This chapter presents a sample implementation of a linked list, encapsulated in a C++ class.

The primary goals of this implementation are:
- to provide a proper separation of functionality.
- to design the list to serve as a container; i.e., the list should be able to store data elements of any type.

First, a node class, SNode class is used to encapsulate the data and pointers.

Second, a SList class is used to encapsulate a list of SNode objects.

Third, an Item class is used to encapsulate the data and separate it from the pointers that define the list structure.

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

Warning: the SList class given in this chapter is intended for instructional purposes. The given implementation contains a number of known flaws, and perhaps some unknown flaws as well. Caveat emptor.

---

### SNode Class

```cpp
// SNode.h

// Singly-linked node class.

// Features:
// - Default SNode contains default Item object and a NULL pointer.
// - Accessor function getData() returns a reference to the stored data element, allowing user editing of the data object.
// - Constructor initializes element to default.
// - Destructor is provided.

// Assumptions:
// - User will supply a header file, Item.h containing a typedef statement mapping some real type to the name Item used in SNode.
// - That type will provide deep copy support and a destructor, if needed.

#ifndef SNODE_H
#define SNODE_H

#include "Item.h" // for typedef

class SNode { // for typedef
private:
  Item  Element;
  SNode *Next;

public:
  SNode();
  SNode(const Item& E, SNode* N = NULL);
  Item& getData();
  void  setData(const Item& E);
  SNode* getNext();
  void  setNext(SNode* N);
};
#endif
```

Why is there no destructor?

The SNode class neither knows nor cares what an Item variable is — an SNode is a container.
SNode Class Constructors

enum\n
SNode constructor implementations:

```cpp
// SNode.cpp
#include <cstdlib> // for NULL
#include "SNode.h" // for declaration of type SNode

////////////////////////////////////////////// SNode()
// Constructs an empty node, with default data
// element and NULL pointer.
// Parameters: none
// Returns: none
// Called by: client code
// SNode::SNode() {
   Next = NULL;
}

///////////////////////////////// SNode(Data, Pointer)
// Constructs a node with specified data
// element and pointer.
// Parameters: E data value to place in node
//            N pointer to next node
// Returns: none
// Called by: client code
// SNode::SNode(const Item& E, SNode *N) {
   Element = E;
   Next = N;
}
```

Uses default construction for Item objects.

When an object is a data member of another object, the data member is automatically initialized using the default constructor for its type.

SNode Class Reporters

```cpp
// Data and Pointer

////////////////////////////////////////////// getData()
// Provides user access to stored data element.
// Parameters: none
// Returns: reference to node's data element
// Calls: none
// Called by: client code
// Item& SNode::getData() {
   return Element;
}

////////////////////////////////////////////// getNext()
// Provides user access to pointer to next node.
// Parameters: none
// Returns: pointer to next node
// Calls: none
// Called by: client code
// SNode* SNode::getNext() const {
   return Next;
}
```

When an object is a data member of another object, the data member is automatically initialized using the default constructor for its type.

Uses default (or overloaded) assignment for Item objects.

Uses const to guarantee no modification occurs.

getData() returns a reference to the data member Element, not a copy of it.

That allows the user of the SNode object to modify the data element it stores, in situ.

Some designers would argue this violates information hiding. Others would ask "who owns the data element anyway?"
### SNode Class Mutators

#### setData()

- **Purpose**: Provides user ability to set data element.
- **Parameters**: 
  - `E` data value to be stored
- **Returns**: none
- **Calls**: none
- **Called by**: client code

```cpp
void SNode::setData(const Item& E) {
    Element = E;
}
```

#### setNext()

- **Purpose**: Provides user ability to modify pointer to next node.
- **Parameters**: value to which pointer will be set
- **Returns**: none
- **Calls**: none
- **Called by**: client code

```cpp
void SNode::setNext(SNode* N) {
    Next = N;
}
```

**Why is the parameter to setNext() not passed as:**

```cpp
const SNode* const N
```
#ifndef SLIST_H
#define SLIST_H
#include <iostream>
#include "Item.h" // for Item declaration
#include "SNode.h" // for SNode declaration

class SList {
private:
    SNode *Head;
    SNode *Tail;
    SNode *Current;
public:
    SList();                      // make an empty list
    SList(const SList& Source);         // copy constructor
    SList& operator=(const SList& RHS); // assignment
    bool Insert(const Item& E);  // insert value E at
                                  // current position
    bool Delete(Item& E);        // delete value at
                                  // current position
    Item& Get() const;            // get reference to
                                  // current data
    bool Advance();              // move current
                                  // position toward tail
    void goToHead();             // move current position
                                  // to head
    void goToTail();             // move current position
                                  // to tail
    bool atEnd() const;          // true if current
                                  // position is NULL
    bool isEmpty() const;        // true if list is empty
    void Display(ostream& Out) const;  // print list
    ~SList();                     // deallocate nodes
};
#endif

One line functions could be "inline" for efficiency.
### SList Destructor

The destructor must deallocate all the SNode objects that were allocated by the SList object.

```cpp
SList::~SList() {
    SNode *toKill = Head;
    while ( toKill != NULL ) {
        Head = Head->getNext();
        delete toKill;
        toKill = Head;
    }
}
```

The destructor is called automatically whenever the lifetime of an SList object ends (i.e. at the end of the function/block in which the objects are defined, when a dynamically allocated object is destroyed with `delete()`, when an object containing a member object is destroyed).

