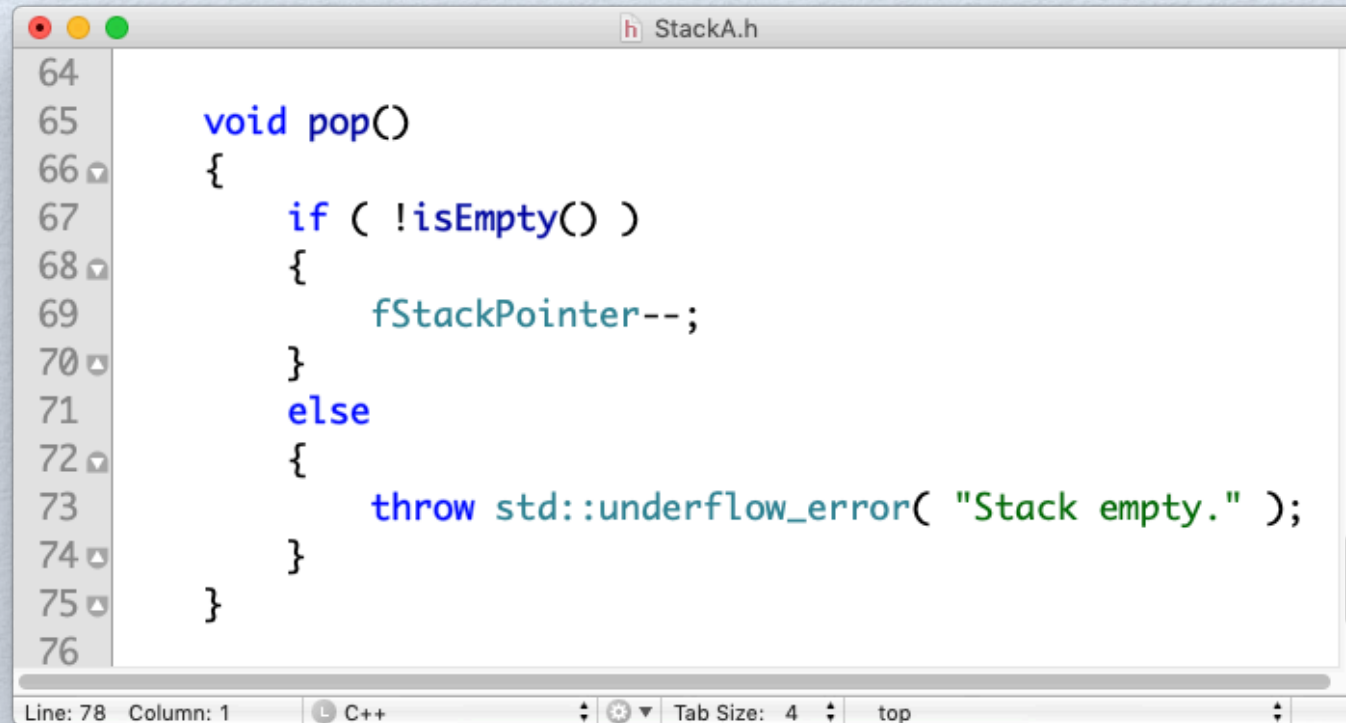


```
41 void push( const T& aItem )
42 {
43     if ( fStackPointer < fStackSize )
44     {
45         fElements[fStackPointer++] = aItem;
46     }
47     else
48     {
49         throw std::overflow_error( "Stack full." );
50     }
51 }
52
53 void push( T&& aItem )
54 {
55     if ( fStackPointer < fStackSize )
56     {
57         fElements[fStackPointer++] = std::move(aItem);
58     }
59     else
60     {
61         throw std::overflow_error( "Stack full." );
62     }
63 }
64
```

- The push method stores a item at the next free slot in the stack, if there is room.



# Pop

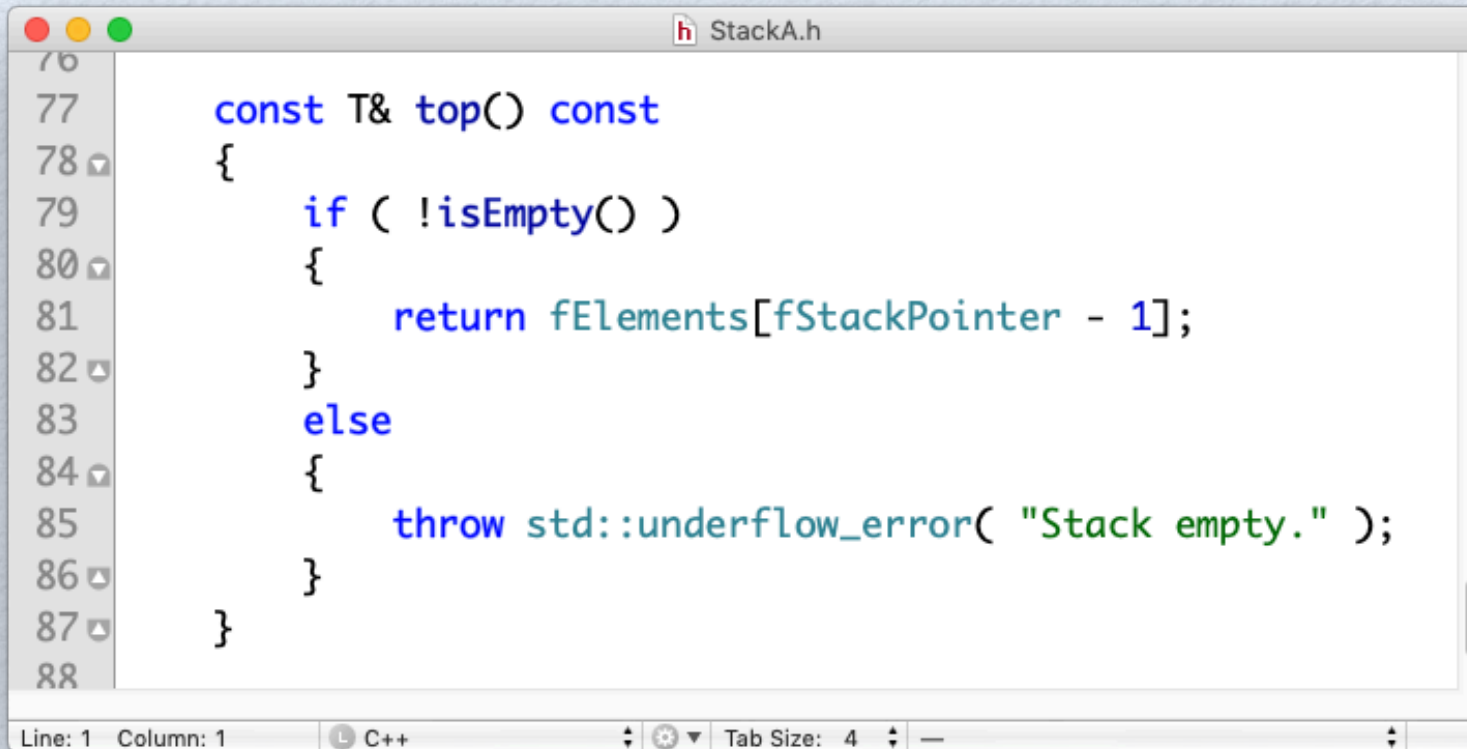


```
64
65 void pop()
66 {
67     if ( !isEmpty() )
68     {
69         fStackPointer--;
70     }
71     else
72     {
73         throw std::underflow_error( "Stack empty." );
74     }
75 }
76
```

The screenshot shows a code editor window titled "StackA.h". The code defines a `pop()` function. It checks if the stack is empty using `isEmpty()`. If not empty, it decrements `fStackPointer`. If the stack is empty, it throws a `std::underflow_error` with the message "Stack empty.". The editor interface includes a line number margin on the left (64-76), a status bar at the bottom showing "Line: 78 Column: 1", "C++", "Tab Size: 4", and "top".

