To:
Tenouk

C LAB WORKSHEET 15
C Pointers, Arrays, Functions, struct Part 1

1. Some of the typical application of C/C++ pointers, the link and linked list.
2. The mystery of the indirection operator.

3. Tutorial references that should be used together with this worksheet are: arrays 1, array 2, pointers 1, pointer 2, pointer 3, functions 1, function 2, function 3, function 4 and structs.

Linked List Using Arrays

This should be an introduction to the pointers applications. In C++ you may use classes available in the Standard Template Library, STL. The following example show the concept of pointers easier to understand because it uses an array called Link[ ], which stores array indexes. Just as each **memory location** has **an address** associated with it, so does each **array element** have an index associated with it.
To follow the steps, create a new **Win32 console mode application**, add a source file and set the project to be compiled as C code. Build and run the following program example.

```
#include <stdio.h>
// The array size
#define SIZE 6

// Function prototype
void LinkThem(int [ ], int [ ], int);

void main(void)
{
    char Composer [SIZE][20] =
    {"    ", "Bach", "Beethoven", "Chopin", "Mozart", "Schubert"};
    int Requests[SIZE] = {0, 145, 150, 70, 110, 95};
    int i, Link[SIZE];

    printf("Original list.\n");
    printf("[i]\tComposer[i]\tRequests[i]\n");
    printf("===\t========\t========\n\n");
    for(i = 0; i <= 5; i++)
        printf("[%d] %9s %15d\n", i, Composer[i], Requests[i]);
    printf("\n");

    // call LinkThem() function, create a linked list
    LinkThem(Requests, Link, SIZE);

    // print out the requests in order, using the links
    // we do some output formatting here...
    printf("\nA sorted linked list...\n");
    printf("L[i]\tComposer\tRequests\n");
    printf("====\t========\t========\n\n");
    for(i = Link[0]; i != 0; i = Link[i])
        // right justified 9 places, then right justified 15 places...
        printf("L[%d]  %9s%15d\n", i, Composer[i], Requests[i]);
}

    // LinkThem() will make a linked list so that the Requests are in order
    // to call this function R[ ] and Max should be initialized and L[ ] is not
    // upon completion, R[ ] and Max are unchanged and L[ ] is a linked list
    // that logically ordered the Requests
void LinkThem(
    int R[ ],       // the number of Requests for each Composer.
    int L[ ],       // will become a linked list.
    int Max)    // this minus 1 is the number of Requests to be ordered.
{
    int i,            // the index of the next Request to be placed in list.
```

```
        j,          // the index used to find the correct place in the list.
    Prevj;          // the previous value of j.
    L[0] = 1;       // the previous value of j.
    L[1] = 0;       // initialize the linked list for the first request.

    // R[i] is the next request to be placed in the list
    for(i = 2; i <= Max - 1; i = i + 1)
    {
        Prevj = 0;
        // find the index j where L[i] should point.
        for(j = L[Prevj]; j != 0; j = L[j])
        {
            printf("Check: R[%d] = %d, R[%d] = %d, If R[%d] < R[%d] insert...\n", i, R[i], j, R[j], i, j);
            if(R[i] < R[j])
            {
                // found the place to insert L[i] in the list.
                printf("Inserting L[%d]\n", i);
                L[Prevj] = i;
                L[i] = j;
                // get the next request to be added to the list.
                break;
            }
            // start traverse/moving forward in the linked list
            printf("Else, move forward, check again...\n");
            Prevj = j;
        }

        // if the point to insert L[i] is not found at all add it to the end of the linked list.
        if(j == 0)
        {
            printf("Point to insert L[%d] not found!\n", i);
            L[i] = 0;
            L[Prevj] = i;
        }
    }
}
```

