KEEPING POINTERS OR REFERENCES UNDER CONTROL: A COMPONENT BASED APPROACH TO LIST BASED DATA STRUCTURES

Joan Krone
Denison University
Granville, Ohio 43023
krone@denison.edu

ABSTRACT

For students and practitioners alike, the use of dynamic variables (pointers or references) introduces not just the probability of more errors than for static variables, but errors that are by far more difficult to find and correct. Here we see a way to address this problem by using a component based approach to building software. This approach designs a generic list-based structure that can be used to build a variety of other components without the need to put in dynamic variables except in the list structure. Of course, this approach is not limited to academic use. It is the prescribed approach for any good software engineering practice.

INTRODUCTION

Many textbooks on data structures make distinctions at an implementation level, rather than at the abstract level. For example, in [Stubbs & Webre], there is a distinction made between stacks according to whether they are implemented using arrays or by using pointers. Similarly, in [Singh and Naps] a distinction is made between queues implemented with arrays and those implemented with pointers (lists). In [weiss] there is a distinction between a list component and a sorted list component. These distinctions are often made not just for lists, stacks and queues, but for a variety of other structures as well. As a result of seeing these distinctions, students do not get a good sense of what a useful component is.

Many textbooks [Corm, Aho,] introduce the concepts of information hiding, component based software, abstract reasoning, and other software engineering principles [Sommerville and Pressman], but fail to follow up with supportive examples. Students need to learn these important principles, and the best way to do so is to see good examples and then to write some of their own components, based on these principles.

Software engineering principles advocate information hiding, a principle clearly violated when these distinctions are made, since a client who needs a stack component should not need to think about how the stack is implemented, but rather what the abstract behavior is. Of course, performance information would be useful to a client, but that should be given in terms of analysis of the operations, not the implementation details. For example, if a client is told that there are two possible stack components available, there should be no distinction with regard to behavior. For performance, the client can be told how long pushes or pops or clears take without revealing the underlying details of implementation. Other useful information would be whether the stacks have a limited depth or whether they can grow as you need them. Then the client can do whatever reasoning needs to be done at a level of abstraction that frees him from being forced to worry about links (pointers).

We show here how it is possible to design a linked structure that can be used in a variety of ways, adhering to the principle of information hiding and allowing reasoning to be done at an appropriate level of abstraction.

A BASIC LIST STRUCTURE

Here we present specifications for a linked structure that can be used by a client who will not need to know anything about implementation details. The most common implementation can be done using pointers (or in Java, references). To make the component generic, one can use a generic template in ++ or use objects as the entries in Java.

We conceptualize the linked structure (a list position) as a pair of strings, called the Prec(eding) and Rem(ainder), suggesting that the current position in the list is at the beginning of the Rem part, i.e., the Prec part has been visited. The elements in the strings are the entries of whatever generic type the client may choose when the list is instantiated. Since this component is generic with regard to the entry type, we think of the component as providing a type family, rather than a single type, hence the Type_Family definition.

For example, if we want to build a list of characters, the conceptualization might look like the pair (rgmq, saf) where Prec = rgmq, indicating that those characters have been visited and we are ready to visit the Rem = saf, s being the next character to visit.

An *Advance* operation would move the current position so that the list position would look like (rgmqs, af). A *Reset* puts the current position back at the beginning of the list, yielding (Λ, rgmqsaf) where Λ indicates the empty string, showing that no part of the list has been visited.

Length_of_Remainder returns the number of entries in the Rem part of the list, so for (rgmq, saf), Length_of_Remainder would return 3. *Advance_to_End* places the current position at the end of the list. Given the list position (rgmq, saf_, Advance_to_End results in (rgmqsaf, Λ), since the Rem part has no elements in it, indicating that all of the elements have been visited.

To put in a new element, there is the *Insert* operation which takes the new entry passed in and places it at the beginning of the Rem. So if we have list position (rgmq, saf) and call for an insert of t, the result is (rgmq, tsaf). If a new element is to be added to the beginning of the list, the end of the list, or the middle, the same operation holds, since one can first call Reset or Advance_to_End and then call Insert. This is a real advantage over many list structures that need to have several ways to insert, depending on where you are in the list.

Finally, we might want to remove an item. The *Remove* operation removes the entry that is currently at the beginning of the Rem part of the list. For example, given list position (rgmq, saf), a Remove would yield (rgmq, af). Just as for inserting, a client can first advance to the position where the new entry is to be placed, when removing and entry, a client can first advance to the entry that is to be removed and then call Remove. This means that one simply calls remove, but does not need to tell which element to remove, since that choice is left to the user of the list. Hence, only one remove operation is needed, since it works whether one wants to remove an element at the beginning or at the end or anywhere in between.

