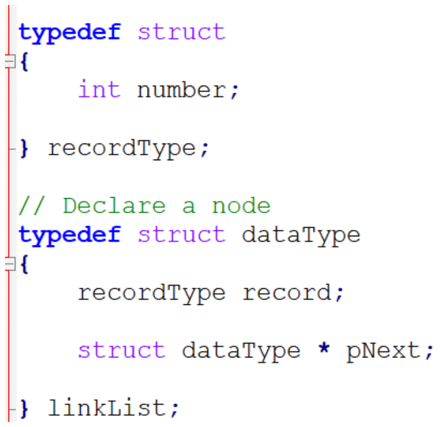
PLAYBOOK NOTES: SIMPLE LINK LIST PROGRAM

Before we get to the main:

Examine the following structure:



```
typedef struct
{
    int number;

} recordType;

// Declare a node
typedef struct dataType
{
    recordType record;

    struct dataType * pNext;

} linkList;
```

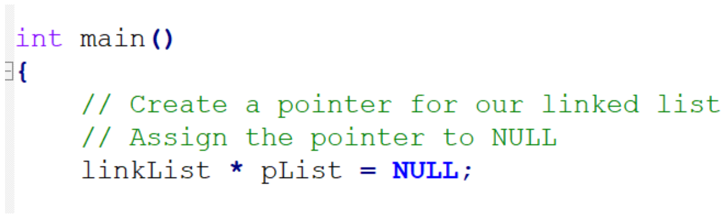
NOTE: Because we declared the structure with the keyword typedef, we don't need to use the word struct when using the structure inside our code. It is also true that linkList will be treated as a data type, not an instance of the structure.

Inside the structure recordType there is an integer named number. In more complex programs (or labs) you will put all the data for your link list in this structure.

In the linklist structure is an instance of our recordType and there is also a pointer of type dataType.

This pointer will be used to point to (contain the address of) another structure.

Inside the main function:



```
int main()
{
    // Create a pointer for our linked list
    // Assign the pointer to NULL
    linkList * pList = NULL;
```

The main function has a pointer of type linkList. This pointer will be the anchor to our link list.
This pointer will hold the address to a node (another word for a structure) or NULL if there is no node to point to.

To begin with, there is nothing in our link list, so the pointer is assigned to NULL. The pointer will look like this:

$$\begin{tikzcd} \text{pList} \arrow[r] & \text{NULL} \end{tikzcd}$$

Besides printing the header, the remaining of four main function looks like this:

```
CreateList(&pList);

// Print message to the screen
printf("Print the list after entering all the data:\n\n");

PrintList(pList);

FreeTheList(&pList);

// Print message to the screen
printf("Print the list after freeing all the data:\n\n");

PrintList(pList);
```

By looking at this code, we can see that this sample program will accomplish the following tasks:

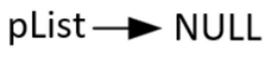
- Create a link list: This function will ask us to enter integers (until a 0 is entered) and store all the numbers in the link list. For this document we will use the values 10, 20, and 0.
- Print the list: This function will print all the numbers that we placed in the link list to the screen
- Free the list: This function deletes all the data from our link list.
- Print the list: This time, this function will print that the link list is empty.

Let's move on to our first function and create the link list.

FUNCTION: CreateList(linkList ** pList)

This function will be changing the address of pList. Because we want to retain those changes (we can't use pass by value) we must pass the address of the pointer. This is known as a double pointer. A double pointer is noted with **.

As we begin, our pointer, pList, points to NULL



To trace this function, we will use the following input: 10 20 0

The value 0 will stop the loop and will not be stored in the list.

FIRST PASS ON THE LOOP (Input = 10)

The following code will read 10 from the keyboard:

```
do
{
    // Enter a number to put into the list
    printf("Enter a number (zero to stop): ");
    scanf("%d", &input);
```

The next line of code is the if-statement:

```
if (input != 0)
{
```

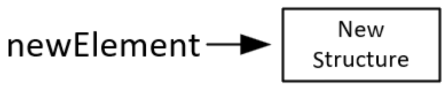
Since 10 does not equal 0, the test for the if-statement passes and we enter the block of code.

The following code creates a pointer a new node.

```
// Create a new node
newElement = (linkList*)malloc(sizeof(linkList));
```

The malloc function designates bytes of memory. The sizof(linkList) tells malloc how much memory to create. A pointer to the new memory is returned, and that pointer is casted to a linkList pointer.

Now we have a pointer called newElement that points to a new node (An instance of the structure).



