3. An Abstraction for Acyclic Pointer Structures

In this section we introduce Chain_Position, an abstraction for acyclic pointer structures. Understanding this abstraction is paramount for understanding the discipline. Why is this abstraction so important? It allows us to get pointers right once and for all, eliminating the troublesome details associated with using programming language pointers when implementing acyclic linked structures.

Experience shows that the abstraction may be difficult to understand, so we begin with a simple example, working up from there. Suppose one is developing a program that requires a singly linked list for storing a character string. There is one problem: the programming language being used does not support pointers. What can be done? Simulate the pointers using an array for storing the characters while maintaining a parallel array for storing the simulated pointer link to the next character. This is common in FORTRAN programs, and is taught in some introductory data structures courses and standard texts (see Horowitz[10]). For example, suppose we have the list (a, x, r). An implementation using simulated pointers and parallel arrays might look like the following:

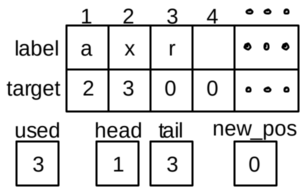


Figure 1 — Simulated pointers using parallel arrays.

To access the first item in the list, ‘a’, use the value stored in the variable head to index into the label array. To find the next item in the list, simply index into the target array and use the value stored there as the index into the label array. The end of the list is reached when the value in the target array is zero.

Abstracting from this implementation leads to a specification for the desired generic abstract data type. There are actually two mappings at work in this example, a mapping from integers to characters, and a mapping from integers to integers. These mappings can be viewed as mathematical functions, label and target, having the following mathematical form:

label:	integer $\rightarrow$ Item
target:	integer $\rightarrow$ integer

By definition label and target are both total functions that form part of a complete mathematical model of the simulated pointer structure. In examples we show only ordered pairs for which the domain value is of interest. For the above example, the functions have the following values:

label =	{(1, a), (2, x), (3, r)}
target =	{(1, 2), (2, 3), (3, 0)}

The Abstraction in the Form of a C++ Template Class

The abstraction Chain_Position presented below (see Figure 2) is based on the two functions label and target. It exports a program type called Position (modeled by the integers used as the domains of the functions), and it exports operations for manipulating the functions (i.e., for evaluating and changing them). The template is parameterized by the type Item. The specification has three parts: C++ code; mathematical specifications (in C++ comments beginning with //!); and English explanation (in C++ comments beginning with only //). The mathematical specifications are similar to those found in Pittel[14], with modifications to facilitate the presentation and to correspond with the C++ implementation. Upon first reading, think of the above example and use the associated explanations as an aid to understanding the specification. Then move on to the example client program found in the next section, referring back to the specification when necessary.

