

Arrays have the following disadvantages:

- Fixed size: specified at compile time or deferred until run time. Dynamically resizing an array with `realloc( )` requires some real programming effort.( Dynamic Arrays we have discussed )
- Because of fixed size programmers tend to allocate very large arrays and as a result either memory is wasted or the program crashes.
- Inserting new elements at the beginning of the array is expensive, as existing elements need to be shifted
- We have discussed the problem of unordered array. Insertion is easy, searching/deletion is expensive.
- We have discussed the maintenance of orderly array. Insertion is costly, and searching/deletion is a bit easy.

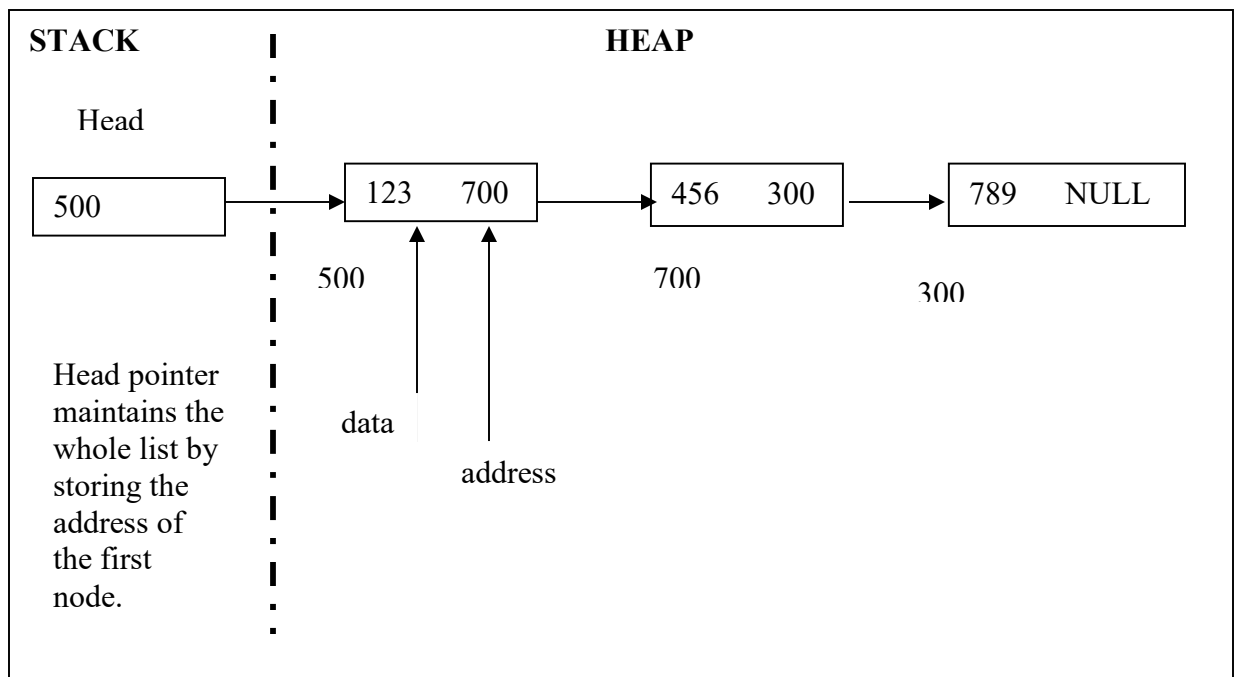
Pointer variables are an extremely useful data type. You can use them to implement the data structure known as a “Linked List”. It has many advantages over arrays.

## Linked List

Linked List allocates space for each element separately. These elements are generally referred to as “Nodes”. Pointers link the nodes to each other. Each node contains two fields – data and address.

Data stores whatever the user wants, whereas address stores the pointer to the next node.

Nodes are allocated by using the **new**, and needs to be freed explicitly using the **delete**. This memory is allocated on the heap and not on the stack.



