[*Linked Lists*]

Linked lists are used when the programmer does not know how much memory the user is going to need when they run the program. Linked lists allows the computer to ask for memory whenever the user needs more of it.

One example of when this is useful is when the user has to enter a series of numbers but does not know at the start how many numbers they have to enter. The programmer can use a vector, however a linked list will require less memory.

A linked list is a series of nodes that point to each other. The first node is typically called **head** and each node will hold information and point to either the next node in the list or to null if it is the last node in the list. This allows the computer to create a new node and have the last node point to the new node for as long as the user needs to enter more information.

First we have to create the linked list class:

```
class ListNode{
    public:
     int content;
     ListNode* pointerToNext;
};
```

In this example we are going to create a linked list that holds integers. The class is called ListNode and in public we create an int called content to hold our integers and a pointer to allow our node to point to the next node.

When creating a linked list we begin by creating our first pointer and call it head. You can call this whatever you want but programmers typically call it head so that it is clear that if refers to the first node in the list. You should also never move head to another node because you will lose access to the first element in the list and possibly lose access to more nodes.

This is how we declare a new ListNode and ask for memory to hold a node:

```
ListNode* head;    //create pointer called head
head = new ListNode;   //ask for memory to hold a node,
                       //head points to this node
```

Once you create a new node, you can store an integer in the content of the node and you should set the pointer to null pointer.

```
(*head).content = 7;
(*head).pointerToNext = nullptr;
```

or you can use the following method to access the content and pointer of each node, both of these methods work the same

```
head->content = 7;
head->pointerToNext = nullptr;
```

***nullptr was introduced in C++ 11, so if you compile your code on a linux machine, you need to add the following flag at the end of the command: **-std=c++11**

If we want to extend our list we need to create another pointer to create the rest of the list, we will call this tail. Some people like to call this runner.

```
ListNode* tail; //creates new pointer called tail
tail = head;   //tail now points to the same node head points to
```

Now to create a new node to hold another integer we can do this:

```
tail->pointerToNext = new ListNode;
//creates new node by asking for memory,
//current node points to this new node

tail = tail->pointerToNext;
//now tail points to the newly created node in the previous step

tail->content = -4;
//stores -4 in the newly created node

tail->pointerToNext = nullptr;
//the newly created node now points to nullptr
```

So for we have two nodes in our list, the first node is called head and contains the integer 7, this node points to the second node which is called tail and contains the integer -4, since tail points to the last node in our list, this node points to nullptr.

You can keep adding elements to your list for as long as you have access to the last node in your list. You should never make head point to another node because you will never be able to access the first node in the list again and cause a memory leak.

Here is how we would use a linked list to help us solve the following problem.

Create a program that accepts integers from the user until the user enters the number zero. Once the user enters the number zero, stop accepting integers, zero will not be included as part of the list unless it is the first number the user enters. Multiply all of the integers the user entered by two and print these integers to the screen.

```
#include <iostream>

class ListNode{
  public:
    int content;
    ListNode* pointerToNext;
};

void eraseLinkedList(ListNode *runner){
 if(runner != nullptr){
     eraseLinkedList(runner->pointerToNext);
     delete runner;
 }
}


int main( )
{
    //get first number from user
    int input;
    std::cin>>input;

    //initialize linked list by creating first node
    ListNode* head = new ListNode;
    head->content = input;
    head->pointerToNext = nullptr;

    //create tail to facilitate the creation of the list
    ListNode* tail;
    tail = head;

    //add each number the user enters to the end of the list
    //keep accepting numbers from user until 0 is entered
    while(input != 0){
        //accept next number from user
        std::cin>>input;
        //create a new node and make tail point to it
        tail->pointerToNext = new ListNode;

        //tail now points to this new node
        tail = tail->pointerToNext;

        //store the value of input in the content of this node
        tail->content = input;

        //since this is the last node in the list,
        //make it point to nullptr
        tail->pointerToNext = nullptr;
    }

    //make tail point to the beginning of the list
    tail = head;

    //go through each element in the list
    //multiply the content by 2 then print
    //repeat this step until you reach the nullptr.
    //the last node points to nullptr,
    //this means we are at the end of the list
    while(tail != nullptr){
        std::cout<< 2 * (tail->content) <<" ";
        tail = tail->pointerToNext;
    }

    eraseLinkedList(head);

    return 0;
}
```

Notice that there is a function called **eraseLinkedList**. Linked lists are created using pointers, this means that we need to delete the list at the end of our program so that we can free the memory and return it back to the computer. This function uses recursion to delete the list, see document for recursion to see how recursion works.