We have some experience introducing Chain_Position to programmers in the classroom The students’ first encounter with Chain_Position was difficult. However, with some explanation of the specs along with an example similar to the one found in the next section, the students became comfortable with the abstraction and easily were able to use it to implement two different linked structure packages (queue and one-way list).

```
template <class Item>
class Chain_Position
//!             mathematics
//!                   math variables

//!                         used: integer
//!                         label: function from integer to math[Item]
//!                         target: function from integer to integer
//    Conceptually Chain_Position maintains these three internal “state” variables.  The exported
//    operations manipulate these variables as well as their actual parameters.  Chain_Position’s
//    implementation is not obligated to represent these variables explicitly,
//    as they are mathematical abstractions, not actual C++ variables.

//!                   initially  "used = 0 and
//!                         for all i: integer (Item.init (label (i))  and  (target (i) = 0))"
//    Conceptually Chain_Position dispenses unused positions beginning at integer number one.
//    Positions are dispensed to the client via the operation Label_New (see below).  Each time
//    a position is dispensed, the math variable used is incremented by one.  There is no danger of
//    eventual overflow because used is a mathematical integer, not a C++ integer.


{
public:
      Chain_Position ();
//!         post: self = 0  and  used = #used  and
//!               label = #label  and  target = #target"

      ~Chain_Position ();

      void operator &= (alters Chain_Position& rhs);
//!         alters: self
//!         post: "self = #rhs  and  rhs = #self
//!               and  used = #used  and  label = #label and target = #target"

      Chain_Position& operator = (preserves Chain_Position& x);
//!         alters: self
//!         post: "self = x  and  used = #used  and  label = #label and target = #target"
//    Conceptually this operation allows a client to create a copy of a position x.  None of the
//    internal mathematical variables (i.e., used and target) are changed.
//    Note: This is similar to aliasing if pointers were being used.  However this is the
//    only way in which a client can create an alias.  Implementations of Chain_Position can
//    take advantage of this situation so that //  dangling references and storage
//    leaks are never created.

      void Label_New (consumes Item& x);
//!         alters: self
//!         post: used = #used + 1  and  self = used  and
//!               for all i: integer (i != self implies label (i) = #label (i))  and
//!               label (self) = #x  and
//!               target = #target
//    Conceptually this operation allocates the next unused integer to be used as a Position value.
//    It alters the label function, mapping the new Position p to the Item x.  The new Position’s
//    target is 0 because target initially maps every integer to 0.  The operation also consume
//    x; i.e., x is changed to an initial value for the type Item.

      Item& operator [] (preserves Integer index);
//!         preserves: self
//!         pre:  "self != 0"
//!         post: "result = &label (self)"
//    Conceptually this operation allows a client access the label function at Position p.
//    The parameter index is not used, but is required by C++.  Neither used nor target
//    is changed.

      void Swap_Label (alters Item& x);
//!         preserves: self
//!         pre:  "self != 0"
//!         post: "used = #used  and
//!                   for all i: integer (i != self implies label (i) = #label (i))  and
//!                   label (self) = #x  and  x = #label (self)  and
//!                   target = #target"
//   Conceptually this operation allows a client to change the label function at Position p and
//   simultaneously to obtain the former label at Position p, by swapping.  Neither used nor target
//   is changed.

     void Apply_Target (void);
//!        alters: self
//!        pre:  "self != 0"
//!        post: "self = target (#self)  and
//!                   used = #used  and  label = #label  and  target = #target"
//   Conceptually this operation applies the target function to self and sets self to the value
//   produced by the application.  This is similar to p = p->next; when using raw C++ pointer types.

     void Swap_With_Target (alters Chain_Position& new_target);
//!        preserves: self
//!        pre:  "self != 0 and self != new_target and
//!              there does not exist k :integer, (k = 0 and (target^k(new_target) = self))
//!        post: "used = #used  and  label = #label  and
//!              for all i: integer (i != self implies target (i) = #target (i)) and
//!              target(self) = #new_target  and  new_target = #target (self)"
--  Conceptually this operation allows a client to alter the target function by swapping target(self)
--  with new_target.  Neither the used nor label functions are changed.
--  Note: The notation target^k(new_target) denotes the iterated application of target
--  to new_target, k times.  For example, target^2(new_target) = target(target(new_target)).
--  The pre condition must be met so that no circular structures will be created!  In other
--  words, the target function is a nilpotent function.

     Boolean operator == (preserves Chain_Position& rhs);
//!        preserves: self
//!        post: "result = true iff (self = rhs)
//!              and  used = #used  and  label = #label and target = #target"
--  Conceptually in raw C++ this is equivalent to testing to see if two pointer variables hold
--

     Boolean operator != (preserves Chain_Position& rhs);
//!        preserves: self
//!        post: "result = true iff (self != rhs)
//!              and  used = #used  and  label = #label and target = #target"

     Boolean Is_Zero (void);
//!        preserves: self
//!        post: "result = true iff (self = 0)
//!              and  used = #used  and  label = #label and target = #target"

private: // Prohibit
     Chain_Position (Chain_Position& x) {};

private:  // Representation

};   // Chain_Position
```

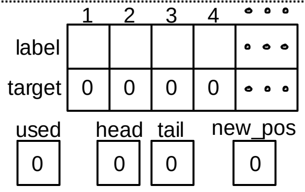
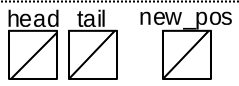
4. A Client of the Chain Position

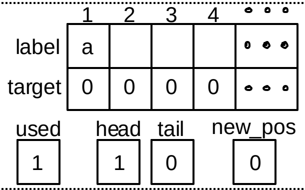
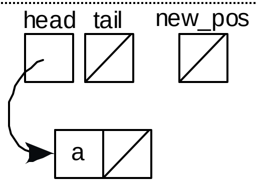
This section demonstrates the use of the Chain_Position introduced in the last section by providing two clients: a C++ procedure that instantiates Chain_Position and a C++ template class layered on Chain_Position.