Two additional operations add a lot of power to this component. First, a binary operation that takes two list positions and swaps their remainders. This operation is quite powerful because it allows the swapping of two complete lists, if resets are performed first, or the exchange of any parts of two lists as desired. For example, given list1 = (rgmq, saf) and list2 = (ca, bprw), a call to *Swap__Remainders* will result in list1 = rgmq, bprw and list2 = (ca, saf).

Finally, we have *Swap_Prev_Entry* that allows one to exchange a new entry with the last entry in the Prec part of the list. If we start with (rgmq, saf) and make a call to Swap_Prev_Entry with the parameter x, the result is (rgmx, saf).

Here are the specifications:

ONE-WAY LIST TEMPLATE

Concept One_Way_List_Template(**type** Entry; **eval** Max_Total_Length: Integer);
uses Std_Integer_Fac, String_Theory;
requires Max_Total_Length > 0;
Family List_Position $\subseteq$ **Cart-Prod**
Prec, Rem: Str(Entry)
end;
exemplar P;
Initialization
ensures P.Prec = Λ **and** P.Rem = Λ;

Def Var List_Length(i: $\mathbb{N}$): $\mathbb{N}$ = (
|List_Position.**Denote**(i).Prec| + |List_Position.**Denote**(i).Rem|);
Def Var Total_Length: $\mathbb{N}$ = ($\sum_{i=1}^{\text{List\_Position.LSN}}$ List_Length(i)) ;
Constraint Total _ Length $\leq$ Max_Total_Length;

Oper Advance(**upd** P: List_Position);
requires P.Rem $\neq \Lambda$;
ensures P.Prec $\circ$ P.Rem = @P.Prec $\circ$ @P.Rem **and** |P.Prec| = |@P.Prec| + 1;
//moves one position to the right

Oper Reset(**upd** P: List_Position);
ensures P.Prec = Λ **and** P.Rem = @P.Prec $\circ$ @P.Rem;
//moves to the beginning

Oper Length_of_Rem(**rest** P: List_Position): Integer;
ensures Length_of_Rem = (|P.Rem|);
//returns the length of the remainder

Oper Insert(**alt** New_Entry: Entry; **upd** P: List Position);
requires Total_Length < Max_Total_Length;
ensures P.Prec = @P.Prec **and** P.Rem = $\langle$@ New _ Entry$\rangle \circ$ @P.Rem;
//inserts a new entry at the current position

Oper Remove(**rpl** Entry_Removed: Entry; **upd** P: List_Position);
requires ⋯
ensures P.Prec = @P.Prec **and** @P.Rem = $\langle \text{Entry\_Removed} \rangle \circ$ P.Rem;
//removes the entry at the current position

Oper Advance_to_End(**upd** P: List_Position);
ensures ⋯
//moves to the end of the list

Oper Swap_Remainders(**upd** P, Q: List_Position);
ensures P.Prec = @P.Prec **and** Q.Prec = @Q.Prec
and P.Rem = @Q.Rem **and** Q.Rem = @P.Rem;
//exchange the remainders of 2 lists

Oper Clear_List(**clr** P: List_Position);
//initialize a new list

end One_Way_List_Template;

The specifications given here are mathematically precise. Depending on student background and department philosophy, one might choose to do informal, natural language specifications instead. However, precision can be achieved only with formal specs. Whatever form one chooses for the description of the list structure, students can follow it up with an implementation in C++ or Java or whatever language the class prefers. Most students do see very early in their academic career the use of requires and ensures clauses, some more formal than others.

