Functions I

1. Content

2. Objective

At the end of these exercises, you should be able to:

- Declare, define and call functions having parameters and return values
- Debug the code using the Visual Studio debug tools
  - Enable/disable breakpoints
  - Execute the code step by step
  - Watch the content of the variables
- Define and use overloading
- Use the elapsed seconds to calculate the displacement of a moving object
- Calculate the velocity and displacement of an object in free fall.
- Do operations with a 2D vector

We advise you to **make your own summary of topics** that are new to you.

3. Some tips about functions

3.1. Name/identifier

Have a look at the document about naming in the folder 00_General/ManualsAndGuides.

Do not hesitate to give functions that perform the same task but with different argument lists the same name (overloading).

3.2. Parameters and return type

Always take a moment to decide on:

- What parameters and their type
- Whether the function should return a value and the type of this value.

4. Exercises

Your name, first name and group should be mentioned at the top of each cpp file.

Give your **projects** the same **name** as mentioned in the title of the exercise, e.g. **FunctionsBasics**. Other names will be rejected.

4.1. FunctionsBasics

Create a new project with name **FunctionsBasics** in your **1DAExx_06_name_firstname** folder.

Add your name and group as comment at the top of the cpp file.

4.1.1. Debugging the code

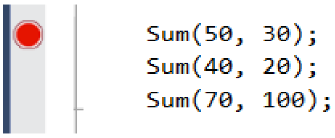
Make a Sum function and step through the code.

The Sum function has two parameters of int type and no return value (for now). It calculates the sum and prints the value of the two arguments and the sum.

Stepping through the code and watching the content of the variables not only helps you in locating errors in your code but also makes you better understand how the program flow (= the order in which code is executed) works, especially when calling functions.

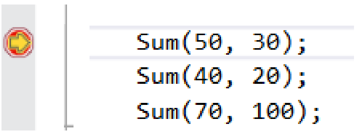
a. Activate a breakpoint

Activate a **breakpoint** next to the first call of the Sum method (just click in the margin next to this line).



```
    Sum(50, 30);
    Sum(40, 20);
    Sum(70, 100);
```

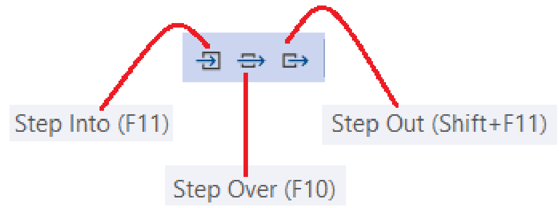
Run the project and notice that the execution stops at the breakpoint indicated by the yellow arrow.



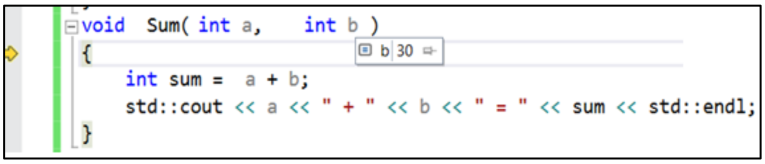
```
    Sum(50, 30);
    Sum(40, 20);
    Sum(70, 100);
```

b. Step through the code

Now you can execute the following code step by step using the buttons or functions keys as mentioned in the picture below.

	Step into – **F11**: steps into the method that is called. Step over – **F10**: executes the method but doesn't step into it Step out – **SHIFT + F11**: executes the remaining part of the method without stepping through it.

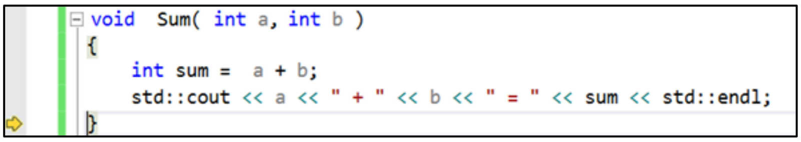
Step into (F11) the first Sum-call and hover with the mouse over the parameters and notice that these contain the values 50 and 30.



```
void  Sum( int a,    int b )
{                        b 30
    int sum =  a + b;
    std::cout << a << " + " << b << " = " << sum << std::endl;
}
```

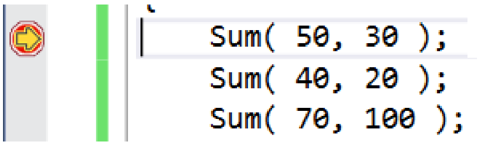
When you step further using **step into** F11 you'll reach code that's part of Microsoft's standard library, you can leave this code using **step out (SHIFT + F11)**

Step further in the Sum method using **step over (F10)**, which will prevent you to step into the calls of functions that you don't need to investigate for now (as those of the standard library).



```
void  Sum( int a, int b )
{
    int sum =  a + b;
    std::cout << a << " + " << b << " = " << sum << std::endl;
}
```

Notice that when having reached the closing curly brace, the next **step** command returns to the code that called the method.

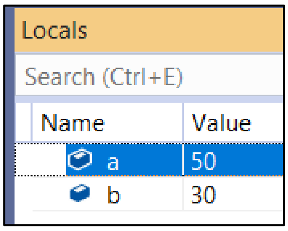


```
    Sum( 50, 30 );
    Sum( 40, 20 );
    Sum( 70, 100 );
```

Then again use **step over** (F10) and notice that the second Sum call is executed without going into it.

c. Content of variables in the Local window

Hovering over a variable shows you its content. However, you can also see the content of the variables in the **Locals window.**

