Software Engineering 2 (C++)

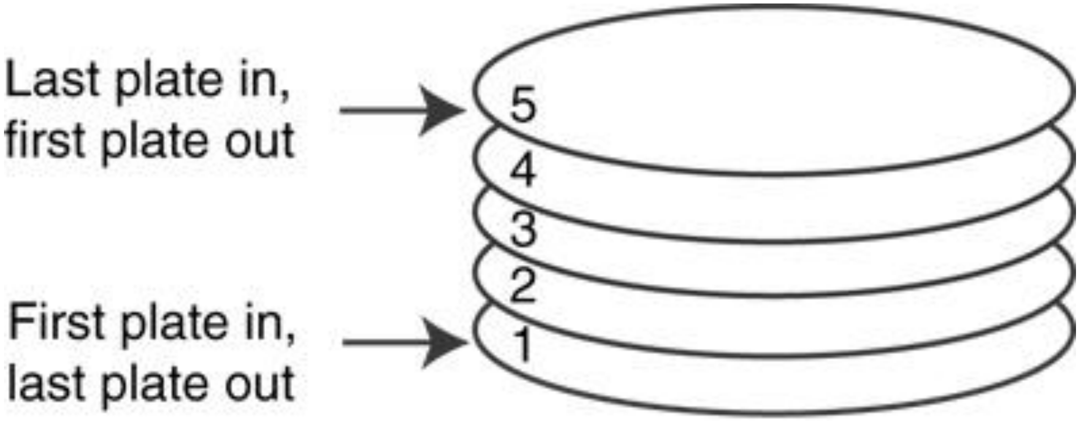
CSY2006 (Week 20)

Dr. Suraj Ajit

Introduction to the Stack ADT

- Stack: a LIFO (last in, first out) data structure
- Examples:
  - plates in a cafeteria
  - return addresses for function calls
- Implementation:
  - static: fixed size, implemented as array
  - dynamic: variable size, implemented as linked list

A LIFO Structure

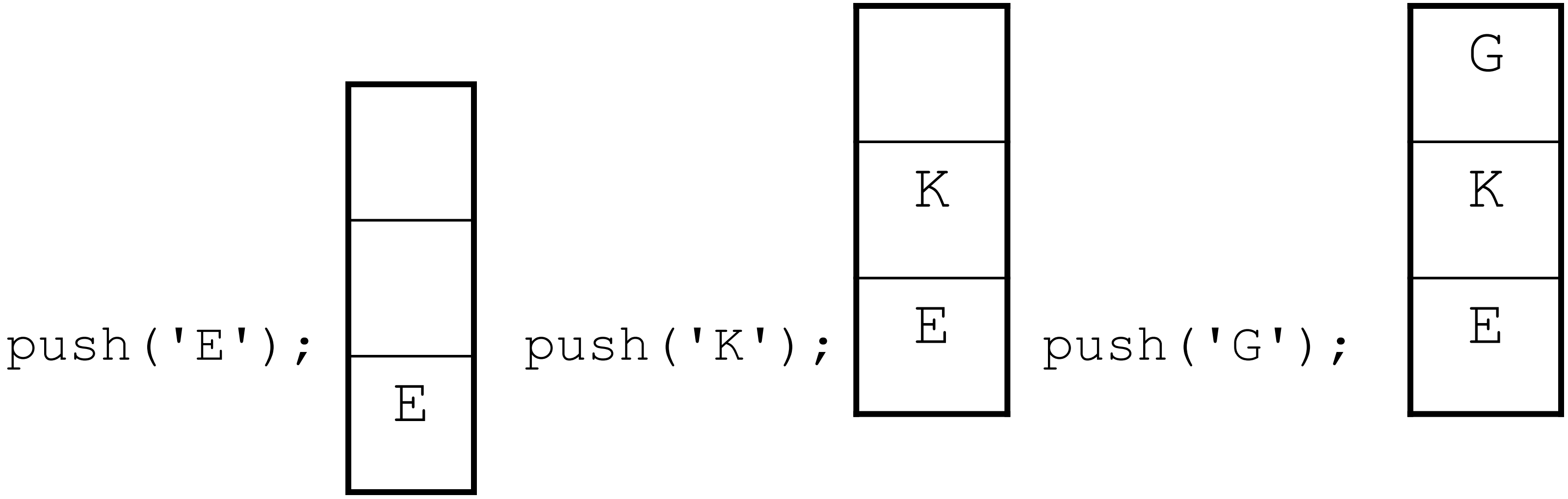


Stack Operations and Functions

- Operations:
  - push: add a value onto the top of the stack
  - pop: remove a value from the top of the stack
- Functions:
  - `isFull`: `true` if the stack is currently full, *i.e.*, has no more space to hold additional elements
  - isEmpty: `true` if the stack currently contains no elements

Stack Operations - Example

- A stack that can hold `char` values:



`push('E');`

E

`push('K');`

K

E

`push('G');`

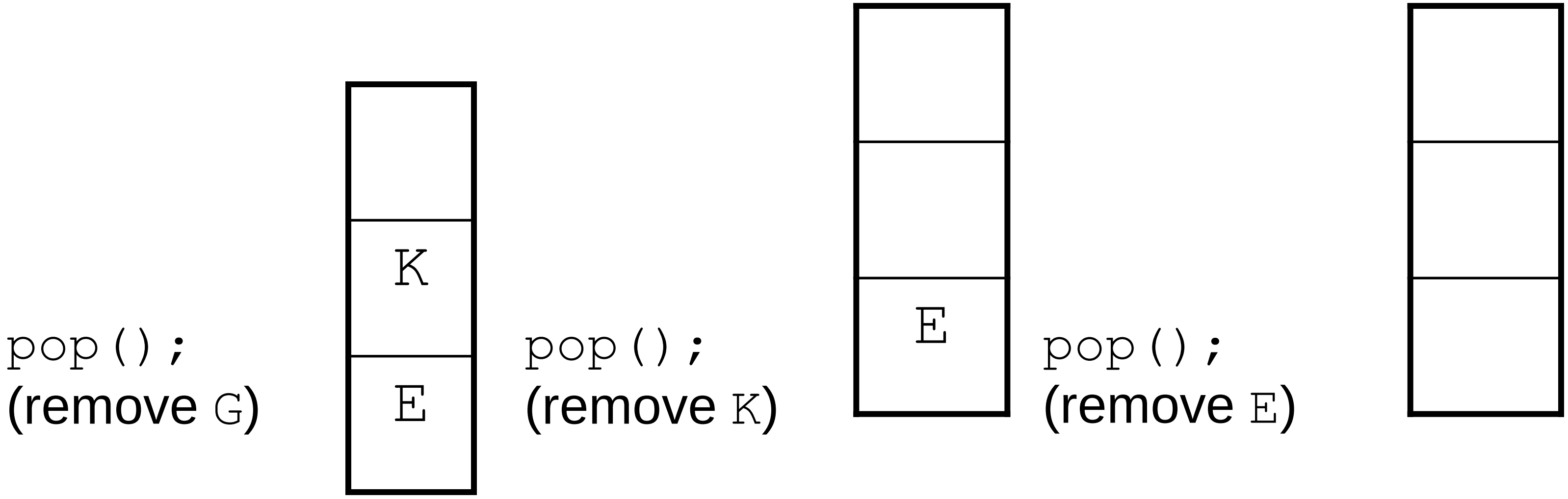
G

K

E

Stack Operations - Example

- A stack that can hold `char` values:



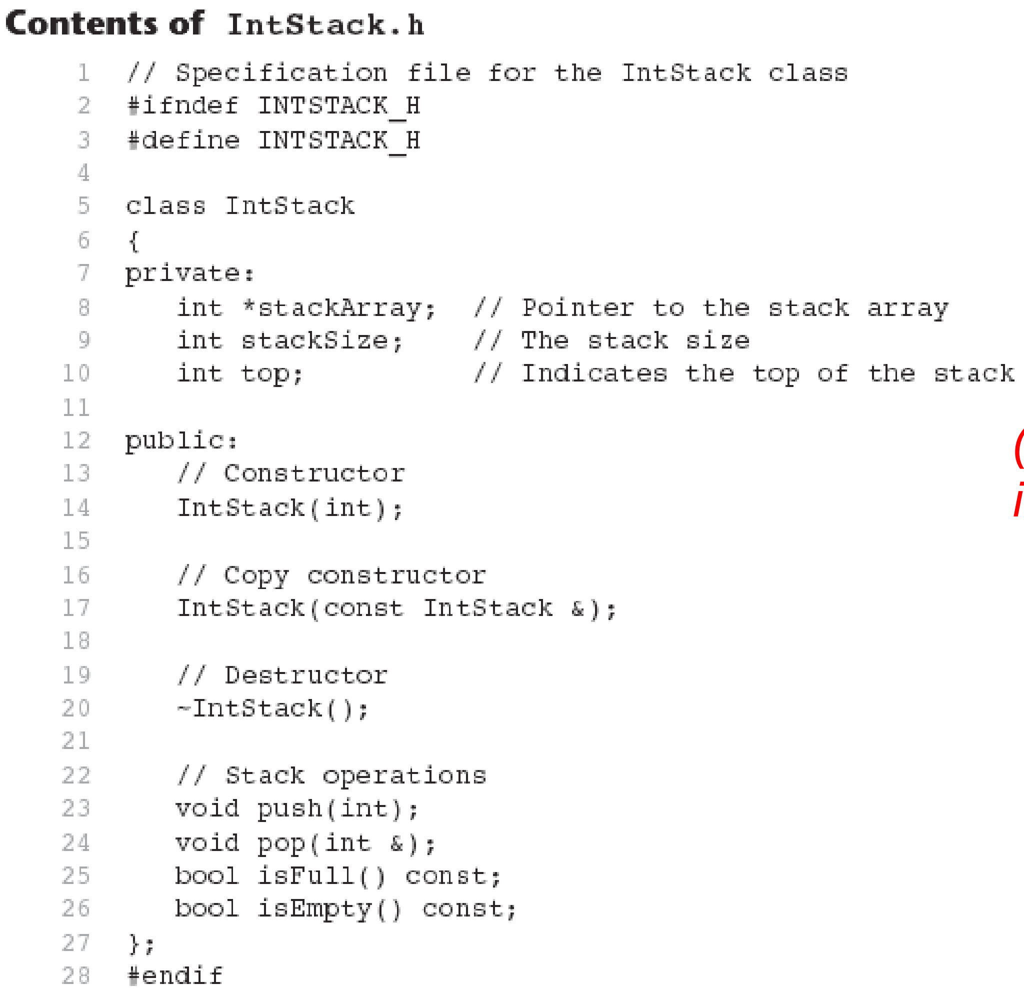
`pop();`
(remove `G`)

`K`
`E`

`pop();`
(remove `K`)

`E`

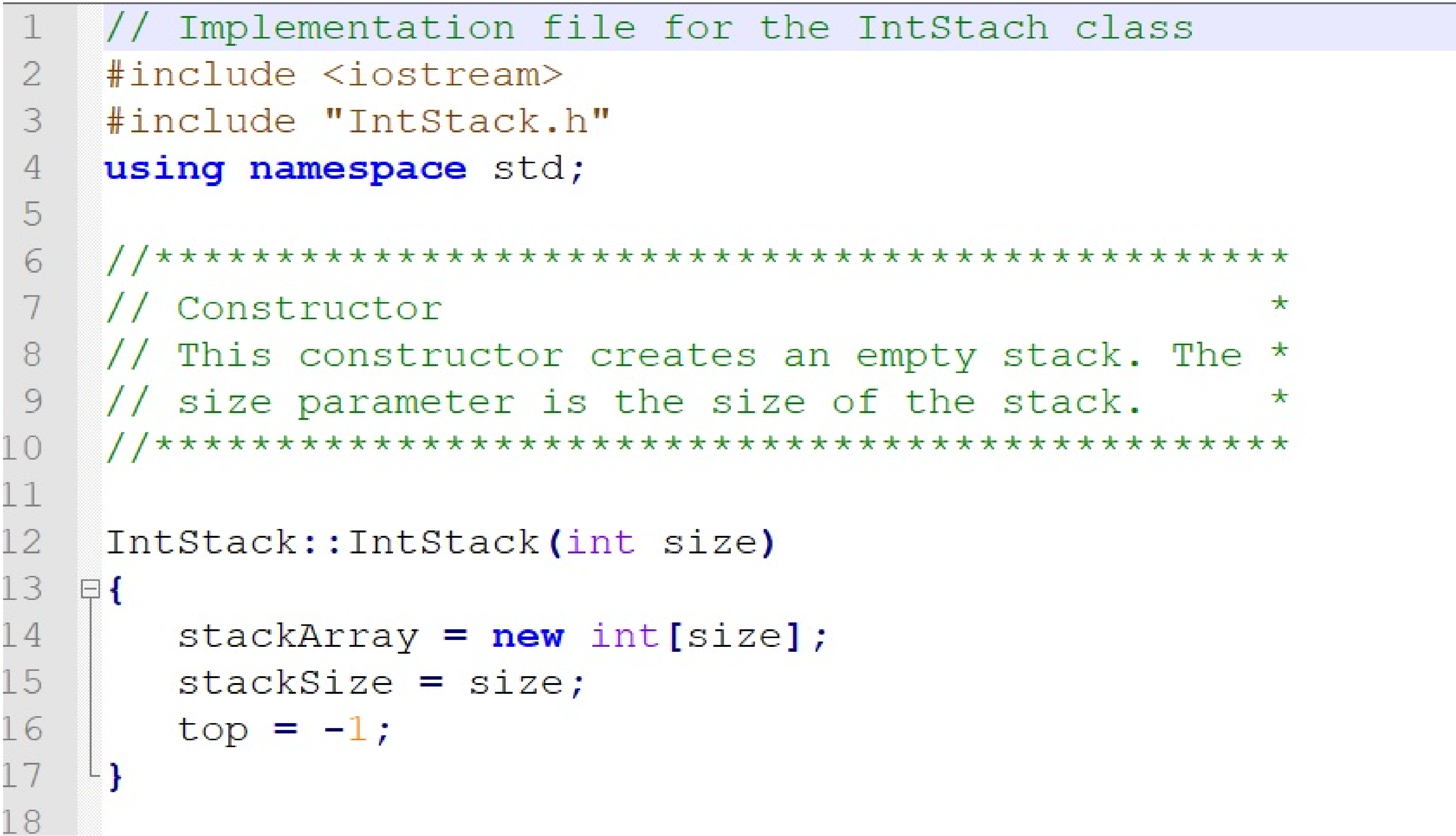
`pop();`
(remove `E`)



