Pointers in C

The language C is a low level language, that can easily access memory locations and perform memory related operations. A pointer is a variable, that contains the address of another variable. It can indirectly access the variable.

Declaration:

```
pointer_type *identifier
```

The pointer_type is the type of data, the pointer will point to.

Pointers should be *initialized* to NULL

```
int *p = NULL;
```

Pointers can be assigned to the *address of a variable* with &

```
int j = 10;
p = &j;
```

To see what the pointer is pointing to, use *

```
printf(„p is pointing to value %d“, *p);
```

Pointers can use *arithmetic* too

```
y = *p + 2;  // y is 12
*p = 13;     // j is assigned by 13;
```

Pointers point to the *first element* of an array if assigned

```
int a[] = {1, 2, 3};
int *ptr = NULL;
ptr = a;       // can be thought of as ptr = &a[0]
```

Pointers pointing on array elements can be *moved forward or backwards*

```
printf("%d %x\n", *ptr, ptr);  // 1
ptr++;
printf("%d %x\n", *ptr, ptr);  // 2
ptr--;
printf(„%d%x\n“, *ptr, ptr);   // 1
```

Pointer addresses can *be compared* with == < and >

An *array name* is a pointer.

An array can not(!) be passed by value to a function. Passing an array to a function is passing a pointer to the array!

```
int add_up (int *a, int num_elements);  // a is an array like
                                        // a[3] = {1, 2, 3}
```

With pointer parameters you can *change actual data* rather than a copy of data

```
void swap (int *num1, int *num2);
int main() {
int x = 25;
int y = 100;

printf("x is %d, y is %d\n", x, y);
swap(&x, &y);
printf("x is %d, y is %d\n", x, y);

return 0;
}

void swap (int *num1, int *num2) {
int temp;

temp = *num1;
*num1 = *num2;
*num2 = temp;
}
```

A function can *return a pointer to an array*

```
int * get_evens() {
static int nums[5];
int k;
int even = 0;

for (k = 0; k < 5; k++) {
nums[k] = even += 2;
}

return (nums);
}
```

Function pointers point to the start of executable code in memory.

Declaration

```
return_type (*func_name)(parameters)
```

```
#include <stdio.h>
void say_hello(int num_times); /* function */

int main() {
void (*funptr)(int); /* function pointer */
funptr = say_hello; /* pointer assignment */
funptr(3); /* function call */

return 0;
}

void say_hello(int num_times) {
int k;
for (k = 0; k < num_times; k++)
printf("Hello\n");
}
```

A *void pointer* if used to refer to any address type in memory.

```
void *ptr;
```

You must type *cast the void pointer* to the appropriate data. Pointer arithmetics can not be perfomed with void pointer.

```
printf(„void ptr points to %d“, *(int *)ptr));
```

Void pointers in a function can *return any type.*

```
#include <stdio.h>

void* square (const void* num);

int main() {
int x, sq_int;
x = 6;
sq_int = square(&x);
printf("%d squared is %d\n", x, sq_int);

return 0;
}

void* square (const void *num) {
int result;
result = (*(int *)num) * (*(int *)num);
return result;
}
```

Function pointers can be used *as parameters*

```
void qsort(void *base, size_t num, size_t width, int (*compare)
(const void *, const void *))
```

void *base A void pointer to the array.
size_t num The number of elements in the array.
size_t width The size of an element.
int (*compare (const void *, const void *) A function pointer which has two arguments and returns 0 when the arguments have the same value, <0 when arg1 comes before arg2, and >0 when arg1 comes after arg2.

Quicksort example, sorting array from high to low

```
#include <stdio.h>
#include <stdlib.h>

int compare (const void *, const void *);

int main() {
int arr[5] = {52, 23, 56, 19, 4};
int num, width, i;

num = sizeof(arr)/sizeof(arr[0]);
width = sizeof(arr[0]);
qsort((void *)arr, num, width, compare);
for (i = 0; i < 5; i++)
printf("%d ", arr[ i ]);

return 0;
}

int compare (const void *elem1, const void *elem2) {
if ((*(int *)elem1) == (*(int *)elem2))
return 0;
else if ((*(int *)elem1) < (*(int *)elem2))
return -1;
else
return 1;
}
```