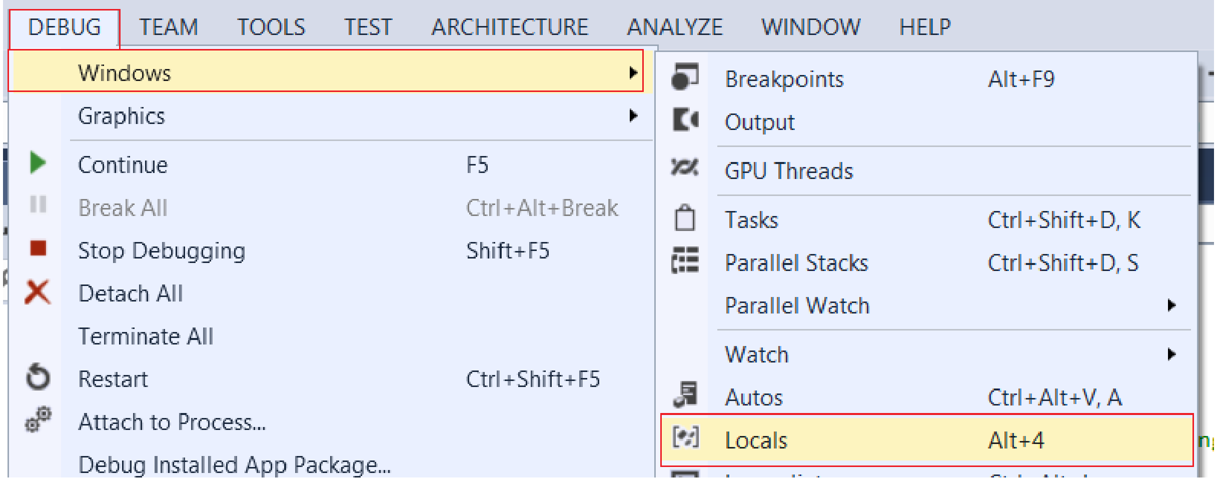


Locals

Search (Ctrl+E)

Name	Value
a	50
b	30

This window becomes visible via menu Debug>Windows>Locals

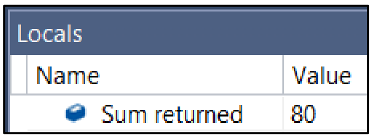


d. Deactivate breakpoint and continue

When you no longer need to investigate you code and no longer want that the execution stops at the breakpoint, you just click on the red circle to make it disappear. Pressing the continue button ▶ Continue ▾ or **F5** continues the execution.

e. Using the return value

Change the Sum function, instead of printing the result on the console, just return it so that this result is usable outside the function. Assign the result of the Sum call to a variable and investigate the execution of this code by stepping through it. Notice that the return value is displayed in the Locals window as soon as you leave the function.

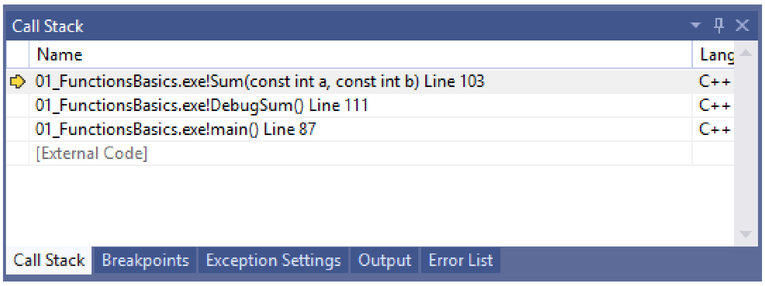


Locals

Name	Value
Sum returned	80

f. Call stack

The call stack can be used to "go back in time". It allows you to see what happenend before the break point was reached. Double clicking a line shows the state of the variables on that line. Explore it because it can be very usefull to trace back function calls and solving bugs. Press alt + F7 if the call stack window is not there when on a break point.



Name	Lang
01_FunctionsBasics.exe!Sum(const int a, const int b) Line 103	C++
01_FunctionsBasics.exe!DebugSum() Line 111	C++
01_FunctionsBasics.exe!main() Line 87	C++
[External Code]	

4.1.2. Function overloading

Given are the following function declarations

```
void PrintInt(int value, char delimiter );
void PrintChar(char value, char delimiter );
void PrintFloat(float value, char delimiter );
void TestPrint( );
```

And the following function definitions

```
void PrintInt(int value, char delimiter )
{
        std::cout << value << delimiter;
}
void PrintChar(char value, char delimiter )
{
        std::cout << value << delimiter;
}
void PrintFloat(float value, char delimiter )
{
        std::cout << value << delimiter;
}


void TestPrint( )
{
        PrintChar( 'a', ',' );
        PrintInt( 20, ',' );
        PrintFloat( 20.0f, ',' );
}
```

Instead of inventing different function names for functions that do the same task but for different argument lists (here: "printing a value followed by a delimiter") use the same name for the Print… functions.

So change the code into Print functions with the same name e.g. Print. Then step through the code execution and watch how the fitting versions of the Print functions are called.

Change the Print function that prints an integer: change the value type into an `unsigned int` instead of `int`. Now you get an error, can you explain the reason of this error ? This is called an ambiguous function call. See Function overloading.

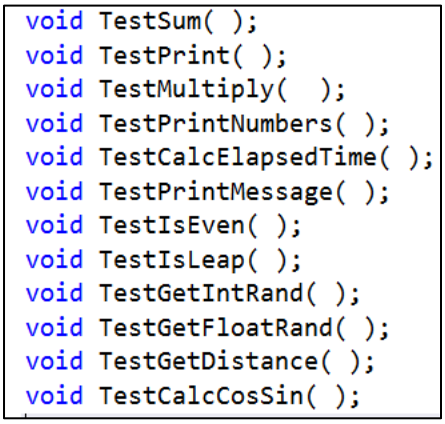
4.1.3. Simple functions

In this section you'll declare and define the following - rather simple – functions, these are their names.

- Multiply
- Modulo
- PrintNumbers
- PrintMessage
- IsEven
- IsLeap
- GetRand
- GetDistance
- CalcElapsedTime

Each function is described in more detail further in this document.

