Branching the code.

basic usage of:

- if()
- else, else if()
- inline if: a = a==b ? a : b
- sitch()

Our ultimate goal for today will be to write a program that solves a quadratic equation. Finds roots of: $ax^2 + bx + c = 0$

if()

if() is based on the concept of TRUE and FALSE, nonzero valus are TRUE and FALSE is zero

First let us see the effect of { } brackets on what is executed

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

 int main()
 {
     if( 1 ) //1 - true

     {printf("This is the first if!!\n");
       printf("Aaaa!\n");
     }

     if( 0 ) //0 - false
     { printf("This is the second if!!\n");
         printf("Bbbb!\n"); }
 }
```

We can use brackets! { }

In [5]:
```
#include <stdio.h>
 int main()
 {
     if(10) //1 - true
     {// I have brackets here!!
         printf("This is the first if!!\n");
         printf("aaaa\n");
     }
     if(0)
     {
         printf("This is the other one if!!\n");
         printf("bbbb\n");
     }
 }
```

```
This is the first if!!
aaaa
```

It seems that brackets set the scope over which the statment apply.

In [7]:
```
#include <stdio.h>
```

```
int main()
{
   int a = 5, b = 4;
   int c = a != b;
   printf("%d\n", c);
}
```

```
0
```

So {} brackets are important since they allow as to execute multi-line instructions

Now for possible operations resulting in a logical value: >, <, >=, <=

In [8]:

```
#include <stdio.h>

int main()
{
    int a = 5;
    int b = 4;
    if(a > b) // this seems to be true!
    {
        printf("A is greater\n");
    }
    if(a < b)//No ; here!!
    {
        printf("B is greater\n");
    }

    if(a >= b)
    {
        printf("A is greater eq\n");
    }
    if(a <= b)//No ; here!!
    {
        printf("B is greater eq\n");
    }

}
```

```
A is greater
A is greater eq
```

and testing the equality is done with == (NOT a single =), inequality is tested with !=

In [9]:

```
#include <stdio.h>

int main()
{
    int a = 5;
    int b = 4;
    //if(!(a = b)) //this is very wrong!!!!
    //{
    //    printf("a=%d b=%d\n", a, b);
    //}
    if(a == b) //this is right!!
    {
        printf("Equal!\n");
    }
    if(a != b) //this is right!!
    {
        printf("Not Equal!\n");
    }
}
```

```
Not Equal!
```

Note: Perform the following exercise. Print the value (using \%d) of comparing ints with <,>,==,!=.
int a = (a == b);

AND and OR

Logical OR and AND are coded as || and &&

In [10]:

```
#include <stdio.h>

int main()
{
    int a = 5;
    int b = 4;


    if(a == 3 || b == 4) // OR
    {
        printf("1 The statment is true\n");
    }


    if(a == 3 && b == 4) // AND
    {
        printf("2 The statment is true\n");
    }
}
```

```
1 The statment is true
```

Precedence of an AND \&\& and OR || is just like for multiplication and addition!

In [13]:

```
#include <stdio.h>

int main()
{
    int a = 5;
    int b = 4;


    if( a == 3 && b == 4 || a == 5 )
    {
        printf("1 The statment is true\n");
    }



    if( (a == 5 || b == 4) && a == 3 )
    {
        printf("2 The statment is true\n");
    }
}
```

```
1 The statment is true
```

if(), else if() and an else:

In [14]:

```
#include <stdio.h>

int main()
```

```
{
    int a = 5;
    int b = 4;

    if(a == b)
    {
        printf("Equal!!\n");
    }
    else
    {
        printf("Else was executed!!\n");
    }
}
```

```
Else was executed!!
```

In [19]:

```
#include <stdio.h>

int main()
{
    int a = 5;
    int b = 5;

    if(a == b)
    {
        printf("Equal!!\n");
    }
    else if( a >= b )
    {
        printf("A is greater!!\n");
    }
    else if( a <= b )
    {
        printf("B is greater!!\n");
    }
    else
    {
        printf("This should not happen\n");
    }
}
```