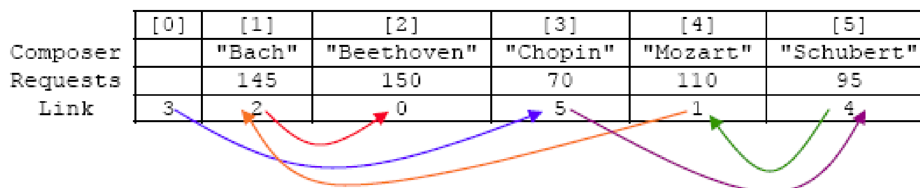
```
C:\WINDOWS\system32\cmd.exe
Original list.
[i]      Composer[i]      Requests[i]
===      ========         ========

[0]                            0
[1]        Bach              145
[2] Beethoven                150
[3]      Chopin               70
[4]      Mozart              110
[5]   Schubert                95

Check: R[2] = 150, R[1] = 145, If R[2] < R[1] insert...
Else, move forward, check again...
Point to insert L[2] not found!
Check: R[3] = 70, R[1] = 145, If R[3] < R[1] insert...
Inserting L[3]
Check: R[4] = 110, R[3] = 70, If R[4] < R[3] insert...
Else, move forward, check again...
Check: R[4] = 110, R[1] = 145, If R[4] < R[1] insert...
Inserting L[4]
Check: R[5] = 95, R[3] = 70, If R[5] < R[3] insert...
Else, move forward, check again...
Check: R[5] = 95, R[4] = 110, If R[5] < R[4] insert...
Inserting L[5]

A sorted linked list...
L[i]     Composer         Requests
====     ========         ========

L[3]      Chopin             70
L[5]     Schubert            95
L[4]      Mozart            110
L[1]        Bach            145
L[2]   Beethoven            150
Press any key to continue . . .
```

The following Figure shows the names of composers who are played on our radio station.

	[1]	[2]	[3]	[4]	[5]
Composer[]:	"Bach"	"Beethoven"	"Chopin"	"Mozart"	"Schubert"
Requests[]:	145	150	70	110	95

How can we keep both the Composers and the number of Requests in order?

They are stored in an array called Composer[ ], starting at an index of 1 instead of 0. Element 0 is not used. Below each composer is the number of listener requests we have received. The requests are stored in an array called Requests[ ] that is parallel to the Composer[ ] array. Sometimes we want to have the composer names in order and other times we would like to have the number of requests in order. The problem is we can't have both arrays in order.
One way we can keep both the composers and their requests in order is by using a **linked list**, a list that has been linked together. In the following Figure we have added a new array called Link[ ]. This array starts at index 0. The value stored in Link[0] always tells where the smallest number of requests is stored. Link[0] is 3, which means that Requests[3], with 70, has the smallest number of requests.



	[0]	[1]	[2]	[3]	[4]	[5]
Composer		"Bach"	"Beethoven"	"Chopin"	"Mozart"	"Schubert"
Requests		145	150	70	110	95
Link	3	2	0	5	1	4

A linked list is one way we can keep both the Composer and the number of Requests in order.

To find the next smallest number of requests, we access the Link[ ] slot parallel to Requests[3]; that is Link[3]. This slot has a 5 in it, which means that Requests[5] is the next smallest number of requests. It is 95. To find the next smallest number of requests, we look at Link[5]. This is 4. Requests[4] or 110 is the next smallest number. Then Link[4] points to 1. Requests[1] is 145 and Link[1] contains 2, meaning that Requests[2] is the next one. Lastly Link[2] is 0. Reaching a 0 means that we are at the end of this ordered chain of indexes.
A proper term for this structure is a “linked list”. Each slot in the Link[ ] array specifies which is the next slot in this linked list. We also need to know where the beginning of the linked is. We use Link[0] to tell us where the beginning of the list is. We also need to know when the list ends. Once a 0 is found in Link[ ], then we know this is the end. If Link[0] is 0, then there are no elements in the list and the linked list is empty. This would be the case if no elements have been added to the list.

As shown in the first Figure, the two arrays Composer[ ] and Requests[ ] are initialized in the program example. The first or the 0th elements of these arrays are not used. Link[ ] is not initialized. This array will become a linked list as shown in the second Figure once we call the function LinkThem(). This is the first executable statement in main(). main() will pass the Link[ ] and Requests[ ] arrays. When arrays are passed to a function, that function may alter its contents and those changes will be reflected in the calling function's arrays. That is what will happen here. LinkThem() will not change the Requests[ ] array; it will be used only to ‘look’ at its numbers. However, the function will place array indexes in the Link[ ] array, making it a linked list. That is the fun part and we will save it for later use. First, let us say that the links have been properly established in LinkThem() and now we want to print out the responses in order as done in main().
Let us follow the following Figure which shows the tracechart for the loop of the linked list that appears in the previous second Figure.

