Lists, Stacks, and Queues

A linear structure is an ordered (not necessarily sorted) arrangement of elements.

There are three common types of linear structures:

list	random insertion and deletion
stack	insertion and deletion only at one end, called the top
queue	insertion only at one end and deletion only at the other end

The underlying organization of a linear structure may be implemented in either of two ways:

contiguous	- constant cost random access but linear cost random insert/delete - storage overhead is unused portion (usually an array)
linked	- linear cost random access but constant cost random insert/delete - storage overhead consists of pointers

Data Structure as a Container

This chapter presents sample implementations of a array-based stack class and a link-based queue template.

The primary goals of these implementation are:

- to provide a proper separation of functionality.
- to design the structure to serve as a container; i.e., the structure should be able to store data elements of any type.

In the case of the stack class, this will be achieved by using a `typedef`.

Warning: the data structure implementations given in these notes are intended for instructional purposes. They contain a number of known flaws, and perhaps some unknown flaws as well. *Caveat emptor*.

Linear Structure as an ADT

From the perspective of an ADT, whether a linear structure is based upon an array or a linked list is unimportant. The client interface may be identical in both cases:

```
enum ErrorType {NO_ERROR, STACK_UNDERFLOW, STACK_OVERFLOW};

class Stack {
private:
   int  Capacity;             // current stack array size
   int  Top;                  // first available index in stack array
   Item* Stk;                 // stack array (allocated dynamically)
   ErrorType errorType;       // indicates last error detected
public:
   Stack(int InitSize);        // construct new stack with Capacity InitSize
   Stack(const Stack& Source);            // copy constructor
   Stack& operator=(const Stack& Source); // assignment overload
   bool Push(Item toInsert);   // push toInsert on top of stack, increasing
                               //      stack array dimension if necessary
   Item Pop();                 // remove and return element at top of stack
   Item Peek();                // return copy of element at top of stack
   bool isEmpty() const;       // indicate whether stack is currently empty
   bool isFull() const;        // indicate whether stack is currently full
   ErrorType getError() const; // return error state
   ~Stack();                   // deallocate stack array
   void Display(ostream& Out) const; // display stack contents to stream
};
```


Independence of Interface

For a user of the `Stack` class, client code would be the same whether the underlying structure were an array or linked.

```
bool SearchStack(Item toFind, Stack S) {

   while ( !S.isEmpty() ) {
      if ( toFind == S.Pop() )
         return true;
   }
   return false;
}
```

Assumes that `Item` has an equality operator.

As a result, changes to the implementation of the class will not mandate changes to the client code (unless the public interface is modified).

©William D McQuain, July 2000

Stack Design Considerations

The `Stack` class has a few noteworthy features:

- The stack elements are of a generic type (`Item`) specified by the client. This could be better done with a template.
- A dynamic array is used to store the stack elements.
- That array is resized dynamically if it is full when `Push()` is called.
- Stack underflow and overflow are handled gracefully.
- An internal flag variable, of an enumerated type is used to record any errors that occur during stack operations.
- It is the client's responsibility to check the value of that flag.
- New-style C++ headers are used throughout, and so `new(nothrow)` is used in order to suppress possible exception throws if an allocation fails.
- Alternatively, Push() and Pop() could throw exceptions on an allocation failure and leave the client with the responsibility to catch and handle those exceptions.

©2000 McQuain WD

Stack Construction and Destruction


```
// Stack.cpp
#include "Stack.h"
#include <new>
using namespace std;
```

Use new-style headers.

```
Stack::Stack(int InitSize) {
   Capacity = InitSize;
   Top = 0;
   errorType = NO_ERROR;
   Stk = new(nothrow) Item[Capacity];
   if (Stk == NULL) Capacity = 0;
}
```

Suppress exception if allocation fails. But be sure to check for failure.

```
Stack::~Stack() {
   delete [] Stk;
}
```

Deallocate stack array.

©2000 McQuain WD

Stack Copy Constructor

Because a `Stack` object has dynamically allocated content, we must provide deep copy operations:

```
Stack::Stack(const Stack& Source) {

   Capacity  = Source.Capacity;
   Top       = Source.Top;
   LastError = Source.LastError;

   Stk = new(nothrow) Item[Capacity];
   if (Stk != NULL) {
      for (int Idx = 0; Idx < Top; Idx++)
         Stk[Idx] = Source.Stk[Idx];
   }
   else {
      Capacity = 0;
      Top      = 0;
   }
}
```

Assumes that `Item` has an appropriate assignment operator.