A POSSIBLE IMPLEMENTATION

Included here is an implementation in C++. A header file (.h) for a class template is presented in which the structure chosen shows a node type and some pointer to node types, all in the private section, so that clients are not forced to worry about those pointers. The idea is that there will be a current pointer that keeps track of where in the list one is. The pre_current travels along behind it, making it easy to do inserts, and the front remains at the beginning of the list, allowing quick resets to take place.

```
#include <iostream.h>
template <class T>
class List
{
   public:
     List();
     // Create empty List
     void Advance();
          /*Requires |Rem|>0
            Ensures @Rem = x + Rem
                        Prec = @Prec + x*/
     T Peek();
          /*Requires List is not empty
            Ensures Peek = Rem.c*/
     int Length_of_Rem();
          /*Requires True
            Ensures Length_of_Rem = |Rem|  */
```

```
      void Reset();
           /*Requires True
             Ensures Prec is empty, Rem = @Prec + @Rem*/
      void Advance_to_End();
           /*Requires True
             Ensures Rem is empty, Prec = @Prec + @Rem*/
      void Insert(T);
           /*Requires True
             Ensures Prec = @Prec, Rem = Entry + @Rem*/
      T Remove();
           /*Requires Rem is not empty
             Ensures Prec = @Prec, @Rem = Remove + Rem*/
      void Swap_Rem(List);
           /*Requires True
             Ensures L1.Rem = @L2.Rem, L2.Rem = @L1.Rem*/
      T Swap_Prev_Entry(T);
           /*Requires |Prec| > 0
             Ensures Swap_Prev_Entry = x, where
             @Prec = A + x and Prec = A + Entry*/
      int  Length_of_Prec();
           /*Requires True
             Ensures Length_of_Prec() = |prec|*/
      void Clear_List();
           /*Requires True
             Ensures Prec is empty and Rem is empty*/
       private:
           struct node
           {
                T c;
                node * next;

           };
           node * prec;
           node * prerem;
           node * rem;
};
```

Also included is a .cpp file showing the constructor that sets up an empty list and the insert that shows how to put in a new entry wherever the current position is. It is significant that there is no need to have several cases. The same insert can be used whether one is at the beginning, the end, or somewhere in the middle of the list.

```
#include "List.h"

template <class T> List<T>::List()
//Constructor for empty list
{
      prec = new(node);
      prec->next = NULL;
      prerem = prec;
      rem = NULL;
}

template <class T> void List<T>::Advance()
//Advance to next Node in list...
//i.e. move first element of rem to end of prec
```

```
{
      if (rem != NULL)  //if rem is not empty
      {
            prerem = prerem->next; //advance rem pointers
            rem = rem->next;
      }
}

template <class T>
T List<T>::Peek()
//Returns element at current position
{
      if (rem != NULL)
            return rem->c;
      else
      {
            cout << "Error:  No value in rem to Peek!";
            return prerem->c;
      }
}

template <class T>
int List<T>::Length_of_Rem()
{
      node * temp;
      temp = rem;
      int n=0;
      while (temp != NULL)
      {
            n++;
            temp = temp->next;
      }
      return n;
}

template <class T>
void List<T>::Reset()
//starts at beginning of list
{
      prerem = prec;
      rem = prec->next;
}

template <class T>
void List<T>::Advance_to_End()
{
      if (prerem->next != NULL)  //checks to see if list is
empty, or already at end
      {
            while (prerem->next->next != NULL)  //advances until
rem is last
                  prerem = prerem->next;
            rem = prerem->next;  //rem points to last element
      }
}

template <class T>
```

```
void List<T>::Insert(T ch)
//Puts item in list at beginning of rem
{
      rem = new(node);//make new node for ch
      rem->c = ch;          //put ch in the new node
      rem->next = prerem->next;        //make new node point to
old rem
      prerem->next = rem;         //make prerem's node point to
new rem
}

template <class T>
T List<T>::Remove()
//Remove node and return item stored there
{
      if (rem != NULL)
      {
            T temp = rem->c;  //hold value to be returned
            rem = rem->next;      //make rem point to node after
one removed
            delete prerem->next;
            prerem->next = rem;
            return temp;
      }
      else
      {
            cout << "Error:  No node at current position to
Remove!";
                  T nil;
            return nil;     //return junk
      }
}
template <class T>
void List<T>::Swap_Rem(List L)
{
      prerem->next = L.rem;
      L.prerem->next = rem;
      rem = prerem->next;
      L.rem = L.prerem->next;
}
template <class T>
T List<T>::Swap_Prev_Entry(T ch)
{
      T temp = prerem->c;
      prerem->c = ch;
      return temp;
}
template <class T>
int  List<T>::Length_of_Prec()
{
      node * temp = prec;
      int n = 0;
      while (temp != rem)
      {
            temp = temp->next;
            n++;
      }
```

```
      return n;
}
template <class T>
void List<T>::Clear_List()
{
      node * temp = prec->next;
      node * oldtemp;
      while (temp != NULL)
      {
            oldtemp = temp;
            temp = temp -> next;
            delete oldtemp;
      }
      rem = NULL;
      prec->next = NULL;
      prerem = prec;
}
```

The usual kinds of pointer manipulation are used in the code, giving students the opportunity to learn how to use them. However, the incentive is to make sure this list template is well done and that it works correctly, with the idea of treating it as reusable an OTS (off the shelf) component in the future.