Statement which changes "i".	i	i != 0?	printout	
i = Link[0]:	3	True		
			Chopin	70
i = Link[3]:	5	True		
			Schubert	95
i = Link[5]:	4	True		
			Mozart	110
i = Link[4]:	1	True		
			Bach	145
i = Link[1]:	2	True		
			Beethoven	150
i = Link[2]:	0	False		

Using the linked list to print the requests in order

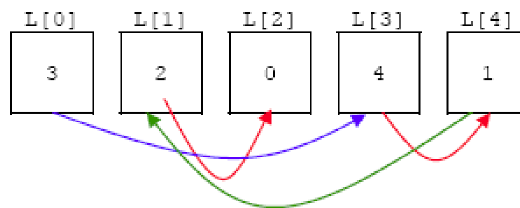
In the first statement of the for loop, i = Link[0], a 3 is assigned to i. The condition, i != 0?, is found to be true, so the printf() is executed. This prints Composer[3] or "Chopin" and Requests[3] or 70. Now we go up to the for statement. i = Link[i] reduces to i = Link[3] since i is 3. After this statement, the number in Link[3], which is 5, becomes the new value of i. 5 isn't equal to 0 so we go back into the body of the loop. Here, "Schubert" and 95 are printed and we go back up to the for loop.

i is 5 so Link[5] is 4. The new value of i becomes 4. Here, "Mozart" and 110 are printed and we go back up to the for statement. i is 4 so Link[4] is 1. The new value of i becomes 1. Here, "Bach" and 145 are printed and we go back up to the for statement.
Likewise, i = Link[1] makes i equal to 2. Printing the composer and requests for slot number 2 shows "Beethoven" and 150. We return to the for statement and i = Link[2] assigns 0 to i. This stops the loop and all the composers and their requests are printed in ascending order. If we had wanted to print their names in order, then a loop that doesn't use the Link[] array would suffice.

LinkThem()

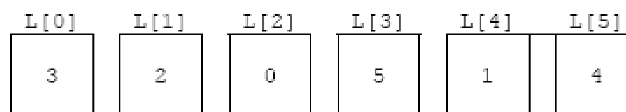
Now back to the LinkThem(). This function receives R[ ], L[ ] and Max as arguments from main(). R[ ] is the same array as Requests[ ] and L[ ] is the same array as Link[ ]. Keeping the brackets empty with the arrays allows us to use a variable length array. One time we can send 5 element arrays and another time send 450 element arrays. One subtracted from the number in Max, gives us the number of elements in the arrays. Initially, R[ ] is already set up, but L[ ] is empty.
The program example code places each element of R[ ] into a linked list, one at a time. Before any loop is encountered, the first element of L[ ] is placed in the linked list. To do this, L[0] points to 1 (or has a 1 for its value) and L[1] is 0, meaning that we are at the end of the list. We have only one element, 145, in the list right now.
Now we go into the i loop. When i is 2, we are placing element 2 in the list; when i is 3, we are placing element 3 in the list and so on. Right now i is 2. 2 is not less than 6, so we go into the body of the i loop. Here Prevj is made 0 and in the j loop, j is made equal to the next link in the list (or 1). j is not equal to 0 so we enter the j loop.
In the j loop, we go down the linked list to see where R[i] should go in the list. Once we find out where it goes, then L[i] is placed in the link. Prevj is needed to know the position of the previous link.
Inside the j loop, we see that R[2] < R[1]? is false. 150 isn't less than 145. Then we march down the link, making Prevj equal to j and advancing j to the next link. However, it is 0. We have gone all the way to the end of the linked list and the number we want to link in, 150, is larger than any number in the list. We have to add 150 at the end of the list.
The j loop is complete. Since j is equal to 0, we make L[2] equal to 0 or mark it as the end of the list and make L[3] point to 1, which is the value of j.
The loop continues as the next R[i] is placed in the linked list. Each time a number is added, the list becomes bigger and continues to be a valid linked list. That is, the beginning of the list is pointed to by L[0] and each successive number is placed in the proper place in the list. The last link has a 0 in it, marking it as the end of the chain. You may make a tracechart for this program to document the details.
Let us see how the last number, 95, is inserted into the list. The outer loop starts with i being 5. Prevj is 0 and j is L[Prevj] or 3. R[i] or 95, isn't less than R[j] or 70, so we go through the loop again.
Now Prevj is 3 and j is 4 as shown in the following Figure (a). R[i] or 95 is less than R[j] or 110, so we insert the slot five by marking L[3] point to 5. An alternative would be to make the number following 70 a 95 and to have L[5] point to 4 or to make the number following 95 a 110 as shown in Figure (b).

```
Just
before the
insertion,
i is 5
Prevj is 3
j is 4
```