©2000 McQuain WD

Stack Assignment Overload


```
Stack& Stack::operator=(const Stack& Source) {

   if (this == &Source)
      return *this;

   delete [] Stk;

   Capacity  = Source.Capacity;
   Top       = Source.Top;
   LastError = Source.LastError;

   Stk = new(nothrow) Item[Capacity];
   if (Stk != NULL) {
      for (int Idx = 0; Idx < Top; Idx++)
         Stk[Idx] = Source.Stk[Idx];
   }
   else {
      Capacity = 0;
      Top      = 0;
   }
   return *this;
}
```

Test for self-assignment.

Avoid memory leak if target of assignment is already initialized.

Return `Stack` object.

©2000 McQuain WD

©William D McQuain, July 2000

Stack Push Operation

Linear Structures 9

The `Stack` object uses a automatically resizable array, rather than a fixed size array:

```
bool Stack::Push(Item toInsert) {
   if (Top == Capacity) {
      Item* tmpStk = new(nothrow) Item[2*Capacity];
      if (tmpStk == NULL) {
         LastError = STACK_OVERFLOW;
         return false;
      }
      for (int Idx = 0; Idx < Capacity; Idx++) {
         tmpStk[Idx] = Stk[Idx];
      }
      delete [] Stk;
      Stk = tmpStk;
      Capacity = 2*Capacity;
   }
   Stk[Top] = toInsert;
   Top++;
   return true;
}
```

If `Stack` array is full, enlarge it on the fly, if possible.

This makes the fact that the underlying structure is an array relatively invisible to the client.

©2000 McQuain WD

Stack Pop Operation

Linear Structures 10

The `Stack` `Pop()` operation must deal with stack underflow:

```
Item Stack::Pop() {
   if ( (Top > 0) && (Top <= Capacity) ) {
      LastError = NO_ERROR;
      Top--;
      return Stk[Top];
   }
   LastError = STACK_UNDERFLOW;
   return Item();
}
```

If `Stack` array is not empty, reset error flag, decrement position of `Top` element, and return the old Top element.

If `Stack` array is empty, record error and return default `Item` object.

This places a burden on the user to test the error state unless the default `Item` is easily recognizable as an invalid return value.

In some stack implementations, the pop operation will use a reference parameter for the stack element and return an indication of success or failure (a bool or enum type value). Here the traditional pop interface is used.

©2000 McQuain WD

Stack Reporters

Linear Structures 11

```
bool Stack::isEmpty() const {
   return (Top == 0);
}
```

Recall `Top` is index of first available cell.

```
bool Stack::isFull() const {
   if (Top < Capacity) return false;
   Item* tmpStack = new(nothrow) Item[2*Capacity];
   if (tmpStack == NULL) return true;
   delete [] tmpStack;
   return false;
}
```

If array is full, determine if it's possible to allocate a larger array.

```
ErrorType Stack::getError() const {
   return LastError;
}
```


©2000 McQuain WD

Stack Utility Functions

Linear Structures 12

```
Item Stack::Peek() {
   if ( (Top > 0) && (Top <= Capacity) ) {
      LastError = NO_ERROR;
      return Stk[Top-1];
   }
   LastError = STACK_UNDERFLOW;
   return 0;
}
```

It's often useful to be able to inspect the `Top` value without actually removing it.

```
void Stack::Display(ostream& Out) const {

   Out << "Capacity:" << setw(3)
       << Capacity << endl;
   Out << "Top:     " << setw(3)
       << Top << endl << endl;
   for (int Idx = Top-1; Idx >= 0; Idx--) {
      Out << setw(3) << Idx << ":  "
          << Stk[Idx] << endl;
   }
}
```

Whether to include a display function is somewhat problematic.

If not, some info used here cannot be displayed by a nonmember function.

If so, then `Item` objects must support stream insertion, or we must force them to implement a print function to call here.

©2000 McQuain WD

Some Observations

Reflecting on the given `Stack` implementation:

- The design assumes the elements are structured at least to the extent that default constructor syntax is supported.
- Effectively, the elements may be `struct` or `class` type variables. However, it is generally preferable use use a `class` type (unless a simple built in type suffices).

  `struct` types should be restricted to situations where member functions are unnecessary (Kruse/Ryba notwithstanding).

  If deep copy issues arise, or element comparisons need to be overloaded, then a `class` should be used.