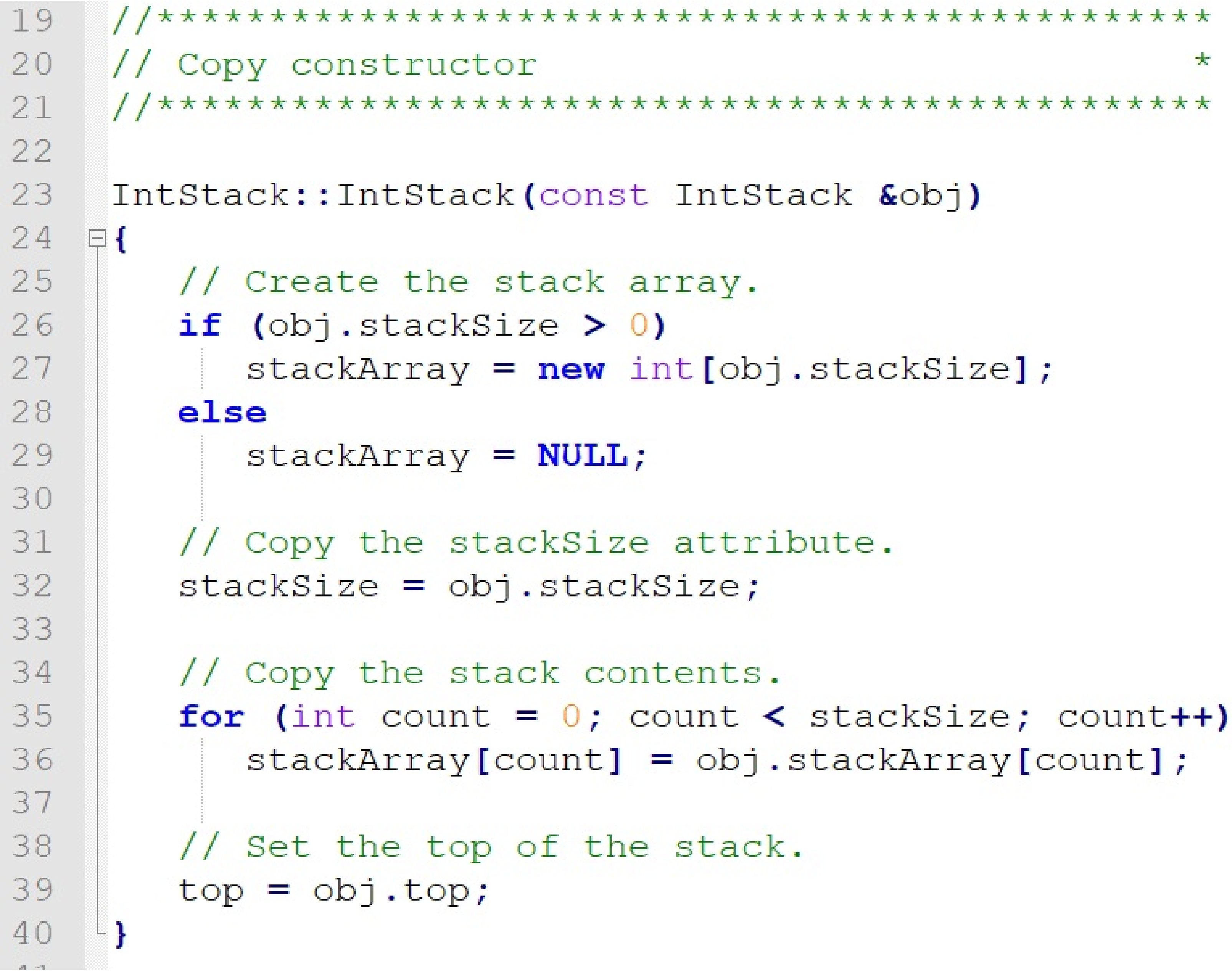
Contents of `IntStack.h`

```
// Specification file for the IntStack class
#ifndef INTSTACK_H
#define INTSTACK_H

class IntStack
{
private:
   int *stackArray;   // Pointer to the stack array
   int stackSize;     // The stack size
   int top;           // Indicates the top of the stack

public:
   // Constructor
   IntStack(int);

   // Copy constructor
   IntStack(const IntStack &);

   // Destructor
   ~IntStack();

   // Stack operations
   void push(int);
   void pop(int &);
   bool isFull() const;
   bool isEmpty() const;
};
#endif
```

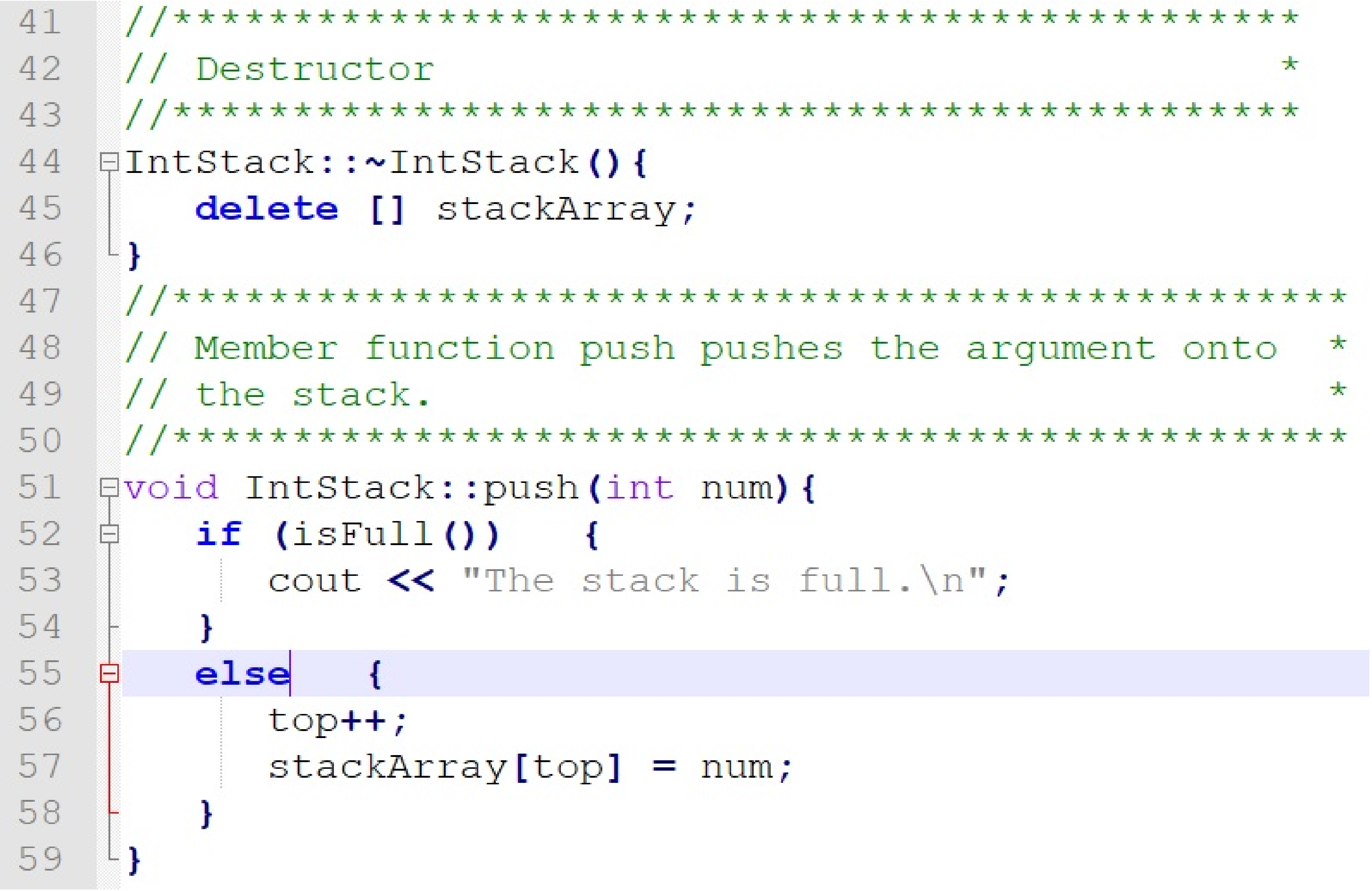
(See IntStack.cpp for the implementation.)