(a)

```
We realize that we have found the
insertion point when
R[i] is < R[i] or 95 < 110
```




```
Hence, we do the insertion.
L[Prevj] becomes i or L[3]
becomes 5 and L[i] becomes j or
L[5] becomes 4.
```

(b)

The Indirection Operator, &

In the Pointers worksheet we learned about pointers and how they store computer memory addresses, how they are similar to arrays and so on. In this and the next section we will learn how to use pointers in a typical C/C++ programming language and the relation with arrays, functions and structure data type. A pointer that stores an address needs **a method to access the location**

of that address. Suppose we have the following declaration.

```
int i = 40, j = 60, *ptr1 = &i, *ptr2 = &j;  // Line 1
```

Then we know that i and j are initialized to 40 and 60, respectively. By referring to the following Figure, we also know that ptr1 contains the address of i, which is A000 and ptr2 contains the address of j, which is A002.

```
Address   Value   Name
        |       |
 A000   |  40   | i
 A002   |  60   | j
 A004   | A000  | ptr1
 A008   | A002  | ptr2
        |       |
```

ptr1 equals A000 and *ptr1 equals 40.

```
printf("ptr1 = %p, *ptr1 = %d\n", ptr1, *ptr1);    // Line 2
```

Then ptr1 will print the address of A000 and *ptr2 will print out the contents of location A000, namely, 40. The asterisks in Line 1 are used to declare pointers, ptr1 and ptr2. The asterisk in Line 2 is used as an **indirection operator**. What this means is that, when we print *ptr1, we are **dereferencing** ptr1 and printing not the value of ptr1 but the **value of the location** whose address is in ptr1. Therefore, from the previous Figure we see that ptr1 is A000 while *ptr1 is 40. After the completion of the following statement:

```
*ptr1 = *ptr + *ptr2 + 1;
```

i will contain 101 because *ptr1 is 40 and *ptr2 is 60 and adding those numbers to 1 gives 101. This number is then assigned to the location of A000 or *ptr1.
The terminology says that the location pointed to by ptr1 is assigned the number 101. The location ptr1 is not assigned a new number, but the **location pointed to** by it, namely, A000, is. We can also say that the location whose address is in ptr1 is assigned a new number.

```
Address   Value    Name
        |       |
 A000   |   3   | i
 A002   |   7   | j
 A004   | A000  | array[0]
 A008   | A002  | array[1]
        |       |
```

An array of pointers.

Arrays

We can store pointers in arrays. Consider the following code:

```
int i = 3, j = 7, *array[2];
array[0] = &i;
array[1] = &j;
```

You can see from the previous Figure that array[ ] is an array of pointers to integers. It has the addresses of i and j. The content of array[1] is A002, but the value of this expression, *array[1], is 7, array[ ] is said to be an array of pointers.

Structure, struct

In the following code, we have defined a new structure with two parts; value and ptr. value stores a floating point number and ptr stores the addresses of variables which are of type struct node. x and y are variables of type struct node and ptr is a pointer to such variables.

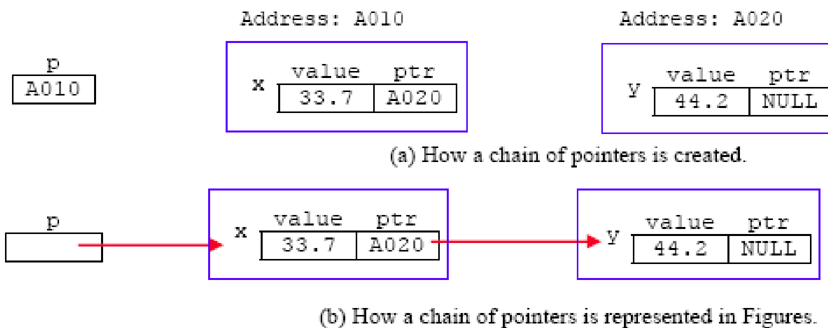
```
struct node
{
  float value;
  struct node *ptr;
};

struct node x, y, *p;
```

If we execute the following statements:

```
p = &x;
```

```
x.ptr = &y;
y.ptr = NULL;
```

We will create a chain of pointers as shown in the following Figure (a). The address of x, which is A010, is stored in p; the address of y is stored in the ptr member of x; and the NULL pointer, an address that points to no location in memory, is stored in y.ptr. Since the actual addresses are not necessary for programming, only arrows are typically shown, as shown in Figure (b).



