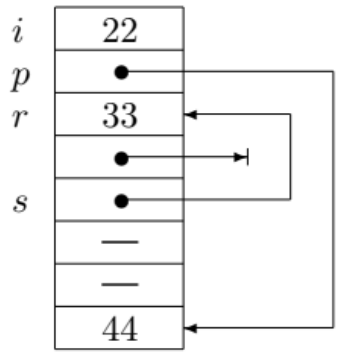
Pointers

What is a pointer?

A pointer is a common technique to allocate storage space dynamically. The compiler only reserves space for addresses to these dynamic parts. The pointer itself points to another location in memory, where something interesting is stored. Every variable has an address and content.

	address	content
i	$1af783$	22
p	$1af784$	$1af78a$
r	$1af785$	33
	$1af786$	NIL
s	$1af787$	$1af785$
	$1af788$	—
	$1af789$	—
	$1af78a$	44



Variable name	Variable type	Variable address	Variable content
i	int	1af783	22
p	pointer to int (int *p)	1af784 -> 1af78a	22
r	int	1af785	33
s	pointer to int (int *s)	1af787 -> 1af785	33

How and when is a pointer used in C?

```c
#include <stdio.h>

int main()
{
  int a = 50;
  int *b = &a;
  int *c = &a;
  d = malloc(sizeof(a))
  printf("%d, %d, %d", a, b, *c);
}
/* prints 50, -520866148, 50 */
```

Here we assign the variable a a value of 50. What we then do is create 2

pointers on that variable called b and c - the difference we can then see while printing our variables b and c, b prints a memory address where as $*c$ shows us the value of variable a to which it's pointing to. We can conclude that when initializing pointers we use `int * pointer_var` and to show which variable we are pointing to we use `int * pointer_var = &pointed_to_var`. With malloc we are asking memory to "reserve" memory in heap for our variable a.

Linked lists

What is a linked list?

Linked lists allow us to dynamically assign memory based on the size of the linked list. We free up memory when an element is deleted from the list and assign memory when a new element is added to the list. This all happens dynamically without the user having to use "malloc" every time.

How and when is a linked list used in C?

List traversal The most important part of the code for a linked list is how traverse and access and eventually print the elements.

```
struct Node *p = first;
while (p != NULL) {
    printf("%d, p-> data);
    p = p->next
}
```

Here we are creating a new struct called Node and giving the first half of the node the value *p. We then ask our print function to print the data of the first node, after that we move the pointer from the data element to the next element of the node. This next element points to the data element of the second node. This keeps going on to our last element. The last element has a "NULL" in it's next element, which makes the while loop stop.

Adding data to list (88, 52, 12)

```
root = malloc(sizeof(struct node));
/* create a node in memory */
root->val = 88;
/* add value to first node */
root->next = malloc(sizeof(struct node));
```

```
/* for next create a new node */

p = root->next;
/* connect root node to second node */
p->val = 52;
/* add value for second node */
p->next = malloc(sizeof(struct node));
/* init third node */

p = p->next;
p->val = 12;
p->next = NULL
/* last element handling */
```

Insert an element Beginning of the list

```
p = malloc(sizeof(struct node));
p->val = 43
p->next = root;
root = p;
```

End of the list

```
if (root == NULL) {
    root = malloc(sizeof(struct node));
    root->val = 43;
    root->next = NULL;
} else {
    q = root;
    while (q->next != NULL) { q = q->next; }
    q->next = malloc(sizeof(struct node));
    q->next->val = 43;
    q->next->next = NULL;
}
```

Delete an element Delete whole list

```
p = root;
while (p != NULL) {
    q = p;
    p = p->next;
    free(q);
}
void del(struct node* p) {
    if (p != NULL) {
        del(p->next );
        free(p);
```

```
    }
}
```

Delete single node from list

```
p = root;
if (p->val == x) {
    root = p->next;
    free(p);
} else {
    while (p->next != NULL &&
p->next->val != x) {
    p = p->next;
    }
    tmp = p->next;
    p->next = tmp->next
    free(tmp);
}
```

Reversing the linked list Here we have two options:

- Reverse elements: We move all elements from the linked list to an array, after that we create a new linked list and then append the values from the newly created array to our linked list backwards
- Reversing links: What we do here is create sliding pointers. With two helper pointers, we keep track of past positions of our linked list and slide through the list, reversing the links between the list elements.

Doubly linked list

Lists with explicit tail

- Do have an extra pointer to the last item of the list
- No need to scan the entire list if an operation applies only to the last item

Doubly linked list

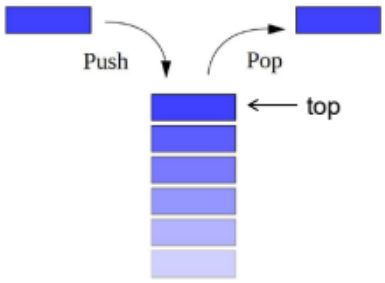
- Each node has a field with a pointer to the previous node of the linked list
- Provides means to quickly navigate forth and back in the linked list

Stacks

What is a stack?

A stack can be seen as collection of variables "stacked upon" each other. Stacks work by the "Last In First Out" (LIFO) principle. This means, the last element which joins the collection is the first one to leave. A good example to visualize this data structure is the browser history of your computer. By going to a new site, we add or $push(x)$ a new element to the collection. While using the "Back"

button, we are removing or $pop(x)$ the most recent elements one by one so that we can see the elements under them.



How and when is a stack used in C?

Typical operations used on stacks are: size(), isEmpty(), push(object), top(), pop(), peek(). Top() and peek() are both used to check the first element of our stack.

We usually use the end of arrays or the beginning of single linked lists to implement stacks.

```
int main () {
    push(x)
    pop(x)
}

int push(x) {
    S[t] = x;
    t = t+1
}

int pop(x) {
    t = t-1
    return S[t];
}
```

Queues

What is a queue?

A queue can be seen as collection of variables "queued behind" each other. Queues work by the "First In First Out" (LIFO) principle. This means, the first element which joins the collection is the first one to leave. A good example to visualize this data structure is a waiting list, the first one to sign up is the first one to recieve the goods. Using $enqueue(x)$ we add people to the waiting

list one after another. Using $dequeue(x)$ we start removing people from the front and push the second person to first place, the third to second etc.

How and when is a queue used in C?

Typical operations include size(), isEmpty(), enqueue(object), dequeue(), front(), back().

```
void enqueue(int x){
    Q[t] = x;
    t = t+1 mod n;
}

int dequeue(){
    int i = h;
    h = h+1 mod n;
    return Q[i];
}
```

Ordered lists

What is a ordered list?

How and when is a ordered list used in C?