  If the external use of the elements justifies having private data, then a `class` should be used.
- The assumptions identified in the discussion of the implementation should be clearly documented in a prefatory comment in the `Stack` class header file.

Use of a Template

The given `Stack` implementation is unsatisfactory for general use because of the mechanism chosen for mapping the element type into the implementation.

(Consider writing a simple program that required stacks holding two different types of data elements.)

In C++, the best solution to this problem is to make use of a template.

See the CS 2704 course notes and/or Stroustrup for an introductory discussion of templates.

We will consider a queue template for illustration… be aware that this implementation has been subjected to light testing, but is in no way guaranteed to be correct (much less optimal).

Template Issues

The syntactic differences between a simple class and a templated class are annoying, but have been covered in previous courses.

The primary logical difference relates to code organization. Because the compiler must generate a specific class from a template instantiation, the implementation of a template cannot be precompiled.

Rather than the usual organization into header and cpp files (such as `Stack.h` and `Stack.cpp` previously), there are three common code organizations for templates:

- place the full implementation in the header file
- place the implementation in a cpp file and `#include` that cpp file at the end of the header file (avoid circular inclusions here!)
- use the `export` mechanism (not covered here)

Encapsulation of Data

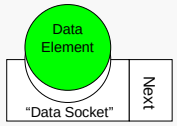
In order to make the data structure flexible, its implementation should neither know, nor care, about the specific properties of the elements to be stored. To achieve this with the queue template, we will make use of a three-level organization:

First, a `NodeT` template is used to encapsulate the low-level pointer operations.

Second, a `QueueT` template is used to encapsulate a list of `NodeT` objects.

Third, a `Foo` class is used to encapsulate the data and separate it from the pointers that define the list structure.

The basic view is that each node provides a data "socket" that is capable of accepting any type of data element:



Queue Template Interface

Linear Structures 17

```
// QueueT.h : a Linked Queue Template
#ifndef QUEUET_H     // conditional compilation directives
#define QUEUET_H

#include <iostream>  // for Display() member function
#include <iomanip>
using namespace std;

#include "NodeT.h"   // Node template (shown later)

enum QueueError {QUEUE_IS_FULL, QUEUE_IS_EMPTY, NO_ERROR};
                     // internal error flags, as in Stack class

template <class Foo> class QueueT {
private:
   NodeT<Foo>* Front;
   NodeT<Foo>* Rear;
   QueueError  LastError;

// . . . private section follows . . .
```


Queue Template Interface

Linear Structures 18

```
   // . . . continuing interface declaration.
public:
   QueueT();                    // create empty queue
   void Enqueue(Foo Item);      // insert at Rear
   Foo  Dequeue();              // remove from Front
   bool isEmpty() const;        // tests for empty queue and
   bool isFull() const;         //      full queue
   void Display(ostream& Out);  // display formatted contents
   QueueError getLastError() const; // return error flag
   ~QueueT();                   // destroy list of nodes
};

// . . . implementation follows in same file . . .
```


Queue Template Constructor/Destructor

Linear Structures 19

```
// . . . continuing implementation . . .

// Queue object is empty if Front and Rear are NULL.
template <class Foo> QueueT<Foo>::QueueT() {
   Front = Rear = NULL;
   LastError = NO_ERROR;
}

// Walk the node list deleting each node.
template <class Foo> QueueT<Foo>::~QueueT() {

   NodeT<Foo>* Next = Front;
   while (Next != NULL) {
      Front = Next->getNext();
      delete Next;
      Next = Front;
   }
}
```

When delete is invoked on a `NodeT<Foo>` object, any subobjects will be destructed automatically.

This assumes that `Foo` has an appropriate destructor.

Queue Template Enqueue

Linear Structures 20

```
// . . . continuing implementation . . .

// Inserts data value into new node at rear of queue list.
template <class Foo> void QueueT<Foo>::Enqueue(Foo Item) {

   NodeT<Foo>* newNode = new NodeT<Foo>(Item);
   if (newNode == NULL) {
      LastError = QUEUE_IS_FULL;
      return;
   }
   if (Front == NULL) {
      Front = Rear = newNode;
   }
   else {
      Rear->setNext(newNode);
      Rear = newNode;
   }
   LastError = NO_ERROR;
}
```

Create new node object and insert data element.

Handle insertion into empty queue…

…and nonempty queue.

