CHAPTER

18 Stacks and Queues

TOPICS

18.1 Introduction to the Stack ADT

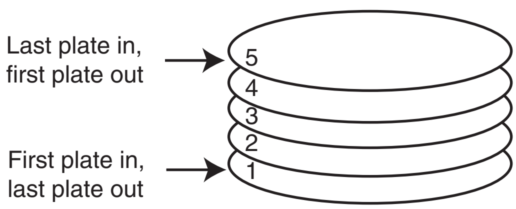
CONCEPT: A stack is a data structure that stores and retrieves items in a last-in-first-out manner.

Definition

Like an array or a linked list, a stack is a data structure that holds a sequence of elements. Unlike arrays and lists, however, stacks are *last-in, first-out (LIFO)* structures. This means that when a program retrieves elements from a stack, the last element inserted into the stack is the first one retrieved (and likewise, the first element inserted is the last one retrieved).

When visualizing the way a stack works, think of a stack of plates at the beginning of a cafeteria line. When a cafeteria worker replenishes the supply of plates, the first one he or she puts on the stack is the last one taken off. This is illustrated in Figure 18-1.

Figure 18-1



The LIFO characteristic of a stack of plates in a cafeteria is also the primary characteristic of a stack data structure. The last data element placed on the stack is the first data retrieved from the stack.

Applications of Stacks

Stacks are useful data structures for algorithms that work first with the last saved element of a series. For example, computer systems use stacks while executing programs. When a function is called, they save the program's return address on a stack. They also create local variables on a stack. When the function terminates, the local variables are removed from the stack and the return address is retrieved. Also, some calculators use a stack for performing mathematical operations.

Static and Dynamic Stacks

There are two types of stack data structure: static and dynamic. Static stacks have a fixed size and are implemented as arrays. Dynamic stacks grow in size as needed and are implemented as linked lists. In this section you will see examples of both static and dynamic stacks.

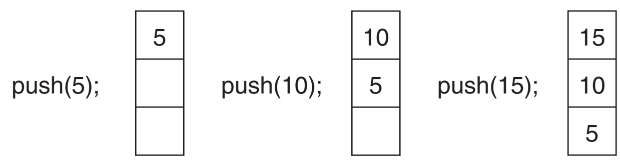
Stack Operations

A stack has two primary operations: *push* and *pop*. The push operation causes a value to be stored, or pushed onto the stack. For example, suppose we have an empty integer stack that is capable of holding a maximum of three values. With that stack we execute the following push operations.

```
push(5);
push(10);
push(15);
```

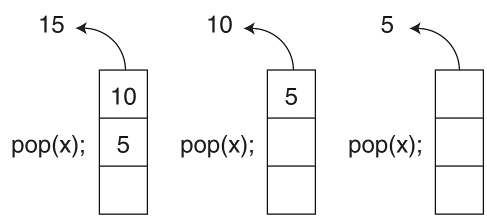
Figure 18-2 illustrates the state of the stack after each of these push operations.

Figure 18-2



The pop operation retrieves (and hence, removes) a value from the stack. Suppose we execute three consecutive pop operations on the stack shown in Figure 18-2. Figure 18-3 depicts the results.

Figure 18-3



As you can see from Figure 18-3, the last pop operation leaves the stack empty.

For a static stack (one with a fixed size), we will need a Boolean `isFull` operation. The `isFull` operation returns `true` if the stack is full and `false` otherwise. This operation is necessary to prevent a stack overflow in the event that a push operation is attempted when all the stack's elements have values stored in them.

For both static and dynamic stacks we will need a Boolean `isEmpty` operation. The `isEmpty` operation returns `true` when the stack is empty and `false` otherwise. This prevents an error from occurring when a pop operation is attempted on an empty stack.

A Static Stack Class

Now we examine a class, `IntStack`, that stores a static stack of integers and performs the `isFull` and `isEmpty` operations. The class has the member variables described in Table 18-1.

Table 18-1

Member Variable	Description
`stackArray`	A pointer to `int`. When the constructor is executed, it uses `stackArray` to dynamically allocate an array for storage.
`stackSize`	An integer that holds the size of the stack.
`top`	An integer that is used to mark the top of the stack.

The class's member functions are listed in Table 18-2.

Table 18-2

Member Functions	Description
Constructor	The class constructor accepts an integer argument that specifies the size of the stack. An integer array of this size is dynamically allocated and assigned to `stackArray`. Also, the variable `top` is initialized to -1.
Destructor	The destructor frees the memory that was allocated by the constructor.
`isFull`	Returns `true` if the stack is full and `false` otherwise. The stack is full when `top` is equal to `stackSize - 1`.
`isEmpty`	Returns `true` if the stack is empty and `false` otherwise. The stack is empty when `top` is set to -1.
`pop`	The `pop` function uses an integer reference parameter. The value at the top of the stack is removed and copied into the reference parameter.
`push`	The `push` function accepts an integer argument, which is pushed onto the top of the stack.

NOTE: Even though the constructor dynamically allocates the stack array, it is still a static stack. The size of the stack does not change once it is allocated.

The code for the class follows.

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
```

Contents of `IntStack.cpp`

```
// Implementation file for the IntStack 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 = nullptr;

    // 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;
}

//*********************************************
// 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;
    }
}
```

```
//****************************************************
// Member function pop pops the value at the top     *
// of the stack off, and copies it into the variable *
// passed as an argument.                            *
//****************************************************

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

//***************************************************
// Member function isFull returns true if the stack *
// is full, or false otherwise.                     *
//***************************************************

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

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

    return status;
}

//****************************************************
// Member function 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;
}
```

The class has two constructors, one that accepts an argument for the stack size (lines 12 through 17 of `IntStack.cpp`) and a copy constructor (lines 23 through 40). The first constructor dynamically allocates the stack array in line 14, initializes the `stackSize` member variable in line 15, and initializes the `top` member variable in line 16. Remember that items are stored to and retrieved from the top of the stack. In this class, the top of the stack is actually the end of the array. The variable `top` is used to mark the top of the stack by holding the subscript of the last element. When `top` holds −1, it indicates that the stack is empty. (See the `isEmpty` function, which returns `true` if `top` is −1, or `false` otherwise.) The stack is full when `top` is at the maximum subscript, which is `stackSize - 1`. This is the value that `isFull` tests for. It returns `true` if the stack is full, or `false` otherwise.

Program 18-1 is a simple driver that demonstrates the `IntStack` class.

Program 18-1

```
// This program demonstrates the IntStack class.
#include <iostream>
#include "IntStack.h"
using namespace std;

int main()
{
    int catchVar; // To hold values popped off the stack

    // Define a stack object to hold 5 values.
    IntStack stack(5);

    // Push the values 5, 10, 15, 20, and 25 onto the stack.
    cout << "Pushing 5\n";
    stack.push(5);
    cout << "Pushing 10\n";
    stack.push(10);
    cout << "Pushing 15\n";
    stack.push(15);
    cout << "Pushing 20\n";
    stack.push(20);
    cout << "Pushing 25\n";
    stack.push(25);

    // Pop the values off the stack.
    cout << "Popping...\n";
    stack.pop(catchVar);
    cout << catchVar << endl;
    stack.pop(catchVar);
    cout << catchVar << endl;
    stack.pop(catchVar);
    cout << catchVar << endl;
    stack.pop(catchVar);
    cout << catchVar << endl;
    stack.pop(catchVar);
    cout << catchVar << endl;
    return 0;
}
```

(program output continues)

Program 18-1 *(continued)*

Program Output