In the implementation a *Peek* operation has been added for convenience to allow looking at the current entry without removing it even though this could be done using the sequence of calls: remove, copy, and insert.

Next we turn to an example that uses the list template component, taking full advantage of the dynamic nature of it without needing to reinvent nodes and pointer. In fact, pointers do not show up at all.

STACKS

We show here a stack component which can grow or shrink as desired without using pointers directly. Suppose we want to satisfy the following specifications for a stack. As before, one might choose a less formal way to write the specifications, these being mathematically formal. Users can think about their stacks at the purely abstract level without regard to implementation details.

First we look at a specification: As with the list, our specifications are mathematically formal, but depending on the level of students and their background, one might write the description is some other form, including natural language, although in order to be precise, mathematical specifications are encouraged.

Concept Bdd_Stack_Template(**type** Entry; **val** Max_Depth: Integer);
uses Std_Boolean_Fac, Std_Integer_Fac, String_Theory;
requires Max_Depth > 0;

Type_Family Stack $\subseteq$ **Str**(Entry);
exemplar S;
constraints $|S| \leq$ Max_Depth;
initialization
ensures S = Λ;

Operation Push(**var** E: Entry; **var** S: Stack);
requires |S| < Max_Depth;
ensures S = <@E> o @S **and** Entry.**Is_Initial** (E);

Operation Pop (**var** R: Entry**; var** S: Stack);
requires S ≠Λ;
ensures @S = <R> o S;

Operation Is_Empty (**preserves** S: Stack): Boolean;
ensures Is_Empty = (S = Λ);

Operation Is_Full **(preserves** S: Stack): Boolean;
ensures Is_ Full = (|S| = Max_Depth);

Operation Clear(**var** S: Stack);
ensures S = Λ
end Bdd_Stack_Template;

Here we define a stack to be a list_position. So a client who is looking for reusable components would see the specifications for a list position and choose it to act as the basis for creating the type stack. To Push an entry, say E, all one needs to do is call S.Reset, followed by S.Insert(E), where S has been declared to be a stack, via syntax Stack<T> S where T is whatever type is desired for the entries. There is no need for setting up nodes or references or moving links. All of those activities are taken care of by the list component. All the stack implementer needs to do is understand the specs for the list and use them accordingly.

A Pop is achieved by first doing a Reset and then a Remove. To check whether the stack is empty, one can do a Reset and then call Length_of_Rem.

What is equally important is that now a client of the stack will not need to know anything about lists or pointers, because the client will simply declare a stack and make appropriate calls to Push and Pop, leaving the implementation details to the writer of the stack component.

This approach results in complete adherence to the principles of information hiding and separation of concerns.

QUEUES

Using the same list component we can implement queues as well. A typical specification for the type Queue includes operations Enaqueue, Dequeue, Length, and possibly a Peek to see what the first element is without having to remove it. In the interest of space, such specs are not included here, since both lists and stacks have been developed in full. Conceptually, a queue is a string whose operations allow access at both ends, one end for putting in new entries, and the other for removing them, hence providing a FIFO (first in first out) capability, as opposed to the LIFO (last in first out) given by the stack.

A programmer who wants to do an implementation that allows dynamic growing and shrinking need not become involved with any pointers or references, but rather can simply set up the type queue as a list_position, similarly to the way we did the stack template.

To do an enqueue, one can make a call to Reset followed by an Insert. To dequeue, we make a call to Advamce_to_End followed by Insert. All of the pointer work was done in the list_position, leaving the queue writer to reason at the appropriate level of abstraction, rather than needing to worry about setting up nodes or links.

A client of the queue component will simply call enqueues and dequeues, doing all thinking and reasoning at that level without any need to worry about low level details.

Perhaps most important is the fact the once the list_position has been correctly implemented, any clients can do their testing and debugging in terms of stacks or queues or whatever other component they are working on, without any need to check the pointer details, since those things have already been checked for correctness before being made available for use as OTS (Off The Shelf) software.

INSERT SORT

As still another example, we consider what is sometimes called "an ordered list," and treated as a different component from a list. Here we see that such a distinction is not necessary at all, since one can produce an ordered list without ever using a pointer or reference. We can simply use the list_position as defined, and as new items are inserted, we advance along the list until we find the position we want to satisfy the ordering requirement and then call Insert.

For example, suppose we have the following characters we want to put into a sorted list: t, a, c, x, e, w.

