# Pointers



## 1. Addresses and Pointers

Every variable has an address in memory. Register variables, expressions - unless they have a result variable -, constants, iterals, and preprocessors **don't** have an adress.

We can acess the adress of the variable using the `&`.

- Referencing: creating a pointer points to the address of a variable.

```
int x = 4;
int *pn = &x; // pn is now a pointer to int points to the address of x
```

`int *p` tells that `p` is a pointer. `int` is not the type of the value of the pointer, instead it means that points to a variable that is `int`.

- Dereferencing: accessing the value of the variable using the pointer.

```
*pn = 5; // now x has a value of 5
```

We can define a pointer to char `char *p` to a `int` variable. When dereferencing the `p`, we will acess only the first byte(size of `char`) of the `int` variable.

```
int x = 0x12345678;
// 78 --> first byte
// 56 --> second byte
// 34 --> third byte
// 12 --> fourth byte
char *p = &x;
*p = 0xaa;
printf("x = %x", x);
```

will result in

```
x = 123456aa
```

**Note:** C is a little indian (least in least and most in most)

This is due that the compiler made an implicit casting.

```
unsigned int x = 0x12345678;
unsigned char y;
y = x;
printf("%x", y);
```

will result in

```
78
```

because the `int x` was implicity casted to the `char y` by assigning only the first byte.

**Why pointers?**

1. Optimization: specially when dealing with large structures instead of creating a new variable for this structure, we pass a pointer that points to the original structure and only creating a 4-byte pointer for the original structure instead of creating a whole new structure.
2. Create sophisticated data structures such as linked lists.

3. Traverse arrays.
4. Modify arguments
5. Have functions with multiple outputs.

## 2. Casting Pointers

A direct casting to an address in the memory is wrong because this address is stored in 4 bytes and writitng the adress is done by writing a number `int`.

```
*(0x1234) = 5; // DON'T
```

Instead, we cast it before derefrencing.

```
*((unsigned char *) 0x1234) = 5; // DO
```

Same for

```
int x = 0x1234;
*x = 3; // DON'T
*((unsigned char *) x) = 3; // DO
```

## 3. Pointers Constructions

3.1. Incrementing

Incrementing the pointer depends on its type (the type it is supposed to be pointing at). The increment of a pointer by 1 increments the value of the pointer by 1 multiplied by the size of the type.

`char` --> 1 `int` --> 4

```
int x;
char *pc = &x;
int *pi = &x;
printf("Char: \n");
printf("%x %x\n", pc, pc+1);
printf("Int: \n");
printf("%x %x\n", pi, pi+1);
```

will result in

```
Char:
61fe0c 61fe0d
Int:
61fe0c 61fe10
```

## 4. Arrays and Pointers

A major difference between the array and a pointer is the following

```
int p[10];
int *ptr;

printf("%d ", p+1);
printf("%d ", ptr+1);
```

Both we will result the same. However, in the machine code, there is no `p+1` in the runtime as it is calculated when compiling but for `ptr+1` there is an addition operation.

Another difference when coming to the increments of the addresses.

For the code

```
char x[10];
printf("%x\n",x);
printf("%x\n",&x);
printf("%x\n",&x[0]);
```

The output will be

```
61fdf0
61fdf0
61fdf0
```

So as a value, `x` is equivalent to `&x` and `&x[0]`.

The code

```
char x[4];
printf("%x\n",x);
printf("%x\n",x+1);
printf("%x\n",(&x)+1);
printf("%x\n",(&x[0])+1);
```

will result in

```
61fe1c
61fe1d
61fe20
61fe1d
```

1. 61fe1c => the value of `x`, or `&x` or `&x[0]`.
2. 61fe1d => the value of `x` incremented by 1 as x is a constant pointer to the first element of the array which is char (1 byte)
3. 61fe20 => the value of `&x` incremented by 1 as &x is now a pointer to an array of 4 char (4 slots * 1 byte/slot = 4 slots).
4. 61fe1d => the value of `&x[0]` incremented by 1 as `x[0]` is the first element in the array `x` which is a `char` (1 byte).

To create a pointer to an array of char

```
unsigned char (*p)[5];
printf("%x\n", p);
printf("%x\n", p+1);
```

will result in

```
10
15
```

The code

```
unsigned char x[5];
unsigned char *p1;
p1=x;
printf("%x %x\n", x, p1);
printf("%x %x\n", x+1, p1+1);
```

will result in

```
61fe13 61fe13
61fe14 61fe14
```

As `x` alone is a pointer to the first element.

But the code

```
unsigned char x[5];
unsigned char *p1;
p1=&x;
printf("%x %x\n", x, p1);
printf("%x %x\n", x+1, p1+1);
```

will result in

```
61fe13 61fe13
61fe14 61fe14
```

SAME ANSWER ? **How?**

`&x` is a pointer to an array of 5 chars. However, in the line `p1 = &x p1` is a pointer to char. So, an implicit casting is done to the &x to be a pointer to char and thus incrementing it by 1 gave that result.

In other words, This code

```
unsigned char x[5];
//unsigned char *p1; //COMMENT THIS LINE
unsigned char (*p1)[5];
p1=&x;
printf("%x %x\n", x, p1);
printf("%x %x\n", x+1, p1+1);
```

will give

```
61fe13 61fe13
61fe14 61fe18
```

## 5. Pointer Arithmetrics

```
unsigned int *p1;
unsigned int *p2;
unsigned int *p3;
p1 = 0x04;
p2 = 0x08;
p3 = 0x0c;
printf("%x ", p2-p1);
printf("%x", p3-p1);
```

will give an ouput

```
1 2
```

As

- 1 ==> 0x08 - 0x04 which is 4 bytes divided by the size of int (4 bytes) which gives 1 slot
- 2 ==> 0x0c - 0x04 which is 8 bytes divided by the size of int which gives 2 slots.

