Lists, Stacks, and Queues

A linear structure is an ordered (e.g., sequenced) arrangement of elements.

There are three common types of linear structures:

list	random insertion and deletion
stack	insertion and deletion only at one end
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 Pure Container

This chapter presents sample implementations of a array-based stack template and a link-based list 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, so a C++ template is used.
- to provide the client with appropriate access to stored data (which is rightly the property of the client) without compromising the encapsulation of the container itself.

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

ADT Case Study

Consider the following interface for a stack:

```
template <typename T> class StackT {
public:
   StackT();                                       // create empty stack
   StackT(const StackT<T>& Source);                // copy logic
   StackT<T>& operator=(const StackT<T>& Source);

   T* const Top();                                 // access top element
   const T* const Top() const;
   void Pop();                                     // remove top element
   void Push(const T& Elem);                       // place new elem on top
   bool isEmpty() const;                           // is stack empty?
   void Clear();                                   // remove stack contents
   ~StackT();                                      // destroy stack
   void Display(std::ostream& Out) const;          // display stack contents

// ... continues ...
```

Note that the public interface gives very few clues as to whether the underlying physical structure will be static or dynamic, array-based or linked.

ADT Case Study

The private section shows the stack is linked rather than contiguous:

```
private:
   class SNodeT {
   public:
      SNodeT();
      SNodeT(const T& Elem, SNodeT* N = NULL) {
         Element = Elem;
         Next    = N;
      }
      T       Element;
      SNodeT* Next;
   };

   SNodeT* mTop;                                   // targets top element
};
```

The node type is private to the container since client code does not require access to it.

Note that although the node type is not declared as a template it is still effectively a template since its declaration is nested within one.

Independence of Interface

For a user of the `StackT` template, client code would be the same whether the underlying structure were an array or linked:

```
bool SearchStack(const string& toFind, StackT<string> S) {

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

Assumes that `T` 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).

Stack Design Considerations

The `StackT` template has a few noteworthy features:

- The use of a template allows the client to create as many stack objects, storing as many different types as desired.
- The use of a C++ template preserves type-checking at compile time.
- A linked structure stores the stack elements, so there is no inherent size limitation.
- Stack underflow is signaled by returning a `NULL` pointer.
- It is the client's responsibility to check the value of that return value.
- The C++ header `<new>` is used, and so that operator `new` will by default throw a `bad_alloc` exception if an allocation fails.

Stack Construction and Destruction

```
#include <new>
#include <iostream>
#include <iomanip>
```

Use new-style headers.

```
template <typename T> StackT<T>::StackT() {

   mTop = NULL;
}
```

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

```
template <typename T> StackT<T>::~StackT() {

   while ( mTop != NULL ) {
      SNodeT* toKill = mTop;
      mTop = mTop->Next;
      delete toKill;
   }
   mTop = NULL;
}
```

Clearing stack contents should restore stack to same state as initial, empty stack.

Stack Copy Constructor

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

```
template <typename T> StackT<T>::StackT(const StackT<T>& Source) {

   mTop = NULL;
   SNodeT* sCurrent = Source.mTop;
   SNodeT* tCurrent;
   while ( sCurrent != NULL ) {
      SNodeT* p = new SNodeT(sCurrent->Element);
      if ( mTop == NULL ) {
         mTop = tCurrent = p;
      }
      else {
         tCurrent->Next = p;
         tCurrent = p;
      }
      sCurrent = sCurrent->Next;
   }
}
```

Assumes that `T` has an appropriate assignment operator.

Stack Assignment Overload

```
template <typename T>
StackT<T>& StackT<T>::operator=(const StackT<T>& Source) {

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

   SNodeT* sCurrent = Source.mTop;
   SNodeT* tCurrent;
   while ( sCurrent != NULL ) {
      SNodeT* p = new SNodeT(sCurrent->Element);
      if ( mTop == NULL ) {
         mTop = tCurrent = p;
      }
      else {
         tCurrent->Next = p;
         tCurrent = p;
      }
      sCurrent = sCurrent->Next;
   }

   return (*this);

}
```

Test for self-assignment.

Avoid memory leak if target of assignment is already initialized.

Return `StackT` object.

Stack Push Operation

```
template <typename T> void StackT<T>::Push(const T& Elem) {

        SNodeT* p = new SNodeT(Elem, mTop);
        mTop = p;
}
```

Note: if the memory allocation fails, `new` will throw an exception of type `bad_alloc`.

We make no effort to deal with that here… the burden is on the client.

Alternatively:

- we could return a `bool` value signifying success/failure
- we could use a `throw` specifier in the interface: `... throw (bad_alloc) ...`

  however, the compiler will <u>not</u> enforce any such restrictions, stated or not

Stack Top and Pop Operations

The StackT `Top()` operation must deal with stack underflow:

