void ArrayTest() {

int scores[100];

// operate on the elements of the scores array...

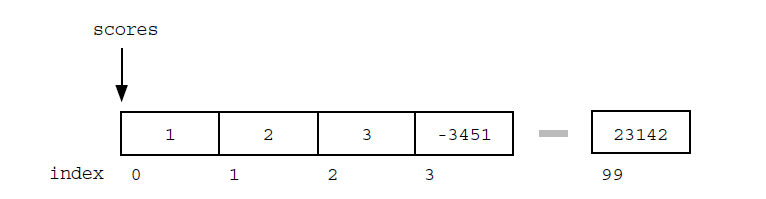
scores[0] = 1;

scores[1] = 2;

scores[2] = 3;

}

The key point is that the entire array is allocated as one block of memory.



Once the array is set up, access to any element is convenient and fast with the [ ] operator. Array access with expressions such as scores[i] is almost always implemented using fast address arithmetic: the address of an element is computed as an offset from the start of the array which only requires one multiplication and one addition.

Some disadvantages of arrays:

* The size of the array is fixed. Most often this size is specified at compile time with a simple declaration such as in the example above . With a little extra effort, the size of the array can be deferred until the array is created at runtime, but after that it remains fixed. You can go to the trouble of dynamically allocating an array in the heap and then dynamically resizing it with realloc(), but that requires more effort.
* Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.

**Linked Lists**

An array allocates memory for all its elements lumped together as one block of memory. In contrast, a linked list allocates space for each element separately in its own block of memory called a "linked list element" or "node". The list gets is overall structure by using pointers to connect all its nodes together like the links in a chain. Think of it like a train. The programmer always stores the first node of the list. This would be the engine of the train. The pointer is the connector between cars of the train. Every time the train adds a car, it uses the connectors to add a new car. This is like a programmer using the keyword new to create a pointer to a new struct or class. 

Each of the big blocks is a struct (or class) that has a pointer to another one. Remember that the pointer only stores the memory location of something, it is not that thing, so the arrow goes to the next one. At the end, there is nothing for the pointer to point to, so it does not point to anything, it should be a null pointer or a dummy node to prevent it from accidentally pointing to a totally arbitrary and random location in memory (which is very bad). 

Each node contains two fields: a "data" field to store whatever element type the list holds for its client, and a "next" field, which is a pointer used to link one node to the next node.

Each node is allocated on the heap with a either malloc( ) or new. The node memory continues to exist until it is explicitly de-allocated using free( ) or delete.

A linked list is a dynamic data structures - grow and shrink during execution

**Node and Pointer (Self-referential)**

*• Node* The type for the nodes which will make up the body of the list.

These are allocated on the heap. Each node contains a single client data element and a pointer to the next node in the list.

DO NOT initalize members inside the structure definition.

struct node {

int data;

struct node\* next;

};

struct node \*ptr

//or

struct node {

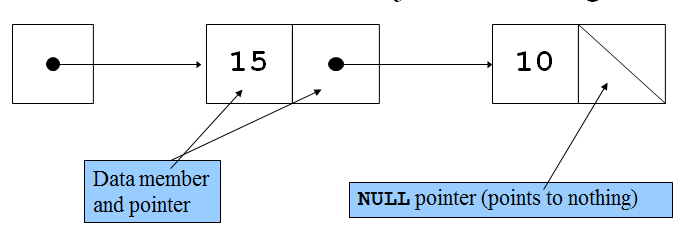
int data;

struct node\* next;

};

typedef struct node node;

node \*ptr;

*• Node Pointer* The type for pointers to nodes. 

* A linked list is a dynamic data structure
* The address of the first node is called the head or first. The address of the last node is called the tail.
* The first (or head) pointer of a linked list is always fixed, pointing to the first node in the list.
* **Linear** **relationship** Each element except the first has a unique predecessor, and each element except the last has a unique successor.
* To traverse a linked list, the program must use a pointer different than the head pointer of the list, initialized to the first node in the list.
* A single linked list is traversed in only one direction
* The search on a linked list is sequential
* The length of a linked list is the number of nodes in the list
* Four common operations associated with lists: Insertion, deletion, retrieval and traversal
* Item insertion and deletion from a linked list do not require data movement; only the pointers are adjusted.