```
Equal!!
```

Note what happens if I remove the **else**

In [20]:

```
#include <stdio.h>

int main()
{
    int a = 5;
    int b = 5;

    if(a == b)
    {
        printf("Equal!!\n");
    }
    if( a >= b )
    {
        printf("A is greater!!\n");
    }
    if( a <= b )
    {
        printf("B is greater!!\n");
    }
```

```
    else
    {
        printf("This should not happen\n");
    }
}
```

```
Equal!!
A is greater!!
B is greater!!
```

Nested if()

An example of nested if(), i.e. if() in an if() in an if() ...

In [ ]:

```
#include <stdio.h>

int main()
{
    int a = 10;
    int b = 5;

    if(a > b) // nested if()
    {
        if(a > 2*b)
        {
            printf("A is very large\n");
        }
        else if(a < 2*b)
        {
            printf("A is greater\n");
        }
        else
        {
            printf("A is 2B\n");
        }
    }
    else
    {
        printf("B is greater than A\n");
    }
}
```

Inline if statment:

value = logical test ? value if true : value if false

I would like d to be the sum of a and greater of b and c

In [ ]:

```
#include <stdio.h>

int main()
{
    int a = 10;
    int b = 5;
    int c = 7;

    int d;

    if(b > c)
        d = a + b;
    else
```

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

With an inline if statment

In [21]:

```
#include <stdio.h>

int main()
{
    int a = 10;
    int b = 5;
    int c = 7;

    int d = b > c ? a + b : a + c;
    printf("%d\n", d);
}
```

17

switch()

is used to select statments to be executed based on the value of an expression evaluating to an intiger

In [ ]:

```
#include <stdio.h>

int main(){
    int a = 1;

    switch( a )
    {
        case 1:
            printf("This is the first case\n");
        //break;

        case 3:
            printf("This is the second case\n");
        //break;

        case 5:
            printf("This is the third case\n");
        break;

        default:
            printf("This is the default\n");
    }
}
```

In [22]:

```
#include <stdio.h>

int main(){
    char c ;
printf("Enter the arithmetic opreation you want to perform \n");
scan("%c", &c);


    switch( c )
    {
        case "+":
```

```
            printf("This is the first case\n");
        //break;

        case "-":
            printf("This is the second case\n");
        //break;

        case "*":
            printf("This is the third case\n");
        break;

        default:
            printf("This is the default\n");
    }
}
```

```
/tmp/tmpesow_zdo.c: In function ‘main’:
/tmp/tmpesow_zdo.c:6:1: warning: implicit declaration of function ‘scan’; did you mean ‘sc
anf’? [-Wimplicit-function-declaration]
    6 | scan("%c", &c);
      | ^~~~
      | scanf
/tmp/tmpesow_zdo.c:11:9: error: case label does not reduce to an integer constant
   11 |         case "+":
      |         ^~~~
/tmp/tmpesow_zdo.c:15:9: error: case label does not reduce to an integer constant
   15 |         case "-":
      |         ^~~~
/tmp/tmpesow_zdo.c:19:9: error: case label does not reduce to an integer constant
   19 |         case "*":
      |         ^~~~
[C kernel] GCC exited with code 1, the executable will not be executed
```

In [ ]:

```
#include <stdio.h>

int main(){
  int a;
  //scanf("%d", &a);

  a = 6;
  int b = 5;

  switch(a-b)
  {
    case 1: // the value here is what needs to be evaluated in the switch statment
      printf("This is the first case\n");
      //possibly many lines
      break;
      //some more instructions?

    case 3:
      printf("This is the second case\n");
      break;

    case 4:
      printf("This is the 4th case\n");
      break;

    default:
      printf("Your choiche is unrecognized!!\n");
  }
}
```

Example

Let us now solve the quadratic equation

$ax^2 + bx + c = 0$

$\Delta = b^2 - 4ac$

$x_{1,2} = \frac{-b \pm \sqrt{\Delta}}{2a}$

In [ ]:

```c
//%cflags:-lm

#include <stdio.h>
#include <math.h>

int main()
{
    double a, b, c;
    // Get the user to input a, b, c
    scanf("%lf", &a);
    scanf("%lf", &b);
    scanf("%lf", &c);

    a = 1; // I need to put those in sice I have no access to the terminal
    b = 4;
    c = 1;

    double x0, x1;

    // 1. Calculate the delta
    double del = b*b - 4*a*c;
    if(del < 0)
    {
        printf("There is no solution\n");
        return 0; // this terminates the main function
    }
    else if(del == 0)
    {
        printf("There is only one solution!\n");
        x0 = -b / (2*a);
        printf("x0=%lf\n", x0);
    }
    else
    {
        printf("There is two solution!\n");
        x0 = (-b-sqrt(del)) / (2*a);
        x1 = (-b+sqrt(del)) / (2*a);
        printf("x0=%lf\n", x0);
        printf("x1=%lf\n", x1);
    }

    //printf("x0=%lf\n", x0);
    //printf("x1=%lf\n", x1);
}
```

Functions in C

The way we program in **C** is the so called *procedural* programming. This means that we go around solving our problems, that are defined by data, by writing down (implementing) *procedures* that

process the data and pass it from a one procedure to another.
When considering a programming (or in fact any) problem one of the most robust and successful approaches is the 'divide and conquer' strategy. This approach comes down to splitting the big problem into small, manageable chunks. We than solve each chunk as a separate problem and construct the final solution by connecting individual solutions together.

How do we achieve this in **C**? We use the concept of *functions*. We have already seen some functions. *main()* is a function (somewhat special), *printf()*, *scanf()* and all the mathematical library procedures (*sqrt()*, *pow()*, *log()* ...) are just functions. We will now get acquainted with writing our own functions and than we will attempt at solving the roots of a quadratic equation example using functions.

A function must have a type. This can be any of the types we already know. The *int*, *double* or even *void* type functions can be written. The type of the function depends on what we want the function to do. For example, if we want a function that calculates a value that we know is an integer than our function will be an *int* type. If we are interested in a floating point value (say, we implement some mathematical formula) it is going to be a double. If the function does not need to provide any value (or we deal with values differently - following lectures) it is going to be a *void* type.
The general syntax is:

In [ ]:
```
type name(list of argument)
{

}
```

A function is defined by its type, the name and also by the arguments it is taking. The arguments are variables that are passed to the function **by value**. This does not mean much now, but it will be a major difficulty to be overcome during our studies.

Let start with a simple example. A single function of type *int* with an empty argument list (does not take any arguments) that prints a message and returns 0 (we could return any integral value, we choose 0 because it is a nice round number).

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

int fun()
{
    printf("Fun has been called\n");
    return 0;
}

int main()
{
    printf("This is main()\n");

    fun(); // this calls the fun function
}
```

We define the function (lines 3-7) before we use (call) it in the main function (line 13). Please note that we do nothing with the value returned by the function. We could, but for now we choose not to.

Let us now augment the example by a new function, also with type *int* and accepting a single *int* type argument. The function returns the argument value increased by 5.

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

int fun()
{
    printf("Fun has been called\n");
    return 0;
}

int fun2(int arg1) // return an argument incresed by 5
{
    return arg1 + 5;
}

int main()
{
    printf("This is main()\n");

    fun(); // this calls the fun function
    int a = 5;
    int b;
    b = fun2(a);
    printf("a=%d b=%d \n", a, b);
}
```

The function is called in line 21, and the returned value is stored on to the variable b. Than printed. Take a second to see if the values printed is what you expect.

We will now illustrate the consequence of th fact that arguments are passed into the function **by value** (or as a copy). We design a function, this time of type *void*, that accepts two integers (referred in the function as *a* and *b*), prints them and modifies them.

In [ ]:
```c
//%cflags:-lm
#include <stdio.h>
#include <math.h>

void fun(int a, double b)
{
    printf("From the function a=%d, b=%lf\n", a, b);
    a = a + 5;
    b = 4.0*atan(1.) * b;
    printf("From the function a=%d, b=%lf\n", a, b);
    // no return!
}

