Lecture 1 review

Things that we have seen:

- a simple "hello World!" program
- and its structure, with minimal explanation
  - but we promised things will become clear as we progress
- preprocessor directives starting with #
  - #include
  - #define
- comments, either multi-line with \/* *\/ or single line \/\/
- semicolons ending the line (but not always)
- brackets!
- we had a look at functions, bu this might have been a bad idea ...

Data types

Our program processes data, and everything it does or uses is stored in computer's memory. Basic data types allow for declaration of variables and allocation of necessary resources (space in memory).

- Used to declare variables or define functions.
- Determine size the variable occupies in memory.
- Need format specifiers to print with *printf()*.

We will discuss some (of many) data types. **Integers**, **Floats**, **Characters**, **Bools** and **Void**.

We start with the program from the previous lecture, and recall that declaration of the *main* function is performed with three letters, i.e. **int**.

In [137…

```c
#include <stdio.h>

int main() // main is designated as an int
{
    printf("Hello\n");
}
```

```
Hello
```

int - integral type

Used for storing integers (2, 6, 676, -1000 are integers and 4.5 is not). Use **int** keyword to define. Nowdays ints occupy 4B.

Format specifiers for printf():

- %d %i - signed integer (%li %ld for long).
- %o - Octal integer.
- %x %X - Hex integer.
- %u - unsigned integer.

Let's start with the using the *sizeof()* operator and finding out the size in memory a single **int** variable uses:

In [139…

```c
#include <stdio.h>

int main()
{
    printf("Storage size for int : %ld B \n", sizeof(int)); // Prints the size of int
}
```

```
Storage size for int : 4 B
```

Note: *sizeof()* is used to obtain the size of a type. We use it since this might differ with the change of a compiler or system.

So an *int* occupies 4B. Let now *declare* some variables of type *int*. We do this with the keyword *int* followed by the variable name - an identifier. like this:
int variable_name.
We can *initialize* the variable with a value immediately at declaration, or at a latter time.
Note: we should not let variables uninitialized. The reason for that is that *C* does not guarantee variables have any value at declaration. This means that freshly declared variable initially has an undefined value and we run into danger of using an undefined variable in our program. Consider an example:

In [140…

```c
#include <stdio.h>

int main()
{
    int a = 20;
    int b, c;

    printf("a=%d, b=%d, c=%d\n", a, b, c);
}
```

```
a=20, b=528139024, c=21989
```

Run the above a couple of times. There are some values printed for b and c, but do they make sense? We should modify the program to initialize b a,d c:

In [141…

```c
#include <stdio.h>

int main()
{
    int a = 20;
    int b, c;
    b = 10;
    c = 40;

    printf("a=%d, b=%d, c=%d\n", a, b, c);
}
```

```
a=20, b=10, c=40
```

To print a value with a *printf()* function we need to us a format specifier. Each type has a format specifier that allows *printf()* to print it in the right way. We have already seen the \%d, when we printed some values.
In case of *integers* we can use (amongst others):

- \%d - for decimal representation - represent a number using 0-9 (10 digits)

- \%o - octal representation - represent a number using 0-7 (8 digits)
- \%x - hex representation - represent a number using 16 digits, 0-9 and a-f

Play with the below examples to get better understanding of the concept:

In [142…

```c
#include <stdio.h>

int main()
{
    int a = 15;
    printf( "in decimal representation a=%d \n", a); // 0-9 to represent a number
    printf( "in octal representation a=%o \n", a); // 0-7 to represent a number
    printf( "in hexadecimal representation a=%x \n", a);// ??
}
```

```
in decimal representation a=15
in octal representation a=17
in hexadecimal representation a=f
```

We could reverse this and input the number using either octal or hexadecimal notation.
To write a number in octal we add a leading 0, and in hex a leading 0x:

In [149…

```c
#include <stdio.h>

int main()
{
    int a = 016; // value put in octal representation
    printf( "in decimal representation a=%d \n", a); // 0-9 to represent a number
    printf( "in octal representation a=%o \n", a); // 0-7 to represent a number
    printf( "in hexadecimal representation a=%x \n", a);// ??

    printf("-----\n");
    a = 0x10;  // value input in hex representation
    printf( "in decimal representation a=%d \n", a); // 0-9 to represent a number
    printf( "in octal representation a=%o \n", a); // 0-7 to represent a number
    printf( "in hexadecimal representation a=%x \n", a);// ??
}
```

```
in decimal representation a=14
in octal representation a=16
in hexadecimal representation a=e
-----
in decimal representation a=16
in octal representation a=20
in hexadecimal representation a=10
```