**Example 1a (Creating a single node using c++)**

struct node {

int data;

node \*next;

};

int main()

{

node \*head;

head = new node; //Now head points to a node struct

head->next = NULL; //The node head points to has its next pointer

//set equal to a null pointer

head->data = 5; //By using the -> operator, you can modify the node

//a pointer (head in this case) points to.

delete head;

}

**Example 1b (Creating a single node using c)**

struct node {

int data;

node \*next;

};

int main()

{

node \*head; //This will be the unchanging first node

head = (node \*)*malloc*(sizeof(struct node));

head->next = NULL; //The node head points to has its next pointer

//set equal to a null pointer

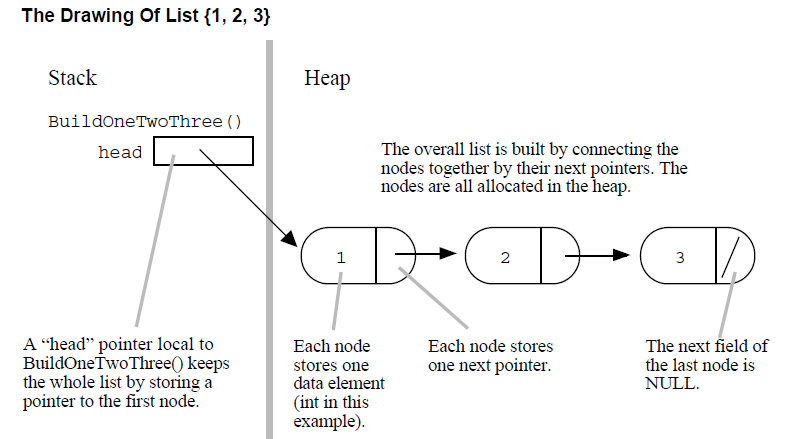
head->data = 5; //By using the -> operator, you can modify the node

//a pointer (head in this case) points to.

*free*(head);

}

**Example 2 (Creating multiple nodes)**



/\*

Build the list {1, 2, 3} in the heap.

Returns the first pointer to the caller.

\*/

struct node {

int data;

node \*next;

};

struct node\* build() {

struct node\* first = NULL;

struct node\* second = NULL;

struct node\* third = NULL;

// allocate 3 nodes in the heap

first = (node \*)malloc(sizeof(struct node));

second = (node \*)malloc(sizeof(struct node));

third = (node \*)malloc(sizeof(struct node));

first->data = 1; // setup first node

first->next = second; // note: pointer assignment rule

second->data = 2; // setup second node

second->next = third;

third->data = 3; // setup third link

third->next = NULL;

return first;

}

int main()

{

node \*head=build();

}

**Example 3 (Adding a node to the beginning)**

struct node {

int data;

node \*next;

};

struct node\* build() {

struct node\* head;

head = (node \*)malloc(sizeof(struct node)); // allocate on the heap

head->data = 2; // setup first node

head->next = (node \*)malloc(sizeof(struct node));

head->next->data = 3; // setup second node

head->next->next = NULL;

return (head);

}

void main() {

struct node\* head = build ();

struct node\* newNode;

newNode= (node \*)malloc(sizeof(struct node)); // allocate

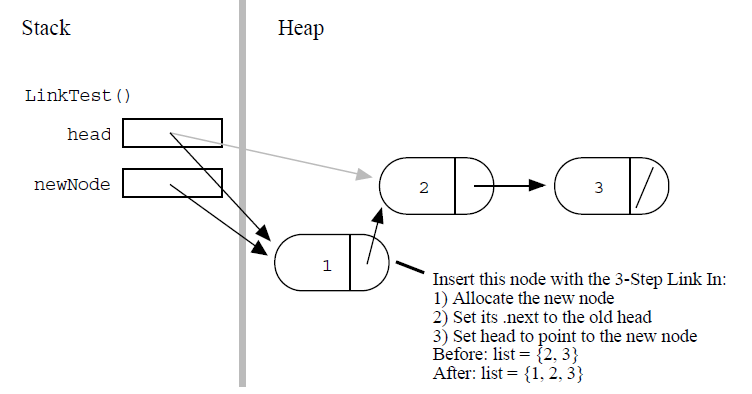
newNode->data = 1;

newNode->next = head; // link next

head = newNode; // link head

// now head points to the list {1, 2, 3}

}



**Example 4 (Traversing and Finding the number of elements in a linked list)**

/\*

Given a linked list head pointer, compute and return the number of nodes in the list. BuildOneTwoThree() creates a linked list with 3 elements. Code is not included

\*/

int Length(struct node\* head) {

struct node\* current = head;

int count = 0;

while (current != NULL) {

count++;

cout<< current ->data;

current = current->next;

}

return count;

}

int main() {

struct node\* myList = BuildOneTwoThree();

int len = Length(myList); // results in len == 3

}

1) The local pointer, current in this case, starts by pointing to the same node as the head pointer with current = head;. When the function exits, current is automatically deallocated since it is just an ordinary local, but the nodes in the heap remain.

2) The while loop tests for the end of the list with (current != NULL). This test smoothly catches the empty list case — current will be NULL on the first iteration and the while loop will just exit before the first iteration.

3) At the bottom of the while loop, current = current->next; advances the local pointer to the next node in the list. When there are no more links, this sets the pointer to NULL. If you have some linked list code which goes into an infinite loop, often the problem is that step (3) has been forgotten.