A class destructor’s names is always the tilde followed by the name of the class. It has no parameters or return type and cannot be overloaded.

SList needs a destructor in order to properly return the dynamically-allocated nodes to the system heap.

### SList Insert Mutator

SList implements insertion to add a new node to the list immediately following the target of the Current pointer, if that is defined.

What limitation does this impose on the client?

```cpp
bool SList::Insert(const Item& E) {
    if ( Head == NULL ) { // inserting in empty list
        SNode *Temp = new SNode(E, NULL); // make node
        Head = Tail = Temp;                   // hook it in
        Current = Head; // current
        return true;
    }
    if ( Current == NULL ) { // no current position
        return false;
    }
    // inserting node in middle or at end
    SNode *Temp = new SNode(E, NULL); // make new node
    Temp->setNext(Current->getNext()); // hook it in
    Current->setNext(Temp);
    return true;
}
```
The Current Position

SList maintains a sense of a "current position" by storing a private pointer that can be moved by the client; this allows the client to use the list in a flexible, natural manner.

```cpp
bool SList::Advance() {
    if (Current == NULL) return false; // no current position
    Current = Current->getNext();
    return true;
}
```

The client may also set the current position to the head or tail of the list, and there is a test to see if the current position is valid; the design corresponds to the STL conventions by making "end" mean "at an imaginary invalid location past the last node".

```
bool SList::atEnd() const {
    return (Current == NULL);
}
```

---

Searching an SList

SList does not provide the client with a search function. However, it's easy for the client to implement one:

```cpp
bool Locate(const Item& Target, SList& L) {
    if (L.isEmpty()) return false;
    L.goToHead();
    while (!L.atEnd()) {
        if (Target == L.Get())
            return true;
        L.Advance();
    }
    return false;
}
```

The implementation assumes that there is an equality comparison for the data type Item, but search would not make much sense otherwise.

Question: will this code terminate properly if the SList doesn't contain a value matching Target?

Question: why isn't the SList object passed to the function by const reference?

Question: could this be a member function of SList?
Client Access to the Data

SList provides the client with an accessor function to the data element in the current list node:

```cpp
///////////////////////////////////////////////// Get()
// Provides user access to data element in current list
// node.
// Parameters: none
// Returns: reference to current data element
// Calls: SNode.getData()
// Called by: client code
// Item& SList::Get() const {
   return (Current->getData());
}
```

Note how SList::Get() and SNode::getData() are designed to work together to give the client a reference to the stored data element.

That allows the client to modify the data element in situ:

```cpp
SList L;
L.Get() = 1;
```

Note also that Get() doesn’t deal well with being called when Current is NULL. The reason for this design is that there’s no good return value for the reference when Current is NULL.

This can be handled by making the return value Item* instead.

Design Discussion

Does the fact that SList::Get() returns a reference to a member of the node violate information hiding?

No. To be picky, the nodes are NOT members of the SList object. But that’s an artificial defense, and misses important points.

First of all, the data elements belong to the client, not to the container. Granted, the container is responsible for organizing the data elements, but it is also responsible for providing the client with flexible, efficient access to the data.

Note that the design here is NOT the same as returning a reference or pointer to a node; that would clearly be unsafe since the client could then interact directly with the node interface, and perhaps even deallocate the node, wreaking havoc with the physical structure of the list.
SList Delete Mutator

7. LL Class

SList deletion removes the current node (if there is one), and returns the data element it contained to the client:

```cpp
bool SList::Delete(Item& E) {
    if ( Current == NULL ) return false;
    if ( Current == Head) {      // deleting first node
        Head = Head->getNext();   //    reset head pointer
        E = Current->getData();  //    save data element
        delete Current;           //    deallocate node
        Current = Head;           //    skip around target
        return true;
    }
    // find preceding node
    SNode *Previous = Head;      // start at head node
    while ( Previous->getNext() != Current)
        Previous = Previous->getNext();
    // make preceding node point to successor
    Previous->setNext(Current->getNext());
    E = Current->getData();      // save data element
    delete Current;              // deallocate node
    Current = Previous->getNext();
    return true;
}
```

Deep Copy for SList

7. LL Class

SList must also provide deep copy support:

```cpp
// copy constructor
// Initializes new SList object as a copy of an existing SList object.
// Parameters: SList object to be copied
// Returns: none
// Calls: SNode.getData() SNode.getNext() SList.goToTail()
// Called by: client code
SList::SList(const SList& Source) {
    Head = Tail = Current = NULL;
    SNode *toCopy = Source.Head;
    while ( toCopy != NULL ) {
        Insert(toCopy->getData());
        goToTail();toCopy = toCopy->getNext();
    }
    // Note that the implementation uses member functions of SList, rather than re-implementing their logic here.
    // As usual, the implementation of SList::operator= is similar to the copy constructor.
```
Lined List Classes

Utility Functions

SList also provides a simple test for an empty list, and display functionality:

```cpp
void SList::Display(ostream& Out) const {
    SNode *Temp = Head;
    int Pos = 0;
    while ( Temp != NULL ) {
        Out << setw(3) << Pos << ":  
            << Temp->getData() << endl;
        Pos++;
        Temp = Temp->getNext();
    }
}
```

Note that the implementation assumes that operator<< can be applied to the data type Item.