Test them profoundly by calling them using different arguments or by asking input from the user. Debug your code. Make your code easy-to-read by splitting up de main function in functions in which you call the new function in several ways. Another advantage is that you don't need to execute them all. Once one of them works you can comment it. We have these test functions.

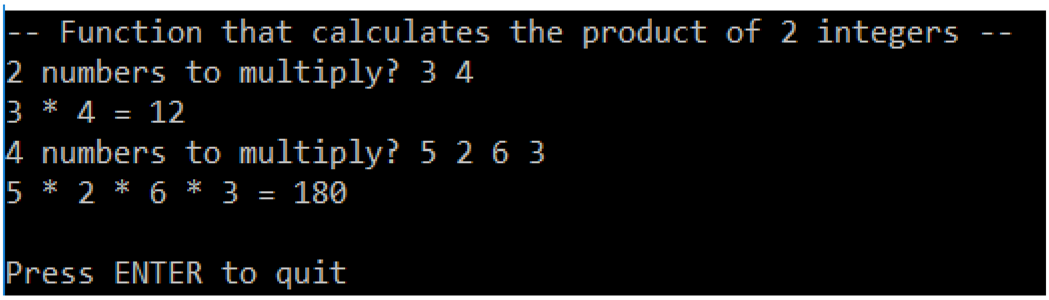


```
void TestSum( );
void TestPrint( );
void TestMultiply(  );
void TestPrintNumbers( );
void TestCalcElapsedTime( );
void TestPrintMessage( );
void TestIsEven( );
void TestIsLeap( );
void TestGetIntRand( );
void TestGetFloatRand( );
void TestGetDistance( );
void TestCalcCosSin( );
```

a. Multiply function

Declare and define a function **Multiply** that has two int parameters and returns the **product** of both parameters.
Declare and define a second function **TestMultiply** and call it in the main function. In TestMultiply you:

- Print a message on the console describing which function you will use
- Ask the user to enter 2 numbers to multiply and show the result after calling your Multiply function.
- Ask the user to enter 4 numbers to multiply and show the result after calling your Multiply function 3 times. Can you do this multiplication without storing intermediate results ? Thus using the result of 2 Multiply calls as arguments for a third Multiply call.



```
-- Function that calculates the product of 2 integers --
2 numbers to multiply? 3 4
3 * 4 = 12
4 numbers to multiply? 5 2 6 3
5 * 2 * 6 * 3 = 180

Press ENTER to quit
```

b. Modulo function

Make this function that returns the modulo of a number. As parameters, there is the initial number, and the divisor. The functions returns the result of the modulo operation.

Use **only** the **multiplication**, **division** and **subtraction** operators.

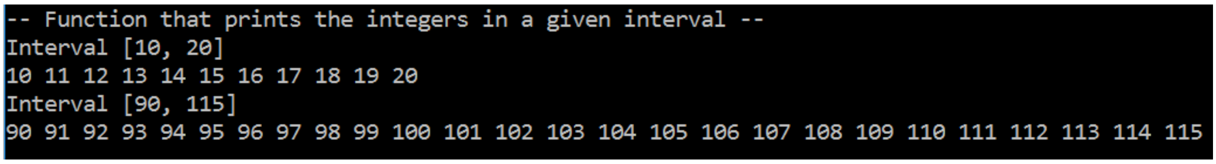
You are not allowed to use the modulo operator (%) except for checking the result.

Google on to do this, or click here.

c. PrintNumbers function

Declare and define a function **PrintNumbers** that **prints the integers** in a given interval [start, end].
Declare and define another function **TestPrintNumbers** and call it in the main function. In TestPrintNumbers you:

- Print a message on the console describing which function you will call
- Call PrintNumbers several times with different arguments



```
-- Function that prints the integers in a given interval --
Interval [10, 20]
10 11 12 13 14 15 16 17 18 19 20
Interval [90, 115]
90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115
```

d. CalcElapsedTime function

Declare and define a function **CalcElapsedTime** that prints how many milliseconds it takes to print the integers in a given interval (you already have a function that prints numbers in an interval, you can call it). Use the **std::chrono::steady_clock** type to measure the time. As the full name implies, this clock resides in the namespace **std::chrono.** And it is defined in the header **<chrono>.**
So the CalcElapsedTime function:

- gets the current time (startTime),
- does the printing,
- gets again the current time (endTime),
- subtracts these 2 time values, which gives you the duration of the printing;
- prints this duration

Below is some more information about chrono. Warning: copying/pasting this code, generates errors!

Use the **now** function of this clock to initialize the time before (startTime) and after (endTime) the printing task, like this:

```
std::chrono::steady_clock::time_point startTime{std::chrono::steady_clock::now( )};
```

As you can see the result of this call is a **time_point** type.

Then get the elapsed time by subtracting both values indicating that you want the result in milliseconds of type float, like this:

```
std::chrono::duration<float, std::milli> elapsedTime{ endTime - startTime};
```

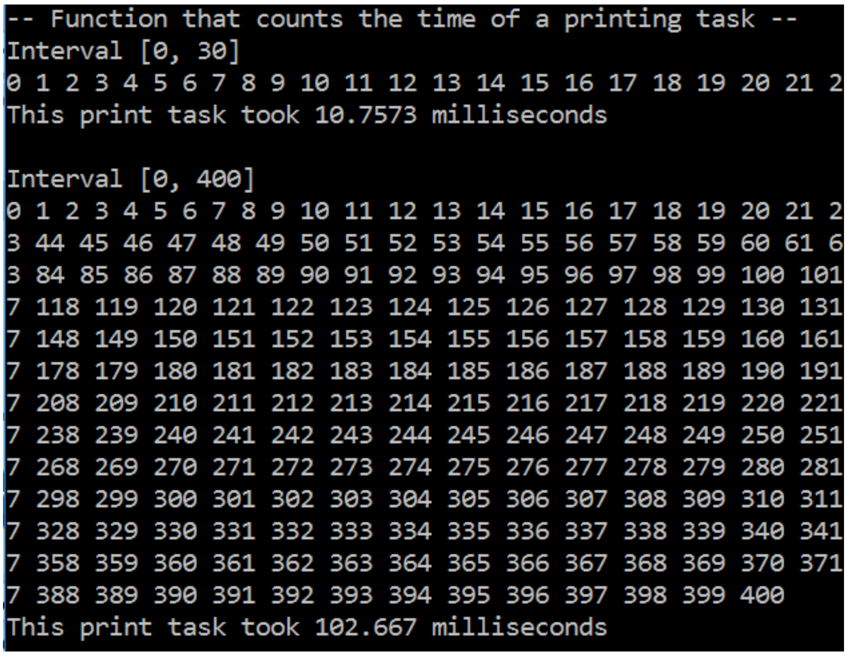
Notice that the result of this call is of **duration** type.