Some questions to consider:

Q: What if we said head = NULL, would that mess up the myList variable in the caller?

Q: What if what was passed in contained no elements, does Length() handle that case properly?

Keeping the head updated if adding to the beginning

* Return contents of head from function (return single pointer \*)
* Pass in the address of the head into the function (pass in double pointer \*\*, return nothing)
* Pass by alias into the function (Pass in \*&, return nothing)

**Example 5 (Adding to beginning: Error with linked list - local variable)**

The key is that the line head = newNode; changes the head local in wrongAdd() but not

the head back in main().

struct node {

int data;

node \*next;

};

struct node\* build() {

struct node\* head;

head = malloc(sizeof(struct node)); // allocate on the heap

head->data = 2; // setup first node

head->next = malloc(sizeof(struct node)); // allocate on the heap

head->next->data = 3; // setup second node

head->next->next = NULL;

return (head);

**}**

//The change to head is not passed back

void wrongAdd (struct node\* head, int data) {

struct node\* newNode = (node \*)malloc(sizeof(struct node));

newNode->data = data;

newNode->next = head;

head = newNode; // NO this line does not work!

}

int main() {

struct node\* head = build();

wrongAdd(head, 1); // try to add 1 to the front -- doesn't work

cout<<head->data;

}

**Example 6 (Corrected version of pervious example)**

//same as before

struct node {

int data;

node \*next;

};

//same as before

struct node\* build() {

struct node\* head;

head = malloc(sizeof(struct node)); // allocate on the heap

head->data = 2; // setup first node

head->next = malloc(sizeof(struct node)); // allocate on the heap

head->next->data = 3; // setup second node

head->next->next = NULL;

return (head);

**}**

//modified: Returns head

struct node\* rightAdd (struct node\* head, int data) {

struct node\* newNode = (node \*)malloc(sizeof(struct node));

newNode->data = data;

newNode->next = head;

head = newNode;

return head

}

//modified: catches head after rightAdd call

int main() {

struct node\* head = build();

head= rightAdd (head, 1);

cout<<head->data;

}

**Example 7 (Correcting the previous example again using \*\*)**

• Design the function to take a pointer to the head pointer. This is the standard technique in C — pass a pointer to the "value of interest" that needs to be changed. To change a struct node\*, pass a struct node\*\*.

• Use '&' in the caller to compute and pass a pointer to the value of interest.

• Use '\*' on the parameter in the callee function to access and change the value of interest.

/\*

Takes a list and a data value. Creates a new link with the given data and pushes it onto the front of the list. The list is not passed in by its head pointer. Instead the list is passed in as a "reference" pointer to the head pointer -- this allows us to modify the caller's memory.

\*/

struct node {

int data;

node \*next;

};

struct node\* build() {

struct node\* head;

head =(node \*) malloc(sizeof(struct node)); // allocate on the heap

head->data = 2; // setup first node

head->next = (node \*)malloc(sizeof(struct node));

head->next->data = 3; // setup second node

head->next->next = NULL;

return (head);

**}**

void add(struct node\*\* headRef, int data) {

struct node\* newNode = (node \*)malloc(sizeof(struct node));

newNode->data = data;

newNode->next = \*headRef;

\*headRef = newNode;

}

void main() {

struct node\* head = build();

add (&head, 1); // note the &

add (&head, 13);

// head is now the list {13, 1, 2, 3}

}

**Example 8 (Correcting the previous example using &)**

struct node\* build() {

struct node\* head;

head = malloc(sizeof(struct node)); // allocate on the heap

head->data = 2; // setup first node

head->next = malloc(sizeof(struct node)); // allocate on the heap

head->next->data = 3; // setup second node

head->next->next = NULL;

return (head);

**}**

//The change to head is not passed back

void add (struct node\* &head, int data) {

struct node\* newNode = (node \*)malloc(sizeof(struct node));

newNode->data = data;

newNode->next = head;

head = newNode;

}

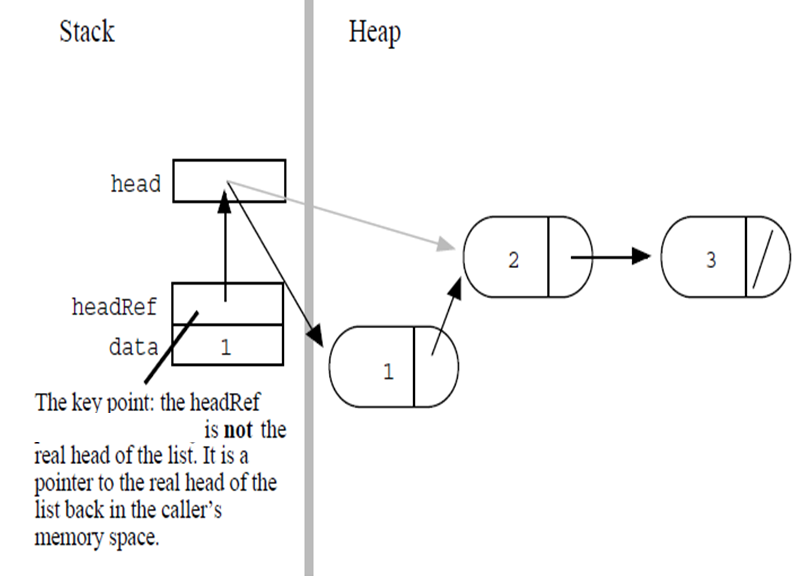
int main() {

struct node\* head = build();