- The pop method shifts the stack pointer to the previous slot in the stack, if there is such a slot. Note, the element in the current slot itself is not yet destroyed.

# Top



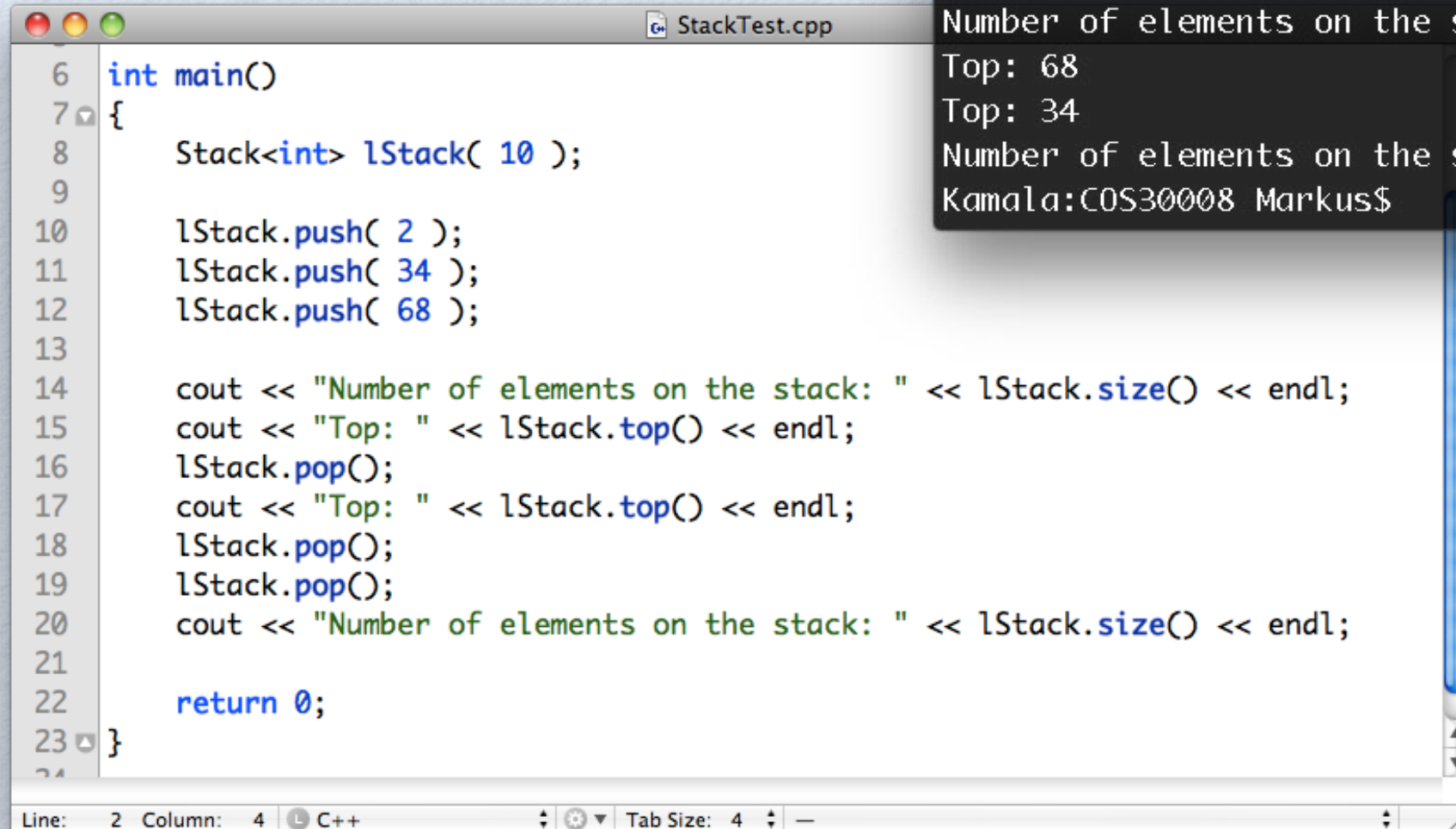
```
76
77     const T& top() const
78     {
79         if ( !isEmpty() )
80         {
81             return fElements[fStackPointer - 1];
82         }
83         else
84         {
85             throw std::underflow_error( "Stack empty." );
86         }
87     }
88
```

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

- The top method returns a const reference to the item in the current slot in the stack, if there is such a slot.



# Stack Sample



The image shows a C++ IDE window titled 'StackTest.cpp' with the following code:

```
6 int main()
7 {
8     Stack<int> lStack( 10 );
9
10    lStack.push( 2 );
11    lStack.push( 34 );
12    lStack.push( 68 );
13
14    cout << "Number of elements on the stack: " << lStack.size() << endl;
15    cout << "Top: " << lStack.top() << endl;
16    lStack.pop();
17    cout << "Top: " << lStack.top() << endl;
18    lStack.pop();
19    lStack.pop();
20    cout << "Number of elements on the stack: " << lStack.size() << endl;
21
22    return 0;
23 }
```

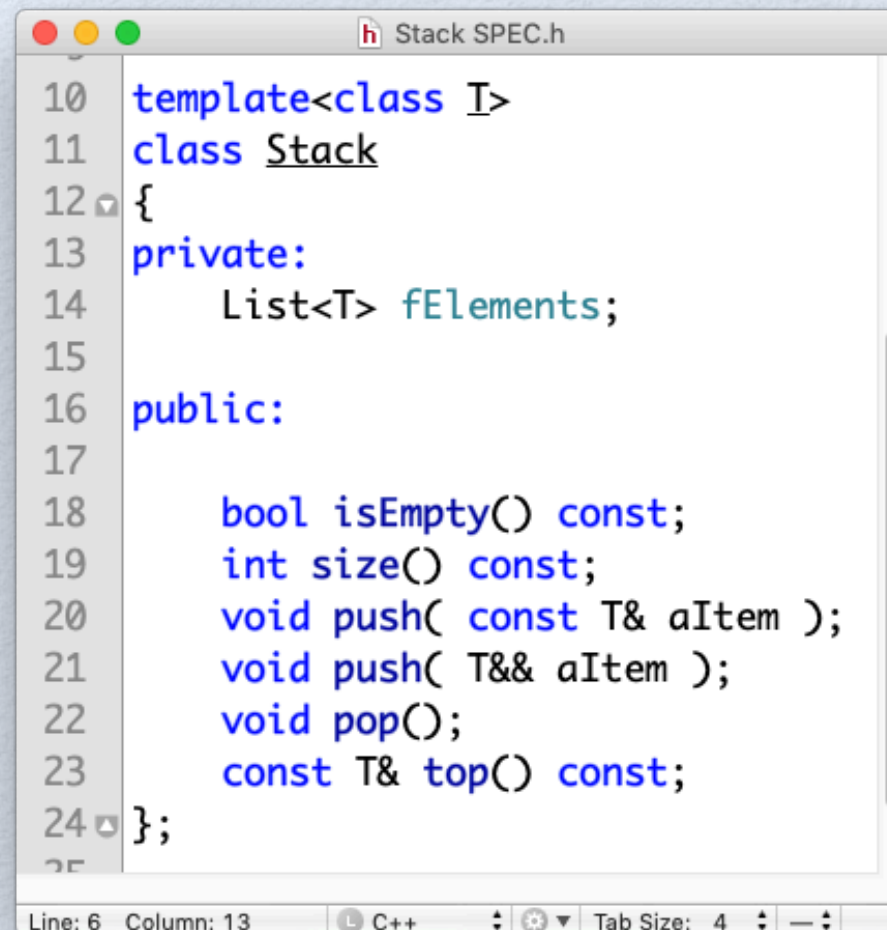
Below the code editor, the status bar shows 'Line: 2 Column: 4 C++' and 'Tab Size: 4'.