Then print this elapsed time, however std::cout doesn't know the duration type, therefor use its count() function which converts it to a float, a type known by std::cout, like this:

```
elapsedTime.count( )
```

Declare and define another function **TestCalcElapsedTime** and call it in the main function. In **TestCalcElapsedTime** you:

- Print a message on the console describing you're testing the CalcElapsedTime function
- Then call CalcElapsedTime several times with different arguments

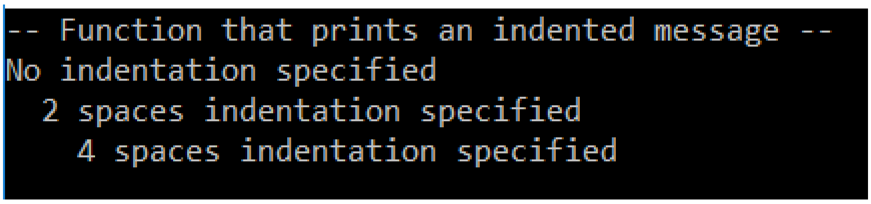


```
-- Function that counts the time of a printing task --
Interval [0, 30]
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 2
This print task took 10.7573 milliseconds

Interval [0, 400]
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 2
3 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 6
3 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101
7 118 119 120 121 122 123 124 125 126 127 128 129 130 131
7 148 149 150 151 152 153 154 155 156 157 158 159 160 161
7 178 179 180 181 182 183 184 185 186 187 188 189 190 191
7 208 209 210 211 212 213 214 215 216 217 218 219 220 221
7 238 239 240 241 242 243 244 245 246 247 248 249 250 251
7 268 269 270 271 272 273 274 275 276 277 278 279 280 281
7 298 299 300 301 302 303 304 305 306 307 308 309 310 311
7 328 329 330 331 332 333 334 335 336 337 338 339 340 341
7 358 359 360 361 362 363 364 365 366 367 368 369 370 371
7 388 389 390 391 392 393 394 395 396 397 398 399 400
This print task took 102.667 milliseconds
```

e. PrintMessage function

Declare and define a function **PrintMessage** that **prints a message** on the window with a specified number of leading spaces and followed by a new line. Make this number of spaces optional, default is no indentation.
Declare and define another function **TestPrintMessage** and call it in the main function. In **TestPrintMessage** you:

- Print a message on the console describing which function you're testing
- Call that function several times with different arguments



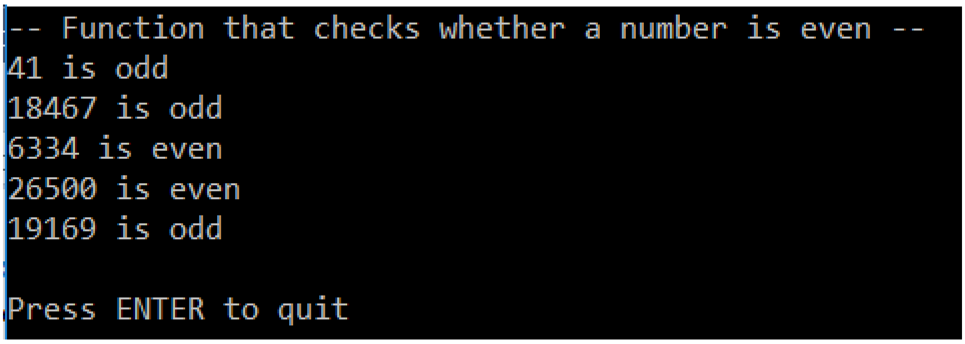
```
-- Function that prints an indented message --
No indentation specified
  2 spaces indentation specified
    4 spaces indentation specified
```

f. IsEven function

Declare and define a function **IsEven** that checks whether a given integer is **even**.

Declare and define another function **TestIsEven** and call it in the main function. This **TestIsEven** function:

- Prints a message “—Function that checks whether a number is even –“ on the console
- Calls the function IsEven several times, and after each call prints the result of this call together with the tested value. So the printing happens outside the IsEven function, this is called **separation of concerns** (**SoC**).



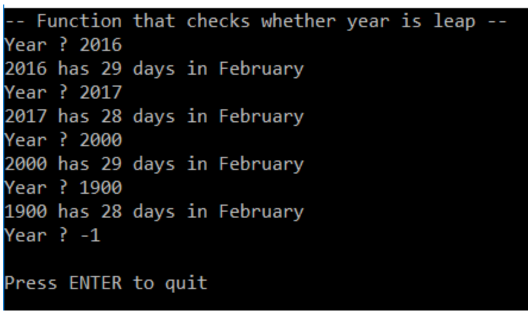
```
-- Function that checks whether a number is even --
41 is odd
18467 is odd
6334 is even
26500 is even
19169 is odd

Press ENTER to quit
```

g. IsLeapYear function

Declare and define a function **IsLeapYear** that checks whether a given year is a **leap year**.