int main()
{
    printf("This is main()\n");

    int var1 = 0;
    double var2 = 10.0;
    printf("a=%d, b=%lf\n", var1, var2);

    fun(var1, var2);
    printf("a=%d, b=%lf\n", var1, var2);
}
```

Follow the values printed by the program. Are all values as you expect?

Let see what happened:

- We declare two variables in lines 18-19 and assign those with some arbitrary values.
- In line 22 the *fun* function is called, and we observe printouts from lines 7 and 9. We note that the values have changed.
- The function call terminates and operation returns to the *main()* function at line 23. Are the values printed in in line 23 what we expect? Those are the original values declared in main, and not the ones resulting from the function!

We note: **Values changed inside of a function do not propagate to the outside. What happens in a function stays in the function.**
Or in other words values placed in the argument list are provided by value (through copy) and all operations are performed on those copies, and not the original variables.

Global variables

To deal with the problem of propagating variable changes made inside functions we will use the concept of **global** variables. Those are variables declared outside any other function, usually at the top of the file and just below *include* statements. Those variables are available to all functions defined in this file, and any change to them is seen everywhere, thus the name **global**.

Note: In future we will try to limit the use of global variables to a minimum as those can be really troublesome in larger projects.

Consider the previous example, but this time with the use of global variables. Note the declaration of global variables a lines 6-7, the empty argument list of the *fun()* function and missing declaration of **local** variables removed from the *main()*:

In [ ]:

```
//%cflags:-lm
#include <stdio.h>
#include <math.h>

//Global variables
int a=0;
double b=10;

void fun()
{
    printf("From the function a=%d, b=%lf\n", a, b);
    a = a + 5;
    b = 4.0*atan(1.) * b;
    printf("From the function a=%d, b=%lf\n", a, b);
}

int main()
{
    printf("This is main()\n");

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

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

This time values have changed as we intend them to.

Example revisited

This time we will try to make it with functions

Let us now solve the quadratic equation

$$ax^2 + bx + c = 0$$

$$\Delta = b^2 - 4ac$$

$$x_{1,2} = \frac{-b \pm \sqrt{\Delta}}{2a}$$

1. Greet the user function?
2. A function to read from keyboard
3. A function to calculate the delta
4. A functions that calculates roots
5. A printout function

In [ ]:

```
//%cflags:-lm
#include <stdio.h>
#include <math.h>

// Global coefficients
double a, b, c;
//global roots
double x1, x2;

void greetings()
{
    printf("Hello user, this is a quadratic function solver\n");
    printf("Start by providing the a, b, and c coefficients.\n");
}

void read_abc()
{
    scanf("%lf", &a);
    scanf("%lf", &b);
    scanf("%lf", &c);
    // I need to provide the values since I am using the Jupyter
    a = 1.0;
    b= 2.0;
    c = 1.0;
}

double calculate_delta()
{
    return b*b - 4.0 * a * c;
}

int roots(double d)
{
    int nr;
    if(d < 0)
    {
        nr = 0;
    }
    else if(d > 0)
    {
        nr = 2;
        x1 = (-b+sqrt(d))/(2.0*a);
```

```
        x2 = (-b-sqrt(d))/(2.0*a);
    }
    else
    {
        nr = 1;
        x1 = x2 = -b/(2.0*a);
    }

    return nr;
}
void print_out(int nr);

int main()
{
    greetings();
    read_abc();
    double delta = calculate_delta();
    int number_of_roots = roots(delta);
    print_out(number_of_roots);
}

void print_out(int nr)
{
    switch(nr)
    {
        case 0:
            printf("There are no roots!\n");
            break;
        case 1:
            printf("There is one root!\n");
            printf("x=%lf!\n", x1);
            break;
        case 2:
            printf("There are two roots!\n");
            printf("x1=%lf!\n", x1);
            printf("x2=%lf!\n", x2);
            break;
        default:
            printf("Stupid user exeption!\n");
    }
}
```