A function call to **CreateNode(linkList* pNewNode, int input)** will set the members of the new node. This function has the following code:

```
// Assign members of the new node
pNewNode->record.number = input;
pNewNode->pNext = NULL;
```

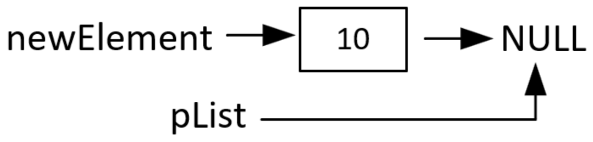
A function call to **InsertNewNode(linkList** pList, linkList* pNewNode)**

The first line of this function has the following code:

```
// Add new node to the link list
pNewNode->pNext = *pList;
```

Since pList is currently set to NULL, the pointer inside the structure (pNext) is set to NULL.

So now our link list looks like this:

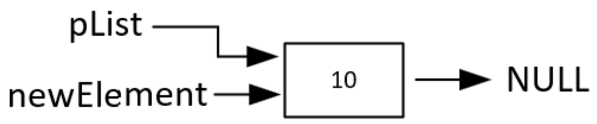


The next line of code will assign pList to our new node (newElement)

```
*pList = pNewNode;
```

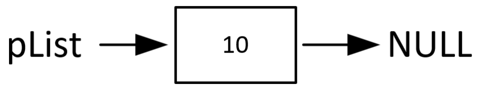
Because pList is a double pointer, you must dereference it.

Now our pointers look like this:



That is the end of the code for the if-statement.

Removing the newElement pointer from our picture (this is just a temporary pointer), our link list now looks like this:



Next is the test for the while-loop:

```
} while (input != 0);
```

The test for the while-loop passes (10 != 0), so we go back to the top and start our second pass through the loop.

<u>SECOND PASS ON THE LOOP (Input = 20)</u>

The following code will read 20 from the keyboard:

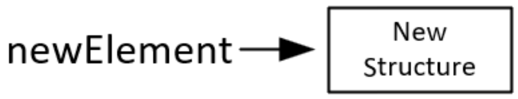
```
do
{
    // Enter a number to put into the list
    printf("Enter a number (zero to stop): ");
    scanf("%d", &input);
```

Since 20 does not equal 0, the if-statement passes, and we go back inside the if-statement.

The following code once again creates a new structure:

```
// Create a new node
newElement = (linkList*)malloc(sizeof(linkList));
```

Once again, our pointer newElement looks like this:



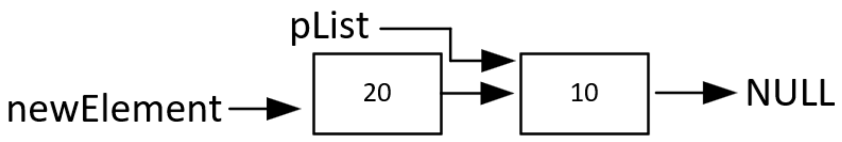
The member of our structure will be set in the CreateNode function just like last time.

The code inside **InsertNewNode** function will insert the new structure into the link list:

```
// Add new node to the link list
pNewNode->pNext = *pList;
```

(Except this time, pList points a structure, not NULL)

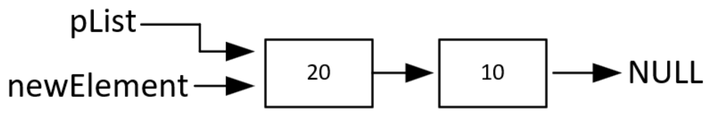
So now our pointers look like this:



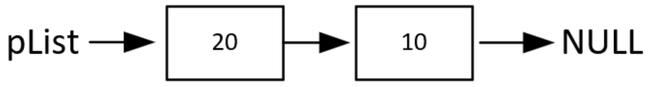
The following code will now re-assign the pointer pList, to point to our new structure (newElement):

```
*pList = pNewNode;
```

Now our link list looks like this:



Removing the newElement pointer from our picture (this is just a temporary pointer), our link list now looks like this:



THIRD PASS ON THE LOOP (Input = 0)

The following code will read 0 from the keyboard:

```
do
{
    // Enter a number to put into the list
    printf("Enter a number (zero to stop): ");
    scanf("%d", &input);
```

The if-statement fails the test and the code for the if-statement is skipped.

The while loop fails the test, and the while loop is now done.

That ends this function. We now have a link list with two elements, 20 and 10.

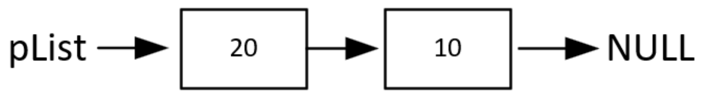
If we had continued to enter more numbers (before the 0), each number would be stored in the front of the list just like the first two numbers.

Next, we will print the list to the screen.

FUNCTION: PrintList(linkList * pList)

Since nothing in the list is going to change, we do not need to use a double pointer.