```
Pushing 5
Pushing 10
Pushing 15
Pushing 20
Pushing 25
Popping...
25
20
15
10
5
```

In Program 18-1, the constructor is called with the argument 5. This sets up the member variables as shown in Figure 18-4. Because `top` is set to −1, the stack is empty.

Figure 18-4

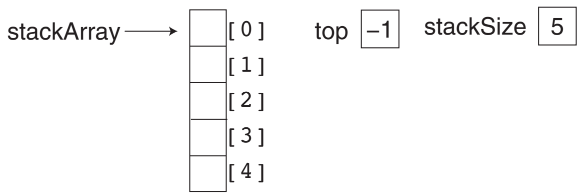


Figure 18-5 shows the state of the member variables after the `push` function is called the first time (with 5 as its argument). The top of the stack is now at element 0.

Figure 18-5

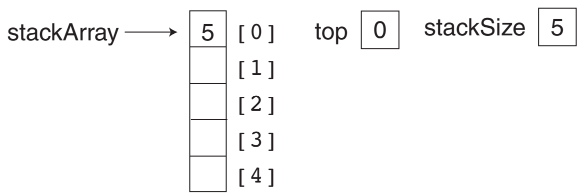
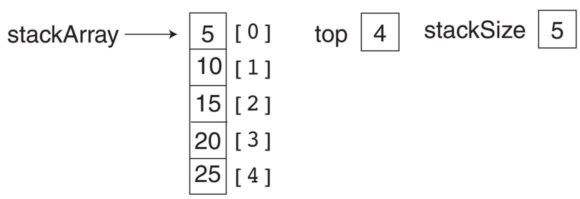


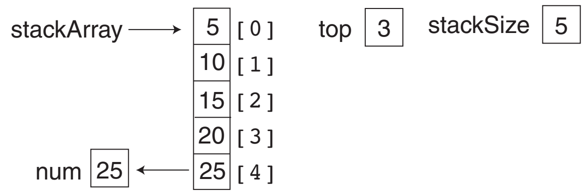
Figure 18-6 shows the state of the member variables after all five calls to the `push` function. Now the top of the stack is at element 4, and the stack is full.

Figure 18-6



Notice that the `pop` function uses a reference parameter, `num`. The value that is popped off the stack is copied into `num` so it can be used later in the program. Figure 18-7 depicts the state of the class members, and the `num` parameter, just after the first value is popped off the stack.

Figure 18-7



The program continues to call the `pop` function until all the values have been retrieved from the stack.

Implementing Other Stack Operations

More complex operations may be built on the basic stack class previously shown. In this section, we will discuss a class, `MathStack`, that is derived from `IntStack`. The `MathStack` class has two member functions: `add()` and `sub()`. The `add()` function pops the first two values off the stack, adds them together, and pushes the sum onto the stack. The `sub()` function pops the first two values off the stack, subtracts the second value from the first, and then pushes the difference onto the stack. The class declaration is as follows.

Contents of `MathStack.h`

```
// Specification file for the MathStack class
#ifndef MATHSTACK_H
#define MATHSTACK_H
#include "IntStack.h"

class MathStack : public IntStack
{
public:
    // Constructor
    MathStack(int s) : IntStack(s) {}

    // MathStack operations
    void add();
    void sub();
};
#endif
```

The definitions of the member functions are shown here:

Contents of `MathStack.cpp`

```
// Implementation file for the MathStack class
#include "MathStack.h"

```

```
//***********************************************
// Member function add. add pops               *
// the first two values off the stack and      *
// adds them. The sum is pushed onto the stack. *
//***********************************************

void MathStack::add()
{
     int num, sum;

     // Pop the first two values off the stack.
     pop(sum);
     pop(num);

     // Add the two values, store in sum.
     sum += num;

     // Push sum back onto the stack.
     push(sum);
}

//**********************************************
// Member function sub. sub pops the          *
// first two values off the stack. The        *
// second value is subtracted from the        *
// first value. The difference is pushed      *
// onto the stack.                            *
//**********************************************

void MathStack::sub()
{
     int num, diff;

     // Pop the first two values off the stack.
     pop(diff);
     pop(num);

     // Subtract num from diff.
     diff -= num;

     // Push diff back onto the stack.
     push(diff);
}
```

The class is demonstrated in Program 18-2, a simple driver.

Program 18-2

```
// This program demonstrates the MathStack class.
#include <iostream>
#include "MathStack.h"
using namespace std;

```

```
int main()
{
    int catchVar; // To hold values popped off the stack

    // Create a MathStack object.
    MathStack stack(5);

    // Push 3 and 6 onto the stack.
    cout << "Pushing 3\n";
    stack.push(3);
    cout << "Pushing 6\n";
    stack.push(6);

    // Add the two values.
    stack.add();

    // Pop the sum off the stack and display it.
    cout << "The sum is ";
    stack.pop(catchVar);
    cout << catchVar << endl << endl;

    // Push 7 and 10 onto the stack
    cout << "Pushing 7\n";
    stack.push(7);
    cout << "Pushing 10\n";
    stack.push(10);

    // Subtract 7 from 10.
    stack.sub();

    // Pop the difference off the stack and display it.
    cout << "The difference is ";
    stack.pop(catchVar);
    cout << catchVar << endl;
    return 0;
}
```

Program Output

```
Pushing 3
Pushing 6
The sum is 9

Pushing 7
Pushing 10
The difference is 3
```

It will be left as a Programming Challenge for you to implement the `mult()`, `div()`, and `mod()` functions that will complete the `MathStack` class.

A Static Stack Template

The stack classes shown previously in this chapter work only with integers. A stack template can be easily designed to work with any data type, as shown by the following example:

Contents of Stack.h

```
#ifndef STACK_H
#define STACK_H
#include <iostream>
using namespace std;

// Stack template
template <class T>
class Stack
{
private:
    T *stackArray;
    int stackSize;
    int top;

public:
    // Constructor
    Stack(int);

    // Copy constructor
    Stack(const Stack&);

    // Destructor
    ~Stack();

    // Stack operations
    void push(T);
    void pop(T &);
    bool isFull();
    bool isEmpty();
};

//*****************************************************
// Constructor                                        *
//*****************************************************

template <class T>
Stack<T>::Stack(int size)
{
    stackArray = new T[size];
    stackSize = size;
    top = -1;
}

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

```

```
template <class T>
Stack<T>::Stack(const Stack &obj)
{
    // Create the stack array.
    if (obj.stackSize > 0)
        stackArray = new T[obj.stackSize];
    else
        stackArray = nullptr;

    // 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;
}

//***********************************************
// Destructor                                   *
//***********************************************

template <class T>
Stack<T>::~Stack()
{
    if (stackSize > 0)
        delete [] stackArray;
}

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

template <class T>
void Stack<T>::push(T item)
{
    if (isFull())
    {
        cout << "The stack is full.\n";
    }
    else
    {
        top++;
        stackArray[top] = item;
    }
}

//***************************************************
// Member function pop pops the value at the top    *
// of the stack off, and copies it into the variable *
// passed as an argument.                           *
//***************************************************
```

```

template <class T>
void Stack<T>::pop(T &item)
{
    if (isEmpty())
    {
        cout << "The stack is empty.\n";
    }
    else
    {
        item = stackArray[top];
        top--;
    }
}

//***********************************************************
// Member function isFull returns true if the stack          *
// is full, or false otherwise.                              *
//***********************************************************

template <class T>
bool Stack<T>::isFull()
{
    bool status;

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

    return status;
}

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

template <class T>
bool Stack<T>::isEmpty()
{
    bool status;

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

    return status;
}
#endif
```

Program 18-3 demonstrates the `Stack` template. It creates a stack of strings, and then presents a menu that allows the user to push an item onto the stack, pop an item from the stack, or quit the program.

Program 18-3

```
// This program demonstrates the Stack template.
#include <iostream>
#include <string>
#include "Stack.h"
using namespace std;

// Constants for the menu choices
const int PUSH_CHOICE = 1,
          POP_CHOICE = 2,
          QUIT_CHOICE = 3;

// Function prototypes
void menu(int &);
void getStackSize(int &);
void pushItem(Stack<string>&);
void popItem(Stack<string>&);

int main()
{
    int stackSize; // The stack size
    int choice;    // To hold a menu choice

    // Get the stack size.
    getStackSize(stackSize);

    // Create the stack.
    Stack<string> stack(stackSize);

    do
    {
        // Get the user's menu choice.
        menu(choice);

        // Perform the user's choice.
        if (choice != QUIT_CHOICE)
        {
            switch (choice)
            {
                case PUSH_CHOICE:
                     pushItem(stack);
                     break;
                case POP_CHOICE:
                     popItem(stack);
            }
        }
    } while (choice != QUIT_CHOICE);

    return 0;
}
```

(program continues)

Program 18-3 *(continued)*

```
//**********************************************
// The getStackSize function gets the desired    *
// stack size, which is assigned to the          *
// reference parameter.                          *
//**********************************************
void getStackSize(int &size)
{
    // Get the desired stack size.
    cout << "How big should I make the stack? ";
    cin >> size;

    // Validate the size.
    while (size < 1)
    {
        cout << "Enter 1 or greater: ";
        cin >> size;
    }
}