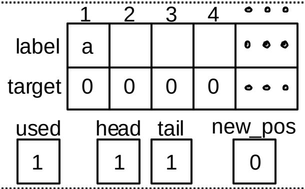
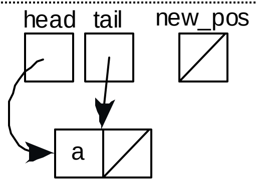
A Procedure to Create a Simple List

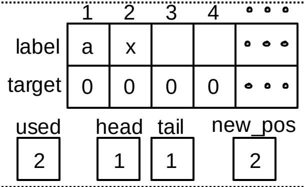
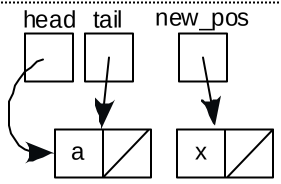
The procedure of Figure 3 creates the example list, (a, x, r), from Section 3. The reader should refer to the remainder of Figure 3 for three different views of the program's execution. Figure 3 has nine rows and four columns. The nine rows correspond with the nine lines in the program tagged with the comment "// #." Column one contains the line number; column two contains an illustration of the abstract state; column three illustrates a simulated pointer representation; column four shows a standard representation using pointers and nodes with "next" fields.

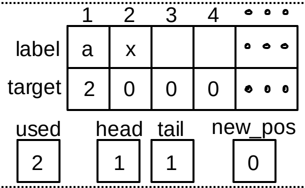
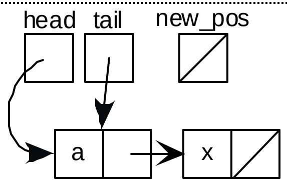
```
#include "wrapper.h"
#include "ChnPos\ChnPos.hpp"

typedef Chain_Position_1 <Character> Character_List_Facility;

void main (void)
{
    Character_List_Facility head, tail;
    Character_List_Facility new_pos;
    Character c;
                                                   // 1
    c = 'a';
    head.Label_New (c);                            // 2
    tail = head;                                   // 3

    c = 'x';
    new_pos.Label_New (c);                         // 4
    tail.Swap_With_Target (new_pos);               // 5
    tail.Apply_Target ();                          // 6

    c = 'r';
    new_pos.Label_New (c);                         // 7
    tail.Swap_With_Target (new_pos);               // 8
    tail.Apply_Target ();                          // 9

    // Walk linked list, display characters
    object Character_List_Facility p;
    p = head;
    while (!p.Is_Zero ()) {
        cout << p[0] << " ";
        p.Apply_Target ();
    } // end while
    cout << endl;
} // main
```

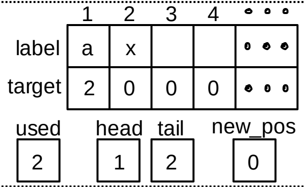
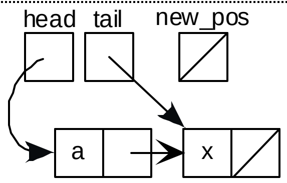
1	head = 0 tail = 0 new_pos = 0
	
	used = 0
	label: { }
	target: { }
			

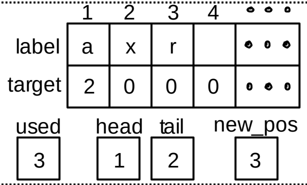
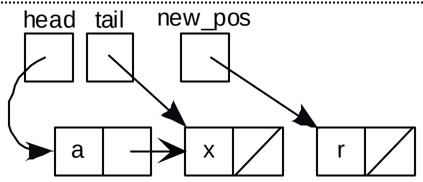
2	head = 1 tail = 0 new_pos = 0
	
	used = 1
	label: { (1, a) }
	target: { (1, 0) }
		 1 2 3 4 label a target 0 0 0 0 used 1 head 1 tail 0 new_pos 0	

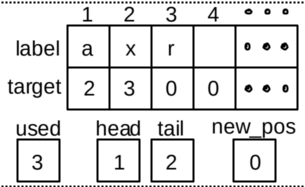
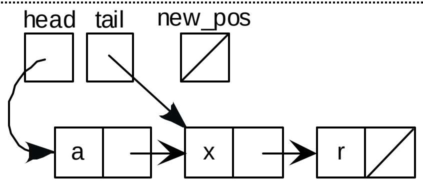
3	head = 1 tail = 1 new_pos = 0
	
	used = 1
	label: { (1, a) }
	target: { (1, 0) }
		 1 2 3 4 label a target 0 0 0 0 used 1 head 1 tail 1 new_pos 0	