Variables are stored in memory using a finite number of bits. Out *int* uses 4B (32 bits). Consequently, there is a limit to the size of a single integer. If the value is to large it will overflow the memory. For 32 integers (4B is 32 b - as in bits) an *int* is in the range: -2^{31} to $2^{31}-1$.
Consider the following example, and run the same calculations to verify the result:

In [150…

```c
#include <stdio.h>

int main()
{
    printf("Storage size for int : %ld B \n", sizeof(int));

    int a = 320*350*360;
    printf( "a =%d \n ", a);
}
```

```
Storage size for int : 4 B
a =40320000
```

In [151…

```c
#include <stdio.h>

int main()
{
    printf("Storage size for int : %ld B \n", sizeof(int));

    int a = 320*350*360*555;
    printf( "a =%d \n ", a);
}
```

```
/tmp/tmpwqnu2otc.c: In function ‘main’:
/tmp/tmpwqnu2otc.c:7:24: warning: integer overflow in expression of type ‘int’ results in
‘902763520’ [-Woverflow]
    7 |     int a = 320*350*360*555;
      |                        ^
Storage size for int : 4 B
a =902763520
```

- **Note:** We have been warned by the compiler that our operation will overflow the int! It does not always has to be so outspoken about this!. Here we have put values that are known during compilation.
- **Note 2:** The actual value should be: 22377600000, is it?

Operations on integers:

We can perform arithmetic operations on integers. We cann add (+), substract (-), multiply (*) and divide (/). We can also request remainder of the divison with %. See an example:

In [64]:

```c
#include <stdio.h>

int main()
{
    int a=5, b=6; //uninitialized variables, have any value

    printf( "a=%d b=%d\n", a, b);
    printf( "%d+%d = %d\n", a, b, a+b);

    int c = a-b;
    printf( "%d-%d = %d\n", a, b, c);
    c = b-a;
    printf( "%d-%d = %d\n", b, a, c);

    printf( "%d*%d = %d\n", a, b, a*b);

    printf( "%d/%d = %d\n", a, b, a/b); // we remembar the result of this <- Data loss he

    printf( "%d%%%d = %d\n", a, b, a%b);
    printf( "%d%%%d = %d\n", b, a, b%a);
}
```

```
a=5 b=6
5+6 = 11
5-6 = -1
6-5 = 1
5*6 = 30
5/6 = 0
```

```
5%6 = 5
6%5 = 1
```

You might have noticed that division gives strange results. That is 5/6=0! The reason is that the integer only stores the integral value of a number and for this case it would be 0. **So rmember 1/2 is 0**! since both 1 and two are integers.

Floats and doubles

Are used to store floating point numbers, such as 14.35 or 5.4 or any other. We will get to know two:

- **floats** that occupy 4B
- **doubles** that take 8B

To print them use:

- %lf for doubles and %f for floats
- %e or %E for scientyfic notation usefull for small and large values

In [67]:
```c
#include <stdio.h>

int main()
{
    printf("Storage size for float : %ld B \n", sizeof(float));
    printf("Storage size for double : %ld B \n", sizeof(double));
}
```

```
Storage size for float : 4 B
Storage size for double : 8 B
```

Let's now print some floats!

In [73]:
```c
#include <stdio.h>

int main()
{
    float a = 4500;
    float b = 0.00009;
    printf("a=%f\n", a);
    printf("b=%f\n", b);
    printf("a=%e\n", a);
    printf("b=%e\n", b);
}
```

```
a=4500.000000
b=0.000090
a=4.500000e+03
b=9.000000e-05
```

and some doubles:

In [75]:
```c
#include <stdio.h>

int main()
{
    double a = 4500;
    double b = 0.00009;
    printf("a=%lf\n", a);
    printf("b=%lf\n", b);
```

```
    printf("a=%le\n", a);
    printf("b=%le\n", b);
}
```

```
a=4500.000000
b=0.000090
a=4.500000e+03
b=9.000000e-05
```

Similarly to ints floating point numbers allow for addition (+), substraction (-), multiplication (*) and division (/):

In [80]:

```
#include <stdio.h>

int main()
{
    double a = 4.56, b=9.345;

    printf( "a=%lf b=%lf\n", a, b);
    printf( "%lf+%lf = %lf\n", a, b, a+b);
    printf( "%lf-%lf = %lf\n", a, b, a-b);
    printf( "%lf*%lf = %lf\n", a, b, a*b);
    printf( "%lf/%lf = %lf\n", a, b, a/b);
}
```

```
a=4.560000 b=9.345000
4.560000+9.345000 = 13.905000
4.560000-9.345000 = -4.785000
4.560000*9.345000 = 42.613200
4.560000/9.345000 = 0.487961
```

Data casting

Now that we know two data types, we can try to change the way data is treated by means of casting.