Declare and define a function **TestIsLeapYear**, in this function:

- Print a message on the console: “—Function that checks …”
- In a loop that stops when the user enters -1
    - Asks the user to enter a year
    - Checks whether this is a leap year by calling the function IsLeapYear
    - Prints the result of this call on the console (so again put the printing code outside the IsLeapYear function).



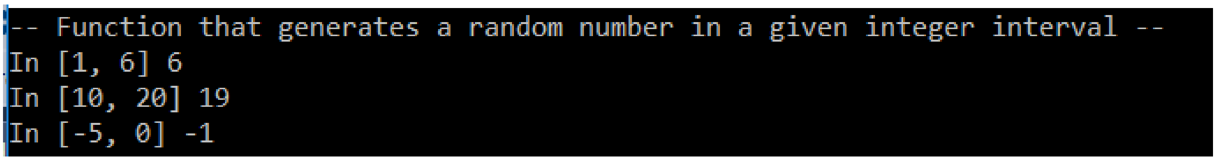
```
-- Function that checks whether year is leap --
Year ? 2016
2016 has 29 days in February
Year ? 2017
2017 has 28 days in February
Year ? 2000
2000 has 29 days in February
Year ? 1900
1900 has 28 days in February
Year ? -1

Press ENTER to quit
```

h. GetRand in integer interval

Declare and define a function **GetRand** that results in a **random integer** number within an inclusive integer interval as defined by 2 parameters.

Again, test the function profoundly. To that end, declare and define a function **TestGetIntRand**, in this function:

- Print a message on the console
- Call the GetRand function several times using different int arguments and display the result together with the interval on the console.



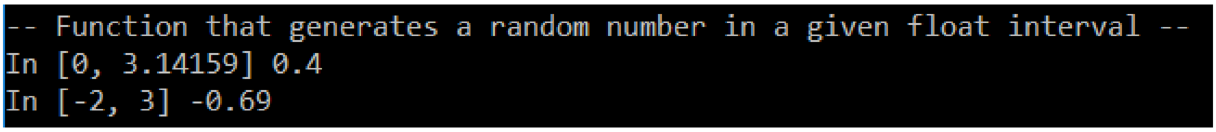
```
-- Function that generates a random number in a given integer interval --
In [1, 6] 6
In [10, 20] 19
In [-5, 0] -1
```

i. GetRand in float interval

Declare and define a function **GetRand** that results in a **random float** number with 2 digits precision after the decimal point within an inclusive float interval.

Again, test the function profoundly. Declare and define a function **TestGetFloatRand**, and in this function:

- Print a message on the console
- Call the GetRand function several times using different float arguments and display the result together with the related interval on the console.



```
-- Function that generates a random number in a given float interval --
In [0, 3.14159] 0.4
In [-2, 3] -0.69
```

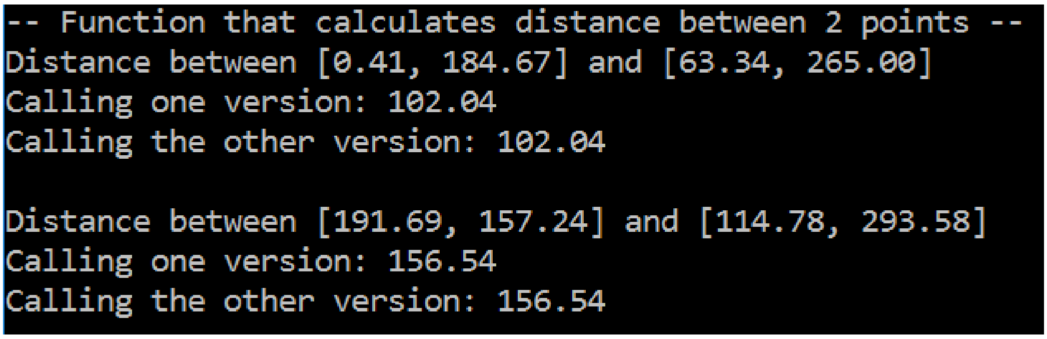
j. GetDistance

Declare and define a function **GetDistance** that calculates and returns the **distance** between two points. Make two versions of this function using overloading:

- One having Point2f struct as parameter types.
- The other one having float parameter types.

Declare and define a function **TestGetDistance**, in this function:

- Print a message on the console
- Call both GetDistance functions several times using different arguments and display the result together with the 2 points coordinates on the console. Hereby use GetRand to get random coordinates.



```
-- Function that calculates distance between 2 points --
Distance between [0.41, 184.67] and [63.34, 265.00]
Calling one version: 102.04
Calling the other version: 102.04

Distance between [191.69, 157.24] and [114.78, 293.58]
Calling one version: 156.54
Calling the other version: 156.54
```

4.2. DrawFunctions

Create a new project with name **DrawFunctions** in your **1DAExx_06_name_firstname** folder.

