

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

This chapter presents sample implementations of an 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.*

Consider the following interface for a stack:

```
template <typename T> class StackT {
private:
    // suppressed for now . . .

public:
    explicit StackT(int Capacity = 100);           // construct new stack
    StackT(const StackT<T>& Source);              // copy constructor
    StackT<T>& operator=(const StackT<T>& Source); // assignment overload
    bool Push(const T& Elem); // insert on top of stack, increasing
                                // stack array dimension if necessary
    bool Pop(T& Elem); // remove and return element at top of stack
    T* const Peek() const; // return pointer to element at top of stack
    bool isEmpty() const; // indicate whether stack is currently empty
    ~StackT(); // deallocate stack array
    void Display(ostream& Out) const; // display stack contents to stream
};
```

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.

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() ) {
        S.Pop(Elem);
        if (toFind == Elem)
            return true;
    }
    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).

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.
- 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 signaled by returning a `bool` indicator.
- It is the client's responsibility to check the value of that return value.
- The C++ header `<new>` is used, and so `new` will throw a `bad_alloc` exception if an allocation fails.

```
#include <new>
#include <iostream>
using namespace std;
```

Use new-style headers.

```
template <typename T>
StackT<T>::StackT(int Capacity) {
    Cap = Capacity;
    Top = 0;
    Stk = new(nothrow) T[Cap];
    if (Stk == NULL) Cap = 0;
}
```

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

```
template <typename T>
StackT<T>::~~StackT() {
    delete [] Stk;
}
```

Deallocate stack array.

Stack Copy Constructor

Linear Structures 7

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

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

    Cap = Source.Cap;
    Top = Source.Top;

    Stk = new T[Cap];
    for (int Idx = 0; Idx < Top; Idx++)
        Stk[Idx] = Source.Stk[Idx];
}
```

Assumes that `T` has an appropriate assignment operator.

Stack Assignment Overload

Linear Structures 8

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

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

    delete [] Stk;

    Cap = Source.Cap;
    Top = Source.Top;

    Stk = new T[Cap];
    for (int Idx = 0; Idx < Top; Idx++)
        Stk[Idx] = Source.Stk[Idx];

    return *this;
}
```

Test for self-assignment.

Avoid memory leak if target of assignment is already initialized.

Return `StackT` object.

Stack Push Operation

Linear Structures 9

The StackT object uses an automatically resizable array, rather than a fixed size array:

```
template <typename T>
bool StackT<T>::Push(const T& Elem) {
    if (Top == Cap) {
        T* tmpStk;
        try {
            tmpStk = new T[2*Cap];
        }
        catch ( bad_alloc e ) {
            return false;
        }
        for (int Idx = 0; Idx < Cap; Idx++) {
            tmpStk[Idx] = Stk[Idx];
        }
        delete [] Stk;
        Stk = tmpStk;
        Cap = 2*Cap;
    }
    Stk[Top] = Elem;
    Top++;
    return true;
}
```

If StackT 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.

Note: Top is the index of the next available cell, not of the top-most element.

Stack Pop Operation

Linear Structures 10

The StackT Pop () operation must deal with stack underflow:

```
template <typename T>
bool StackT<T>::Pop(T& Elem) {
    if ( (Top > 0) && (Top <= Cap) ) {
        Top--;
        Elem = Stk[Top];
        return true;
    }
    return false;
}
```

If StackT array is not empty, decrement position of Top element, set the parameter and return true.

If StackT array is empty, return false.

This places a burden on the user to test the return value.

In some stack implementations, the pop operation will return the data element instead of a bool value. This poses some difficulties in signaling failure.

Stack Reporters

Linear Structures 11

```
template <typename T>
bool StackT<T>::isEmpty() const {
    return (Top == 0);
}
```

Recall `Top` is index of first available cell.

```
template <typename T>
T* const StackT<T>::Peek() const {
    if ( (Top > 0) && (Top <= Cap) ) {
        return &Stk[Top-1];
    }
    return NULL;
}
```

This allows the client to access the top-most element (if any), and even modify it *in situ*.

Why is the return value a const pointer instead of just a pointer? or a reference?

Displaying the Contents

Linear Structures 12

```
template <typename T>
void StackT<T>::Display(ostream& Out) const {
    Out << "Cap:" << setw(3) << Cap << 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 `T` objects must support stream insertion.

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.

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

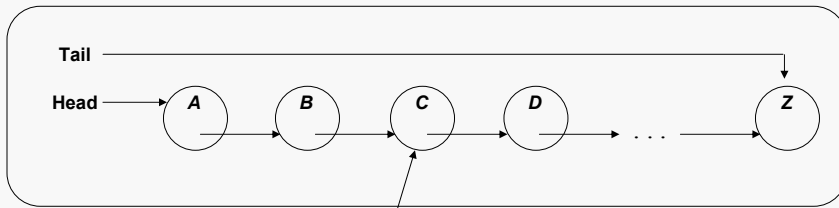
Containers may declare iterator classes as public member types. This gives the iterator privileged access to the container, while hiding the iterator's internal structure from clients.

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.

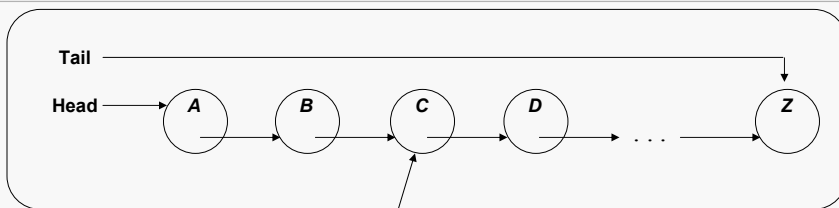
Suppose we have a linked list object:



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).



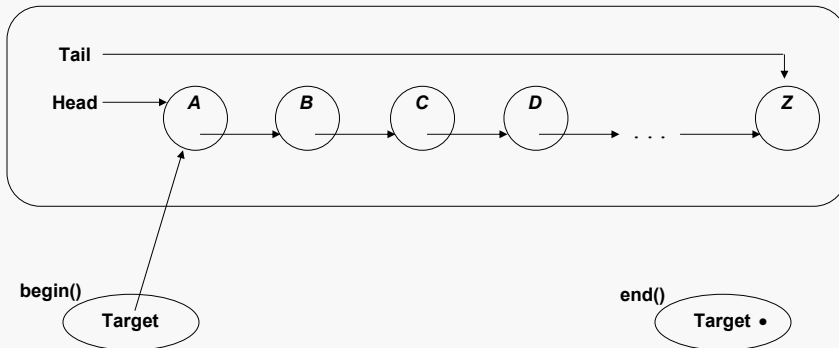
The iterator can be dereferenced like a pointer, but this yields a reference to the corresponding data element: `*it`



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--`

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



This 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.

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

```
template <typename T> class DListT {
private:
    DNodeT<T>* Head;    // pointer to head node of list
    DNodeT<T>* Tail;   // pointer to tail node of list

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"

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

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*();              // access data element

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

```

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

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

Linear Structures 21

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

Linear Structures 22

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*() {  
    if ( Position == NULL )  
        throw IllegalAccess();  
    return (Position->Element);  
}
```

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

The class `IllegalAccess` is a trivial class declared within the `DListT` header file.

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

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;
}
```

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++) {
        cout << *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.

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 not the reverse).

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.

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;
```

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.