Overlaid on the right is a terminal window titled 'COS30008' showing the execution output:

```
Kamala: COS30008 Markus$ ./StackTest
Number of elements on the stack: 3
Top: 68
Top: 34
Number of elements on the stack: 0
Kamala: COS30008 Markus$
```

# Dynamic Stack

- We can define a dynamic stack that uses a list as underlying data type to host an arbitrary number of elements:



The screenshot shows a code editor window titled "Stack SPEC.h". The code defines a template class `Stack` for a generic type `T`. It uses a `List<T>` as its underlying data structure, stored as a private member `fElements`. The `Stack` class provides several public methods: `isEmpty()`, `size()`, `push()` (with two overloads for const and non-const references), `pop()`, and `top()`. The code is as follows:

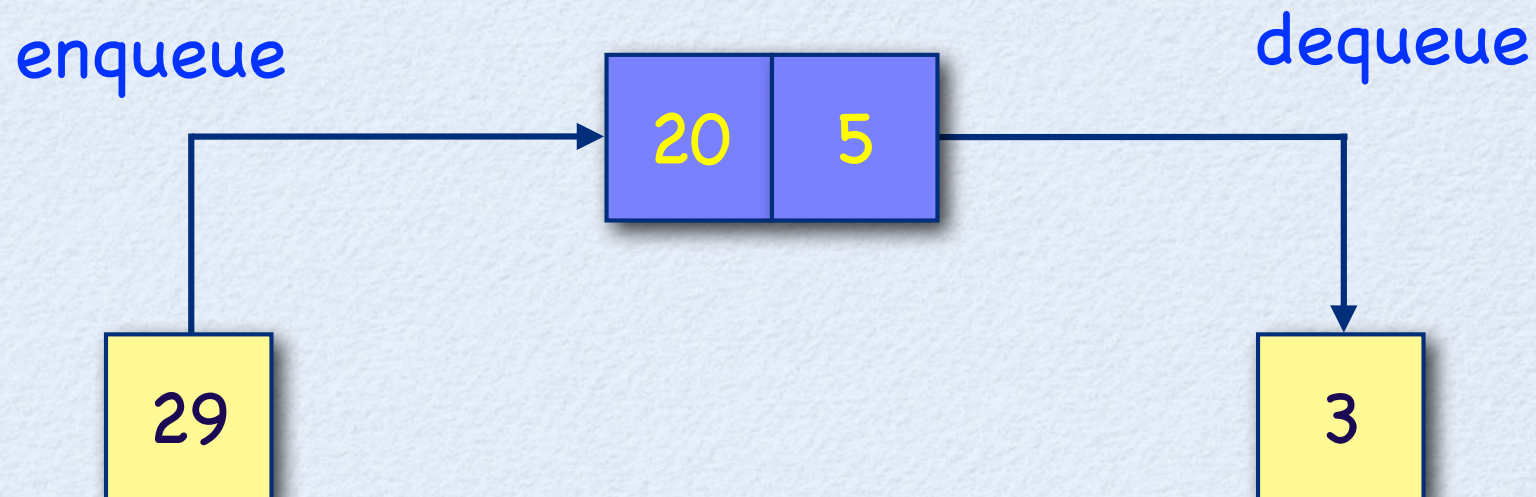
```
10 template<class T>
11 class Stack
12 {
13 private:
14     List<T> fElements;
15
16 public:
17
18     bool isEmpty() const;
19     int size() const;
20     void push( const T& aItem );
21     void push( T&& aItem );
22     void pop();
23     const T& top() const;
24 };
25
```

The editor's status bar at the bottom indicates "Line: 6 Column: 13", "C++", "Tab Size: 4", and other standard editor controls.



# Queues

- A **queue** is a special version of a linear list where access to items is only possible at its front and end.



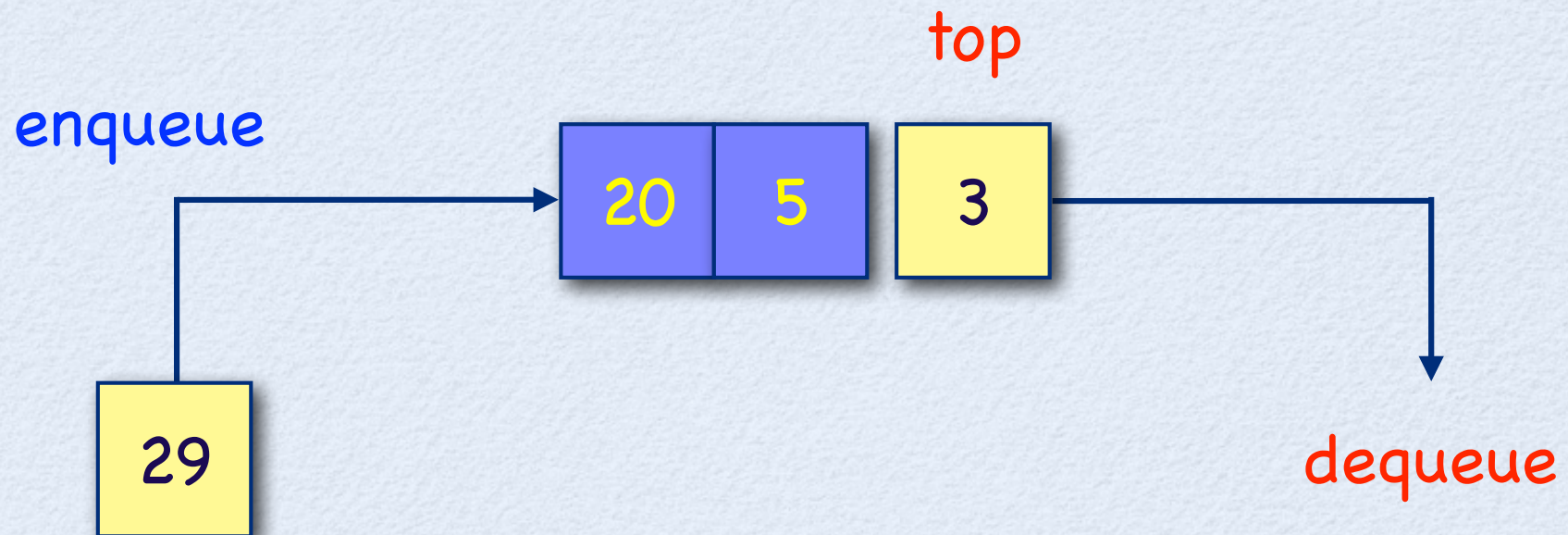
# Queue Behavior

- Queues manage elements in first-in, first-out (FIFO) manner.
- A queue underflow happens when one tries to dequeue on an empty queue.
- A queue overflow happens when one tries to enqueue on a full queue.