//**********************************************
// The menu function displays the menu and gets *
// the user's choice, which is assigned to the   *
// reference parameter.                          *
//**********************************************
void menu(int &choice)
{
    // Display the menu and get the user's choice.
    cout << "\nWhat do you want to do?\n"
         << PUSH_CHOICE
         << " - Push an item onto the stack\n"
         << POP_CHOICE
         << " - Pop an item off the stack\n"
         << QUIT_CHOICE
         << " - Quit the program\n"
         << "Enter your choice: ";
    cin >> choice;

    // Validate the choice
    while (choice < PUSH_CHOICE || choice > QUIT_CHOICE)
    {
        cout << "Enter a valid choice: ";
        cin >> choice;
    }
}

//**********************************************
// The pushItem function gets an item from the  *
// user and pushes it onto the stack.           *
//**********************************************
void pushItem(Stack<string> &stack)
```

```
{
    string item;

    // Get an item to push onto the stack.
    cin.ignore();
    cout << "\nEnter an item: ";
    getline(cin, item);
    stack.push(item);
}

//***************************************************
// The popItem function pops an item from the stack *
//***************************************************
void popItem(Stack<string> &stack)
{
    string item = "";

    // Pop the item.
    stack.pop(item);

    // Display the item.
    if (item != "")
       cout << item << " was popped.\n";
}
```

Program Output with Example Input Shown in Bold

```
How big should I make the stack? 3 [Enter]
What do you want to do?
1 - Push an item onto the stack
2 - Pop an item off the stack
3 - Quit the program
Enter your choice: 1 [Enter]

Enter an item: The Adventures of Huckleberry Finn [Enter]

What do you want to do?
1 - Push an item onto the stack
2 - Pop an item off the stack
3 - Quit the program
Enter your choice: 1 [Enter]

Enter an item: All Quiet on the Western Front [Enter]

What do you want to do?
1 - Push an item onto the stack
2 - Pop an item off the stack
3 - Quit the program
Enter your choice: 1 [Enter]

Enter an item: Brave New World [Enter]
```

(program output continues)

Program 18-3 *(continued)*

```
What do you want to do?
1 - Push an item onto the stack
2 - Pop an item off the stack
3 - Quit the program
Enter your choice: 2 [Enter]
Brave New World was popped.

What do you want to do?
1 - Push an item onto the stack
2 - Pop an item off the stack
3 - Quit the program
Enter your choice: 2 [Enter]
All Quiet on the Western Front was popped.

What do you want to do?
1 - Push an item onto the stack
2 - Pop an item off the stack
3 - Quit the program
Enter your choice: 2 [Enter]
The Adventures of Huckleberry Finn was popped.

What do you want to do?
1 - Push an item onto the stack
2 - Pop an item off the stack
3 - Quit the program
Enter your choice: 2 [Enter]
The stack is empty.

What do you want to do?
1 - Push an item onto the stack
2 - Pop an item off the stack
3 - Quit the program
Enter your choice: 3 [Enter]
```

18.2 Dynamic Stacks

CONCEPT: A stack may be implemented as a linked list and expand or shrink with each `push` or `pop` operation.

A dynamic stack is built on a linked list instead of an array. A linked list–based stack offers two advantages over an array-based stack. First, there is no need to specify the starting size of the stack. A dynamic stack simply starts as an empty linked list, then expands by one node each time a value is pushed. Second, a dynamic stack will never be full, as long as the system has enough free memory.

In this section we will look at a dynamic stack class, `DynIntStack`. This class is a dynamic version of the `IntStack` class previously discussed. The class declaration is shown here:

Contents of DynIntStack.h

```
// Specification file for the DynIntStack class
#ifndef DYNINTSTACK_H
#define DYNINTSTACK_H

class DynIntStack
{
private:
    // Structure for stack nodes
    struct StackNode
    {
        int value;        // Value in the node
        StackNode *next;  // Pointer to the next node
    };

    StackNode *top;       // Pointer to the stack top

public:
    // Constructor
    DynIntStack()
        { top = nullptr; }

    // Destructor
    ~DynIntStack();

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

The `StackNode` structure is the data type of each node in the linked list. It has a `value` member and a `next` pointer. Notice that instead of a `head` pointer, a `top` pointer is defined. This member will always point to the first node in the list, which will represent the top of the stack. It is initialized to `nullptr` by the constructor, to signify that the stack is empty.

The definitions of the other member functions are shown here:

Contents of DynIntStack.cpp

```
#include <iostream>
#include "DynIntStack.h"
using namespace std;

//**************************************************
// Destructor                                      *
// This function deletes every node in the list.   *
//**************************************************

DynIntStack::~DynIntStack()
{
    StackNode *nodePtr = nullptr, *nextNode = nullptr;

```

```
    // Position nodePtr at the top of the stack.
    nodePtr = top;

    // Traverse the list deleting each node.
    while (nodePtr != nullptr)
    {
        nextNode = nodePtr->next;
        delete nodePtr;
        nodePtr = nextNode;
    }
}

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

void DynIntStack::push(int num)
{
    StackNode *newNode = nullptr; // 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 = nullptr;
    }
    else  // Otherwise, insert NewNode before top.
    {
        newNode->next = top;
        top = newNode;
    }
}

//****************************************************
// Member function pop pops the value at the top     *
// of the stack off, and copies it into the variable *
// passed as an argument.                            *
//****************************************************

void DynIntStack::pop(int &num)
{
    StackNode *temp = nullptr; // Temporary pointer

    // First make sure the stack isn't empty.
    if (isEmpty())
    {
        cout << "The stack is empty.\n";
    }
    else // pop value off top of stack
```

```
69      {
70          num = top->value;
71          temp = top->next;
72          delete top;
73          top = temp;
74      }
75  }
76
77  //***************************************************
78  // Member function isEmpty returns true if the stack *
79  // is empty, or false otherwise.                     *
80  //***************************************************
81
82  bool DynIntStack::isEmpty()
83  {
84      bool status;
85
86      if (!top)
87          status = true;
88      else
89          status = false;
90
91      return status;
92  }
```

Let's look at the push operation in lines 31 through 51 of `DynIntStack.cpp`. First, in lines 36 and 37, a new node is allocated in memory, and the function argument is copied into its value member:

```
newNode = new StackNode;
newNode->value = num;
```

Next in line 41, an `if` statement calls the `isEmpty` function to determine whether the stack is empty:

```
if (isEmpty())
{
   top = newNode;
   newNode->next = nullptr;
}
```

If `isEmpty` returns `true`, `top` is made to point at the new node, and the new node's `next` pointer is set to `nullptr`. After these statements execute, there will be one node in the list (and one value on the stack).

If `isEmpty` returns `false` in the `if` statement, the following statements in lines 46 through 50 are executed.

```
else // Otherwise, insert newNode before top
{
   newNode->next = top;
   top = newNode;
}
```

Notice that `newNode` is being inserted in the list before the node that `top` points to. The `top` pointer is then updated to point to the new node. When this is done, `newNode` is at the top of the stack.

Now let's look at the pop function in lines 59 through 75. Just as the push function must insert nodes at the head of the list, pop must delete nodes at the head of the list. First, the function calls isEmpty in line 64 to determine whether there are any nodes in the stack. If there are none, an error message is displayed:

```
if (isEmpty())
{
   cout << "The stack is empty.\n";
}
```

If isEmpty returns false, then the following statements in lines 68 through 74 are executed.

```
else   // pop value off top of stack
{
   num = top->value;
   temp = top->next;
   delete top;
   top = temp;
}
```

First, the value member of the top node is copied into the num reference parameter. This saves the value for later use in the program. Next, a temporary StackNode pointer, temp, is made to point to top->next. If there are other nodes in the list, this causes temp to point to the second node. (If there are no more nodes, this will cause temp to point to nullptr.) Now it is safe to delete the top node. After the top node is deleted, the top pointer is set equal to temp. This action moves the top pointer down the list by one node. The node that was previously second in the list becomes first.

The isEmpty function, in lines 82 through 92, is simple. If top is a null pointer, then the list (the stack) is empty.

Program 18-4 is a driver that demonstrates the DynIntStack class.