First we call Insert on t, creating a list of one entry that conceptually looks like (Λ, t). Next, to insert the a, we use a loop to Advance until we find an entry the is alphabetically past the a. In this case, that happens right away. As soon as we see the t, we stop the advancing loop and do an Insert, resulting in (Λ, at). Next we Reset and then Advance until we find an element past c. This happens at the position, (a, t). Calling Insert results in (a, ct). Now Reset and continue with the x. This causes us to advance until we reach the end of the list where we do an insert, getting (act, x). We continue in this way until all of the elements have been entered.

There is no need to even think about references or pointers, since all of those details are in the list_position component. The programmer can concentrate on the task at hand, namely putting values in alphabetical order (or whatever order the type requires), the appropriate level of abstraction for this problem.

Here are two versions for implementing insert sort, one with character entries, one with integer entries:

```
#include <iostream.h>
#include "List.cpp"
```

```
void main()
{
////////////////////////////
//Insert sort for characters//
////////////////////////////
     cout << "Enter a string of characters terminated with
'0'." <<
                                                          // endl;
     List<char> L;   //Declare a list of characters named L
     char c;
     cin >> c;
     L.Insert(c);          //Puts first value in L
     cin >> c;
     while (c != '0')  //Waits for 0 input to stop
     {
          if (c < L.Peek()) L.Reset();     //if c is too
small, reset
                                  //and start from the beginning
          while (L.Length_of_Rem() && (c > L.Peek()))
                     L.Advance();    //advance to the correct
spot

          L.Insert(c);   //now that we are at the correct
postition,
                                        //insert the character
          cin >> c;       //and take the next character
     }
     L.Reset(); //Reset the list so we can output from the
beginning
     cout << "The sorted data is as follows:" << endl;
     while (L.Length_of_Rem())        //while Rem is not empty
     {
          cout << L.Peek() << ' '; //output each element of
the list
          L.Advance();
     }
     cout << endl;

////////////////////////////
//Insert sort for integers//
////////////////////////////

     cout << "Enter integers, terminated by 0." <<endl;
     List<int> B;          //declare a list of integers named
B
     int d;
     cin >> d;
     B.Insert(d);          //Puts first value in B
     cin >> d;
     while (d != 0)  //Waits for 0 input to stop
     {
          if (d < B.Peek()) B.Reset();
          while (B.Length_of_Rem() && (d > B.Peek()))
                                  B.Advance();
          B.Insert(d);
          cin >> d;
```

```
      }
      B.Reset();
      cout << "The sorted data is as follows:" << endl;
      while (B.Length_of_Rem())
      {
            cout << B.Peek() << ' ';
            B.Advance();
      }
      cout << endl;

} //end
```

SUMMARY

In order to encourage separation of concerns, information hiding, and the ability to reason about programs at the appropriate level, a generic, all purpose list component has been introduced. Several other structures, such as stacks, queues, and ordered lists have been built using the basic list component. In the broader sense, the idea of carefully planning reusable components is introduced and promoted. Using this approach teaches students to do careful planning so that they are not constantly reinventing the same ideas over and over, but rather they are thinking about constructing whatever they need in terms of reusable components that have already been shown correct.

One of the best results of using this approach is that students who write some basic components in early courses find it beneficial to use them in later courses, allowing them to concentrate on whatever new concepts they are learning in those courses, rather than needing to go back and reinvent ideas they have already mastered.

In case there is concern that students should be reinforcing old ideas, there are two ways to look at that. First, if one has the goal of reinforcing, it is quite reasonable to disallow the use of formerly written components. However, if students are writing new components for any ideas (old or new), it is still important that they do so with the idea of promoting correctness and reuse.

The second significant result of using such an approach is that students really do master the ideas, probably more deeply than with other approaches, because they have to give so much thought to how they can set up their object classes to achieve desired information hiding and reuse potential.

Employers of students who have been educated using this approach have given (in many cases unsolicited) feedback, indicating their great appreciation of those students and their contributions to whatever company they are working for.

REFERENCES

Cormen, T., Leiserson, C., Rivest, R., Stein, C. **Introduction to Algorithms**, MIT Press, Cmbridge, Massachusetts, 2001.

Pressman, Roger, **Software Engineering: A Practitioner's Approach**, McGraw-Hill, 2005.

Sahni, Sartaj, Data Structures, **Algorithms, and Applications in C++**, McGraw-Hill, 1998.

Sommerville, Ian, **Software Engineering**, Addison-Wesley, 2004.

Stubbs, D., Webre, N., **Data Structures with Abstract Data Types**, Brooks/Cole, Pacific Grove, California.

Weiss, Mark, **Data Structures & Problem Solving using Java**, Addison Wesley, New York, NY, 2002.