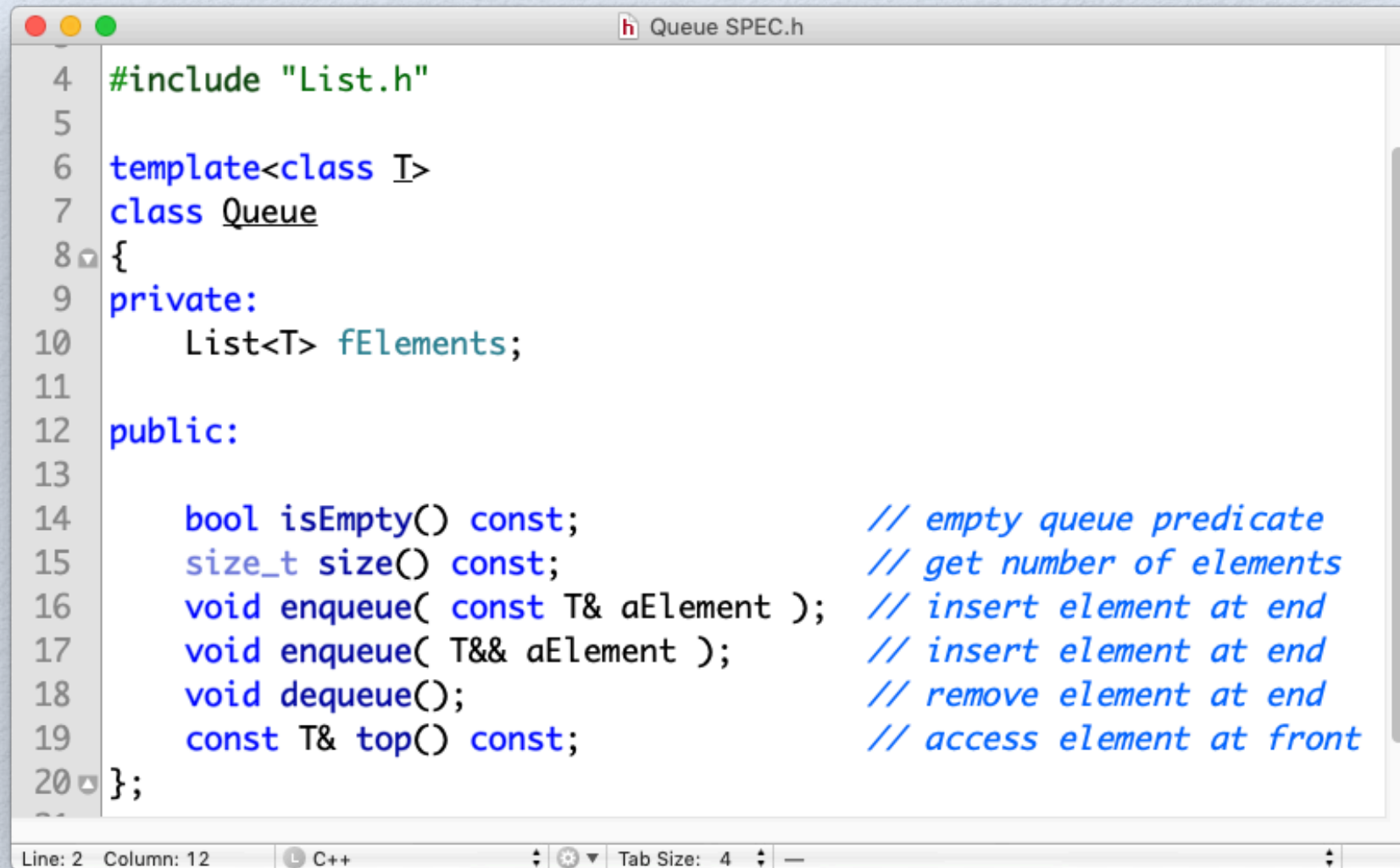


# Implementation of Queues

- A concrete **queue** implementation requires us to split the dequeue operation into two steps: access to the first element (via **top**) and removal of first element (the actual **dequeue**).
- If we were to perform both steps as one (just dequeue), we would create a memory leak in C++ (i.e., we would create a reference to released memory). Hence, we need:



# A Queue Interface

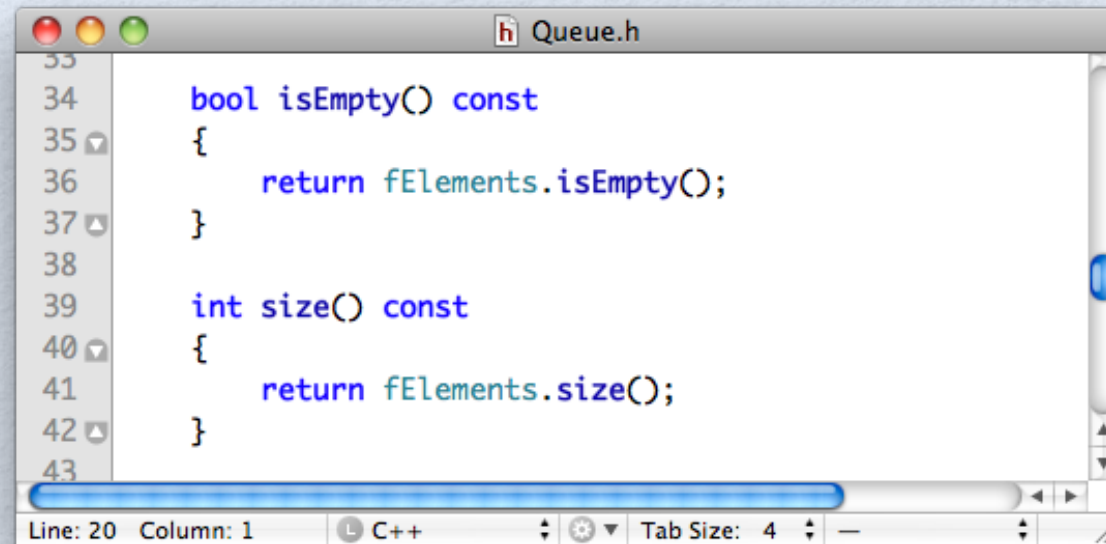


```
1
2
3
4 #include "List.h"
5
6 template<class T>
7 class Queue
8 {
9     private:
10         List<T> fElements;
11
12     public:
13
14         bool isEmpty() const;           // empty queue predicate
15         size_t size() const;           // get number of elements
16         void enqueue( const T& aElement ); // insert element at end
17         void enqueue( T&& aElement );     // insert element at end
18         void dequeue();                  // remove element at end
19         const T& top() const;           // access element at front
20 };
21
```