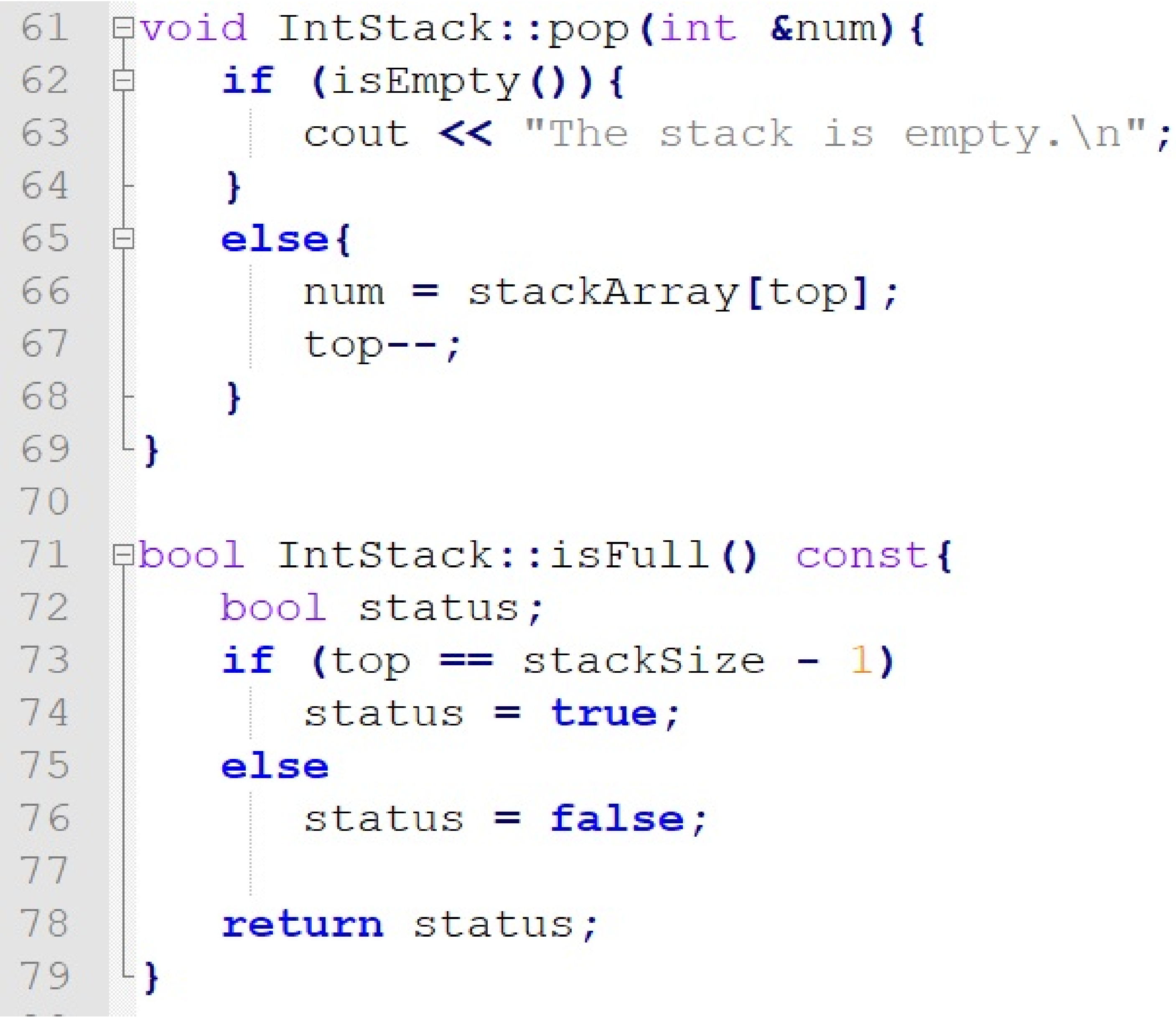
```cpp
// Implementation file for the IntStach class
#include <iostream>
#include "IntStack.h"
using namespace std;

//***********************************************
// Constructor                                  *
// This constructor creates an empty stack. The *
// size parameter is the size of the stack.     *
//***********************************************

IntStack::IntStack(int size)
{
   stackArray = new int[size];
   stackSize = size;
   top = -1;
}

```



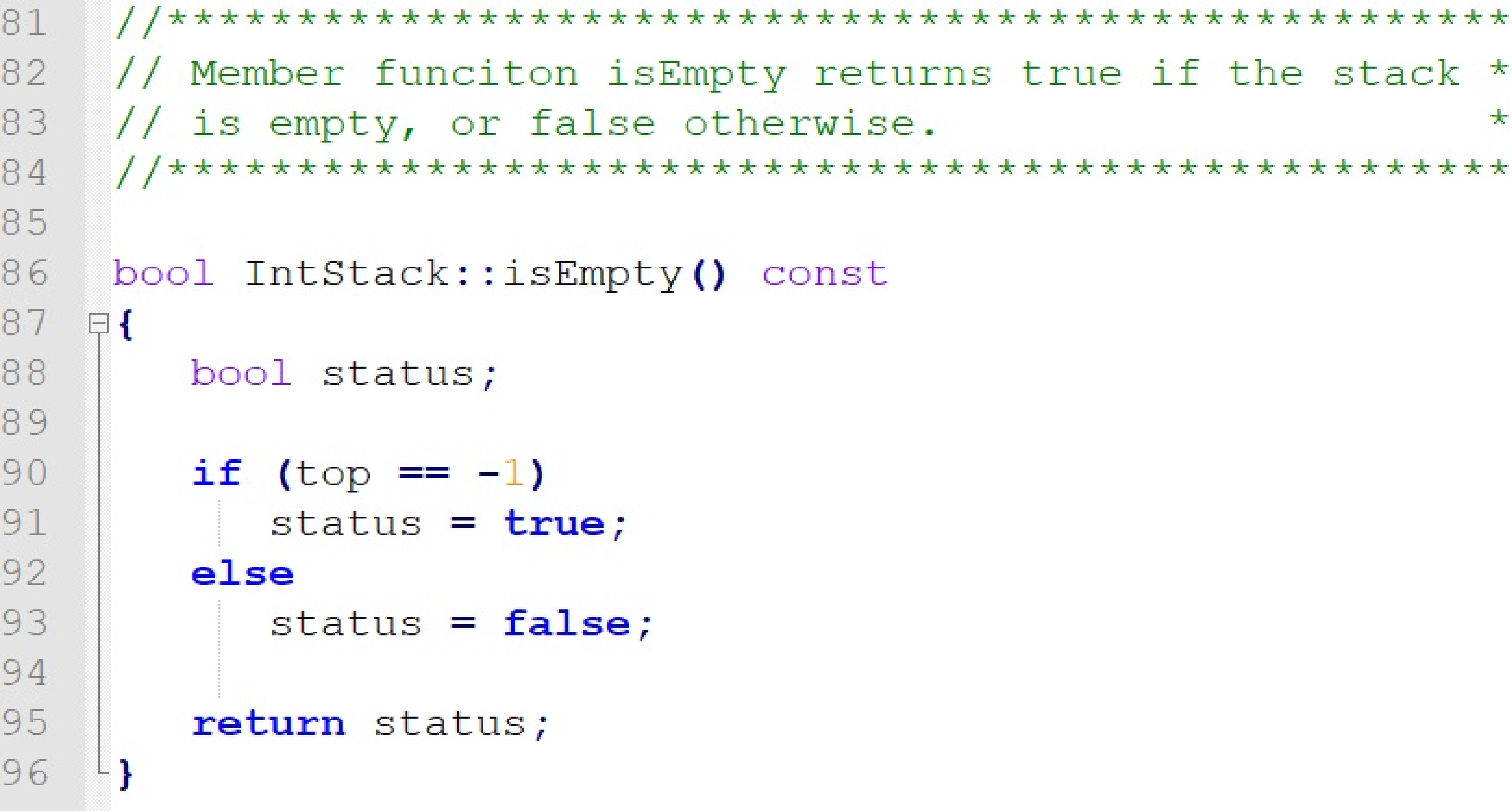
```
//************************************************
// Copy constructor                              *
//************************************************

IntStack::IntStack(const IntStack &obj)
{
   // Create the stack array.
   if (obj.stackSize > 0)
      stackArray = new int[obj.stackSize];
   else
      stackArray = NULL;

   // Copy the stackSize attribute.
   stackSize = obj.stackSize;

   // Copy the stack contents.
   for (int count = 0; count < stackSize; count++)
      stackArray[count] = obj.stackArray[count];

   // Set the top of the stack.
   top = obj.top;
}
```



```cpp
//***********************************************
// Destructor                                   *
//***********************************************
IntStack::~IntStack(){
   delete [] stackArray;
}
//*************************************************
// Member function push pushes the argument onto  *
// the stack.                                     *
//*************************************************
void IntStack::push(int num){
   if (isFull())   {
      cout << "The stack is full.\n";
   }
   else   {
      top++;
      stackArray[top] = num;
   }
}
```



```cpp
void IntStack::pop(int &num){
    if (isEmpty()){
        cout << "The stack is empty.\n";
    }
    else{
        num = stackArray[top];
        top--;
    }
}

bool IntStack::isFull() const{
    bool status;
    if (top == stackSize - 1)
        status = true;
    else
        status = false;

    return status;
}
```



```
//****************************************************
// Member funciton isEmpty returns true if the stack *
// is empty, or false otherwise.                     *
//****************************************************

bool IntStack::isEmpty() const
{
   bool status;

   if (top == -1)
      status = true;
   else
      status = false;

   return status;
}
```