Delete the generated file **DrawFunctions.cpp.**

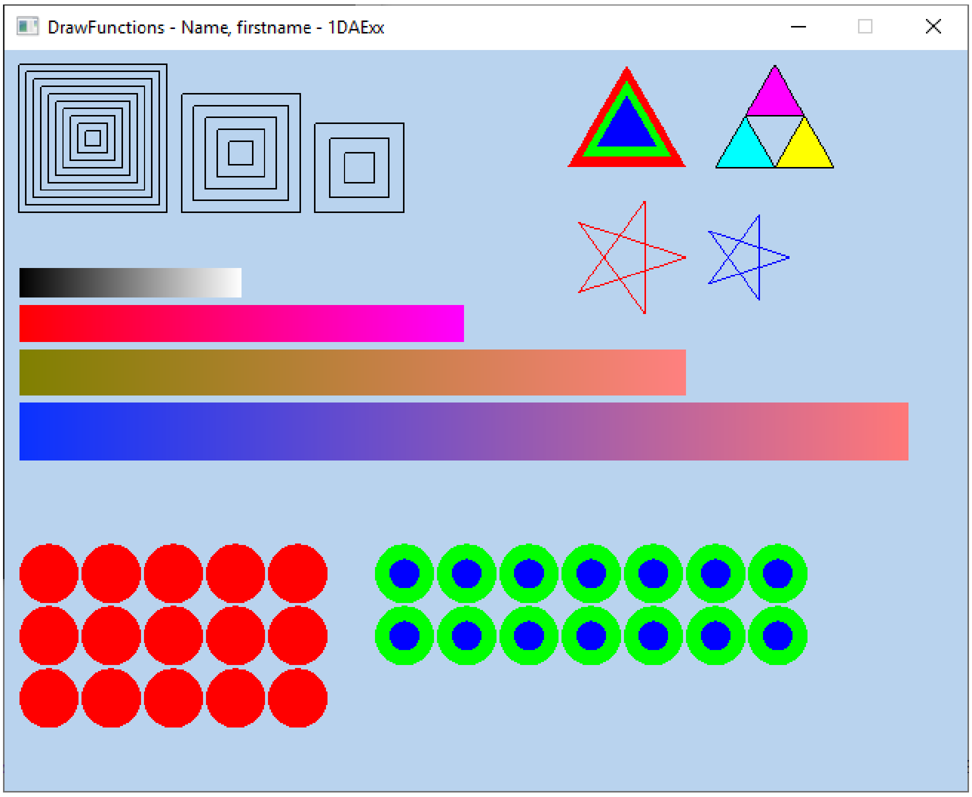
Use the framework.

Adapt the window title.

In this project you'll **declare**, **define** and **call functions that draw various geometric shapes**. The functions are described in more detail further in this document. However:

- You should decide yourself about the parameter lists.
- Don't hesitate to make overloaded functions, e.g.: 2 float parameters vs 1 Point2f parameter in case of 2D coordinates of a point.
- Use pass by value, default values whenever appropriate.

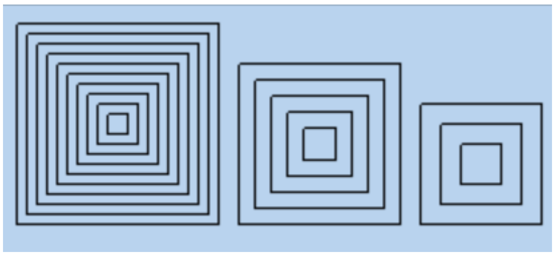
At the end we have a window that looks like this. Make these exercises as explained in the following paragraphs.



4.2.1. DrawSquares function

This function draws some concentric squares. The parameter list contains the position and size of the outermost square and the number of squares. Make use of the DrawRect function in this function.

Then call this **DrawSquares** function 3 times to draw 3 sets of squares next to each other, like this.



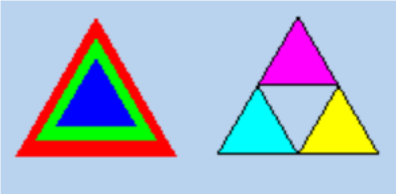
4.2.2. DrawEquilateralTriangle function

Here you can read more about an equilateral triangle Equilateral triangle

This function draws an equilateral triangle with one side being parallel to the x-axes. In utils, there is a FillTriangle and a DrawTriangle function.

The parameter list contains the position of one of the vertices (e.g. the bottom left one), the length of the sides and whether it should be filled or not. The function calculates the other 2 vertices using trigonometry.

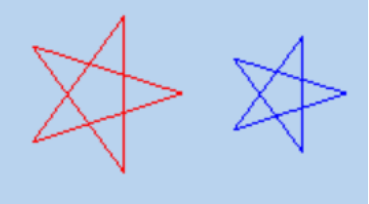
Call this function several times until you get e.g. next result.



4.2.3. DrawPentagram function

Define a function that draws a pentagram.

Decide yourself about the required parameters. Then call this function twice to draw 2 pentagrams.

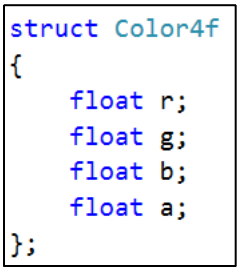


4.2.4. DrawLinearGradient function

Define a function that draws a linear gradient rectangle.

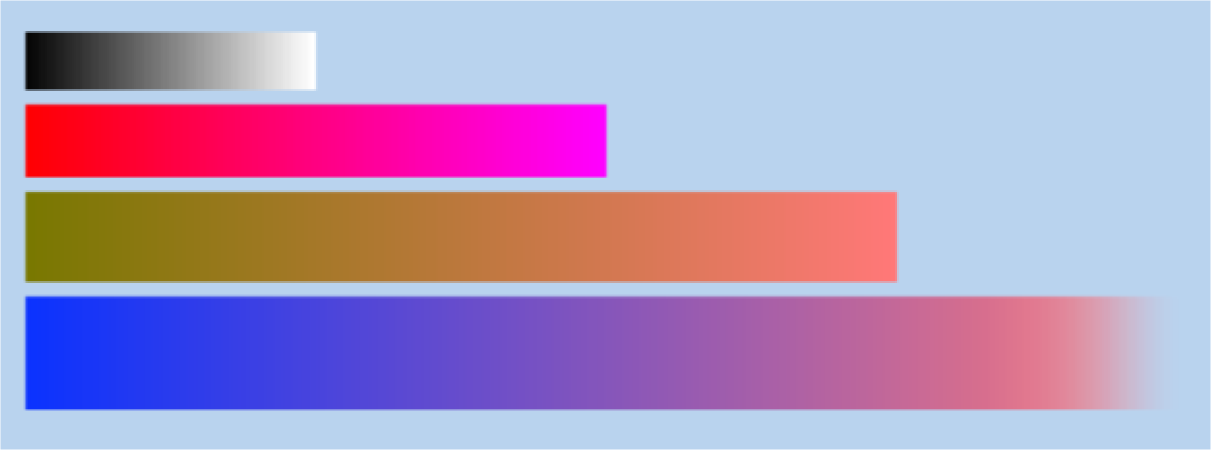
The parameters are: position of the bottom left corner, the width and height (use Rectf), the color at the left side and the color at the right side.