Line: 2 Column: 12 C++ Tab Size: 4



# Queue Service Members

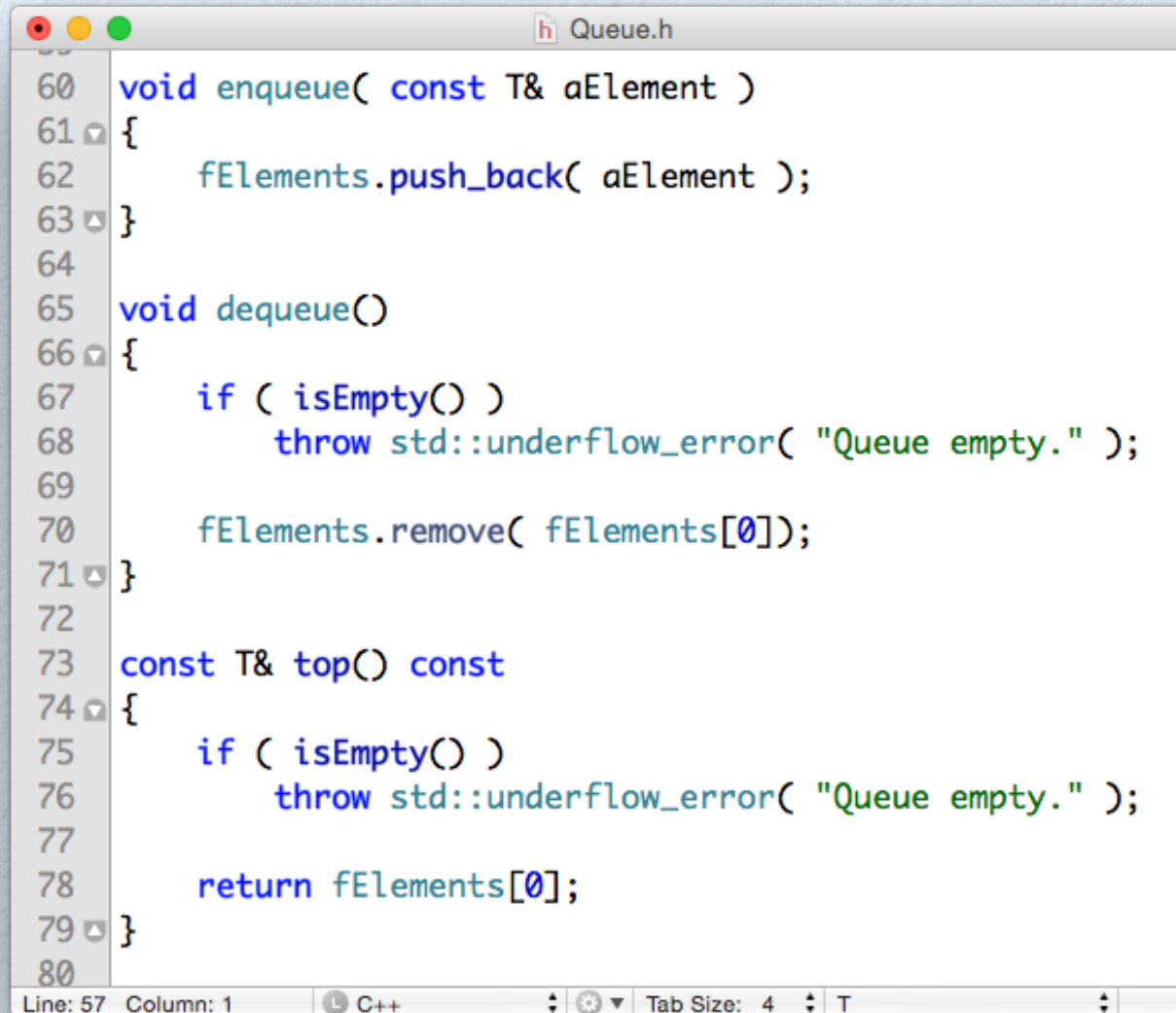


The screenshot shows a code editor window titled "Queue.h". The code defines two constant member functions for a Queue class. The first function, `isEmpty()`, returns `fElements.isEmpty()`. The second function, `size()`, returns `fElements.size()`. The code is as follows:

```
33  
34     bool isEmpty() const  
35     {  
36         return fElements.isEmpty();  
37     }  
38  
39     int size() const  
40     {  
41         return fElements.size();  
42     }  
43
```

The editor's status bar at the bottom indicates "Line: 20 Column: 1", "C++", and "Tab Size: 4".

# Queue Semantics

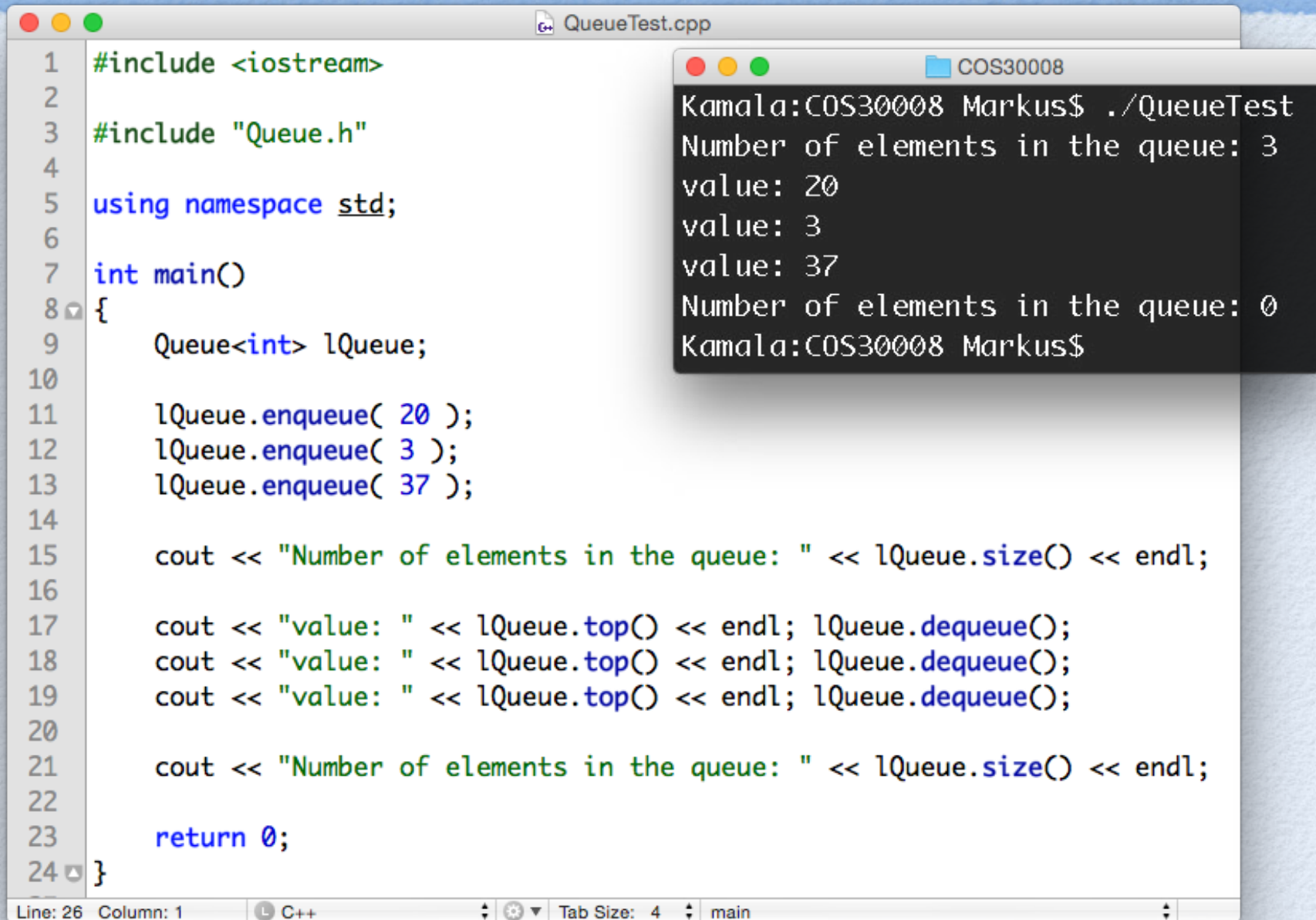


```
60 void enqueue( const T& aElement )
61 {
62     fElements.push_back( aElement );
63 }
64
65 void dequeue()
66 {
67     if ( isEmpty() )
68         throw std::underflow_error( "Queue empty." );
69
70     fElements.remove( fElements[0] );
71 }
72
73 const T& top() const
74 {
75     if ( isEmpty() )
76         throw std::underflow_error( "Queue empty." );
77
78     return fElements[0];
79 }
80
```