## Class Node

```
template <class T>
class Node {
    // declare class List a friend so that it can access
    //Node's private vars.
    friend class List<T>;
public:
    // constructors
    Node() { nextPtr = 0; }
    Node(const T & d) { data = d; nextPtr = 0; }
    Node(const T & d, Node<T> *n)
        {data = d; nextPtr = n; }

private:
    T data; // data
    Node<T> *nextPtr; // next node in the list
};
```

## Class List

```
// The List will contain nodes linked together by pointers
template <class T>
class List {
public:
    List() { firstPtr = 0; size = 0; } // constructor
    List(const List<T> &); // copy constructor
    ~List(); // destructor
    void deleteList();
    void insertAtFront( const T & );
    void removeAtFront();
    void print() const;
    int getSize() const { return size; }
    List& operator = (const List<T> &);
private:
    Node<T> *firstPtr; // pointer to first node
    int size;
};
```

## Destructor

```
// Destructor
template<class T>
List<T>::~~List()
{
    deleteList();
    size = 0;
}
```

## Delete all Node from the List

```
template<class T>
void List<T>::deleteList()
{
    if (firstPtr != 0) {    // List is not empty
        cout << "Destroying nodes ...\n";
        Node<T> *currentPtr = firstPtr;
        Node<T> *tempPtr;

        while ( currentPtr != 0 ) {    // not end of list
            tempPtr = currentPtr;
            cout << "Deleting " << tempPtr->data << endl;
            currentPtr = currentPtr->nextPtr; // move ptr
            delete tempPtr; // delete last node
        }
    }
    cout << "All nodes destroyed\n\n";
}
```

## Insert At the Front of the List

```
// Insert a node at the front of the list
template<class T>
void List<T>::insertAtFront( const T &value )
{
    // create a new node with the value in it.
    Node<T> *newPtr = new Node<T>(value);
    assert(newPtr != 0);

    if ( firstPtr == 0 ) // if List is empty
        firstPtr = newPtr; // point to new node
    else {                // if List is not empty
        newPtr->nextPtr = firstPtr; // point to list
        firstPtr = newPtr; // move up firstPtr
    }
    size++;
}
```

## Delete From the Front of the List

```
// Delete a node from the front of the list
template<class T>
void List<T>::removeAtFront()
{
    if ( firstPtr == 0 ) // List is empty
        cout << "Error: list is empty!\n";
    else {
        Node<T> *tempPtr = firstPtr;
        cout << "Deleting " << tempPtr->data << endl;
        firstPtr = firstPtr->nextPtr; // move to next node
        delete tempPtr;
        size--;
    }
}
```

## Print the contents Of the List

```
// Display the contents of the List
template<class T>
void List<T>::print() const
{
    if ( firstPtr == 0 ) {
        cout <<"The list is empty\n\n";
    }
    else {
        Node<T> *currentPtr = firstPtr;
        cout <<"The list's Contents are : ";
        while ( currentPtr != 0 ) { // not end of list
            cout <<currentPtr->data << " -> ";
            currentPtr = currentPtr->nextPtr;
        }
        cout <<"\n";
    }
}
```

## Copy Constructor

```
// copy constructor
template<class T>
List<T>::List(const List<T> &original) {
    size = original.size;
    firstPtr = 0;
    Node<T> *ptr = original.firstPtr;
    Node<T> *newPtr, *lastPtr;
    while (ptr != 0) {
        // create a new node with the value in it.
        newPtr = new Node<T>(ptr->data);
        assert(newPtr != 0);
        if(firstPtr == 0)
            firstPtr = newPtr; // first node
        else
            lastPtr->nextPtr = newPtr; // set up last link
        lastPtr = newPtr; // save last ptr
        ptr = ptr->nextPtr; // move up ptr
    }
}
```

## Assignment Operator

```
// assignment operator
template<class T>
List<T> & List<T>::operator =(const List<T> &original) {
    if (this != &original) {
        deleteList();
        size = original.size;
        firstPtr = 0;
        Node<T> *ptr = original.firstPtr;
        Node<T> *newPtr, *lastPtr;
        while (ptr != 0) {
            // create a new node with the value in it.
            newPtr = new Node<T>(ptr->data);
            assert(newPtr != 0);
            if(firstPtr == 0)
                firstPtr = newPtr; // first node
            else
                lastPtr->nextPtr = newPtr; // set up last link
            lastPtr = newPtr; // save last ptr
            ptr = ptr->nextPtr; // move up ptr
        }
    }
}
```

# Driver Program

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

int main()
{
    List<int> L1;
    cout << "The size of the list is: " << L1.getSize() <<
endl;
    L1.insertAtFront(123);
    L1.insertAtFront(456);
    L1.insertAtFront(789);
    cout << "The size of the list now is: " << L1.getSize()
<< endl;
    L1.print();
    L1.removeAtFront();
    cout << "The size of the list now is: " << L1.getSize()
<< endl;
    return 0;
}
```