**NOTE:** 0x04 + 2 int slots gives 0x0c not 0x12

When dealing with 2 different types of pointers like this code

```
unsigned int *p1;
unsigned char *p2;
p1 = 0x04;
p2 = 0x08;
printf("%x ", p2-p1);
```

will raise an error

```
error: invalid operands to binary - (have 'unsigned char *' and 'unsigned int *')
```

**NOTE**: There is no addition, multiplication, or division of pointers as they don't give meaningful results.

## 6. Pointer to Pointer

To define a pointer to pointer to unsigned int, we write:

```
unsigned int **p2;
```

then we can

```
unsigned int **p2;
unsigned int *p1;
unsigned int x;
p1 = &x;
p2 = &p1;
```

We can access through the pointer to pointer by double dereferncing

```
unsigned char x[3];
unsigned char *p1;
unsigned char **p2;
p1 = &x[0];
p2 = &p1;

// to change x[1] through p2...
(*((*p2)+1)) = 3;
```

## 7. Array of Pointers

To create array of 5 pointers to unsigned char, we write

```
unsinged char * pArr[5];
```

To access a variable from a pointer from array of pointer, we do

```
// We want to change the second element
unsigned int x[2] = {1, 2};
unsigned int (*p1)[2] = &x;
unsigned int * p2[2] = {&x[0], &x[1]};

// Using p1...
(*p1)[1] = 3;
// OR
// *((*p1)+1) = 3

// Using p2...
*(p2[1]) = 3;
// OR
// *(*(p2+1)) = 3
```

## 8. Void Pointers

- They are pointers to void `void *x`.
- They can be used as a parameter in a function or return of a function.
- Can't be deferenced unless they were casted. `*( (unsigned char *) p)`

For example,

```
unsigned char x = 0x12;
void * ptr;
ptr = &x;
printf("%x", *ptr);
```

will give

```
error: invalid use of void expression
```

To fix it, we cast `ptr` before derefrencing it.

```
unsigned char x = 0x12;
void * ptr;
ptr = &x;
printf("%x", *((unsigned char *)ptr));
```

This will result in

```
12
```

## 9. Call By Refrence

If we want to not use the return function or return more than one variable, we can modify the function to receive parameters as pointers.

```
void add(int x, int y, int *sum)
{
    *sum = x + y;
}

int main()
{
    int sum;
    add(5, 6, &sum);
    printf("%d", sum);

}
```

will return

```
11
```

The address of `sum` `&sum` was sent through the function parameters as a refernce of `sum` as pointer to `int` to be modified directly in the function by dereferncing the pointer (refrence) `*sum`.

## 10. WARNING: Pointers can Preserve Scope

```
char *get_message()
{
    char msg[] = "Are not pointers fun?"; //NO
    return msg;
}

int main(void)
{
    char *string = get_message();
    puts(string);
    return 0;
}
```

is a buggy code. **Why?**

Because `msg` is an automatic variable and when the function ends, it is deleted from the stack. Any the address returned by this function can be overwritten by another function. A better explaination can be found here.

## 11. Pointer to Function

We can define a pointer to a function using the following statement.

```
void (*ptr) (int, int, int);
// This defined ptr as a pointer to a function that receives 3 int parameters and doesn't return.
```

Be careful between the above statement and the following

```
void *ptr (int, int, int);
// this is a function called ptr that takes 3 int parameters and return pointer to void.
```

We can assign the pointer to function by

```
void add(int x, int y, int *sum)
{
    *sum = x + y;
}

int main(void)
{
    void (*ptr) (int, int, int*);
    ptr = &add; // exactly the same as ptr = add;
    int m;
    ptr(5, 6, &m); // exactly the same as (*ptr)(5, 6, &m);
}
```

## 12. Structure

Structure is a collection of variables that can be of different data types.

A structure can be created by

```
struct employee
{
    char firstName[30];
    char lastName[30];
    int age;
    char gender;
    double hourlySalary;
};

int main()
{
    struct employee emp;
}
```

Also `typedef` keyword can be used to simplfy the definiton

```
struct employee
{
    char firstName[30];
    char lastName[30];
    int age;
    char gender;
    double hourlySalary;
};

typedef struct employee Employee;
```

or

```
typedef struct employee
{
    char firstName[30];
    char lastName[30];
    int age;
    char gender;
    double hourlySalary;
} Employee;
```

The members of the struct can be accessed using the period `.` then the member name.

```
Employee emp;
//emp.firstName = "Ahmed"; // Not applicable in instead we do...
strcpy(emp.firstName, "Ahmed");
emp.age = 20;
```

A strucute has an addrees, thus, can be referenced by a pointer.

```
Employee emp;
Employee *p;
p = &emp;
strcpy(p->firstName, "Ahmed");
// Same as
strcpy((*p).firstName, "Ahmed");
```

## 13. Bitfields

Bitfields is a set of ajacent bits within a single word. It **must** be used with int, but can be used in char also, by adding a colon `:` after the int variable name then the number of bits to be used for this variable.

```
typedef struct myStruct{
    char mobile[20];
    char *email;
    int age: 3;
} Struct;

int main(void)
{
    Struct s;
    s.age = 5;
    printf("%d", s.age);
}
```

This will result in

```
-3
```

**WHY -3 not 5 although 5 is 101 ?** Because the age is a **signed** 3 bits. So 101 has an MSB of 1 so it's a negative number and to get the value we invert the 2's compliment back to -3.

If we defined `age` as `unsigned int age:3` , the result would be `5` .