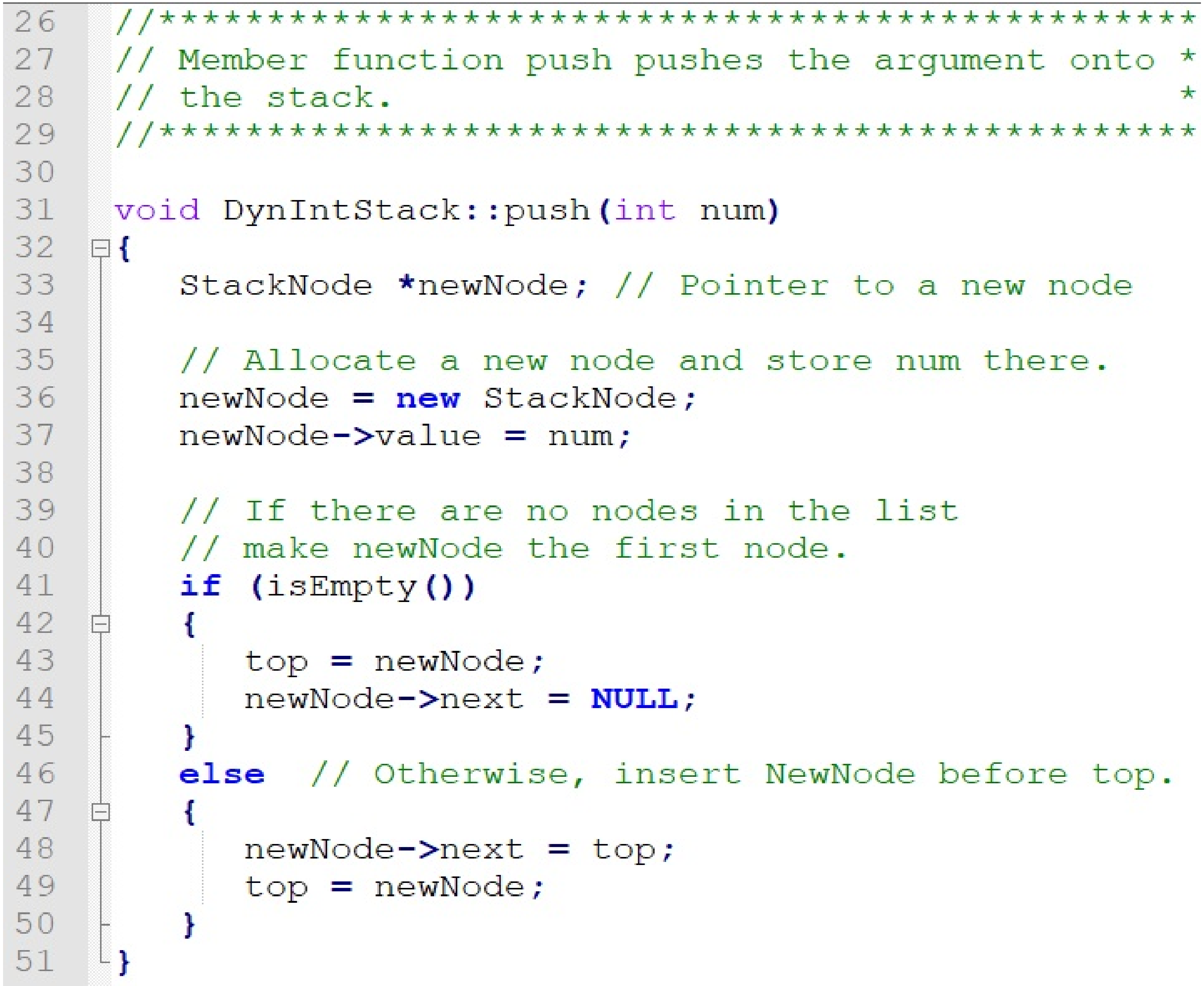
Dynamic Stacks

- Grow and shrink as necessary
- Can't ever be full as long as memory is available
- Implemented as a linked list

Implementing a Stack

- Programmers can program their own routines to implement stack functions
- See `DynIntStack` class in the book for an example.
- Can also use the implementation of stack available in the STL
- Other implementations: See Malik folder (Sample Programs)
- Application: See Applications folder (Sample Programs)

Implementing a Stack



```cpp
//************************************************
// Member function push pushes the argument onto *
// the stack.                                    *
//************************************************

void DynIntStack::push(int num)
{
   StackNode *newNode; // Pointer to a new node

   // Allocate a new node and store num there.
   newNode = new StackNode;
   newNode->value = num;

   // If there are no nodes in the list
   // make newNode the first node.
   if (isEmpty())
   {
      top = newNode;
      newNode->next = NULL;
   }
   else  // Otherwise, insert NewNode before top.
   {
      newNode->next = top;
      top = newNode;
   }
}
```

The STL `stack` container

- Stack template can be implemented as a `vector`, a linked list, or a `deque`
- Implements `push`, `pop`, and `empty` member functions
- Implements other member functions:
  - `size`: number of elements on the stack
  - `top`: reference to element on top of the stack

Defining a `stack`

- Defining a stack of `char`s, named `cstack`, implemented using a `vector`:

  ```
  stack< char, vector<char> > cstack;
  ```

- implemented using a list:

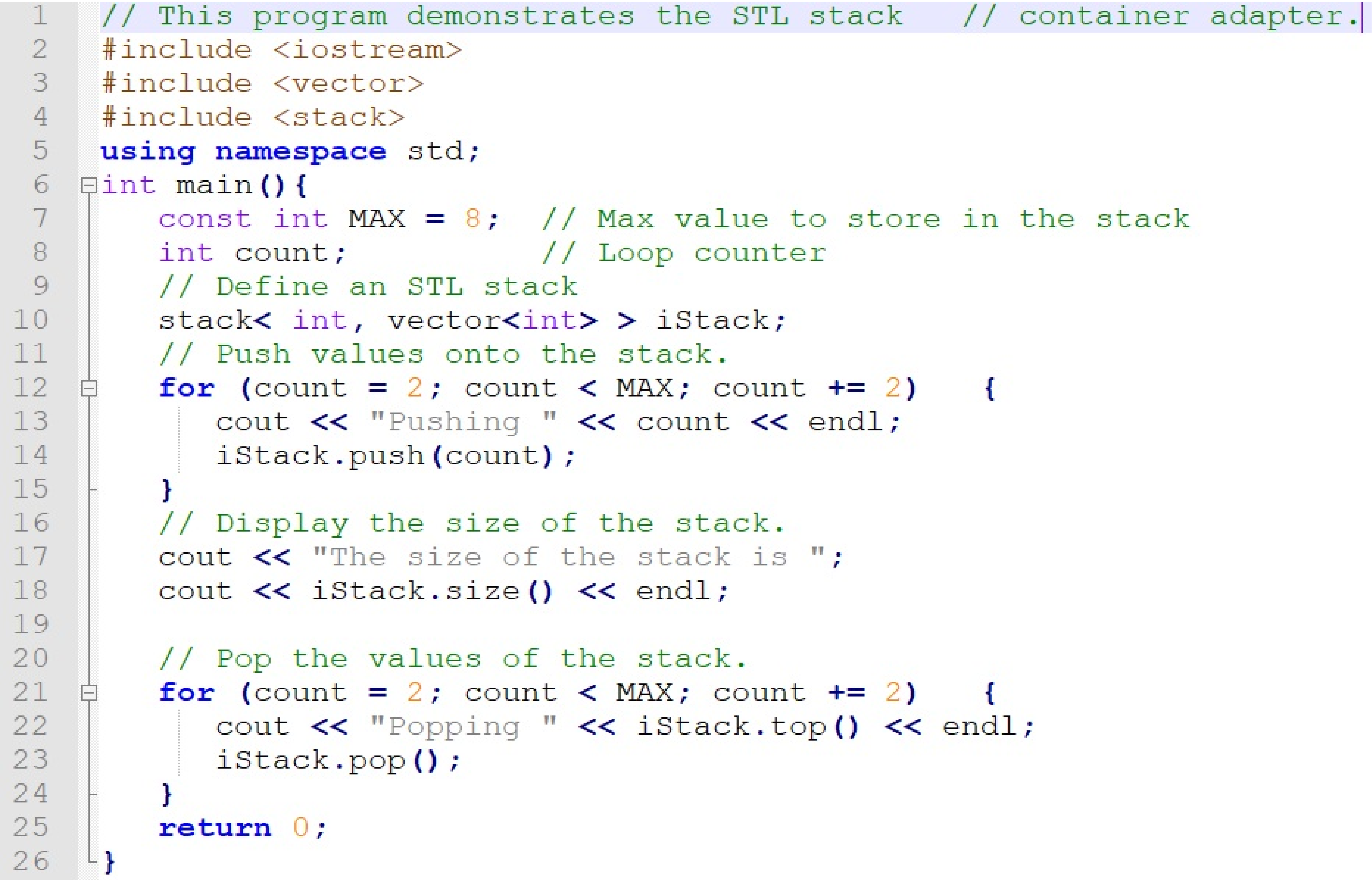
  ```
  stack< char, list<char> > cstack;
  ```

- implemented using a `deque`:

  ```
  stack< char > cstack;
  ```

- Spaces are required between consecutive >>, << symbols

Defining a `stack`



```cpp
// This program demonstrates the STL stack   // container adapter.
#include <iostream>
#include <vector>
#include <stack>
using namespace std;
int main(){
   const int MAX = 8;  // Max value to store in the stack
   int count;          // Loop counter
   // Define an STL stack
   stack< int, vector<int> > iStack;
   // Push values onto the stack.
   for (count = 2; count < MAX; count += 2)   {
      cout << "Pushing " << count << endl;
      iStack.push(count);
   }
   // Display the size of the stack.
   cout << "The size of the stack is ";
   cout << iStack.size() << endl;

   // Pop the values of the stack.
   for (count = 2; count < MAX; count += 2)   {
      cout << "Popping " << iStack.top() << endl;
      iStack.pop();
   }
   return 0;
}
```

Introduction to the Queue ADT

- Queue: a FIFO (first in, first out) data structure.
- Examples:
  - people in line at the theatre box office
  - print jobs sent to a printer
- Implementation:
  - static: fixed size, implemented as array
  - dynamic: variable size, implemented as linked list

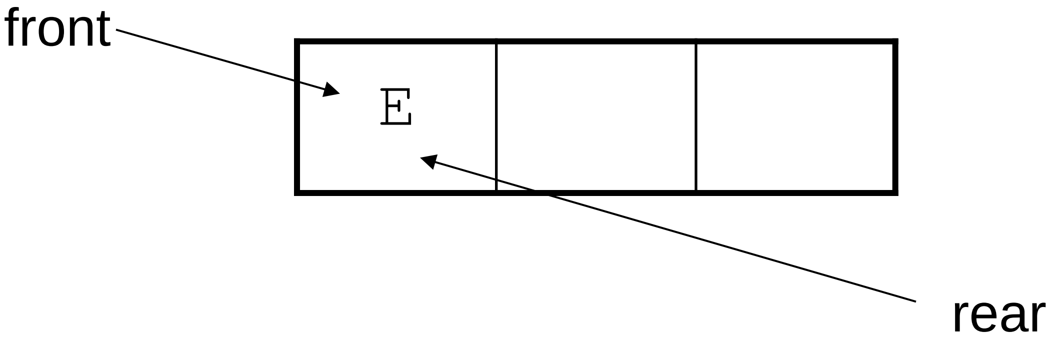
Queue Locations and Operations

- rear: position where elements are added
- front: position from which elements are removed
- enqueue: add an element to the rear of the queue
- dequeue: remove an element from the front of a queue

Queue Operations - Example

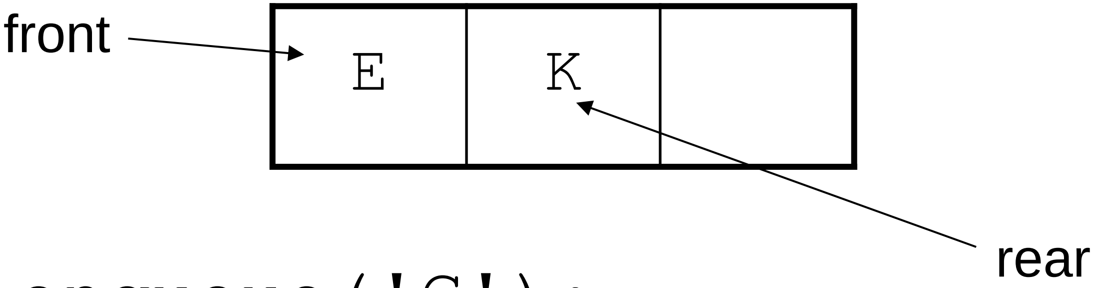
- A currently empty queue that can hold `char` values:

- `enqueue('E');`

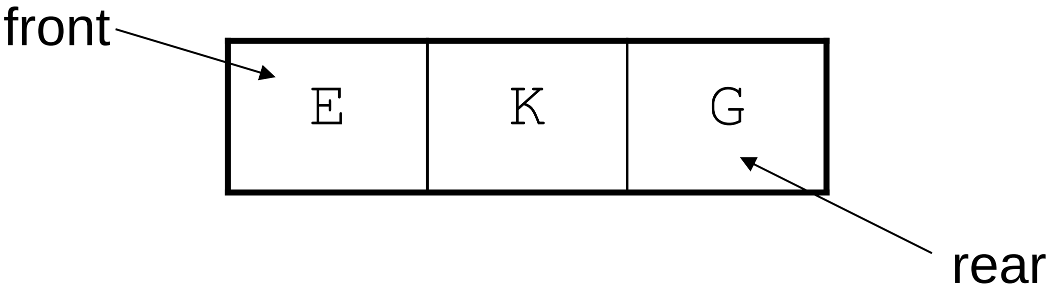


Queue Operations - Example

- `enqueue('K');`

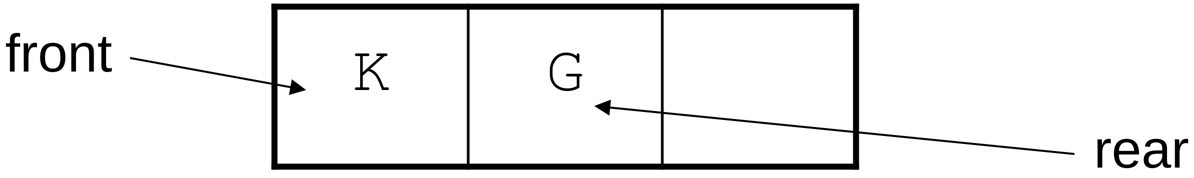


- `enqueue('G');`

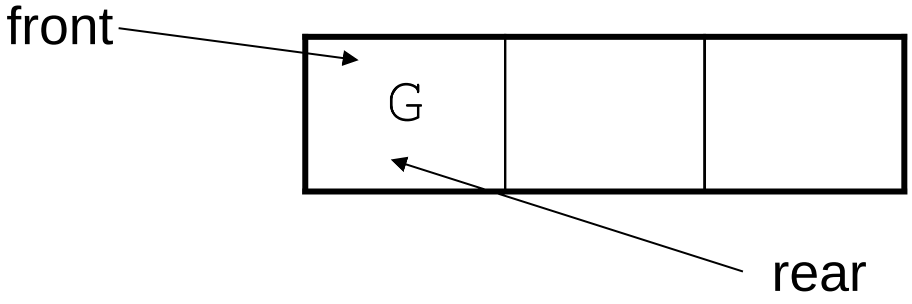


Queue Operations - Example

- `dequeue(); // remove E`



- `dequeue(); // remove K`



`dequeue` Issue, Solutions

- When removing an element from a queue, remaining elements must shift to front
- Solutions:
  - Let front index move as elements are removed (works as long as rear index is not at end of array)
  - Use above solution, and also let rear index "wrap around" to front of array, treating array as circular instead of linear (more complex enqueue, dequeue code)

Contents of `IntQueue.h`

```
// Specification file for the IntQueue class
#ifndef INTQUEUE_H
#define INTQUEUE_H

class IntQueue
{
private:
   int *queueArray;  // Points to the queue array
   int queueSize;    // The queue size
   int front;        // Subscript of the queue front
   int rear;         // Subscript of the queue rear
   int numItems;     // Number of items in the queue
```

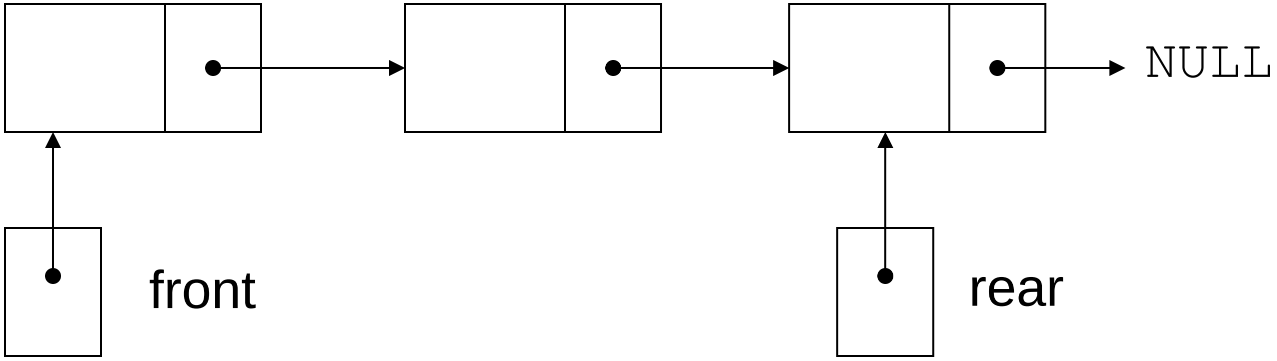
Contents of `IntQueue.h` (Continued)

```
public:
   // Constructor
   IntQueue(int);

   // Copy constructor
   IntQueue(const IntQueue &);

   // Destructor
   ~IntQueue();

   // Queue operations
   void enqueue(int);
   void dequeue(int &);
   bool isEmpty() const;
   bool isFull() const;
   void clear();
};
#endif
```

(See IntQueue.cpp for the implementation)

Dynamic Queues

- Like a stack, a queue can be implemented using a linked list
- Allows dynamic sizing, avoids issue of shifting elements or wrapping indices