Line: 57 Column: 1 C++ Tab Size: 4 T



# Queue Test



The image shows a C++ IDE window titled "QueueTest.cpp" with the following code:

```
1 #include <iostream>
2
3 #include "Queue.h"
4
5 using namespace std;
6
7 int main()
8 {
9     Queue<int> lQueue;
10
11     lQueue.enqueue( 20 );
12     lQueue.enqueue( 3 );
13     lQueue.enqueue( 37 );
14
15     cout << "Number of elements in the queue: " << lQueue.size() << endl;
16
17     cout << "value: " << lQueue.top() << endl; lQueue.dequeue();
18     cout << "value: " << lQueue.top() << endl; lQueue.dequeue();
19     cout << "value: " << lQueue.top() << endl; lQueue.dequeue();
20
21     cout << "Number of elements in the queue: " << lQueue.size() << endl;
22
23     return 0;
24 }
```

Overlaid on the bottom right is a terminal window titled "COS30008" showing the execution output:

```
Kamala: COS30008 Markus$ ./QueueTest
Number of elements in the queue: 3
value: 20
value: 3
value: 37
Number of elements in the queue: 0
Kamala: COS30008 Markus$
```

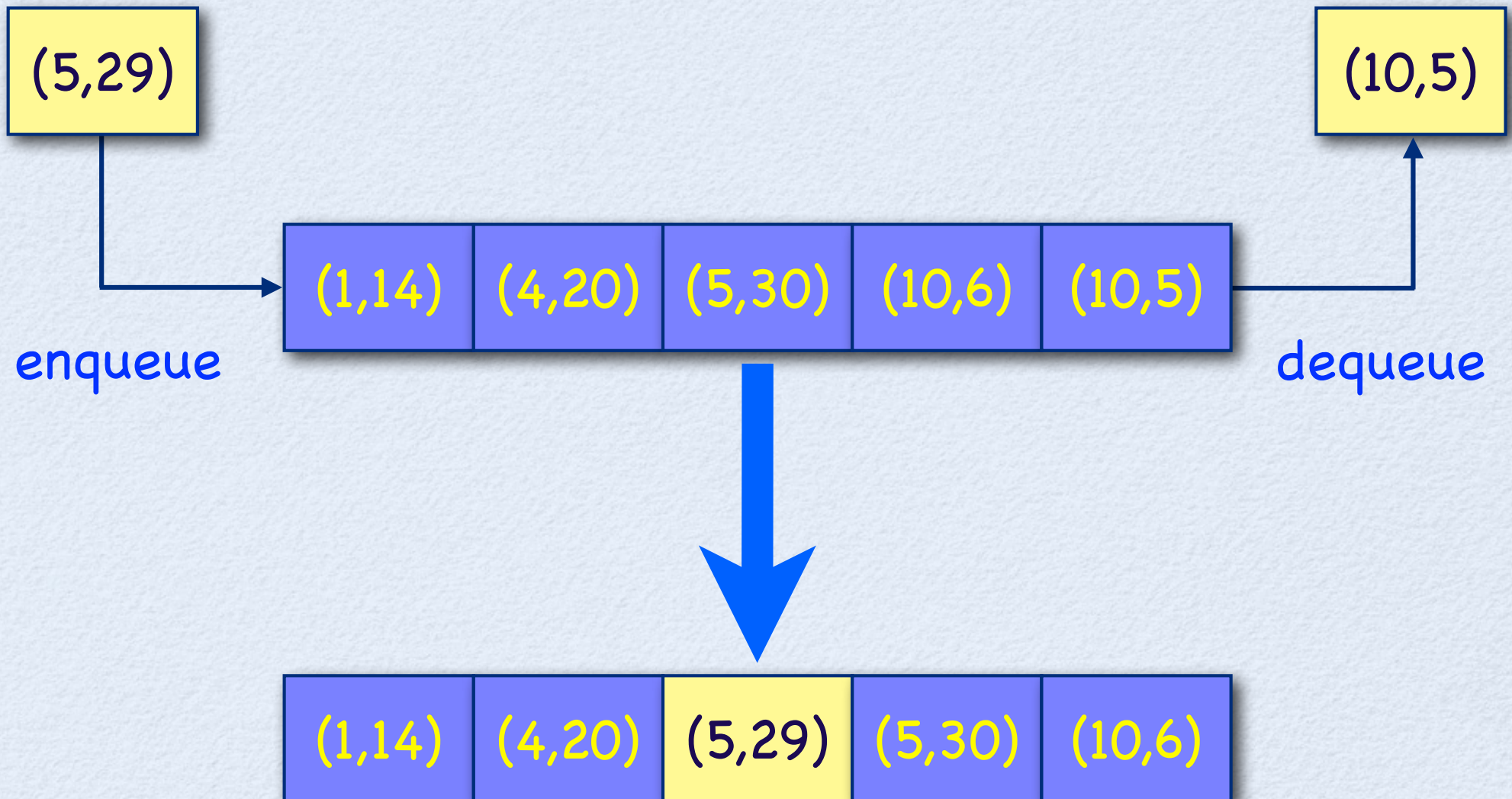
The IDE status bar at the bottom indicates "Line: 26 Column: 1", "C++", "Tab Size: 4", and "main".

# Requirements for a Priority Queue

- The underlying data structure for a priority queue must be sorted (e.g., `SortedList<T>`).
- Elements are queued using an integer to specify priority. We use a `Pair<Key, T>` to store elements with their associated priority.
- We need to provide a matching `operator<` on key values to sort elements in the priority queue.



# Priority Queue



```
SortedList SPEC.h
6 #include "DoublyLinkedList.h"
7 #include "DoublyLinkedListIterator.h"
8
9 #include <stdexcept>
10
11 template<class T>
12 class SortedList
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     SortedList(); // default constructor - creates empty list
26     SortedList( const SortedList& aOtherList ); // copy constructor
27     SortedList( SortedList&& aOtherList ); // move constructor
28     SortedList& operator=( const SortedList& aOtherList ); // assignment operator
29     SortedList& operator=( SortedList&& aOtherList ); // move assignment operator
30     ~SortedList(); // destructor - frees all nodes
31
32     bool isEmpty() const; // Is list empty?
33     int size() const; // list size
34
35     void insert( const T& aElement ); // adds aElement at proper position
36     void insert( T&& aElement ); // adds aElement at proper position
37     void remove( const T& aElement ); // remove first match from list
38
39     const T& operator[]( size_t aIndex ) const; // list indexer
40
41     Iterator begin() const; // return a forward iterator
42     Iterator end() const; // return a forward end iterator
43     Iterator rbegin() const; // return a backwards iterator
44     Iterator rend() const; // return a backwards end iterator
45 };
46
```