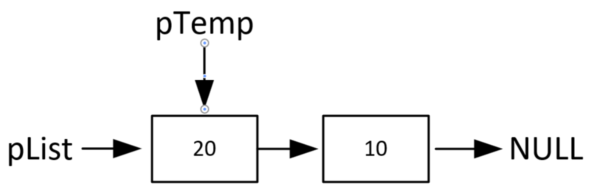
We will use the list that we created in the previous function:



In the first line of this function, we create pTemp variable and assign it to pList:

```
// Assign a temp pointer to the beginning of our list
linkList* pTemp = pList;
```

Now our link list looks like this:



I drew the pointer on top to make it easier to see.

The next line of code is an if-statement to see if the list is empty:

```
// Check to see if the list is empty
if (IsEmpty(pList))
{
```

The function IsEmpty(linkList * pList) will return pList == NULL.

Since pTemp does not equal NULL, this test will fail, and we will enter the else part of the code.

Next, we start a while-loop and since pTemp does not equal NULL, the test passes.

```
while (pTemp != NULL)
{
```

<u>FIRST PASS ON THE LOOP (pTemp points to 20)</u>

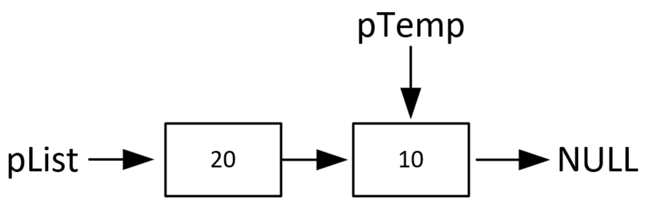
The first line of code in the while-loop will print 20 to the screen:

```
// Print the number of the current number
printf("%d ", pTemp->record.number);
```

The next line of code in the while-loop will move the pointer to the next item in the link list:

```
// Move pointer to the next number in the list
pTemp = pTemp->pNext;
```

Now our link list looks like this:



<u>SECOND PASS ON THE LOOP (pTemp points to 10)</u>

Since pTemp does not equal NULL, the test pass and we enter the code for the while-loop.

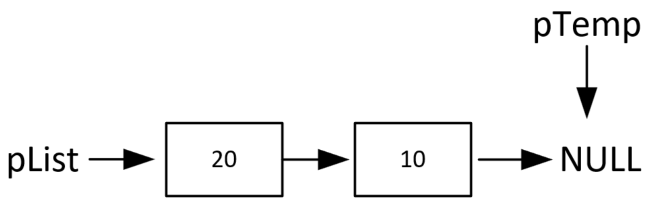
The first line of code I the loop will print 10 to the screen:

```
// Print the number of the current number
printf("%d ", pTemp->record.number);
```

The last line of code in the while-loop will move the pointer to the next item in the link list.

```
// Move pointer to the next number in the list
pTemp = pTemp->pNext;
```

Now our link list looks like this:



The while loop now fails the test and the loop has finished.

Before exiting the function, a new line character is sent to the screen to make the output easier to read.

You could make the argument that because we are not using a double pointer and any changes made to the link list will not be retained after the function exits, that we did not need to use the pointer pTemp. We could have just moved the pointer pList down the list. I chose to use the pTemp pointer because I believe it makes the code easier to understand.

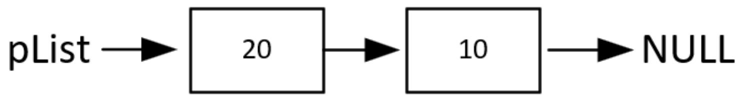
Next, we will free (or delete) the list!

FUNCTION: FreeTheList(linkList ** pList)

In this function we are going to free (or delete) every node in the list.

When we are done, pList will equal NULL. Since we are making changes to the link list we will once again be using a double pointer.

Let's start with our linked list:



The first line of code will create a pointer called pTemp but does not assign it to anything.

```
linkList* pTemp;
```

The pointer looks like this:

pTemp

Notice there is no arrow, that is because pTemp was never assigned a value. *It is important to know that if you don't assign a pointer a value, you cannot assume that it equals NULL.*

The next line of code starts our while loop, the test checks the pointer pList to see if it equals NULL.

```
while (*pList != NULL)
{
```

You can see in the picture above that the pointer pList doesn't equal NULL so the test passes and we enter the code for the loop.