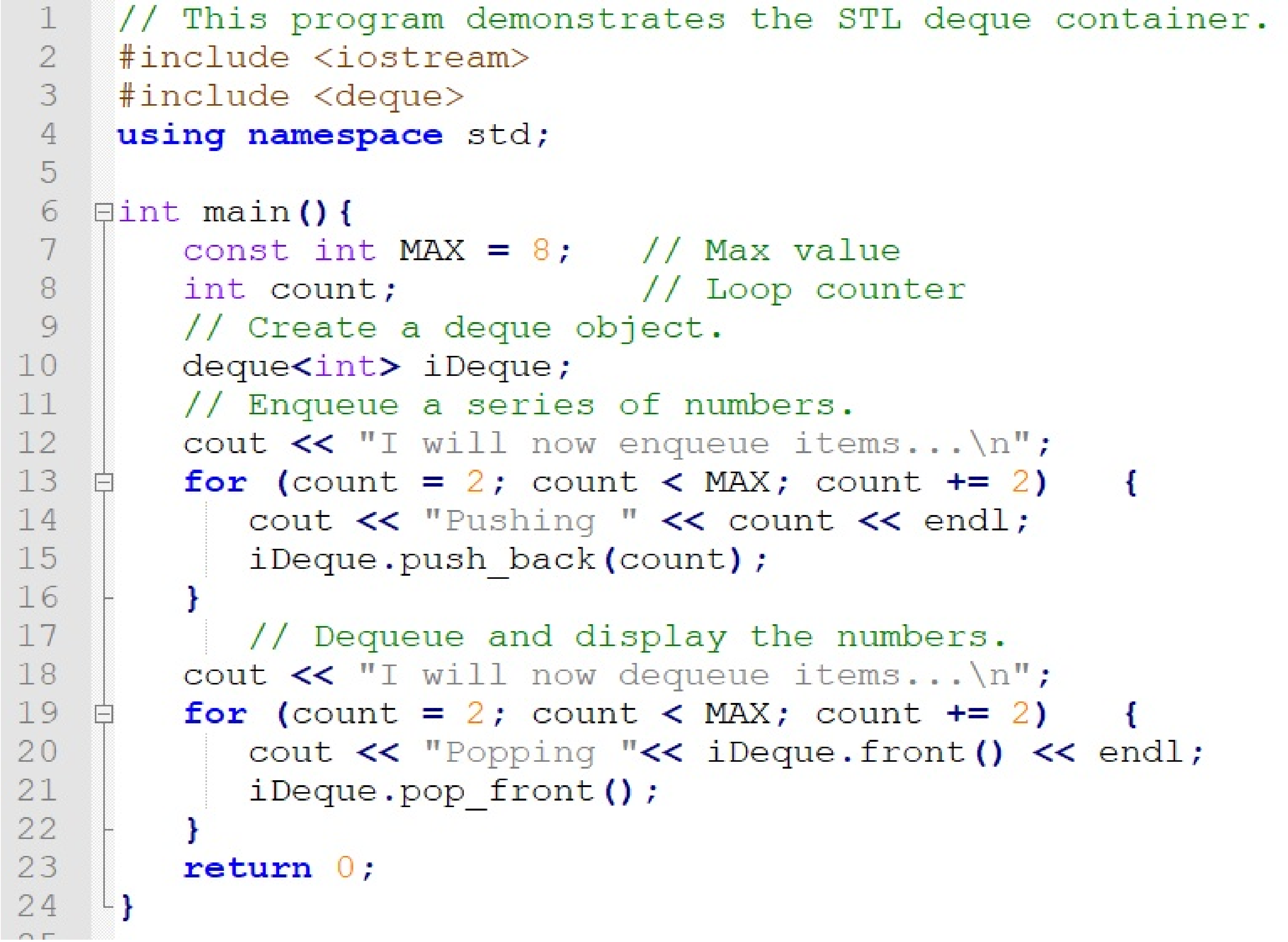
Implementing a Queue

- Programmers can program their own routines to implement queue operations
- See the `DynIntQue` class in the book for an example of a dynamic queue
- Can also use the implementation of queue and dequeue available in the STL
- Other implementations: See Malik folder (Sample programs)
- Application: See Applications folder (See Sample Programs)

The STL `deque` and `queue` Containers

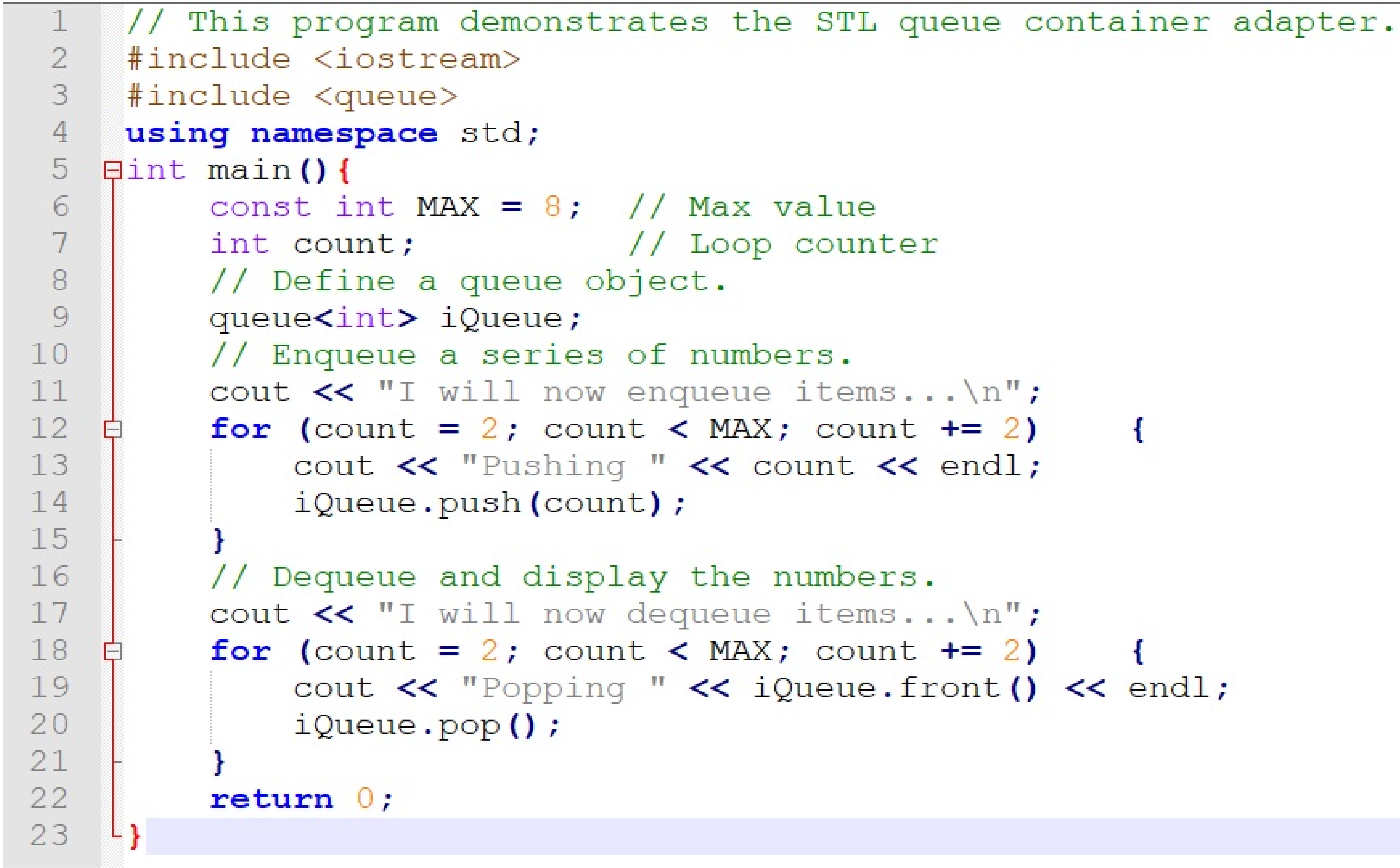
- `deque`: a double-ended queue. Has member functions to enqueue (`push_back`) and dequeue (`pop_front`)
- `queue`: container ADT that can be used to provide queue as a `vector`, list, or `deque`. Has member functions to enque (`push`) and dequeue (`pop`)

The STL deque and queue Containers



```
// This program demonstrates the STL deque container.
#include <iostream>
#include <deque>
using namespace std;

int main(){
    const int MAX = 8;     // Max value
    int count;             // Loop counter
    // Create a deque object.
    deque<int> iDeque;
    // Enqueue a series of numbers.
    cout << "I will now enqueue items...\n";
    for (count = 2; count < MAX; count += 2)   {
        cout << "Pushing " << count << endl;
        iDeque.push_back(count);
    }
        // Dequeue and display the numbers.
    cout << "I will now dequeue items...\n";
    for (count = 2; count < MAX; count += 2)   {
        cout << "Popping "<< iDeque.front() << endl;
        iDeque.pop_front();
    }
    return 0;
}
```

The STL deque and queue Containers



```
// This program demonstrates the STL queue container adapter.
#include <iostream>
#include <queue>
using namespace std;
int main(){
    const int MAX = 8;  // Max value
    int count;          // Loop counter
    // Define a queue object.
    queue<int> iQueue;
    // Enqueue a series of numbers.
    cout << "I will now enqueue items...\n";
    for (count = 2; count < MAX; count += 2)    {
        cout << "Pushing " << count << endl;
        iQueue.push(count);
    }
    // Dequeue and display the numbers.
    cout << "I will now dequeue items...\n";
    for (count = 2; count < MAX; count += 2)    {
        cout << "Popping " << iQueue.front() << endl;
        iQueue.pop();
    }
    return 0;
}
```

Defining a `queue`

- Defining a queue of `char`s, named cQueue, implemented using a deque:

```
deque<char> cQueue;
```

- implemented using a queue:

```
queue<char> cQueue;
```

- implemented using a `list`:

```
queue< char, list<char> > cQueue;
```

- Spaces are required between consecutive `>>`, `<<` symbols