©William D McQuain, July 2000

Queue Template Dequeue

Linear Structures 21

```
// . . . continuing implementation . . .

// Returns data element from first node in queue list.
template <class Foo> Foo QueueT<Foo>::Dequeue() {

   if (Front == NULL) {
      LastError = QUEUE_IS_EMPTY;
      return Foo();
   }
   Foo toReturn = Front->getData();
   NodeT<Foo>* SaveFront = Front;
   Front = Front->getNext();

   if (Front == NULL)
      Rear = NULL;

   delete SaveFront;
   LastError = NO_ERROR;
   return toReturn;
}
```

If queue is empty set flag and return default value…

…otherwise, snag return value from node, and reset `Front` pointer…

… and check for empty queue.

Delete obsolete node object.

©2000 McQuain WD

Queue Template Tests

Linear Structures 22

```
// . . . continuing implementation . . .

// Return true if queue list is empty.
template <class Foo> bool QueueT<Foo>::isEmpty() const {

   return (Front == NULL);
}

// Return true if new node object cannot be created.
template <class Foo> bool QueueT<Foo>::isFull() const {

   NodeT<Foo>* Try = new NodeT<Foo>;
   if (Try == NULL) return true;
   delete Try;
   return false;
}
```

Same logic as full test for `Stack` class shown earlier.

©2000 McQuain WD

Queue Template State Functions

Linear Structures 23

```
// . . . continuing implementation . . .

// Return true if queue list is empty.
template <class Foo> bool QueueT<Foo>::isEmpty() const {

   return (Front == NULL);
}

// Return true if new node object cannot be created.
template <class Foo> bool QueueT<Foo>::isFull() const {

   NodeT<Foo>* Try = new NodeT<Foo>;
   if (Try == NULL) return true;
   delete Try;
   return false;
}

// Return error flag.
template <class Foo>
QueueError QueueT<Foo>::getLastError() const {

   return LastError;
}
```

Same logic as full test for `Stack` class shown earlier.

©2000 McQuain WD

Queue Template Display Function

Linear Structures 24

```
// . . . continuing implementation . . .

template <class Foo>
void QueueT<Foo>::Display(ostream& Out) {

   NodeT<Foo>* Current = Front;
   int Idx = 0;
   while (Current != NULL) {
      Out << setw(3) << Idx << ":  "
          << Current->getData()
          << endl;
      Idx++;
      Current = Current->getNext();
   }
}

#endif
// end of QueueT.h
```


©2000 McQuain WD

©William D McQuain, July 2000

Queue Template Driver


```
// Driver to test QueueT.
#include <iostream>
#include <iomanip>

using namespace std;

#include "QueueT.h"

void main() {

   const int Size = 10;

   QueueT<int> intQ;

   for (int i = 0; i < Size; i++) {
      intQ.Enqueue(i*i);
   }

   intQ.Display(cout);
}
```

```
0:  0
1:  1
2:  4
3:  9
4:  16
5:  25
6:  36
7:  49
8:  64
9:  81
```


Node Template Interface


```
// NodeT.h
// Node Template
#ifndef NODET_H
#define NODET_H

template <class Foo> class NodeT {
private:
   Foo Data;            // data element
   NodeT<Foo>* Next;    // ptr to next node

public:
   NodeT();                          // empty, isolated node
   NodeT(Foo newData);               // filled, isolated node
   void setData(Foo newData);        // set data element
   void setNext(NodeT<Foo>* newNext);// set pointer element
   Foo getData() const;              // get data element
   NodeT<Foo>* getNext() const;      // get pointer element
   ~NodeT();                         // empty destructor
};

#include "NodeT.cpp"                 // include implementation
#endif
```


Node Template Implementation


```
// NodeT.cpp

//////////////////////////////////////////////////////////
// Constructor for NodeT objects with default data field.
//
// Parameters: none
// Pre:        none
// Post:       new NodeT has been created with default
//                 data field value and NULL pointer
//
template <class Foo> NodeT<Foo>::NodeT() {
   Data = Foo();
   Next = NULL;
}
```

The implementation of the `NodeT` member functions is straightforward and omitted here.

Containers as Templates

As a general rule, the data structures we will discuss are most usefully implemented as templates.

The additional syntactic ugliness is more than repaid by the ease with which the resulting code may be simply plugged into a variety of designs, housing a variety of types, with no internal modifications.

The popularity of the STL attests to the appeal of such genericity in OO design and development.

©William D McQuain, July 2000