add(head, 1);

cout<<head->data;

}



**Example 9 (Adding to the end)**

Think back to the train. Let’s imagine a conductor who can only enter the train through the engine, and can walk through the train down the line as long as the connector connects to another car. This is how the program will traverse the linked list. The conductor will be a pointer to node, and it will first point to root, and then, if the root's pointer to the next node is pointing to something, the "conductor" (not a technical term) will be set to point to the next node. In this fashion, the list can be traversed. Now, as long as there is a pointer to something, the traversal will continue. Once it reaches a null pointer, meaning there are no more nodes (train cars) then it will be at the end of the list, and a new node can subsequently be added if so desired.

Adding a node at the tail of a list most often involves locating the last node in the list, and then changing its next field from NULL to point to the new node, such as the tail variable.

struct node {

int data;

node \*next;

};

struct node\* build() {

struct node\* head;

head = (node \*)malloc(sizeof(struct node)); // allocate on the heap

head->data = 2;

head->next = (node \*)malloc(sizeof(struct node));

head->next->data = 3; // setup second node

head->next->next = NULL;

return (head);

}

void addToEnd (struct node \* current, int data){

struct node\* newNode;

while ( current->next != NULL)

current = current->next;

newNode = new node; // Sets it to actually point to something

newNode->data = data;

newNode->next = NULL;

current->next=newNode;

}

int main (){

struct node \* head=build();

if (head) //Make sure head is valid

addToEnd(head, 5);

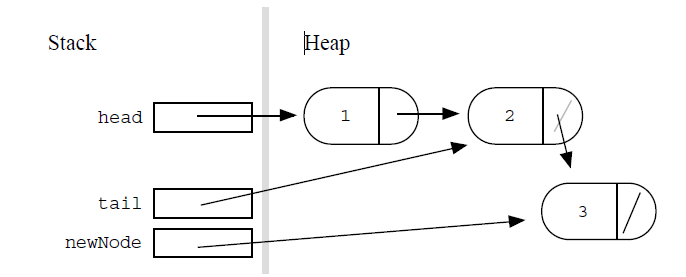
while ( head != NULL ) {

cout<< head->data;

head=head->next;

}

}



**Example 10 (Adding to the end, another example)**

struct node {

int data;

node \*next;

};

typedef struct node node;

node\* add(node\*& end, int data) {

node\* newNode = (node \*)malloc(sizeof(struct node));

newNode->data = data;

newNode->next = NULL;

end = newNode;

return (end);

}

node\* build () {

node\* head, \*current;

head=current=NULL;

for (int i=1; i<4; i++) {

if (!head)

current=add(head, i);

else

current=add(current->next, i);

}

return(head);

}

int main(){

node\* head, \*current;

current=head=build ();

while (current!= NULL ) {

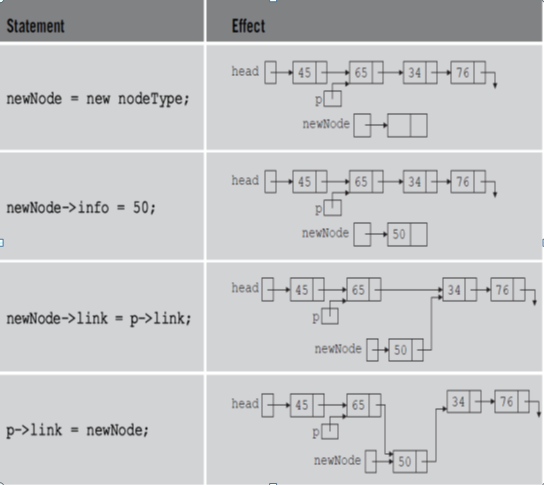
cout<< current ->data;

current = current ->next;

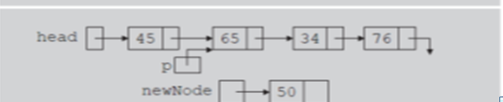
}

}

**Adding to the middle (Visual)**



Question: write the code that places the new node between 65 and 34 (Hard code)



Question: Write the code that points you to the last node (Hard code)

Question: Write the code that points you to the last node (Generic code)