Consider the following:

We can use *explicit* data casting. This is done by specifying the type to which the result should be *cast*.
You do this by: **(type)value**.

In [100…

```
#include <stdio.h>

int main()
{
    int a = 1, b=2;

    printf( "%d / %d = %d\n", a, b, a/b); // we loose data here
    printf( "%d %d %lf\n", a, b, (double)a/b); // explicit cast a into a double

    printf( "%d\n", 1/2); // this is wrong to do with %lf!
    printf( "%lf\n", 1/2.);
}
```

```
1 / 2 = 0
1 2 0.500000
0
0.500000
```

What happens there?

1) So in line 7 we try to print a result of division of two ints. Since both operands are of type int the result is assumed to be of the same type. The operation cuts out any decimal part and only the integral part is left. That is why we see a zero.
2) In line 8 we perform an explicit cast by adding (double) to one of the variables (we write (double)a/b) this informs the complier that the result should be treated as a double. This way we see a rpoper result.
3) In lines 9 and 10 we print the result of 1/2, please note that adding a '.' changes the type from int to double! so 1 - an integer, but 1. - a double!

Characters

Our next data type are characters. Those are used to store just that, characters. Those are 1B in size and can be interpreted as integers or symbols. Have a look here: ASCI table for an ASCI table, that is a translator from integer to a character kind of thing.

(ASCI - American Standard Code for Information Interchange)

In [104…

```c
#include <stdio.h>

int main()
{
    printf ( "Storage size for char : %ld B \n" , sizeof(char));
}
```

```
Storage size for char : 1 B
```

In [109…

```c
#include <stdio.h>

int main()
{
    char a = 'b'; // mind that a character is marked with '' not with ""
    printf("%c\n", a);
    printf("%d\n", a);

    a = 0;  // this is a NULL
    printf("as a character %c and as an int %d\n", a, a);

    a = 8; // backspace
    printf("as a character %c and as an int %d\n", a, a);

    a = 48; // backspace
    printf("as a character %c and as an int %d\n", a, a);
}
```

```
b
98
as a character  and as an int 0
as a character □ and as an int 8
as a character 0 and as an int 48
```

bool

Used to represent logical value of *true* and *false*. Introduced into C in C99 standard. in order to use *stdbool.h* needs to be included. Use %d format specifier to print.

The example below illustrates declaration of a boolean variable, assigment and printing.

In [113…

```c
#include <stdio.h>
#include <stdbool.h>

int main(){
    printf ( "Storage size for char : %ld B \n" , sizeof(bool));

    bool a = true; // equivalent to 1
    printf ( "a=%d \n" , a );

    a = false; //equivalent to 0
    printf ( "a=%d \n" , a );
}
```

```
Storage size for char : 1 B
a=1
a=0
```

Void

Void is a special type that represents *nothing*. That is a lot, I hope you agree? It is used to distinguish functions that return no value and therfore are *void*. Variables of this type can not be declared! We will be having a look at *void* type when we get to work with functions and pointers.

In [126…

```c
#include <stdio.h>

void main(){
    printf( " Storage size for void : %ld B \n" , sizeof ( void ) ) ;
    void a; // variables of type void are not allowed
    //printf ( " %d \n" , a ) ;
}
```

```
/tmp/tmphoxu4nf3.c: In function ‘main’:
/tmp/tmphoxu4nf3.c:5:10: error: variable or field ‘a’ declared void
    5 |     void a; // variables of type void are not allowed
      |          ^
[C kernel] GCC exited with code 1, the executable will not be executed
```

Data assigment, data loss and casting

Mixing of types should be, if possible avoided. If necessary it should be done with care.
Here we interpret a double as an int and than back again. As a result we suffer data loss since the output vale is not what it originally was. It is possible that your compiler will complain with a warning.

In [129…

```c
#include <stdio.h>

int main(){
    double x1 = 6.28;
    int a = 2;
    printf("%d %lf \n", a, x1);
    a = x1; // loss of data since a = 6!
    printf("%d %lf \n", a, x1);

    x1 = a;
    printf("%d %lf \n", a, x1);
}
```

```
2 6.280000
6 6.280000
```

```
6 6.000000
```

And here we do the same, but "more consciously", i.e we perform a data cast we have seen earlier.

