The fundamentals of C programming

Part 1

Primitive Data Types

Integers and Floating points

```
int foo;          # Defines an uninitialized integer. What does it mean?
                    We can use expression sizeof(foo) to find out the size.
```

Loops

do/while

I prefer **for** loof

Conditional statements

switch-statement

Is cleaner and can be implemented more efficiently than several if-else cases.

After each case we need to break out of the switch with break.

If you want several cases to run skip break.

Default case doesn't need break as it is the last case.

Write a function called expo that computes the exponential value xˆn.
Example: expo(4,3) = 64.

```
int expo( int a, int b) {
        int sum = 1;
        for( int i = 0; i < b; i++ ){
                        sum *= a;
                }
        return sum
}
```

Works only for positive integers. Can also cause overflow. We can change the **int** declaration to **unsigned int**.

```
unsigned int expo( unsigned int a, unsigned int b )
 {
        int sum = 1;
        for( int i = 0; i < b; i++ )
        {
                sum *= a;
        }
        return sum
}
```

Local and Global Variables

Global variables should in general be avoided. Violates the principle of modularity.
Local Variable can only be used inside a function.
void type means that it is a **procedure**, it does not return a value.

Recursive functions

Write a function that the n factorial, **n!** .
Imperative

```
unsigned int fact( unsigned int n ) {
        sum = 1;
        for( int i = 1; i <= n; i++ ){
                sum *= i;
        }
        return sum;
}
```

Functional, Recursive

```
unsigned int fact( unsigned int n ){
        if ( n <= 1 )
                return 1;
        return  n*fact(n-1);
}
```

Part 2

Arrays, Pointers

Declaration

```
int b[3];
int b[0] = 10;
etc..
```

The number of elements can be computed like this:

sizeof(b) / sizeof(int) = 3

Accessing elements

```
int a[] = {1,2,3,4};        # Implicit size declaration
int k0 = a[0];
int k5 = a[5];              # random number from memory as it is out of bounds
```

Casting has higher precedence than division.

Example:

```
(double) b / a              # will perform the cast before the division
```

Multi-Dimensional Arrays

```
int a[2][4] = {{1,2,3,4},{5,6,7,8}};
```

Note that a print function can have a **const** parameter(read only), but not the function with side effects.

Pointers
Swap values of two variables.
How can we write a function that performs the swap?
Problem 1: A function can only return one value.
Problem 2: Can't change variables outside of function.

Solution: Use pointers

A pointer is defined with the ***** symbol before the variable name in a variable definition.
NOTE: we can also write:
int* p;
& symbol is used for an expression for getting the memory address of a variable.
***** before a pointer variable **dereferences** a pointer, i.e., returns or assigns the value that the pointer points to.

A safer and simpler programming style with reference types is available in C++, but not in C.

Dynamic Memory Allocation

All examples so far have been using statically defined variables or allocating on the stack (local variables).

```
int n = 100;
int *buf = malloc( sizeof( int ) * n);
buf[4] = 10;
printf( "%d\n", buf[4] );
free( buf );
```

malloc dynamically allocates N number of bytes, where N is the argument.

It returns a pointer to the new data.

We can access the array using array indexing.

When the buffer is not needed anymore, it must be deallocated using **free()** .

Floating-Point Numbers

Floating point numbers can represent an approximation of real numbers in a computer. Used heavily in high performance scientific computing.

```
float x = 3.7e-3;
float y = 0.0037;
x == y                  # True, 1
```

Summary:
Arrays and **pointers** are expressive, low-level data structures.

Floating-Point numbers are very useful, but should be used carefully when comparing numbers.