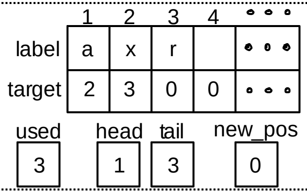
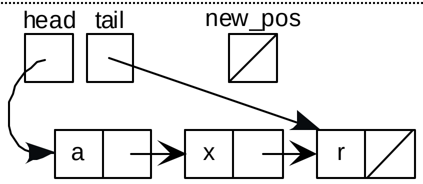
4	head = 1 tail = 1 new_pos = 2
	
	used = 2
	label: { (1, a), (2, x) }
	target: { (1, 0), (2, 0) }
		 1 2 3 4 label a x target 0 0 0 0 used 2 head 1 tail 1 new_pos 2	

5	head = 1 tail = 1 new_pos = 0
	
	used = 2
	label: { (1, a), (2, x) }
	target: { (1, 2), (2, 0) }
		 1 2 3 4 label a x target 2 0 0 0 used 2 head 1 tail 1 new_pos 0	

6	head = 1 tail = 2 new_pos = 0
	
	used = 2
	label: { (1, a), (2, x) }
	target: { (1, 2), (2, 0) }
		 1 2 3 4 label a x target 2 0 0 0 used 2 head 1 tail 2 new_pos 0	

7	head = 1 tail = 2 new_pos = 3
	
	used = 3
	label: { (1, a), (2, x), (3, r) }
	target: { (1, 2), (2, 0), (3, 0) }
		 1 2 3 4 label a x r target 2 0 0 0 used 3 head 1 tail 2 new_pos 3	

8	head = 1 tail = 2 new_pos = 0
	
	used = 3
	label: { (1, a), (2, x), (3, r) }
	target: { (1, 2), (2, 3), (3, 0) }
		 1 2 3 4 label a x r target 2 3 0 0 used 3 head 1 tail 2 new_pos 0	

9	head = 1 tail = 3 new_pos = 0
	
	used = 3
	label: { (1, a), (2, x), (3, r) }
	target: { (1, 2), (2, 3), (3, 0) }
		 1 2 3 4 label a x r target 2 3 0 0 used 3 head 1 tail 3 new_pos 0	

```
// Filename: Stk1.hpp
#pragma once
#include "CHNPOS\ChnPos.hpp"

template <class Item>
class Stack_1
{
public: // Standard Operations
        Stack_1 ();
        ~Stack_1 ();
        void Clear (void);
        void operator &= (Stack_1& rhs);

public: // Stack Specific Operations
        void Push (consumes Item& x);
        void Pop (produces Item& x);
        Item& operator [] (preserves Integer index);
        Integer Size (void);

private: // Representation
        typedef Chain_Position_1 <Item> Item_Chain_Position;
        Item_Chain_Position top;

private: // Prohibited Operations
        Stack_1& operator = (const Stack_1& rhs);
        Stack_1 (const Stack_1& s);
};
```

```
template <class Item>
Stack_1<Item>::Stack_1 ()
{} // Stack_1

//---------------------------------

template <class Item>
Stack_1<Item>::~Stack_1 ()
{} // ~Stack_1

//---------------------------------

template <class Item>
void Stack_1<Item>::operator &= (
           Stack_1& rhs
       )
{
   top &= rhs.top;
} // operator &=

//---------------------------------

template <class Item>
void Stack_1<Item>::Clear (void)
{
   top.Clear ();
} // Clear

//---------------------------------

template <class Item>
void Stack_1<Item>::Push (Item& x)
{
   Item_Chain_Position new_pos;

   new_pos.Label_New (x);
   new_pos.Swap_With_Target (top);
   top = new_pos;
} // Push
```

```
template <class Item>
void Stack_1<Item>::Pop (Item& x)
{
   Item_Chain_Position old_pos;

   old_pos = top;
   top.Apply_Target ();
   old_pos.Swap_Label (x);
} // Pop

//---------------------------------

template <class Item>
Item& Stack_1<Item>::operator [] (
           Integer index
       )
{
   return (top[0]);
} // operator []

//---------------------------------

template <class Item>
Integer Stack_1<Item>::Size (void)
{
   Item_Chain_Position p;
   Integer c;

   p = top;
   while (!p.Is_Zero ()) {
      p.Apply_Target ();
      c++;
   } // end while
   return (c);
} // Size
```