(a) How a chain of pointers is created.

(b) How a chain of pointers is represented in Figures.

To access 33.7, we can specify either x.value or (*p).value. *p points you to x, but x has two members, so the member of interest must be specified. Hence, the value pointed to by p is coded as (*p).value. A shorter way of saying "(*p).value", is "p→value". The value of p→ptr is A020.

The malloc() and free() Functions

We often need to create storage dynamically. While the program is executing, it may find out that the size of an array needs to be expanded. To allocate storage on-the-fly, malloc() is commonly used. Unfortunately, the syntax of this function is involved. Suppose that we wanted to create another storage place of type struct node; we can do it this way:

```
p = (struct node *) malloc(sizeof(struct node));
```

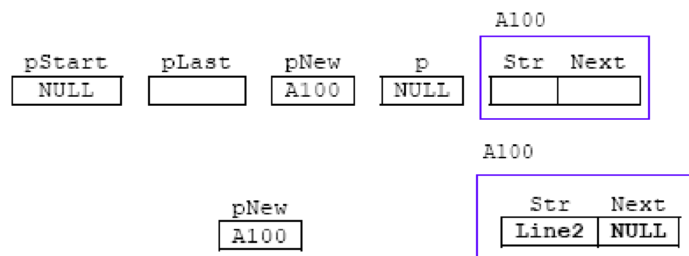
The sizeof() operator although it looks like a function, actually finds out for us how many bytes of Random Access Memory (RAM) a location of type struct node takes. Let us say that sizeof (struct node) is equal to 6 bytes on our computer system. Then that many bytes of RAM are given to the program by the operating system using the malloc() function.
However, this space in memory is useless unless the program can access it. What malloc() does is to return the address of the first byte of that location as a pointer to characters. We want to convert the pointer to characters to pointer to struct node, so the pointer type is cast appropriately by preceding the malloc() by (struct node *), Now this address can be assigned to p, our pointer to store type struct node.
A simple, explicit typecasting allows one to convert one data type to another. For example the following statement:

```
x = (float) i /3 ;
```

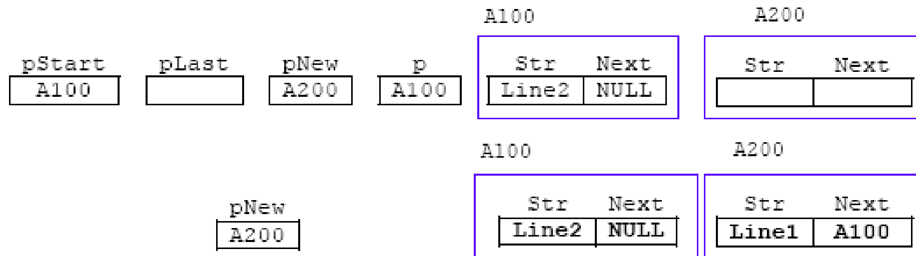
will convert the integer i to a float, divide by 3 and assign the floating point number to x. Similarly with the above malloc() example, we are converting pointer types. A function called free() will simply return the chunk of memory pointed to by the address given in free(). This should be done every time memory is no longer needed.

A Linked List Revisited

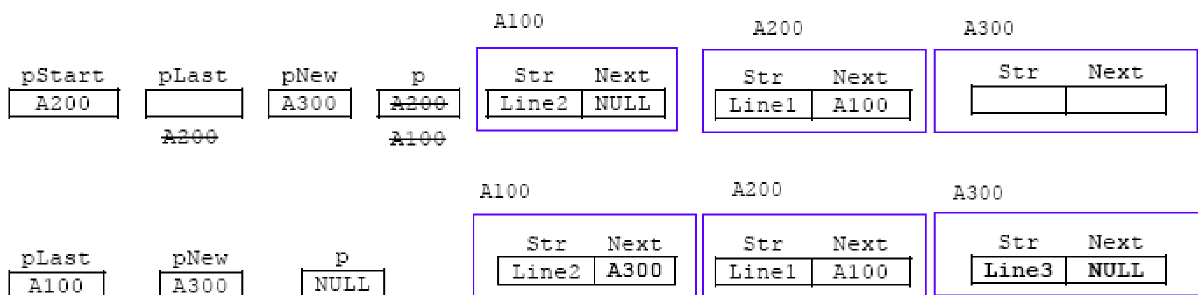
At the beginning of this worksheet we were introduced to linked list using an array. Again we will use a linked list and chunks of memory which are created as we need them. With the array example at the beginning of this worksheet it would have been possible to run out of available slots if we had needed more. If we were to make the array too large, it may take away memory needed by other arrays and variables.
We will start the linked list that occupies hardly any memory. As the linked list becomes larger, we will use the malloc() function to obtain more memory. If we delete an item from the linked list, then we will give the memory occupied by that item back to RAM/system for other variables and uses.
Data in our linked list will be stored in items called Nodes.
To see how data is added to a linked list, look at the following Figure. In Figure (a), initially there is nothing in the list. Then a new node is added that has Line2 in it. At first, the pStart pointer, which gives the address of the node where the list starts, is set to NULL. This means that there is nothing in the list. Using malloc(), memory for a new node is taken from RAM. Its address, A100, is stored in the pointer called pNew.