Program 18-4

```
// This program demonstrates the dynamic stack.
// class DynIntClass.
#include <iostream>
#include "DynIntStack.h"
using namespace std;

int main()
{
    int catchVar; // To hold values popped off the stack

    // Create a DynIntStack object.
    DynIntStack stack;

    // Push 5, 10, and 15 onto the stack.
    cout << "Pushing 5\n";
    stack.push(5);
    cout << "Pushing 10\n";
    stack.push(10);
```

```
    cout << "Pushing 15\n";
    stack.push(15);

    // Pop the values off the stack and display them.
    cout << "Popping...\n";
    stack.pop(catchVar);
    cout << catchVar << endl;
    stack.pop(catchVar);
    cout << catchVar << endl;
    stack.pop(catchVar);
    cout << catchVar << endl;

    // Try to pop another value off the stack.
    cout << "\nAttempting to pop again... ";
    stack.pop(catchVar);
    return 0;
}
```

Program Output

```
Pushing 5
Pushing 10
Pushing 15
Popping...
15
10
5

Attempting to pop again... The stack is empty.
```

A Dynamic Stack Template

The dynamic stack class shown previously in this chapter works only with integers. A dynamic stack template can be easily designed to work with any data type, as shown by the following example:

Contents of `DynamicStack.h`

```
#ifndef DYNAMICSTACK_H
#define DYNAMICSTACK_H
#include <iostream>
using namespace std;

// Stack template
template <class T>
class DynamicStack
{
private:
    // Structure for the stack nodes
    struct StackNode
    {
        T value;          // Value in the node
        StackNode *next;  // Pointer to the next node
    };
```

```

    StackNode *top; // Pointer to the stack top

public:
    //Constructor
    DynamicStack()
    { top = nullptr; }

    // Destructor
    ~DynamicStack();

    // Stack operations
    void push(T);
    void pop(T &);
    bool isEmpty();
};

//**************************************************
// Destructor                                      *
//**************************************************
template <class T>
DynamicStack<T>::~DynamicStack()
{
    StackNode *nodePtr, *nextNode;

    // Position nodePtr at the top of the stack.
    nodePtr = top;

    // Traverse the list deleting each node.
    while (nodePtr != nullptr)
    {
        nextNode = nodePtr->next;
        delete nodePtr;
        nodePtr = nextNode;
    }
}

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

template <class T>
void DynamicStack<T>::push(T item)
{
    StackNode *newNode = nullptr; // Pointer to a new node

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

    // If there are no nodes in the list
    // make newNode the first node.
    if (isEmpty())
```

```
    {
        top = newNode;
        newNode->next = nullptr;
    }
    else  // Otherwise, insert NewNode before top.
    {
        newNode->next = top;
        top = newNode;
    }
}

//*********************************************************
// Member function pop pops the value at the top          *
// of the stack off, and copies it into the variable      *
// passed as an argument.                                 *
//*********************************************************

template <class T>
void DynamicStack<T>::pop(T &item)
{
    StackNode *temp = nullptr; // Temporary pointer

    // First make sure the stack isn't empty.
    if (isEmpty())
    {
        cout << "The stack is empty.\n";
    }
    else // pop value off top of stack
    {
        item = top->value;
        temp = top->next;
        delete top;
        top = temp;
    }
}

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

template <class T>
bool DynamicStack<T>::isEmpty()
{
    bool status;

    if (!top)
        status = true;
    else
        status = false;

    return status;
}
#endif
```

Program 18-5 demonstrates the `DynamicStack` template. This program is a modification of Program 18-3. It creates a stack of strings and then presents a menu that allows the user to push an item onto the stack, pop an item from the stack, or quit the program.

Program 18-5

```
#include <iostream>
#include <string>
#include "DynamicStack.h"
using namespace std;

// Constants for the menu choices
const int PUSH_CHOICE = 1,
          POP_CHOICE = 2,
          QUIT_CHOICE = 3;

// Function prototypes
void menu(int &);
void getStackSize(int &);
void pushItem(DynamicStack<string> &);
void popItem(DynamicStack<string> &);

int main()
{
    int choice;  // To hold a menu choice

    // Create the stack.
    DynamicStack<string> stack;

    do
    {
        // Get the user's menu choice.
        menu(choice);

        // Perform the user's choice.
        if (choice != QUIT_CHOICE)
        {
            switch (choice)
            {
                case PUSH_CHOICE:
                     pushItem(stack);
                     break;
                case POP_CHOICE:
                     popItem(stack);
            }
        }
    } while (choice != QUIT_CHOICE);

    return 0;
}

```

```
//***********************************************
// The menu function displays the menu and gets *
// the user's choice, which is assigned to the  *
// reference parameter.                         *
//***********************************************
void menu(int &choice)
{
    // Display the menu and get the user's choice.
    cout << "What do you want to do?\n"
         << PUSH_CHOICE
         << " - Push an item onto the stack\n"
         << POP_CHOICE
         << " - Pop an item off the stack\n"
         << QUIT_CHOICE
         << " - Quit the program\n"
         << "Enter your choice: ";
    cin >> choice;

    // Validate the choice
    while (choice < PUSH_CHOICE || choice > QUIT_CHOICE)
    {
        cout << "Enter a valid choice: ";
        cin >> choice;
    }
}

//***********************************************
// The pushItem function gets an item from the  *
// user and pushes it onto the stack.           *
//***********************************************
void pushItem(DynamicStack<string> &stack)
{
    string item;

    // Get an item to push onto the stack.
    cin.ignore();
    cout << "\nEnter an item: ";
    getline(cin, item);
    stack.push(item);
}

//***************************************************
// The popItem function pops an item from the stack *
//***************************************************
void popItem(DynamicStack<string> &stack)
{
    string item = "";

    // Pop the item.
    stack.pop(item);

```

(program continues)

Program 18-5 *(continued)*

```
    // Display the item.
    if (item != "")
        cout << item << " was popped.\n";
}
```

Program Output with Example Input Shown in Bold

```
What do you want to do?
1 - Push an item onto the stack
2 - Pop an item off the stack
3 - Quit the program
Enter your choice: 1 [Enter]

Enter an item: The Catcher in the Rye [Enter]

What do you want to do?
1 - Push an item onto the stack
2 - Pop an item off the stack
3 - Quit the program
Enter your choice: 1 [Enter]

Enter an item: Crime and Punishment [Enter]
What do you want to do?
1 - Push an item onto the stack
2 - Pop an item off the stack
3 - Quit the program
Enter your choice: 2 [Enter]
Crime and Punishment was popped.

What do you want to do?
1 - Push an item onto the stack
2 - Pop an item off the stack
3 - Quit the program
Enter your choice: 2 [Enter]
The Catcher in the Rye was popped.

What do you want to do?
1 - Push an item onto the stack
2 - Pop an item off the stack
3 - Quit the program
Enter your choice: 2 [Enter]
The stack is empty.

What do you want to do?
1 - Push an item onto the stack
2 - Pop an item off the stack
3 - Quit the program
Enter your choice: 3 [Enter]
```

18.3 The STL `stack` Container

CONCEPT: The Standard Template Library offers a stack template, which may be implemented as a `vector`, a `list`, or a `deque`.

VideoNote
Storing Objects in an STL `stack`

So far, the STL containers you have learned about are `vector`s and `list`s. The STL `stack` container may be implemented as a `vector` or a `list`. (It may also be implemented as a `deque`, which you will learn about later in this chapter.) Because the `stack` container is used to adapt these other containers, it is often referred to as a *container adapter*.

Here are examples of how to define a stack of `int`s, implemented as a `vector`, a `list`, and a `deque`.

```
stack<int, vector<int>> iStack; // Vector stack
stack<int, list<int>> iStack;   // List stack
stack<int> iStack;              // Default - deque stack
```

NOTE: If you are using a compiler that is older than C++ 11, be sure to put spaces between the angled brackets that appear next to each other. Older compilers will mistake the >> characters for the stream extraction operator, >>. Here is an example of how to write the definitions for an older compiler:

```
stack< int, vector<int> > iStack; // Vector stack
stack< int, list<int> > iStack;   // List stack
stack< int > iStack;              // Default - deque stack
```

Table 18-3 lists and describes some of the stack template's member functions.

Table 18-3

Member Function	Examples and Description
`empty`	`if (myStack.empty())` The `empty` member function returns `true` if the `stack` is empty. If the `stack` has elements, it returns `false`.
`pop`	`myStack.pop();` The `pop` function removes the element at the top of the stack.
`push`	`myStack.push(x);` The `push` function pushes an element with the value `x` onto the stack.
`size`	`cout << myStack.size() << endl;` The `size` function returns the number of elements in the `list`.
`top`	`x = myStack.top();` The `top` function returns a reference to the element at the top of the stack.