To avoid a long parameter-list, pass the **Color4f** struct that is declared/defined in structs.h/cpp



```
struct Color4f
{
    float r;
    float g;
    float b;
    float a;
};
```

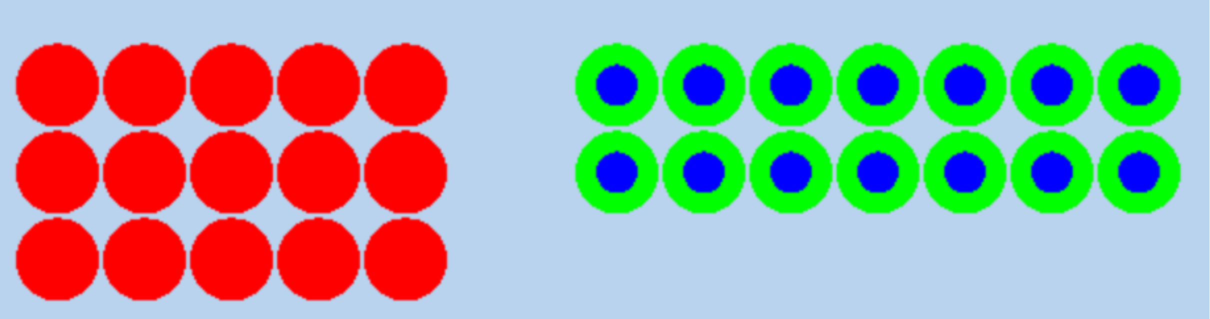
Then test your function by calling it several times with different parameter values.



4.2.5. DrawDotGrid function

Make a function that draws a grid of filled circles. To make this function usable in several cases you should decide wisely about the properties that should be configurable, because this will be your parameter-list. We think that the position, the dots ’ radius, the space between the dots, the number of rows and columns should be configurable.

Then call this function 3 times to get next result.



4.2.6. Centralize often needed functions

In previous exercises you defined some basic drawing functions that you might need again in future applications. Instead of copying them every time it is better to put the declarations and definitions in a header(.h) and implementation (.cpp) file and add those files to a project whenever you need them. If that is the case, use e.g. MyUtils.cpp and .h files.

4.3. FrameTime

Create a new SDL project with name **FrameTime** in your **1DAExx_06_name_firstname** folder.

Delete the generated file **FrameTime.cpp.**

Adapt the window title.

Until now you could count the number of frames, but you had no idea about the **number of frames per second** or about **the elapsed time between 2 consecutive frames.** This resulted in animation speeds that were dependent on the framerate. We do not want this in game!

The code in the core.cpp solves this problem. The **Run** function calls the **Update** function with a parameter that indicates how much time (in seconds) was elapsed since the last call. Notice that the functionality of the **std::chrono** namespace is used to measure the time.

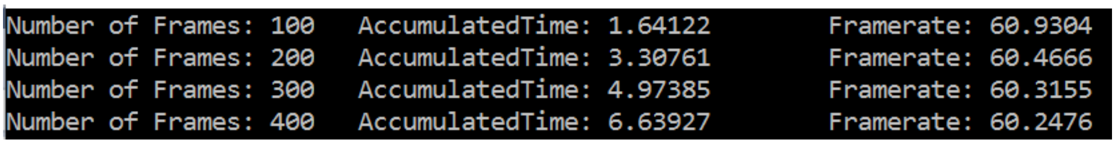
The elapsed seconds knowledge allows us to calculate the new position/velocity of an object in a game:

new velocity = previous velocity + acceleration * elapsed time

new position = previous position + velocity * elapsed time

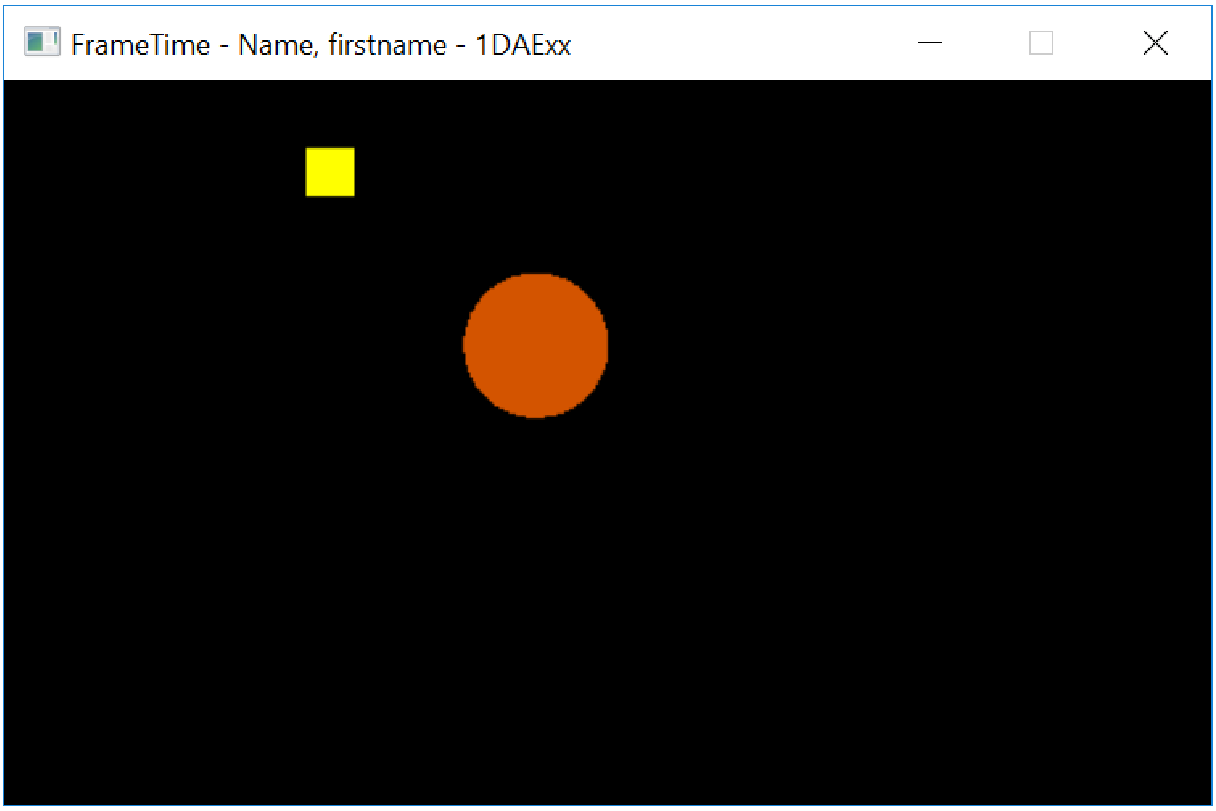
Test the new Update function. Every hundredth frame, show on the console:

- the number of frames,
- the total elapsed time and
- the number of frames per second.



```
Number of Frames: 100   AccumulatedTime: 1.64122       Framerate: 60.9304
Number of Frames: 200   AccumulatedTime: 3.30761       Framerate: 60.4666
Number of Frames: 300   AccumulatedTime: 4.97385       Framerate: 60.3155
Number of Frames: 400   AccumulatedTime: 6.63927       Framerate: 60.2476
```

Let’s apply this new functionality in the following 2 exercises, in the end you should have a window that looks like this.



4.3.1. Bouncing ball

In this exercise you draw a ball that moves at a given velocity and that changes direction when reaching the window boundaries.

The ball has a horizontal speed of 100 pixels per second and vertical speed of 80 pixels per second. Create a function **UpdateBall** that calculates the new position using the elapsed seconds and call it in the Update function.

Create a function **DrawBall** that draws the ball (a filled circle) and call it in the Draw function, remember that you can use the functionality defined in utils. So add these files to this project.

Build, run and watch how the ball leaves the window.

Adapt the UpdateBall function: **after** having calculated the new position, change the sign of the horizontal/vertical speed when the ball reaches the left or right / top or bottom window boundaries.

In core.cpp, there is the global variable g_IsVSyncOn. If true, the framerate is locked to the display frequency, might be 60Hz or 144Hz resulting in 60 or 144 fps. Setting it to false, unlocks the fps and the fps goes to extreme heights, depending on what is executed in the update and draw functions.
Goto that Boolean, and set it to false. The file may be read only. Use the explorer to un-check the read-only checkbox. The visible speed should be the same.
Without the elapsedTime, well…check it out!

4.3.2. Free-falling object

In this exercise you ‘ll calculate the position of a free-falling object supposing that there is no air resistance.

A free-falling object undergoes an acceleration due to the gravity; this acceleration is 9.8m/sec^2. This means that every second its vertical velocity increases with -9.8 meters per second (negative because the gravity acceleration goes from top to bottom).

More info is available here Gravity.

Consider a square object in rest – vertical velocity is zero - at the top of the window.

When the s-key is pressed, let it fall:

Define a function **UpdateFreeFall** that does executes these steps and call it from the Update function:

- Calculates the new velocity: add the gravity acceleration multiplied with the number of elapsed seconds to it.
- Calculates the new position: add the velocity multiplied with the number of elapsed seconds to it.

Define a function **DrawFreeFall** that draws the object and call it in the Draw function. This function:

- Draws the square using its current position, 1 meter being 1 pixel. Tip: Add a FillRectangle function in utils.
- In the test phase: Prints on the console the value of the vertical velocity.

Test and admire your first falling object. Explain why it seems to fall so slow. Think about units.

After a while the object falls outside the window boundaries. When this happens put it again in rest at the top until the s-key is pressed again, add this code to the UpdateFreeFall function, **after** having calculated the new position.

4.4. FreeGame

Continue improving your game of previous week.

When the i-key is pressed then print some information on the console:

- Short description of the game
- Which keys to use
- …

5. Submission instructions

You have to upload the folder *1DAExx_06_name_firstname, however first clean up each project. Perform the steps below for each project in this folder:*

- In Solution Explorer: Select the solution, RMB, choose **Clean Solution**.
- Then **close** the project in Visual Studio.
- Delete the .vs folder.

Compress this *1DAExx_06_name_firstname* folder and upload it before the start of the first lab next week.

6. References

6.1. Functions

6.1.1. Introduction

http://www.learncpp.com/cpp-tutorial/14-a-first-look-at-functions/

http://www.learncpp.com/cpp-tutorial/1-4a-a-first-look-at-function-parameters/

http://www.learncpp.com/cpp-tutorial/1-4b-why-functions-are-useful-and-how-to-use-them-effectively/

http://www.learncpp.com/cpp-tutorial/1-4c-a-first-look-at-local-scope/

6.1.2. Function overloading

http://www.learncpp.com/cpp-tutorial/76-function-overloading/

6.1.3. Default parameters

http://www.learncpp.com/cpp-tutorial/77-default-parameters/

6.1.4. Forward Function declaration

http://www.learncpp.com/cpp-tutorial/17-forward-declarations/

6.2. Equilateral triangle

http://en.wikipedia.org/wiki/Equilateral_triangle

6.3. Gravity

http://www.physicsclassroom.com/class/1DKin/Lesson-5/How-Fast-and-How-Far