(a)



(b)



(c)

Before and after clips of adding to a linked list.

To place this node in the list, we first take the data labeled Line2 and place it in the Str member of the node pointed to by pNew. Then we take the pointer stored in pStart and place it in the next member of this new node. It is NULL. Finally, instead of the pStart pointer pointing to NULL, we make it point to this node stored at A100. Now the first node is added to the linked list.

In Figure (b), we add another node that will be added to the beginning of the list. First we grab some memory and get its address, A200, stored in pNew. The data of Line1 is placed in the Str member of the new node. The address of the pStart, A100, is placed in the next member of the new node and pStart now points to A200.

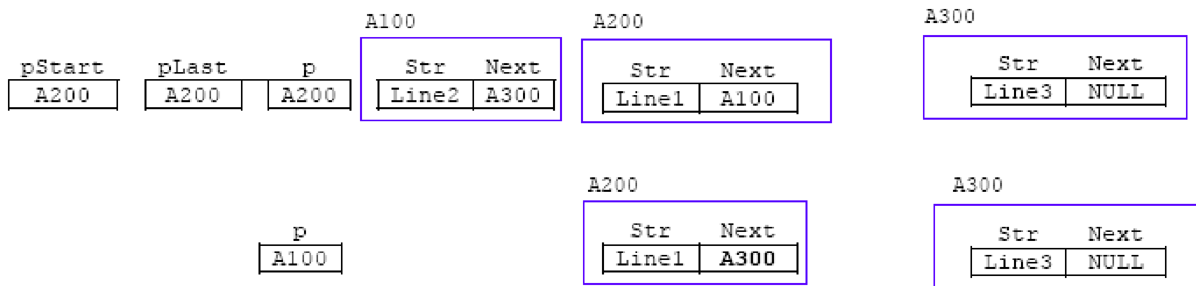
Let us follow the linked list we have so far in order. Starting from pStart, we go to A200 which has Line1 stored in it. Then following the next pointer we go to A100, which has Line2 stored in it. Finally, following its next pointer, which is NULL, we stop traversing the list.

So far we have added nodes at the beginning of the list. In Figure (c), we are adding a node at the end. First, pNew gets the address of A300, where the new node is located. Then we place the data, Line3, in the Str member of A300. Now we have to traverse the list to see where Line3 should go in the list.

pStart starts us at A200. The pointer, p, will be used to go down the list, while pStart will hold the address of the beginning of the list. There Line1 is stored. Next, we go to A100. Before we change p to A100, we store the previous pointer of p in pLast. pLast will contain A200. pLast and p are needed to add a new link between these two nodes. A100 has Line2 stored in it. We need to march forward. Before we advance p, we save its value in pLast, which becomes A100. Then the next member of A100 becomes NULL. There are no more nodes in the list. The new one must be placed at the end.

p, or NULL, is stored in the next member of A300. The address in pLast gets the address of pNew. That is, the next member of A300. Now the linked list has three nodes in it.

Let us now remove a node. Refer to the following Figure.



Before and after clips of deleting the A100 node from the linked list.

Here we were removing Line2 from the list. Starting at pStart, with a value of A200, we go down the list, storing the A200 in pLast. p becomes A100. Here Line2, which we want to delete, is stored. We must get the next member of p. This is the A300 stored in A100. It must be place in the next of pLast. Hence, the A300 is stored in A200. Lastly the memory at A100 is free and the list still starts at pStart. When the first node is deleted, the pStart has to be moved up to the next node.