// Example illustrating member functions
// ECE3090
// George F. Riley, Georgia Tech, Spring 2009

#include <iostream>
using namespace std; // We will discuss namespaces later

// Define an arbitrary fixed size limit for queues
#define MAX_QUEUE_SIZE 100

class LIFOQueue
{// Define a last-in-first-out queue with fixed size
 public:
 LIFOQueue(); // Constructor
 // Define member functions to manage the elements of the queue
 // Note the use of the "const" keyword after some of the member function
 // declarations. This indicates that the member function promises
 // not to change any of the member variables.
   bool Enque(int); // Enque a new integer element
   int Deque(); // Remove an element
   unsigned Length() const; // Return the number of elements in the queue
   bool Empty() const; // Returns true if the queue is empty
   void Clear(); // Remove all elements in the queue
   void Print() const; // Print the queue contents
 // Does it make sense to define an addition operator for LIFOQueue objects?
 // We will define one using "member function operators"
 // Note we don’t need the "left hand side" argument. Why?
   LIFOQueue operator+(const LIFOQueue& rhs);
 // Also add an "indexing operator". See discussion in the implementation
 // also note the return type. Will discuss in class
   int& operator[](unsigned);

 private:
 // Note the use of "private:" above. This is discussed in the
 // comments in the main program.
 // Define the variables needed for the queue
   int queue[MAX_QUEUE_SIZE]; // The actual data in the queue
   unsigned length; // Number of elements in the queue
};

// Implement the member functions for the LIFOQueue object
// Constructor
LIFOQueue::LIFOQueue()
 : length(0) // Set length to zero
 { // We can leave the "queue" variable uninitialized. Why?
 }

bool LIFOQueue::Enque(int newElement)
{
 // First check if queue is full.
 if (length == MAX_QUEUE_SIZE) return false; // Cannot enqueue
 // Ok to enqueue this new value
 queue[length++] = newElement;
 // The line below (commented out) is equivalent

 Program member-functions.cc
/this->queue[this->length++] = newElement;

// Note that when we refer to a variable in a member function, the compiler automatically assumes we mean the member variable if there is one by that name. Also notice that all member functions always define a pointer "this", that points to an object of the correct type (LIFOQueue in this case).
return true;
}

int LIFOQueue::Deque()
{
    // First check if queue is empty.
    if (length == 0)
        { // It’s hard to decide what to do here. "This should never happen".
            cout << "OOps, Deque called on empty queue" << endl;
            // For this example we will just exit the program.
            exit(0);
        }
    return queue[--length];
    // As above, we are refering to member variables by name. eg. "queue" and "length". Also as before we could have said:
    return this->queue[--(this->length)];
}

unsigned LIFOQueue::Length() const
{ // Return the length (number of elements) in the queue
    return length;
}

bool LIFOQueue::Empty() const
{ // Return true if empty (length is zero)
    return length == 0;
}

void LIFOQueue::Clear()
{ // Remove all elements in the queue
    // Here we would be tempted to write:
    // while (!Empty()) { Deque();}
    // Does the below do exactly the same thing?
    // Which is best?
    length = 0;
}

void LIFOQueue::Print() const
{ // Notice that in a member function, we can refer to other member functions in the same class by name. eg. "Empty()" below.
    if (Empty())
        { cout << "Queue is empty" << endl;
            return;
        }
    for (int i = 0; i < Length(); ++i)
        { cout << "Element " << i << " is " << queue[i] << endl;
        }
// Implement the addition operator
LIFOQueue LIFOQueue::operator+(const LIFOQueue& rhs)
{ // Notice the left hand side argument is "this"
    // Make a new LIFOQueue and add all elements in lhs + rhs
    LIFOQueue returnValue;
    // Add the left-hand-side values (from "this")
    for (int i = 0; i < Length(); ++i)
        { // Add all elements from "this"
            returnValue.Enque(queue[i]);
        }
    // Add the right-hand-side values
    for (int i = 0; i < rhs.Length(); ++i)
        { // Add all elements from "rhs"
            returnValue.Enque(rhs.queue[i]);
        }
    return returnValue;
}

// Implement the indexing operator
int& LIFOQueue::operator[](unsigned index)
{ // return the value at the "index" element
    // The length check below, uncomment if desired
    //if (index >= length) exit(); // !! Not clear what to do here
    return queue[index];
}

int main()
{ // Illustrate the use of the member functions
    LIFOQueue q1;
    // Add elements until the queue fills up
    int i = 0;
    while(true)
    {
        if (!q1.Enque(i++)) break; // If q1.Enque() is false, queue is full
    }
    // Note the syntax above and below. variable name DOT function name
    // It calls the "Print()" function with "this" as "address of q1"
    q1.Print();
    // Remove elements one at a time
    while (!q1.Empty())
    {
        int k = q1.Deque();
        cout << "Dequeued element " << k << endl;
    }
    // Add element 2 ten times
    for (int i = 0; i < 10; ++i)
    {
        q1.Enque(2);
    }
    // Define a second queue
    LIFOQueue q2;
    // Add element 3 ten times to q2
    Program member-functions.cc (continued)
for (int i = 0; i < 10; ++i)
{
    q2.Enqueue(3);
}

// Use the addition operator
LIFOQueue q3 = q1 + q2;
// The above called the addition operator with "address of q1" as the
// "this", and q2 as the "right-hand-side"
cout << "Printing q3 after addition" << endl;
q3.Print();
// Add again in reverse order
LIFOQueue q4 = q2 + q1;
cout << "Printing q4 after addition" << endl;
q4.Print();
// So, what about the addition operator? Did our implementation make sense?
// What might have been a better choice?
// Call the copy constructor
LIFOQueue q5(q4);
// We did not define a copy constructor, so what happened above?
cout << "Printing q5 after copy constructor" << endl;
q5.Print();
// Illustrate the indexing operator
LIFOQueue q6;
// First populate the queue with some values
for (int i = 0; i < 10; ++i)
{
    q6.Enqueue(i);
}
// Now use indexing operator to retrieve them
for (unsigned i = 0; i < q6.Length(); ++i)
{
    int v = q6[i]; // Return the "i'th" element of q6
    cout << "Element " << i << " is " << v << endl;
}
// Now use indexing operator on LEFT HAND SIDE!
for (unsigned i = 0; i < q6.Length(); ++i)
{
    q6[i] = q6[i] * 10;
}
// And print it out
cout << "After LHS indexing loop" << endl;
for (unsigned i = 0; i < q6.Length(); ++i)
{
    int v = q6[i]; // Return the "i'th" element of q6
    cout << "Element " << i << " is " << v << endl;
}
// The fact that member variables "length" and "queue" are declared
// "private" means that they cannot be referenced except in member
// functions. In other words, the below (commented out) would not
// compile if it was uncommented.
// int q1Length = q1.length;
// Even though q1.length definitely exists, since it is private it

Program member-functions.cc (continued)
// cannot be referenced here. However, we defined a member function "Length() const" that returned the value of length. Why is this better?