This could also easily be written as a non-member function, however the ability to easily display the contents of a container is so useful in testing and debugging that it is common to build that into containers that are under development.

Sample Data Element Class

```
// CreditCard.h
#ifndef CREDITCARD_H
#define CREDITCARD_H
#include <iostream>
using std::ostream;
#include <string>
using std::string;
class CreditCard {
private:
    string Number;
    double Balance;
public:
    CreditCard(const string& Num = ",
        double Amount = 0.0);
    void Payment(double Amount);
    void Charge(double Amount);
    double CardBalance() const;
    bool operator==(const CreditCard& RHS) const;
    bool operator!=(const CreditCard& RHS) const;
    bool operator<(const CreditCard& RHS) const;
    bool operator<=(const CreditCard& RHS) const;
    bool operator>(const CreditCard& RHS) const;
    bool operator>=(const CreditCard& RHS) const;
    friend ostream& operator<<(ostream& Out,
        const CreditCard& Card);
};
#endif
```

friend operators and functions can access private members as if they were class members themselves.
Aside: friends

There are some circumstances in which an operator or function needs to have direct access to private data of a class, but it cannot itself be a class member.

The most common example is an overloaded operator<<.

An operator can only be a member of the class that appears as its left operand.

The left operand of operator<< is an output stream object.

The problem may be solved by having the (right operand) class declare the operator to be a friend.

Friends have privileged access to the private section of a class:

```cpp
ostream& operator<<(ostream& Out, const CreditCard& Card) {
    Out << fixed << showpoint;
    Out << Card.Number << setw(11) << setprecision(2) << Card.Balance;
    return Out;
}
```

Normally, the implementation of a friend operator or function will be placed in the same file as the class implementation.

Data Comparison Operators

Sometimes the relational operators will consider only some, or one, of the data members of a class:

```cpp
bool CreditCard::operator==(const CreditCard& RHS) const {
    return (Number == RHS.Number);
}
```

This overloaded operator is required in order for search code to work. The other relational operators, such as operator<, may be needed for sorting or other operations.

As a general rule, if you implement operator== for a class, you should also supply operator!=.

And, if you implement operator<, you should also supply the other four comparisons.

The implementation cost is trivial, and it will make the resulting class much more natural to use.
Non-destructive List Merge

In some applications it is useful to be able to merge two lists into a third list.

The function below does that, making extensive use of the SList interface:

```cpp
// Given two SList objects, return a new ordered list // which contains all of the elements of both lists, // (the original lists must NOT be destroyed by the // merging).
// SList MergeLists(const SList& L1, SList L2){
    Item toCopy;
    SList Merger = L1;
    L2.goToHead();
    while ( !L2.atEnd() ) {
        toCopy = L2.Get();
        Merger.Insert(toCopy);
        Merger.goToTail();
        L2.Advance();
    }
    return Merger;
}
```

Question: how would you modify the function above to avoid storing duplicates in the merged list?

Question: what would happen if the function returned SList&?
The implementation of SList could be modified to maintain the data elements in ascending (or descending) order:

```cpp
bool SList::Insert(const Item& E) {
    if ( Head == NULL ) {     // inserting to empty list
        SNode *Temp = new SNode(E, NULL);
        Head = Tail = Temp;
        Current = Head;
        return true;
    }

    SNode *Predecessor = NULL;  // find preceding node
    SNode *Look = Head;
    while( Look != NULL && Look->Element < E ) {
        Predecessor = Look;
        Look = Look->Next;
    }

    if ( Predecessor == NULL ) {
        SNode *Temp = new SNode(E, Head);
        Head = Temp;
        return true;
    }

    // inserting in middle or at end
    SNode *Temp = new SNode(E, Predecessor);
    Predecessor->Next = Temp;
    return true;
}
```

There is now some risk associated with SList::Get(); if the client uses it to modify a data element in the wrong way then the list ordering would be incorrect.

The data element should be ordered on an immutable key field.

The interface of SList reflects a particular design philosophy; other developers would make different decisions.

For example, the user-controlled "bookmark" approach could be abandoned in favor of a less open approach. That would require some additional interface changes:

- at least one search function would be necessary.
- if the client is to have any control over the ordering of the data elements, there would have to be at least some variation of the insertion function, such as a prefix and/or a suffix insertion.
- deletion would have to allow the client to specify the data element to be found and removed.