NOTE: The `pop` function in the stack template does not retrieve the value from the top of the stack, it only removes it. To retrieve the value, you must call the `top` function first.

Program 18-6 is a driver that demonstrates an STL `stack` implemented as a `vector`.

Program 18-6

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

Program Output

```
Pushing 2
Pushing 4
Pushing 6
The size of the stack is 3
Popping 6
Popping 4
Popping 2
```

Checkpoint

18.1 Describe what LIFO means.

18.2 What is the difference between static and dynamic stacks? What advantages do dynamic stacks have over static stacks?

18.3 What are the two primary stack operations? Describe them both.

18.4 What STL types does the STL stack container adapt?

18.4 Introduction to the Queue ADT

CONCEPT: A queue is a data structure that stores and retrieves items in a first-in-first-out manner.

Definition

Like a stack, a queue (pronounced "cue") is a data structure that holds a sequence of elements. A queue, however, provides access to its elements in *first-in*, *first-out (FIFO)* order. The elements in a queue are processed like customers standing in a grocery checkout line: The first customer in line is the first one served.

Application of Queues

Queue data structures are commonly used in computer operating systems. They are especially important in multiuser/multitasking environments where several users or tasks may be requesting the same resource simultaneously. Printing, for example, is controlled by a queue because only one document may be printed at a time. A queue is used to hold print jobs submitted by users of the system, while the printer services those jobs one at a time.

Communications software also uses queues to hold data received over networks and dial-up connections. Sometimes data is transmitted to a system faster than it can be processed, so it is placed in a queue when it is received.

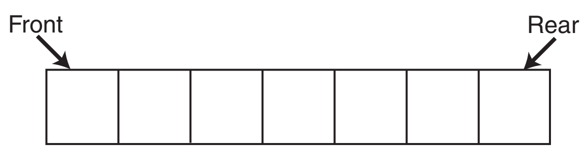
Static and Dynamic Queues

Just as stacks are implemented as arrays or linked lists, so are queues. Dynamic queues offer the same advantages over static queues that dynamic stacks offer over static stacks. In fact, the primary difference between queues and stacks is the way data elements are accessed in each structure.

Queue Operations

Just like checkout lines in a grocery store, think of queues as having a front and a rear. This is illustrated in Figure 18-8.

Figure 18-8



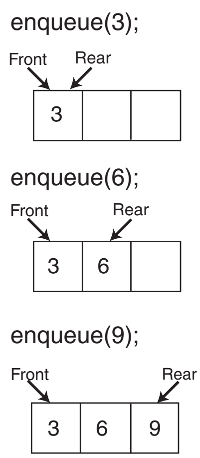
When an element is added to a queue, it is added to the rear. When an element is removed from a queue, it is removed from the front. The two primary queue operations are *enqueuing* and *dequeuing*. To enqueue means to insert an element at the rear of a queue, and to dequeue means to remove an element from the front of a queue. There are several different algorithms for implementing these operations. We will begin by looking at the most simple.

Suppose we have an empty static integer queue that is capable of holding a maximum of three values. With that queue we execute the following enqueue operations.

```
enqueue(3);
enqueue(6);
enqueue(9);
```

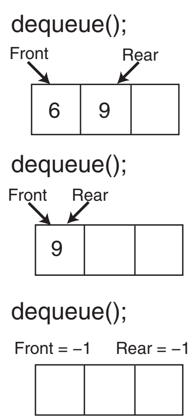
Figure 18-9 illustrates the state of the queue after each of these enqueue operations.

Figure 18-9



Notice in this example that the front index (which is a variable holding a subscript or perhaps a pointer) always references the same physical element. The rear index moves forward in the array as items are enqueued. Now let's see how dequeue operations are performed. Figure 18-10 illustrates the state of the queue after each of three consecutive dequeue operations.

Figure 18-10



In the dequeuing operation, the element at the front of the queue is removed. This is done by moving all the elements after it forward by one position. After the first dequeue operation, the value 3 is removed from the queue and the value 6 is at the front. After the second dequeue operation, the value 6 is removed and the value 9 is at the front. Notice that when only one value is stored in the queue, that value is at both the front and the rear.

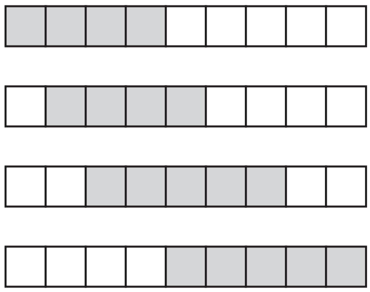
When the last dequeue operation is performed in Figure 18-10, the queue is empty. An empty queue can be signified by setting both front and rear indices to −1.

The problem with this algorithm is its inefficiency. Each time an item is dequeued, the remaining items in the queue are copied forward to their neighboring element. The more items there are in the queue, the longer each successive dequeue operation will take.

Here is one way to overcome the problem: Make both the front and rear indices move in the array. As before, when an item is enqueued, the rear index is moved to make room for it. But in this design, when an item is dequeued, the front index moves by one element toward the rear of the queue. This logically removes the front item from the queue and eliminates the need to copy the remaining items to their neighboring elements.

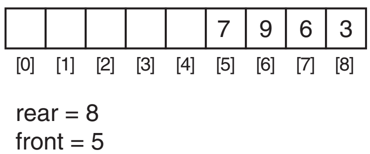
With this approach, as items are added and removed, the queue gradually "crawls" toward the end of the array. This is illustrated in Figure 18-11. The shaded squares represent the queue elements (between the front and rear).

Figure 18-11



The problem with this approach is that the rear index cannot move beyond the last element in the array. The solution is to think of the array as circular instead of linear. When an item moves past the end of a circular array, it simply "wraps around" to the beginning. For example, consider the queue depicted in Figure 18-12.

Figure 18-12



The value 3 is at the rear of the queue, and the value 7 is at the front of the queue. Now, suppose an enqueue operation is performed, inserting the value 4 into the queue. Figure 18-13 shows how the rear of the queue wraps around to the beginning of the array.

Figure 18-13

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
4					7	9	6	3

rear = 0
front = 5

So, what is the code for wrapping the rear marker around to the opposite end of the array? One straightforward approach is to use an `if` statement such as

```
if (rear == queueSize - 1)
   rear = 0;
else
   rear++;
```

Another approach is with modular arithmetic:

```
rear = (rear + 1) % queueSize;
```

This statement uses the `%` operator to adjust the value in `rear` to the proper position. Although this approach appears more elegant, the choice of which code to use is yours.

Detecting Full and Empty Queues with Circular Arrays

One problem with the circular array algorithm is that, because both the front and rear indices move through the array, detecting whether the queue is full or empty is a challenge. When the rear index and the front index reference the same element, does it indicate that only one item is in the queue, or that the queue is full? A number of approaches are commonly taken, two of which are listed below.

- When moving the rear index backward, always leave one element empty between it and the front index. The queue is full when the rear index is within two positions of the front index.
- Use a counter variable to keep a total of the number of items in the queue.

Because it might be helpful to keep a count of items in the queue anyway, we will use the second method in our implementation.