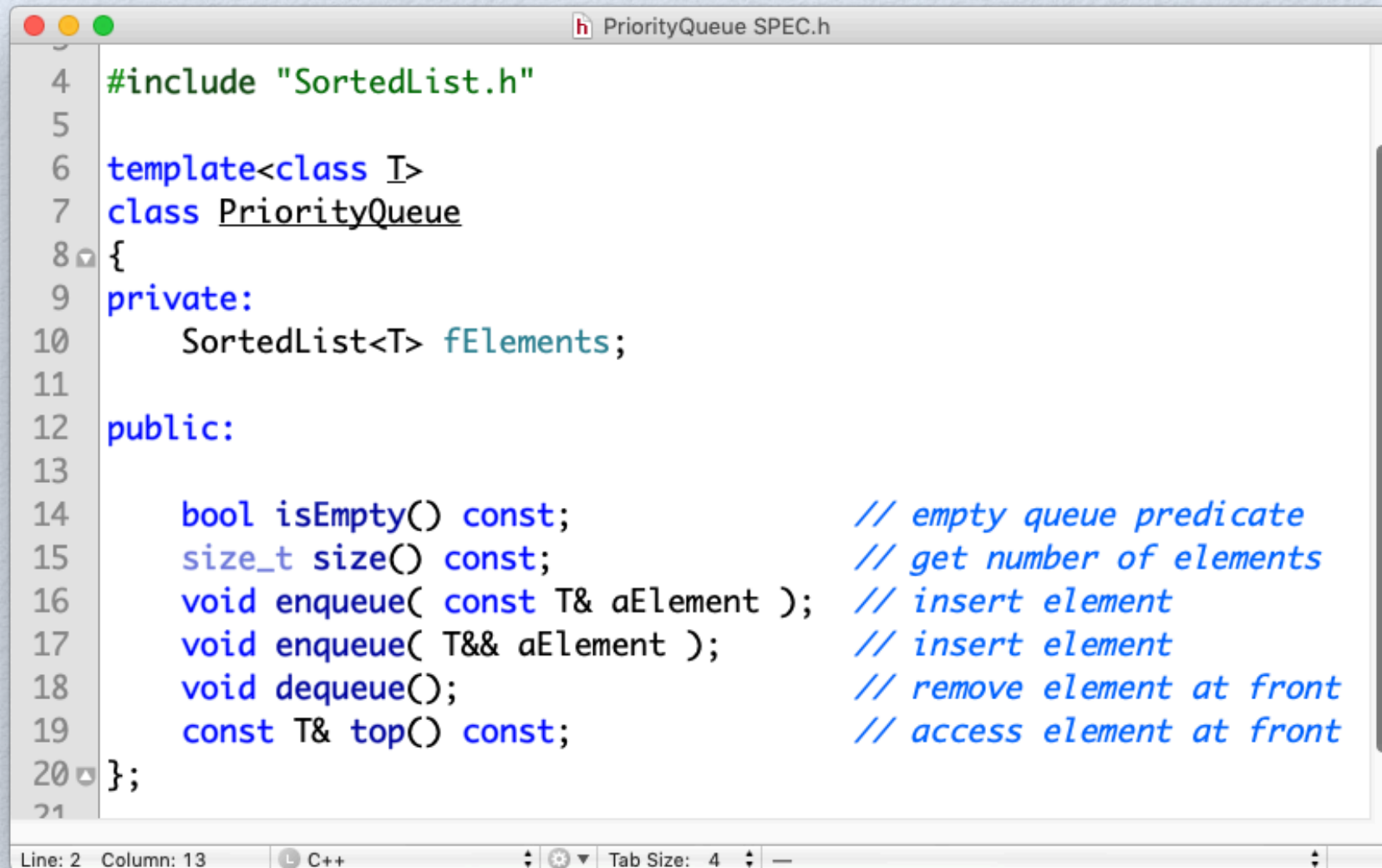
# The Pair Class

```
Pair.h
4  template<class K, class V>
5  struct Pair
6  {
7      K key;
8      V value;
9
10     Pair( const K& aKey, const V& aValue ) : key(aKey), value(aValue)
11     {}
12
13     bool operator<( const Pair<K,V>& aOther ) const
14     {
15         return key < aOther.key;
16     }
17
18     bool operator==( const Pair<K,V>& aOther ) const
19     {
20         return key == aOther.key && value == aOther.value;
21     }
22 };
```

SortedList uses an increasing order.

Line: 2 Column: 13 C++ Tab Size: 4

# A Priority Queue

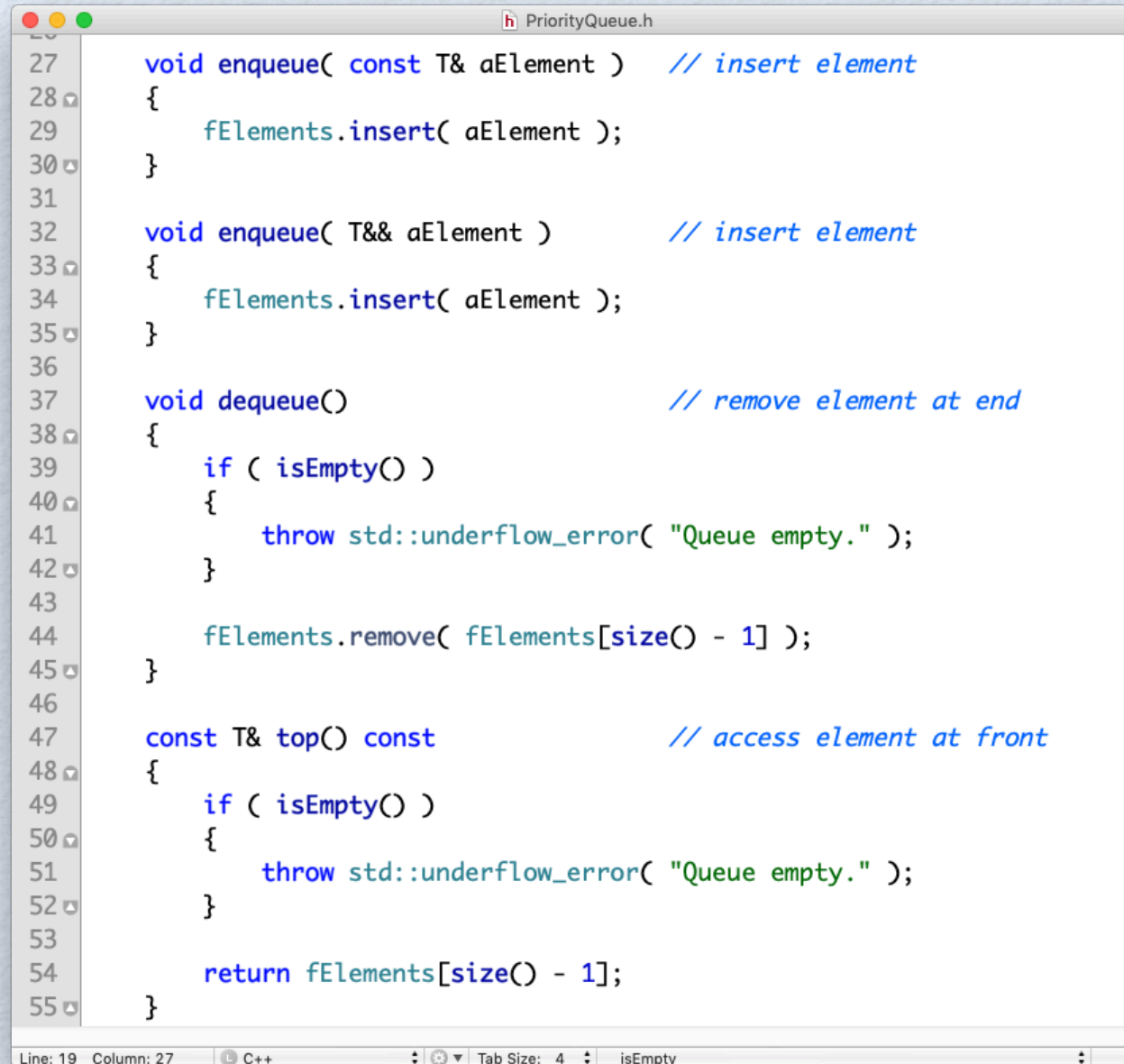


```
1
2
3
4 #include "SortedList.h"
5
6 template<class T>
7 class PriorityQueue
8 {
9     private:
10         SortedList<T> fElements;
11
12     public:
13
14         bool isEmpty() const;           // empty queue predicate
15         size_t size() const;           // get number of elements
16         void enqueue( const T& aElement ); // insert element
17         void enqueue( T&& aElement );    // insert element
18         void dequeue();                 // remove element at front
19         const T& top() const;           // access element at front
20 };
21
```