```
template <typename T> T* const StackT<T>::Top() {

   if ( mTop == NULL )
     return NULL;
   else
      return &(mTop->Element);
}
```

Here, we return NULL if the stack underflows… the client has the burden of checking the return value.

The StackT `Pop()` operation simply removes the top element, if any:

```
template <typename T> void StackT<T>::Pop() {

   if ( mTop != NULL ) {
      SNodeT* Temp = mTop;
      mTop = mTop->Next;
      delete Temp;
   }
}
```


Displaying the Contents

```
template <typename T>
void StackT<T>::Display(std::ostream& Out) const {

   SNodeT* Current = mTop;
   unsigned int Pos = 0;
   while ( Current != NULL ) {

      Out << setw(5) << Pos << ":  " << Current->Element << endl;
      Current = Current->Next;
      Pos++;
   }
}
```

Whether to include a display function is somewhat problematic.

If not, some info used here cannot be displayed by a nonmember function without destroying the stack or making a copy to destroy.

If we do include this, then `T` objects must support stream insertion.

Some Observations

Reflecting on the given `StackT` implementation:

- Effectively, the stored elements may be simple, `struct` or `class` type variables. However, it is generally preferable use use a `class` type rather than a `struct` (unless a simple built in type suffices).

  `struct` types should be restricted to situations where member functions are unnecessary (some authors 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 `StackT` class header file.

Iterators

An iterator is an object, associated with a particular container type, that provides safe, controlled access to data stored in the container.

Containers may declare iterator classes as public member types. This gives the iterator privileged access to the container, while hiding the iterator's internal structure from clients.

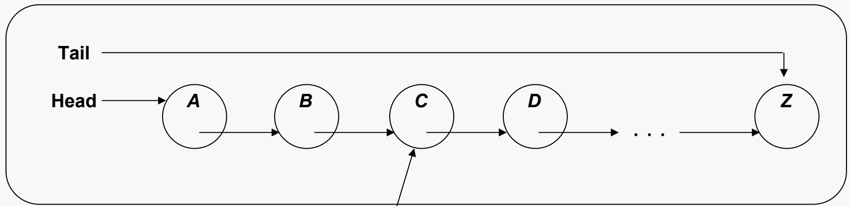
Container objects are designed to provide iterator objects to clients.

Iterator objects provide the user with the ability to traverse the container, and to access data elements by dereferencing the iterator.

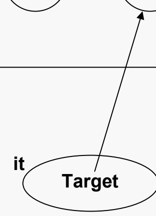
Iterators are similar to pointers, but provide a level of information hiding and error-checking that simple C++ pointers do not.

Iterator Basics

Suppose we have a linked list object:

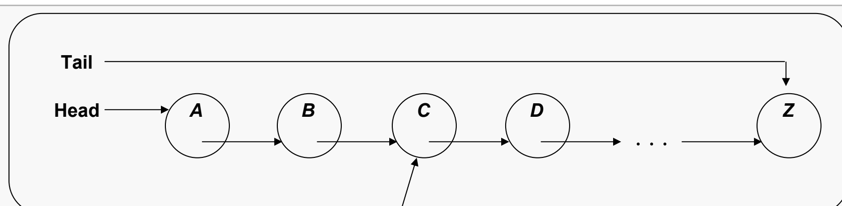


Then an iterator object would store a pointer to a particular list node:

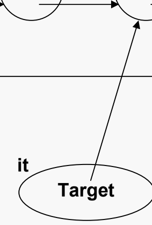


The iterator client can move it within the list, and dereference it to access the data within the list node (but not to access the node itself).

Iterator Operations



The iterator can be dereferenced like a pointer, but this yields a reference to the corresponding data element: `*it`

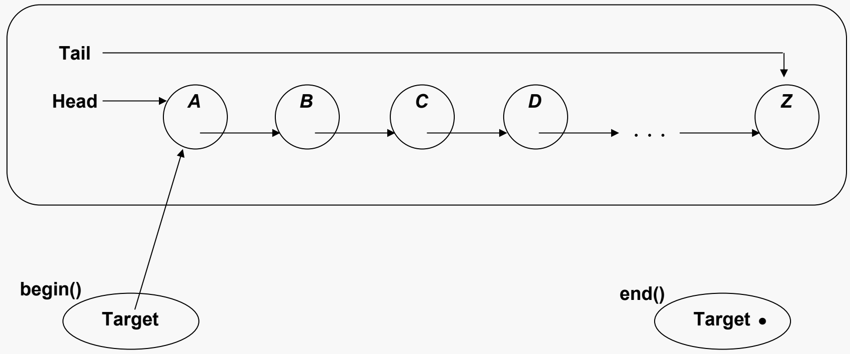


It is possible an iterator does not have a target, a case which may be represented by storing a `NULL` value within the iterator object.

The client can move the iterator by using the increment and decrement operators: `it++` `it--`

Container Operations

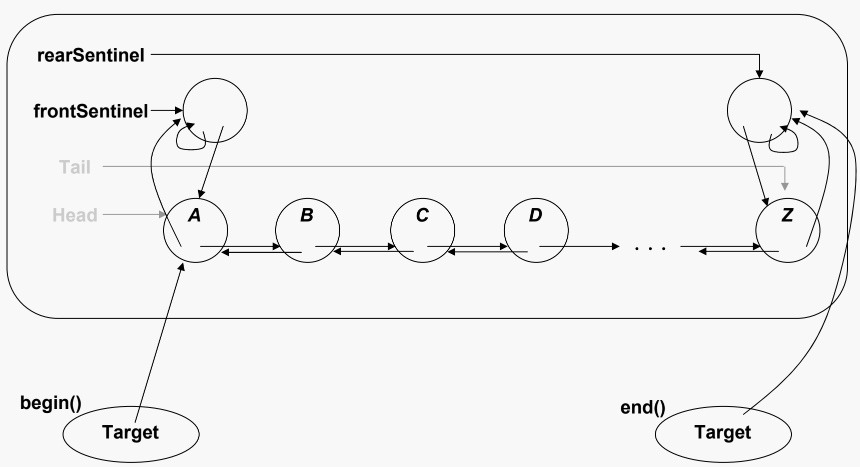
The container will typically provide public functions that return useful iterators to the client.



This sort-of shows the STL convention that `end()` returns an iterator that is "one-past-the-end" of the container… as we will see, this is useful in designing traversals in client code.

Sentinel Nodes

The container may also employ data-free sentinel nodes at the front or at both ends. This slightly complicates some operations, but may be preferred when including iterator support. Here is one approach:



Container with Iterator

Consider the following interface for a doubly-linked list with iterator:

```
template <typename T> class DListT {
private:
   DNodeT<T>* Fore;    // pointer to front sentinel
   DNodeT<T>* Aft;     // pointer to rear sentinel

public:
   DListT();                                // create empty list object
   DListT(const DListT<T>& Source);                // deep copy support
   DListT<T>& operator=(const DListT<T>& Source);
   ~DListT();                                      // deallocate nodes

   bool isEmpty() const;              // return true if empty
   void Clear();                      // deallocate nodes and reset ptrs
   void Display(ostream& Out) const;  // write formatted contents

   // iterator class declaration/implementation goes here

   iterator begin();                  // return iterator to first elem
   iterator end();                    // return iterator "one-past-last"

   iterator Insert(iterator It, const T& Elem);    // insert elem
   iterator Find(const T& Elem);                   // locate data elem
   bool Delete(iterator It);                       // remove data elem
};
```


The DListT Iterator

Here's the declaration for the `DListT::iterator` class:

```
/////////////////////////////////////////// iterator
class iterator {
private:
   DNodeT<T>* Position;          // pointer to DListT node

   iterator(DNodeT<T>* P) {      // make an iterator from a node ptr
      Position = P;
   }

public:
   iterator() { Position = NULL; } // invalid iterator
   iterator operator++();          // step to next data element
   iterator operator++(int Dummy);
   iterator operator--();          // step to previous data element
   iterator operator--(int Dummy);
   bool operator==(const iterator& RHS) const; // comparisons
   bool operator!=(const iterator& RHS) const;
   T& operator*() throw( range_error );      // access data element

   friend class DListT<T>;     // let DListT create useful iterators
};
```

Note: the iterator function implementations must be (implicitly) inlined.

Incrementing the Iterator

Here are the two `operator++` implementations for the `DListT` iterator:

```
iterator operator++() {
   if ( Position != NULL /* && ??? */ )
      Position = Position->Next;
   return (*this);
}

iterator operator++(int Dummy) {

   iterator Now(Position);
   if ( Position != NULL /* && ??? */ )
      Position = Position->Next;
   return (Now);
}
```

The implementations are essentially straightforward.

Recall that the postfix version must declare a dummy parameter in order to be distinguishable (by the compiler) from the prefix version.

The implementations of the decrement operators are similar.

Comparing Iterators

Obviously, two iterators are equal if, and only if, they point to the same element of the data structure, as opposed to merely to the same data value:

```
bool operator==(const iterator& RHS) const {

   return ( Position == RHS.Position );
}
```

There are also reasonable definitions for less-than comparisons for iterators, at least on a linear structure. However, we will not consider them here.

Dereferencing the Iterator

Dereferencing the iterator yields a reference to the stored data element, NOT to the list node… that's the key to safely providing the client with access to the data.