A Static Queue Class

The declaration of the `IntQueue` class is as follows:

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
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
```

Notice that in addition to the operations discussed in this section, the class also declares a member function named `clear`. This function clears the queue by resetting the `front` and `rear` indices and setting the `numItems` member to 0. The member function definitions are listed below.

Contents of `IntQueue.cpp`

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

//*****************************************************************
// This constructor creates an empty queue of a specified size. *
//*****************************************************************

IntQueue::IntQueue(int s)
{
    queueArray = new int[s];
    queueSize = s;
    front = -1;
    rear = -1;
    numItems = 0;
}
```

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

IntQueue::IntQueue(const IntQueue &obj)
{
    // Allocate the queue array.
    queueArray = new int[obj.queueSize];

    // Copy the other object's attributes.
    queueSize = obj.queueSize;
    front = obj.front;
    rear = obj.rear;
    numItems = obj.numItems;

    // Copy the other object's queue array.
    for (int count = 0; count < obj.queueSize; count++)
        queueArray[count] = obj.queueArray[count];
}

//*********************************************************
// Destructor                                             *
//*********************************************************

IntQueue::~IntQueue()
{
    delete [] queueArray;
}

//*********************************************************
// Function enqueue inserts a value at the rear of the queue. *
//*********************************************************

void IntQueue::enqueue(int num)
{
    if (isFull())
        cout << "The queue is full.\n";
    else
    {
        // Calculate the new rear position
        rear = (rear + 1) % queueSize;
        // Insert new item
        queueArray[rear] = num;
        // Update item count
        numItems++;
    }
}

//*********************************************************
// Function dequeue removes the value at the front of the queue *
// and copies t into num.                                 *
//*********************************************************

```

```
void IntQueue::dequeue(int &num)
{
    if (isEmpty())
       cout << "The queue is empty.\n";
    else
    {
         // Move front
         front = (front + 1) % queueSize;
         // Retrieve the front item
         num = queueArray[front];
         // Update item count
         numItems--;
    }
}

//*************************************************************
// isEmpty returns true if the queue is empty, otherwise false. *
//*************************************************************

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

    if (numItems)
         status = false;
    else
         status = true;

    return status;
}

//*************************************************************
// isFull returns true if the queue is full, otherwise false. *
//*************************************************************

bool IntQueue::isFull() const
{
    bool status;

    if (numItems < queueSize)
         status = false;
    else
         status = true;

    return status;
}

//*************************************************************
// clear sets the front and rear indices, and sets numItems to 0. *
//*************************************************************

```

```
void IntQueue::clear()
{
    front = queueSize - 1;
    rear = queueSize - 1;
    numItems = 0;
}
```

Program 18-7 is a driver that demonstrates the `IntQueue` class.

Program 18-7

```
// This program demonstrates the IntQueue class.
#include <iostream>
#include "IntQueue.h"
using namespace std;

int main()
{
    const int MAX_VALUES = 5;  // Max number of values

    // Create an IntQueue to hold the values.
    IntQueue iQueue(MAX_VALUES);

    // Enqueue a series of items.
    cout << "Enqueuing " << MAX_VALUES << " items...\n";
    for (int x = 0; x < MAX_VALUES; x++)
        iQueue.enqueue(x);

    // Attempt to enqueue just one more item.
    cout << "Now attempting to enqueue again...\n";
    iQueue.enqueue(MAX_VALUES);

    // Dequeue and retrieve all items in the queue
    cout << "The values in the queue were:\n";
    while (!iQueue.isEmpty())
    {
        int value;
        iQueue.dequeue(value);
        cout << value << endl;
    }
    return 0;
}
```

Program Output

```
Enqueuing 5 items...
Now attempting to enqueue again...
The queue is full.
The values in the queue were:
0
1
2
3
4
```

A Static Queue Template

The queue class shown previously works only with integers. A queue template can be easily designed to work with any data type, as shown by the following example:

Contents of `Queue.h`

```
#ifndef QUEUE_H
#define QUEUE_H
#include <iostream>
using namespace std;

// Stack template
template <class T>
class Queue
{
private:
    T *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
public:
    // Constructor
    Queue(int);

    // Copy constructor
    Queue(const Queue &);

    // Destructor
    ~Queue();

    // Queue operations
    void enqueue(T);
    void dequeue(T &);
    bool isEmpty() const;
    bool isFull() const;
    void clear();
};

//***************************************************************
// This constructor creates an empty queue of a specified size. *
//***************************************************************
template <class T>
Queue<T>::Queue(int s)
{
    queueArray = new T[s];
    queueSize = s;
    front = -1;
    rear = -1;
    numItems = 0;
}

```

```
//********************************************************
// Copy constructor                                      *
//********************************************************
template <class T>
Queue<T>::Queue(const Queue &obj)
{
    // Allocate the queue array.
    queueArray = new T[obj.queueSize];

    // Copy the other object's attributes.
    queueSize = obj.queueSize;
    front = obj.front;
    rear = obj.rear;
    numItems = obj.numItems;

    // Copy the other object's queue array.
    for (int count = 0; count < obj.queueSize; count++)
        queueArray[count] = obj.queueArray[count];
}

//********************************************************
// Destructor                                            *
//********************************************************
template <class T>
Queue<T>::~Queue()
{
    delete [] queueArray;
}

//********************************************************
// Function enqueue inserts a value at the rear of the queue. *
//********************************************************
template <class T>
void Queue<T>::enqueue(T item)
{
    if (isFull())
        cout << "The queue is full.\n";
    else
    {
        // Calculate the new rear position
        rear = (rear + 1) % queueSize;
        // Insert new item
        queueArray[rear] = item;
        // Update item count
        numItems++;
    }
}

//********************************************************
// Function dequeue removes the value at the front of the queue *
// and copies t into num.                                *
//********************************************************
template <class T>
void Queue<T>::dequeue(T &item)
```

```
{
    if (isEmpty())
       cout << "The queue is empty.\n";
    else
    {
        // Move front
        front = (front + 1) % queueSize;
        // Retrieve the front item
        item = queueArray[front];
        // Update item count
        numItems--;
    }
}

//*************************************************************
// isEmpty returns true if the queue is empty, otherwise false. *
//*************************************************************
template <class T>
bool Queue<T>::isEmpty() const
{
    bool status;

    if (numItems)
       status = false;
    else
       status = true;

    return status;
}

//*************************************************************
// isFull returns true if the queue is full, otherwise false. *
//*************************************************************
template <class T>
bool Queue<T>::isFull() const
{
    bool status;

    if (numItems < queueSize)
        status = false;
    else
        status = true;

    return status;
}

//*************************************************************
// clear sets the front and rear indices, and sets numItems to 0. *
//*************************************************************
template <class T>
void Queue<T>::clear()
{
    front = queueSize - 1;
    rear = queueSize - 1;
    numItems = 0;
}
#endif
```

Program 18-8 demonstrates the `Queue` template. It creates a queue that can hold strings, and then prompts the user to enter a series of names that are enqueued. The program then dequeues all of the names and displays them.

Program 18-8

```
// This program demonstrates the Queue template.
#include <iostream>
#include <string>
#include "Queue.h"
using namespace std;

const int QUEUE_SIZE = 5;

int main()
{
    string name;

    // Create a Queue.
    Queue<string> queue(QUEUE_SIZE);

    // Enqueue some names.
    for (int count = 0; count < QUEUE_SIZE; count++)
    {
        cout << "Enter a name: ";
        getline(cin, name);
        queue.enqueue(name);
    }

    // Dequeue the names and display them.
    cout << "\nHere are the names you entered:\n";
    for (int count = 0; count < QUEUE_SIZE; count++)
    {
        queue.dequeue(name);
        cout << name << endl;
    }
    return 0;
}
```

Program Output with Example Input Shown in Bold

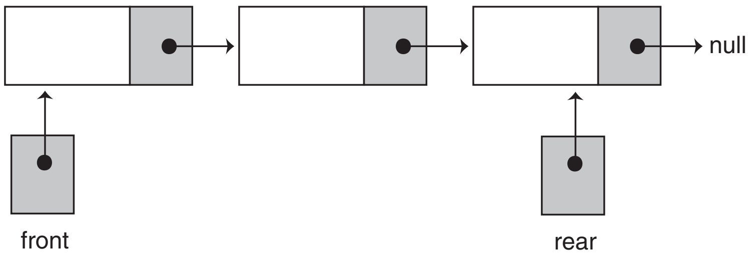
```
Enter a name: Chris [Enter]
Enter a name: Kathryn [Enter]
Enter a name: Alfredo [Enter]
Enter a name: Lori [Enter]
Enter a name: Kelly [Enter]

Here are the names you entered:
Chris
Kathryn
Alfredo
Lori
Kelly
```

18.5 Dynamic Queues

CONCEPT: A queue may be implemented as a linked list and expand or shrink with each enqueue or dequeue operation.

Dynamic queues, which are built around linked lists, are much more intuitive to understand than static queues. A dynamic queue starts as an empty linked list. With the first enqueue operation, a node is added, which is pointed to by the `front` and `rear` pointers. As each new item is added to the queue, a new node is added to the rear of the list, and the `rear` pointer is updated to point to the new node. As each item is dequeued, the node pointed to by the `front` pointer is deleted, and `front` is made to point to the next node in the list. Figure 18-14 shows the structure of a dynamic queue.

Figure 18-14



A dynamic integer queue class is listed here.

Contents of `DynIntQueue.h`

```
#ifndef DYNINTQUEUE_H
#define DYNINTQUEUE_H

class DynIntQueue
{
private:
     // Structure for the queue nodes
     struct QueueNode
     {
          int value;        // Value in a node
          QueueNode *next; // Pointer to the next node
     };