Line: 2 Column: 13 C++ Tab Size: 4



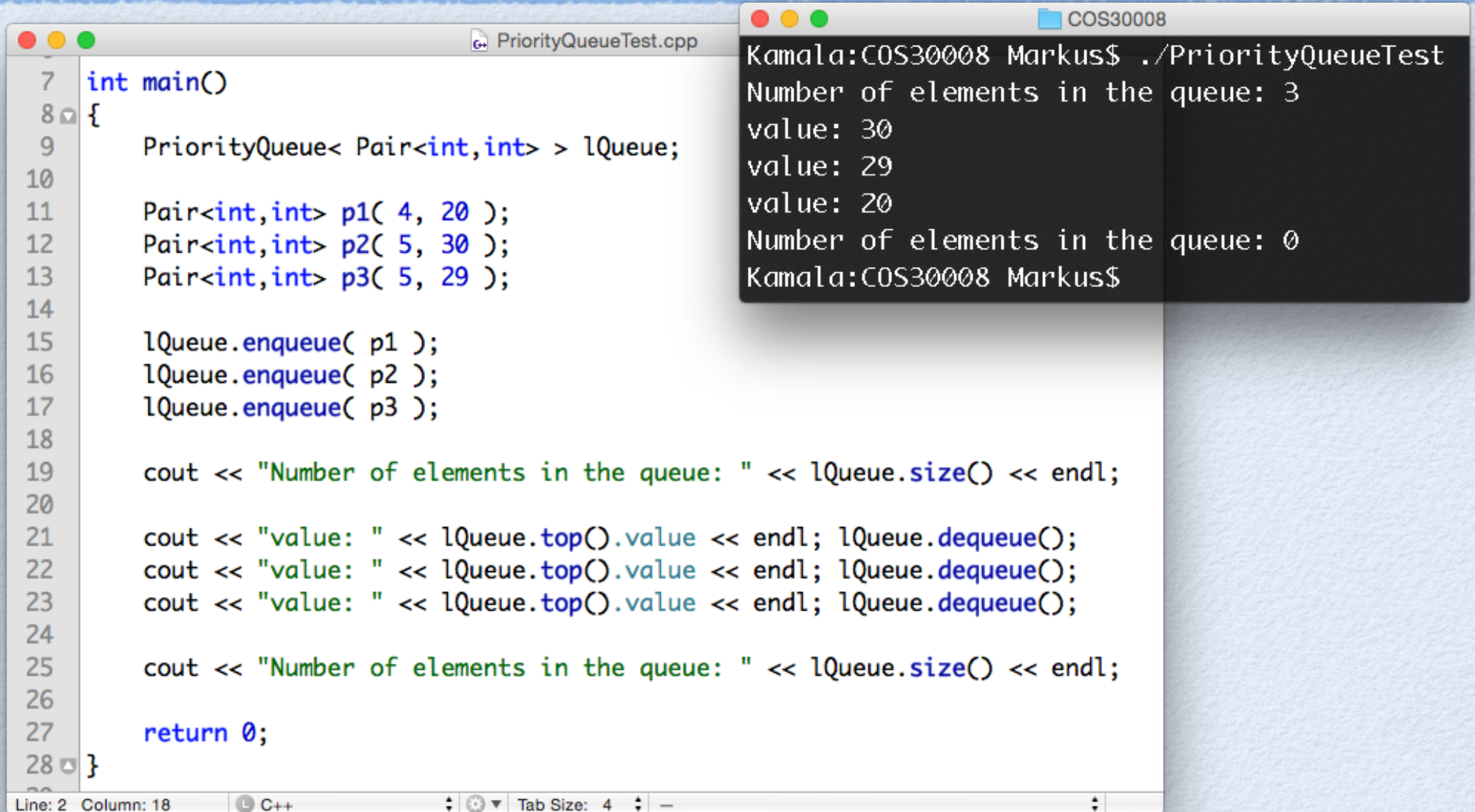
# Priority Queue Semantics



```
27 void enqueue( const T& aElement )    // insert element
28 {
29     fElements.insert( aElement );
30 }
31
32 void enqueue( T&& aElement )          // insert element
33 {
34     fElements.insert( aElement );
35 }
36
37 void dequeue()                       // remove element at end
38 {
39     if ( isEmpty() )
40     {
41         throw std::underflow_error( "Queue empty." );
42     }
43
44     fElements.remove( fElements[size() - 1] );
45 }
46
47 const T& top() const                 // access element at front
48 {
49     if ( isEmpty() )
50     {
51         throw std::underflow_error( "Queue empty." );
52     }
53
54     return fElements[size() - 1];
55 }
```

Line: 19 Column: 27 C++ Tab Size: 4 isEmpty

# A PriorityQueue Test



The image shows a C++ code editor window titled "PriorityQueueTest.cpp" and a terminal window titled "COS30008".

**Code Editor (PriorityQueueTest.cpp):**

```
7 int main()
8 {
9     PriorityQueue< Pair<int,int> > lQueue;
10
11     Pair<int,int> p1( 4, 20 );
12     Pair<int,int> p2( 5, 30 );
13     Pair<int,int> p3( 5, 29 );
14
15     lQueue.enqueue( p1 );
16     lQueue.enqueue( p2 );
17     lQueue.enqueue( p3 );
18
19     cout << "Number of elements in the queue: " << lQueue.size() << endl;
20
21     cout << "value: " << lQueue.top().value << endl; lQueue.dequeue();
22     cout << "value: " << lQueue.top().value << endl; lQueue.dequeue();
23     cout << "value: " << lQueue.top().value << endl; lQueue.dequeue();
24
25     cout << "Number of elements in the queue: " << lQueue.size() << endl;
26
27     return 0;
28 }
```

**Terminal (COS30008):**

```
Kamala: COS30008 Markus$ ./PriorityQueueTest
Number of elements in the queue: 3
value: 30
value: 29
value: 20
Number of elements in the queue: 0
Kamala: COS30008 Markus$
```

The code editor shows line numbers 7 to 28. The terminal shows the output of the program, which is a test of a PriorityQueue. The program enqueues three elements (p1, p2, p3) and then dequeues them in order of their priority (value). The output shows the number of elements in the queue and the value of the top element at each step.