C programming part 3

More on pointers

```
void *name
```

↑ we use "void" when we desire that the pointer "name" to be utilized with more than one types of data!

Example:

```
int x;
float y;
char z;
void *p;
...
p = &x;
...
p = &y;
...
p = &z;
```

`void *` increases the flexibility of utilizing pointers!

Note: However, this may represent a source of errors! Try to avoid them!

Call by reference vs. Call by value

Call or pass by reference:

```
void my_adder (int *a, int *b, int *c){
    int res = 0;
    res = *a + *b;
    *c = res;
}
```

Description of Function "my_adder"

a, b, c are pointers to data of type integer!

Example:

```
int main ( ) {
    int x = 3;
    int y = 7;
```

```
    int z=0;                          call by reference!
    my_adder(&x, &y, &z); // z gets value 10;
    printf("Our z has value %d !\n", z);
}
```

Note: Using "call by reference", we can modify variables that are passed via reference as arguments of functions!

Example:

```
int my_adder( int a, int *b) {
    int res=0;
    res = a + *b;
    return res;
}
int main( ) {
    int x=3, y=7, z=0;
    z= my_adder (x, &y);
    return 0;
}
```

call or pass by reference we "transfer" the address of y

call by value we "transfer" the value of x.

Relation between pointers and arrays

- The name of an array <u>is</u> a pointer! to which we <u>cannot</u> assign values later in our program! (it is also called a <u>constant pointer</u>). It has as value the address of the <u>first element of the array!</u>

Example:

```
int t[5] = {7, 4, 3, 11, 13};
int *p;
int x;
```

```
p = t;  // p has the same value as t, that is
        // the address of t[0]

x = *p; // x gets assigned the value 7 !
```

Note: Because the name of an array is a (constant) pointer the function declarations:

```
void my_function (int t[]);
void my_function (int *t);
```

are equivalent when we know that we'll utilize (call) "my_function" with an effective parameter/argument that is the name of a one-dimentional array!

Example:

```
void my_function (int t[]){
    int i=0;
    for ( i=0; i < 10; i++){ //assume we know 10 is fixed;
        t[i] = t[i] + 2;
    }
}
int main ( ) {
    int tab[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 101, 102 };
    // add 2 to each element of tab:
    my_funtion (tab);
}
```

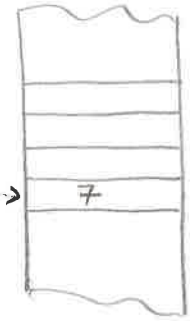
- The point here is that we could have declared (and described) the function "my_function" also as:

```
void my_function (int *t) {
    ...
}
```

Operations on pointers

ⓐ increment and decrement
(++) (--)

```
int a=7; // 16 bits = 2 bytes;
int *p;
p=&a;
p++; // assings to pointer p the address
     // of a to which 2 is added!
```



p has address of variable a

- Operator increment (++) applied to a "pointer to type t", increases the address (i.e, the value of the pointer) with a number equal to the number of bytes necessary (or taken) to store a data of type t!
- Similar statement about operator (--)
- These operations are useful when we work with arrays:

Example:

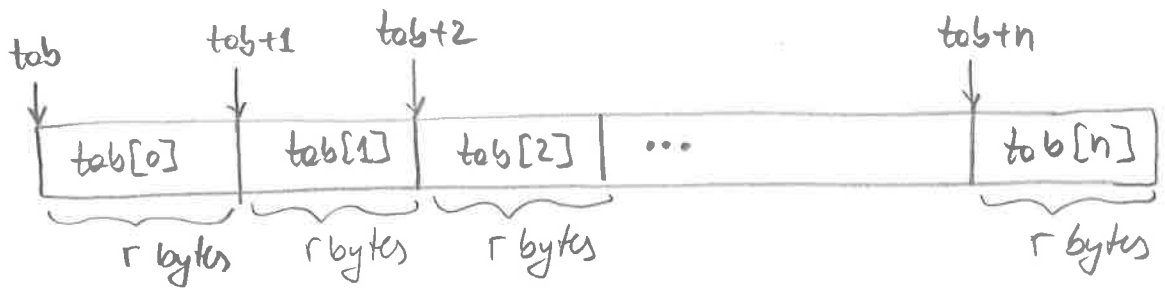
```
double tab[10];
double *p;
int i=3;

p = &tab[i];
p++; //value of p is increased with 8 ! p is the address of tab[4]
printf("value of next element is %f ", *p);
```

ⓑ addition and subtraction of an int from a pointer

```
int n=7; int a=12;
int *p;
p=&a;
p+n; // increases value of p with 7*2 = 14
p-n; // decreases value of p with 14
```

number of bytes to represent an integer!



r = number of bytes (allocated) needed to represent a data of type t (e.g., int, char, double, etc.)

r = 2 1 8 ...

```
double x;
double tab[10];    (n = 9 in figure above)
```

tab + 1 has the address of the element tab[1]

tab + n has as value the address of element tab[n]

- Therefore:

```
x = tab[n]; // is equivalent to:
x = *(tab+n);
```

© **Comparison**

- Two pointers that point to elements of the same array can be compared using operators: relational, equality

```
p = &tab[i];
q = &tab[j];
```

p < q , p <= q , p >= q , p > q , p == q , p != q are legal.

- Operators == and != can be used to compare a pointer to NULL (defined in stdio.h)

```
p == NULL
p != NULL
```

⑥ difference of two pointers

- Two pointers that point to elements of the same array can be substracted.

```
p = &tab[i];
q = &tab[i+k];
```

then:

q-p has value k

Pointers to functions

- The name of a function is a pointer to the function!
- Therefore, the name of a function can be called by its pointer as the argument of a different function for example!

Example:

```
int g(void);
double f( int (*)(void)); // function, which when called receives
                          // as parameter/argument the function g
                          // and returns a value of type double
...
f(g); // f gets as parameter a pointer to function g
...   // because the name of the function is a pointer to that
      // function!
```

Arrays of pointers

```
char *days[] = {
        "illegal",
        "mon",
        "tue",
        "wed",
        "thu", "fri", "sat",
        "sun"
    };
```

days[2] is a pointer to an array of characters "tue"

- we can also utilize:

char *days[8];

but we need to later initialize all elements of the array:

days[0], days[1], days[2], ..., days[7];

with pointers to arrays of characters: "illegal", "mon", ...

Examples:

```
void my_display(char *p[]);
...
my_display(days);
void my_display(char **p);   // equivalent to the above!
                             // because p[] is a pointer that
                             // we can replace with *p.
```

- Because p[i] is a pointer to an array of characters, the characters of this array can be accessed with expressions like:

*(p[i]), *(p[i]+1), *(p[i]+2) ...

or like:

p[i][0], p[i][1], p[i][2] ...

- Arrays of pointers can be utilized for any type.

Example:

```
int *tab[];
```

which can also be replaced by:

```
int **tab;   // preffered! It can be used as a 2 dimensional
             // array!
```

Expressions: tab[i][j] ≡ *(*(tab+i)+j) ≡ *(tab[i]+j)