     QueueNode *front; // The front of the queue
     QueueNode *rear;  // The rear of the queue
     int numItems;     // Number of items in the queue
public:
     // Constructor
     DynIntQueue();

     // Destructor
     ~DynIntQueue();
```

```

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

Contents of `DynIntQueue.cpp`

```
#include <iostream>
#include "DynIntQueue.h"
using namespace std;

//*******************************************
// The constructor creates an empty queue.  *
//*******************************************

DynIntQueue::DynIntQueue()
{
    front = nullptr;
    rear = nullptr;
    numItems = 0;
}

//*******************************************
// Destructor                               *
//*******************************************

DynIntQueue::~DynIntQueue()
{
    clear();
}

//********************************************
// Function enqueue inserts the value in num *
// at the rear of the queue.                 *
//********************************************

void DynIntQueue::enqueue(int num)
{
    QueueNode *newNode = nullptr;

    // Create a new node and store num there.
    newNode = new QueueNode;
    newNode->value = num;
    newNode->next = nullptr;

    // Adjust front and rear as necessary.
    if (isEmpty())
    {
        front = newNode;
        rear = newNode;
    }
```

```
    else
    {
        rear->next = newNode;
        rear = newNode;
    }

    // Update numItems.
    numItems++;
}

//**********************************************
// Function dequeue removes the value at the   *
// front of the queue, and copies it into num. *
//**********************************************

void DynIntQueue::dequeue(int &num)
{
    QueueNode *temp = nullptr;

    if (isEmpty())
    {
        cout << "The queue is empty.\n";
    }
    else
    {
        // Save the front node value in num.
        num = front->value;

        // Remove the front node and delete it.
        temp = front;
        front = front->next;
        delete temp;

        // Update numItems.
        numItems--;
    }
}

//*********************************************
// Function isEmpty returns true if the queue *
// is empty, and false otherwise.             *
//*********************************************

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

    if (numItems > 0)
        status = false;
    else
        status = true;
    return status;
}

```

```
//*********************************************
// Function clear dequeues all the elements *
// in the queue.                            *
//*********************************************

void DynIntQueue::clear()
{
    int value;  // Dummy variable for dequeue

    while(!isEmpty())
        dequeue(value);
}
```

Program 18-9 is a driver that demonstrates the `DynIntQueue` class.

Program 18-9

```
// This program demonstrates the DynIntQueue class.
#include <iostream>
#include "DynIntQueue.h"
using namespace std;

int main()
{
    const int MAX_VALUES = 5;

    // Create a DynIntQueue object.
    DynIntQueue iQueue;

    // Enqueue a series of numbers.
    cout << "Enqueuing " << MAX_VALUES << " items...\n";
    for (int x = 0; x < 5; x++)
        iQueue.enqueue(x);

    // Dequeue and retrieve all numbers in the queue
    cout << "The values in the queue were:\n";
    while (!iQueue.isEmpty())
    {
        int value;
        iQueue.dequeue(value);
        cout << value << endl;
    }
    return 0;
}
```

Program Output

```
Enqueuing 5 items...
The values in the queue were:
0
1
2
3
4
```

A Dynamic Queue Template

The dynamic queue class shown previously in this chapter works only with integers. A dynamic queue template can be easily designed to work with any data type, as shown by the following example:

Contents of `DynamicQueue.h`

```
#ifndef DYNAMICQUEUE_H
#define DYNAMICQUEUE_H
#include <iostream>
using namespace std;

// DynamicQueue template
template <class T>
class DynamicQueue
{
private:
    // Structure for the queue nodes
    struct QueueNode
    {
        T value;         // Value in a node
        QueueNode *next; // Pointer to the next node
    };

    QueueNode *front;   // The front of the queue
    QueueNode *rear;    // The rear of the queue
    int numItems;       // Number of items in the queue
public:
    // Constructor
    DynamicQueue();

    // Destructor
    ~DynamicQueue();

    // Queue operations
    void enqueue(T);
    void dequeue(T &);
    bool isEmpty() const;
    bool isFull() const;
    void clear();
};

//********************************************
// The constructor creates an empty queue.   *
//********************************************
template <class T>
DynamicQueue<T>::DynamicQueue()
{
    front = nullptr;
    rear = nullptr;
    numItems = 0;
}

```

```
//********************************************
// Destructor                                *
//********************************************
template <class T>
DynamicQueue<T>::~DynamicQueue()
{
    clear();
}

//********************************************
// Function enqueue inserts the value in num *
// at the rear of the queue.                 *
//********************************************
template <class T>
void DynamicQueue<T>::enqueue(T item)
{
    QueueNode *newNode = nullptr;

    // Create a new node and store num there.
    newNode = new QueueNode;
    newNode->value = item;
    newNode->next = nullptr;

    // Adjust front and rear as necessary.
    if (isEmpty())
    {
        front = newNode;
        rear = newNode;
    }
    else
    {
        rear->next = newNode;
        rear = newNode;
    }

    // Update numItems.
    numItems++;
}

//**********************************************
// Function dequeue removes the value at the   *
// front of the queue, and copies it into num. *
//**********************************************
template <class T>
void DynamicQueue<T>::dequeue(T &item)
{
    QueueNode *temp = nullptr;

    if (isEmpty())
    {
        cout << "The queue is empty.\n";
    }
    else
    {
        // Save the front node value in num.
        item = front->value;
```

```

        // Remove the front node and delete it.
        temp = front;
        front = front->next;
        delete temp;

        // Update numItems.
        numItems--;
    }
}

//**********************************************
// Function isEmpty returns true if the queue *
// is empty, and false otherwise.             *
//**********************************************
template <class T>
bool DynamicQueue<T>::isEmpty() const
{
    bool status;

    if (numItems > 0)
        status = false;
    else
        status = true;
    return status;
}

//********************************************
// Function clear dequeues all the elements *
// in the queue.                            *
//********************************************
template <class T>
void DynamicQueue<T>::clear()
{
    T value;  // Dummy variable for dequeue

    while(!isEmpty())
        dequeue(value);
}
#endif
```

Program 18-10 demonstrates the DynamicQueue template. This program is a modification of Program 18-8. It creates a queue that can hold strings and then prompts the user to enter a series of names that are enqueued. The program then dequeues all of the names and displays them. (The program's output is the same as that of Program 18-8.)

Program 18-10

```
// This program demonstrates the DynamicQueue template.
#include <iostream>
#include <string>
#include "DynamicQueue.h"
using namespace std;

```

(program continues)

Program 18-10 *(continued)*

```
const int QUEUE_SIZE = 5;

int main()
{
    string name;

    // Create a Queue.
    DynamicQueue<string> queue;

    // Enqueue some names.
    for (int count = 0; count < QUEUE_SIZE; count++)
    {
        cout << "Enter a name: ";
        getline(cin, name);
        queue.enqueue(name);
    }

    // Dequeue the names and display them.
    cout << "\nHere are the names you entered:\n";
    for (int count = 0; count < QUEUE_SIZE; count++)
    {
        queue.dequeue(name);
        cout << name << endl;
    }
    return 0;
}
```

Program Output

(Same as Program 18-8's output.)

18.6 The STL deque and queue Containers

CONCEPT: The Standard Template Library provides two containers, `deque` and `queue`, for implementing queue-like data structures.

In this section we will examine two ADTs offered by the Standard Template Library: `deque` and `queue`. A `deque` (pronounced "deck" or "deek") is a double-ended queue. It is similar to a `vector`, but allows efficient access to values at both the front and the rear. The `queue` ADT is like the `stack` ADT: It is actually a container adapter.

The deque Container

Think of the `deque` container as a `vector` that provides quick access to the element at its front as well as at the back. (Like `vector`, `deque` also provides access to its elements with the `[]` operator.)

Programs that use the deque ADT must include the deque header. Because we are concentrating on its queue-like characteristics, we will focus our attention on the push_back, pop_front, and front member functions. Table 18-4 describes them.

Table 18-4

Member Function	Examples and Description
push_back	`iDeque.push_back();` Accepts as an argument a value to be inserted into the deque. The argument is inserted after the last element. (Pushed onto the back of the deque.)
pop_front	`iDeque.pop_front();` Removes the first element of the deque.
front	`cout << iDeque.front() << endl;` front returns a reference to the first element of the deque.