In [131…

```c
#include <stdio.h>

int main(){
    double x1 = 6.28;
    int a = 2;
    a = (int) x1; // loss of data , but no warning
    printf("%d %lf \n", a, x1);
    x1 = (double) 2/3; // x1 is not zero since I use a cast
    printf("%d %lf \n", a, x1);
}
```

```
6 6.280000
6 0.666667
```

Precedence of operators

It is simple. More less what you know from your math class about the order of execution. So brackets () before multiplication, sumation/substraction and so on. There can be some new concepts on the way though.

In [135…

```c
#include <stdio.h>

int main(){
    double a = 6, b = 9;
    double c = (a+b) / a + b / (a * b);
    printf("%lf \n", c);

    c = (a+b) / a + b / a * b; // it is different!
    printf("%lf \n", c);
}
```

```
2.666667
16.000000
```

Increment / decrement

C offers some operators that are good to know. Incrementation ++ and decremantation -- operators increase or decrease *int* value by 1. Consider the following:

In [5]:

```c
#include <stdio.h>

int main(){
    int a = 1;
    printf("%d \n", a);
    ++a;
    printf("%d \n", a);
    ++a;
    printf("%d \n", a);
    ++a;
    printf("%d \n", a);
    a++;
    printf("%d \n", a);
    a++;
    printf("%d \n", a);
```

```
    a--;
    printf("%d \n", a);
    a--;
    printf("%d \n", a);
    --a;
    printf("%d \n", a);
    --a;
    printf("%d \n", a);
    --a;
    printf("%d \n", a);
    --a;
    printf("%d \n", a);
    --a;
    printf("%d \n", a);
    --a;
    printf("%d \n", a);
}
```

```
1
2
3
4
5
6
5
4
3
2
1
0
-1
-2
```

and:

In [46]:

```
#include <stdio.h>

int main(){
    int a = 1;
    printf("%d \n", a);
    a++;
    printf("%d \n", a);
    a++;
    printf("%d \n", a);

    a--;
    printf("%d \n", a);
    a--;
    printf("%d \n", a);
}
```

```
1
2
3
2
1
```

There are two versions of those operators and apparently the two do the same. But there is a difference. The ++a is called prefix and a++ the postfix. The first is performed first and than passed for possible assigment. The second is first passed for assigment and only than incremented. Consider the two examples and note the resulting values:

In [6]:

```
#include <stdio.h>

int main(){
    int a = 1;
    int b = ++a; // prefix: first increment than copy
    printf("%d %d\n", a, b);
}
```

```
2 2
```

In [8]:

```
#include <stdio.h>

int main(){
    int a = 1;
    int b = a+1; // prefix: first increment than copy
    ++a;
    printf("%d %d\n", a, b);
}
```

```
2 2
```

In [12]:

```
#include <stdio.h>

int main(){
    int a = 1;
    int b = a++; // postfix: first copy than increment
    printf("a=%d b=%d\n", a, b);
}
```

```
a=2 b=1
```

We note that it is better to use the prefix version sice it only invikes coping the value, while the postfix requiers first to create a copy that is passed and than the increamentation, which in case of composite data types can be expensive.

Compound Assignment Operators += -= *= /=

Onother type of operators are the compound operators. Those are used to replace operation with the assigment. Such as a = a + b. Consider the following:

In [15]:

```
#include <stdio.h>

int main(){
    int a = 1;
    int b = 2;
    a += b; // a = a + b;
    printf("a=%d b=%d\n", a, b);
    a -= b; // a = a - b;
    printf("a=%d b=%d\n", a, b);
    a *= b; // a = a * b;
    printf("a=%d b=%d\n", a, b);
    a /= b; // a = a / b;
    printf("a=%d b=%d\n", a, b);
}
```

```
a=3 b=2
a=1 b=2
a=2 b=2
a=1 b=2
```

In [17]:

```c
#include <stdio.h>

int main(){
    double a = 3.14;
    double b = 2.73;
    a *= b; // a = a * b;
    printf("%lf %lf\n", a, b);
    a /= b; // a = a / b;
    printf("%lf %lf\n", a, b);
    a += b; // a = a + b;
    printf("%lf %lf\n", a, b);
    a -= b; // a = a - b;
    printf("%lf %lf\n", a, b);
}
```

```
8.572200 2.730000
3.140000 2.730000
5.870000 2.730000
3.140000 2.730000
```