```
T& operator*() throw ( range_error ) {

   if ( Position == NULL /* || other conditions?? */ )
      throw range_error("dereferenced bad iterator");

   return (Position->Element);
}
```

If the client dereferences a NULL iterator, an exception of type `range_error` will be thrown. This allows the client to attempt to recover from such a mistake.

The class `range_error` is a Standard class declared within the Standard header file `stdexcept`. This is one sensible way to respond to an attempt to dereference a bad pointer.

Note that the constructor logic guarantees that each iterator will either store NULL, or the address of one of the sentinel nodes, or the address of an actual `DListT` node, assuming the `DListT` implementation is correct.

Using `DListT` Iterators

Here's a client loop that prefixes values to a `DListT` object:

```
DListT<int> L;

for (int Idx = 0; Idx < 10; Idx++) {

    L.Insert(L.begin(), rand() % 10 );
}
```

Here's a client loop that prints the contents of a `DListT` object:

```
DListT<int>::iterator It;

for (It = L.begin(); It != L.end(); It++) {
   cout << *It << endl;
}
```

The Need for const Iterators

Consider the following function implementation:

```
void H(const DListT<int>& L, ostream& Out) {

   DListT<int>::iterator It;

   for (It = L.begin(); It != L.end(); It++) {
      Out << *It << endl;
   }
}
```

The use of a `DListT::iterator` leads to a compile-time error because it conflicts with the fact that the `DListT` parameter is declared as `const`.

To fix the problem, we must add a second iterator class to the `DListT` implementation, one that is designed to preserve `const`-ness.

The DListT const_iterator

Here's part of the declaration for the `DListT::const_iterator` class:

```
////////////////////////////////////////// const_iterator
class const_iterator {
private:
   DNodeT<T>* Position;             // pointer to DListT node

   const_iterator(DNodeT<T>* P) {  // make an iterator from a node ptr
      Position = P;
   }

public:
   const_iterator() { Position = NULL; } // invalid iterator
   // increment/decrement and comparison operators go here

   const T& operator*();         // access (but not change) data element
   const_iterator(const iterator& It);
   const_iterator& operator=(const iterator& It);
   friend class DListT<T>;
};
```

The primary differences are that the dereference returns a constant reference, and that we need conversion operations from `iterator` to `const_iterator` (but <u>not</u> the reverse).

Using `const` Iterators

The `const_iterator` is needed in the copy constructor for the `DListT` template:

```
template <typename T>
DListT<T>::DListT(const DListT<T>& Source) {

   Head = Tail = NULL;
   DListT<T>::const_iterator Current = Source.begin();
   while ( Current != Source.end() ) {
      Insert(this->end() , *Current);
      Current++;
   }
}
```

By providing a `const_iterator`, we make it possible to use const passes of `DListT` objects whenever it makes sense in the design of client code.

Of course, it is still up to the client to use the `const_iterator` when it is needed.

Conversions

We provide conversions from `iterator` to `const_iterator` so that a regular `iterator` can be passed when a `const_iterator` is expected:

```
const_iterator(const iterator& It) {
   Position = It.Position;
}

const_iterator& operator=(const iterator& It) {
   Position = It.Position;
   return (*this);
}
```

Note that these require that a `const_iterator` object be able to access the private data member of the `iterator` object it receives. So, we must add another `friend` declaration to the `iterator` class declaration:

```
friend class const_iterator;
```

Other Issues

The STL distinguishes between several logically different kinds of iterators, including:

- forward and backward iterators that can be moved in only one direction
- bidirectional iterators that can move in either of two directions
- input and output iterators associated with a stream

At this level, we are primarily concerned with bidirectional iterators associated with a storage container.

Many STL containers also provide additional iterator-supplying functions. Typical examples are:

`rbegin()` returns an iterator to the final data value

`rend()` returns an iterator to one-before-the-first data element

These are useful for making reverse traversals of a container.

Summary

We now have a robust, client-friendly doubly-linked list template.

The addition of iterators allows us to safely provide client access to data without risking a compromise of the integrity of the class protections.

The iterators are easy to use, after a brief learning curve, and impose no strenuous burdens on the client.

As a general rule, we will expect iterators to be used for the container templates that are implemented in this course.