Program 18-11 demonstrates the deque container.

Program 18-11

```
// 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;
}
```

(program output continues)

Program 18-11 *(continued)*

Program Output

```
I will now enqueue items...
Pushing 2
Pushing 4
Pushing 6
I will now dequeue items...
Popping 2
Popping 4
Popping 6
```

The queue Container Adapter

VideoNote
Storing Objects in an STL queue

The queue container adapter can be built upon vectors, lists, or deques. By default, it uses deque as its base.

The insertion and removal operations supported by queue are the same as those supported by the stack ADT: push, pop, and top. There are differences in their behavior, however. The queue version of push always inserts an element at the rear of the queue. The queue version of pop always removes an element from the structure's front. The top function returns the value of the element at the front of the queue.

Program 18-12 demonstrates a queue. Because the definition of the queue does not specify which type of container is being adapted, the queue will be built on a deque.

Program 18-12

```
// 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;
}
```

Program Output

```
I will now enqueue items...
Pushing 2
Pushing 4
Pushing 6
I will now dequeue items...
Popping 2
Popping 4
Popping 6
```

Review Questions and Exercises

Short Answer

1. What does LIFO mean?
2. What element is always retrieved from a stack?
3. What is the difference between a static stack and a dynamic stack?
4. Describe two operations that all stacks perform.
5. Describe two operations that static stacks must perform.
6. The STL `stack` is considered a container adapter. What does that mean?
7. What types may the STL `stack` be based on? By default, what type is an STL `stack` based on?
8. What does FIFO mean?
9. When an element is added to a queue, where is it added?
10. When an element is removed from a queue, where is it removed from?
11. Describe two operations that all queues perform.
12. What two queue-like containers does the STL offer?

Fill-in-the-Blank

13. The __________ element saved onto a stack is the first one retrieved.
14. The two primary stack operations are __________ and __________.
15. __________ stacks and queues are implemented as arrays.
16. __________ stacks and queues are implemented as linked lists.
17. The STL stack container is an adapter for the __________, __________, and __________ STL containers.
18. The __________ element saved in a queue is the first one retrieved.

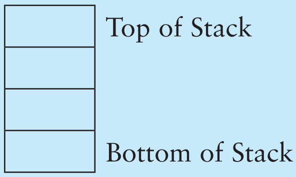
19. The two primary queue operations are __________ and __________.
20. The two ADTs in the Standard Template Library that exhibit queue-like behavior are __________ and __________.
21. The queue ADT, by default, adapts the __________ container.

Algorithm Workbench

22. Suppose the following operations are performed on an empty stack:

```
push(0);
push(9);
push(12);
push(1);
```

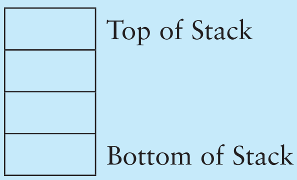
Insert numbers in the following diagram to show what will be stored in the static stack after the operations above have executed.



23. Suppose the following operations are performed on an empty stack:

```
push(8);
push(7);
pop();
push(19);
push(21);
pop();
```

Insert numbers in the following diagram to show what will be stored in the static stack after the operations above have executed.



24. Suppose the following operations are performed on an empty queue:

```
enqueue(5);
enqueue(7);
enqueue(9);
enqueue(12);
```

Insert numbers in the following diagram to show what will be stored in the static stack after the operations above have executed.

front rear

25. Suppose the following operations are performed on an empty queue:

```
enqueue(5);
enqueue(7);
dequeue();
enqueue(9);
enqueue(12);
dequeue();
enqueue(10);
```

Insert numbers in the following diagram to show what will be stored in the static stack after the operations above have executed.

front rear

26. What problem is overcome by using a circular array for a static queue?
27. Write two different code segments that may be used to wrap an index back around to the beginning of an array when it moves past the end of the array. Use an `if/else` statement in one segment and modular arithmetic in the other.

True or False

28. T F A static stack or queue is built around an array.
29. T F The size of a dynamic stack or queue must be known in advance.
30. T F The push operation inserts an element at the end of a stack.
31. T F The pop operation retrieves an element from the top of a stack.
32. T F The STL stack container's pop operation does not retrieve the top element of the stack, it just removes it.

Programming Challenges

1. Static Stack Template

Write your own version of a class template that will create a static stack of any data type. Demonstrate the class with a driver program.

2. Dynamic Stack Template

Write your own version of a class template that will create a dynamic stack of any data type. Demonstrate the class with a driver program.

3. Static Queue Template

Write your own version of a class template that will create a static queue of any data type. Demonstrate the class with a driver program.

4. Dynamic Queue Template

Write your own version of a class template that will create a dynamic queue of any data type. Demonstrate the class with a driver program.

5. Error Testing

The `DynIntStack` and `DynIntQueue` classes shown in this chapter are abstract data types using a dynamic stack and dynamic queue, respectively. The classes do not currently test for memory allocation errors. Modify the classes so they determine whether new nodes cannot be created by handling the `bad_alloc` exception.

NOTE: If you have already done Programming Challenges 2 and 4, modify the templates you created.

6. Dynamic String Stack

Design a class that stores strings on a dynamic stack. The strings should not be fixed in length. Demonstrate the class with a driver program.

7. Dynamic MathStack

The `MathStack` class shown in this chapter only has two member functions: `add` and `sub`. Write the following additional member functions:

Function	Description
`mult`	Pops the top two values off the stack, multiplies them, and pushes their product onto the stack.
`div`	Pops the top two values off the stack, divides the second value by the first, and pushes the quotient onto the stack.
`addAll`	Pops all values off the stack, adds them, and pushes their sum onto the stack.
`multAll`	Pops all values off the stack, multiplies them, and pushes their product onto the stack.

Demonstrate the class with a driver program.

8. Dynamic MathStack Template

Currently the `MathStack` class is derived from the `IntStack` class. Modify it so it is a template, derived from the template you created in Programming Challenge 2.

9. File Reverser

Write a program that opens a text file and reads its contents into a stack of characters. The program should then pop the characters from the stack and save them in a second text file. The order of the characters saved in the second file should be the reverse of their order in the first file.

10. File Filter

Write a program that opens a text file and reads its contents into a queue of characters. The program should then dequeue each character, convert it to uppercase, and store it in a second file.

11. File Compare

VideoNote
Solving the File Compare Problem

Write a program that opens two text files and reads their contents into two separate queues. The program should then determine whether the files are identical by comparing the characters in the queues. When two nonidentical characters are encountered, the program should display a message indicating that the files are not the same. If both queues contain the same set of characters, a message should be displayed indicating that the files are identical.

12. Inventory Bin Stack

Design an inventory class that stores the following members:

`serialNum:`	An integer that holds a part's serial number.
`manufactDate:`	A member that holds the date the part was manufactured.
`lotNum:`	An integer that holds the part's lot number.

The class should have appropriate member functions for storing data into, and retrieving data from, these members.

Next, design a stack class that can hold objects of the class described above. If you wish, you may use the template you designed in Programming Challenge 1 or 2.

Last, design a program that uses the stack class described above. The program should have a loop that asks the user if he or she wishes to add a part to inventory, or take a part from inventory. The loop should repeat until the user is finished.

If the user wishes to add a part to inventory, the program should ask for the serial number, date of manufacture, and lot number. The data should be stored in an inventory object, and pushed onto the stack.

If the user wishes to take a part from inventory, the program should pop the top-most part from the stack and display the contents of its member variables.

When the user finishes the program, it should display the contents of the member values of all the objects that remain on the stack.

13. Inventory Bin Queue

Modify the program you wrote for Programming Challenge 12 so it uses a queue instead of a stack. Compare the order in which the parts are removed from the bin for each program.

14. Balanced Parentheses

A string of characters has balanced parentheses if each right parenthesis occurring in the string is matched with a preceding left parenthesis, in the same way that each right brace in a C++ program is matched with a preceding left brace. Write a program that uses a stack to determine whether a string entered at the keyboard has balanced parentheses.

15. Balanced Multiple Delimiters

A string may use more than one type of delimiter to bracket information into "blocks." For example, A string may use braces { }, parentheses ( ), and brackets [ ] as delimiters. A string is properly delimited if each right delimiter is matched with a preceding left delimiter of the same type in such a way that either the resulting blocks of information are disjoint, or one of them is completely nested within the other. Write a program that uses a single stack to check whether a string containing braces, parentheses, and brackets is properly delimited.