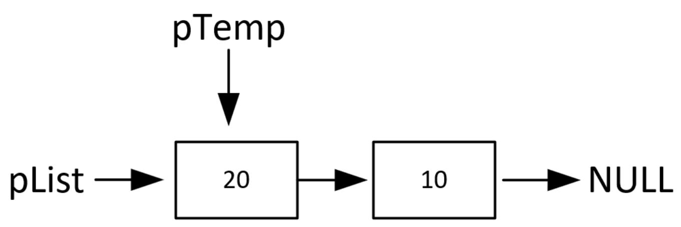
FIRST PASS ON THE LOOP (pList points to 20)

The first line of code assigns the pointer pTemp to the pointer pList.

```
// Assign pTemp to pList
pTemp = *pList;
```

Don't forget that are parameter is a double pointer, so we must dereference the pointer pList.

Now our link list looks like this:

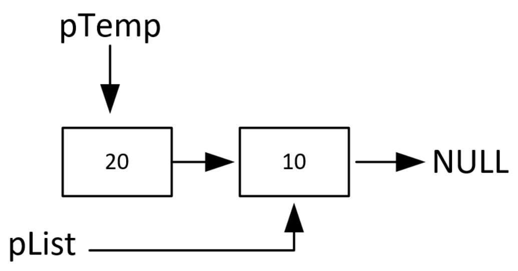


The next line of code will move the pList to next item in the list:

```
// Move pList to the next element
*pList = (*pList)->pNext;
```

Notice I used ( ) to make sure that the dereference (*) happens first.

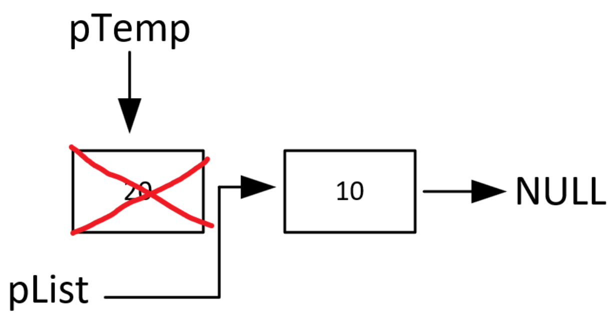
The link list now looks like this:



The last line of code in the while loop will free (or delete) the node pointed to by pTemp:

```
// Delete pTemp
free(pTemp);
```

The link list now looks like this:



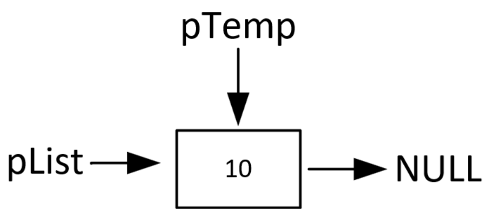
Notice that the node pointed to by pTemp has an X over it, this is because the node no longer exists.

SECOND PASS ON THE LOOP (pList points to 10)

The first line of code assigns pTemp to pList

```
// Assign pTemp to pList
pTemp = *pList;
```

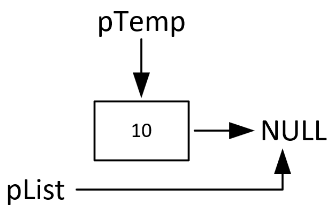
Now our link list looks like this:



The next line of code will move pList to the next node (or this case NULL)

```
// Move pList to the next element
*pList = (*pList)->pNext;
```

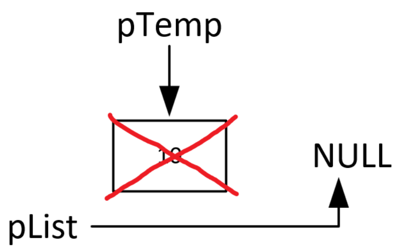
Now our link list looks like this:



The last line in the loop will once again free (or delete) the pTemp node:

```
// Delete pTemp
free(pTemp);
```

Now our link list look like this:



Notice that that the node containing the 10 has an X on it because the node no longer exists.

Now our link list looks like this:

$$\begin{tikzcd}\text{pList} \arrow[r] & \text{NULL}\end{tikzcd}$$

The test for the while loop will now fail. Since there is no code after the while-loop, the function will exit.

The last task this program will do is re-call the function PrintList.

FUNCTION: PrintList(linkList * pList)

This time when we start the function, pList is equal to NULL.

$$\begin{tikzcd}\text{pList} \arrow[r] & \text{NULL}\end{tikzcd}$$

The first line of code will create pointer called pList and assign it to pList:

$$\begin{tikzcd} & \text{pTemp} \arrow[d] \\ \text{pList} \arrow[r] & \text{NULL}\end{tikzcd}$$

The next line of code will check to see if the list is empty.

```
// Check to see if the list is empty
if (pTemp == NULL)
{
```

Since the list is empty, this if-statement will pass, and a message will be printed to the screen.

```
// Print message to the screen
printf("The list is empty.\n");
```

Because the if-statement is true, the while loos is skipped. The function exits and the program is done.

That concludes this sample program!