Debugging using breakpoints

In this document we learn how to add breakpoints, which will help us interrupt the execution of our program and check the values of the variables at that moment.

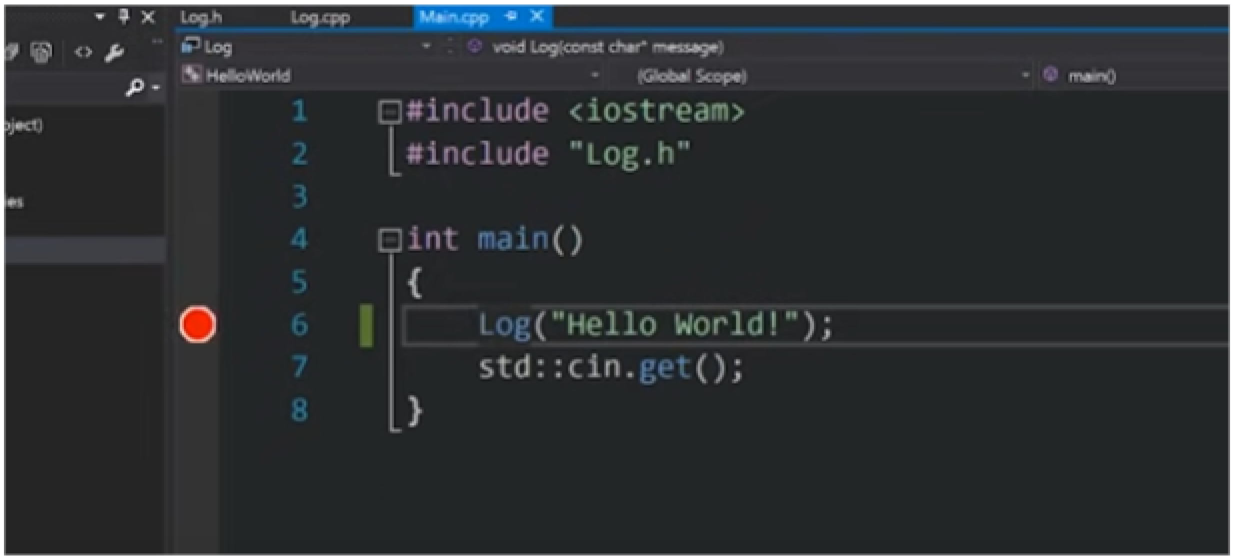
A breakpoint is a point in our program where the debugger will pause execution in the line where we have placed this breakpoint, and this allows us to see the state of our program, or in other words, see which values, in memory, contain the variables and structures of our program.

As we will see in the next examples, knowing the state of the memory is incredibly useful for diagnosing the problem in our code, since we can check at any time what the values of the variables are and diagnose whether this value is the correct one or not.

How to add a breakpoint in VStudio 2022

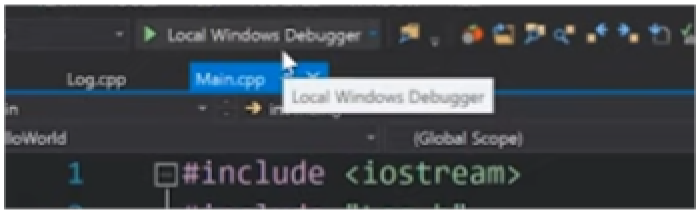
A useful tip is to try to debug in "Debug" mode, since the other modes change a little how the code is interpreted, and it is possible that some breakpoints do not correspond to the selected line or it simply does not stop.

A breakpoint can be added, just by clicking F9, in this way a red dot will appear in the selected line of code, or simply by clicking on the left bar with the left mouse button.



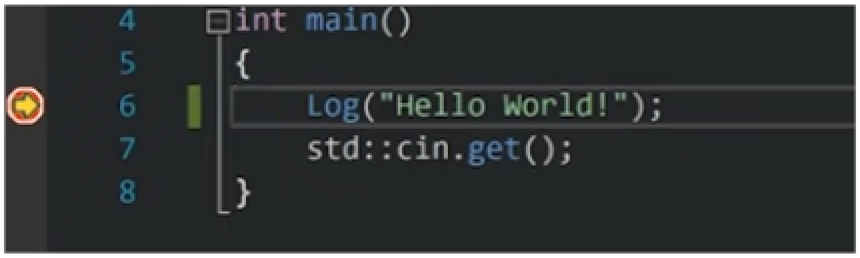
```cpp
#include <iostream>
#include "Log.h"

int main()
{
    Log("Hello World!");
    std::cin.get();
}
```

If we then click on the "Local Windows Debugger", which ensures that we run our program with the debugger attached to our program



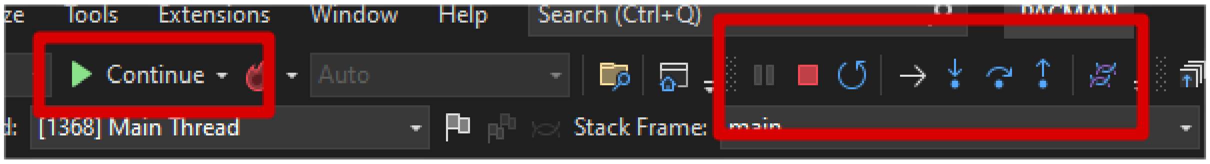
```
#include <iostream>
```

We will see that the execution of our program stops where the red dot is located, where a yellow arrow now appears, which indicates where the execution point of our program is located. The yellow arrow indicates the line that you want to debug, even though this line has not yet been executed.



```
int main()
{
    Log("Hello World!");
    std::cin.get();
}
```

At the top of the menu we will find tools that will make it easier for us to handle the execution, for example:



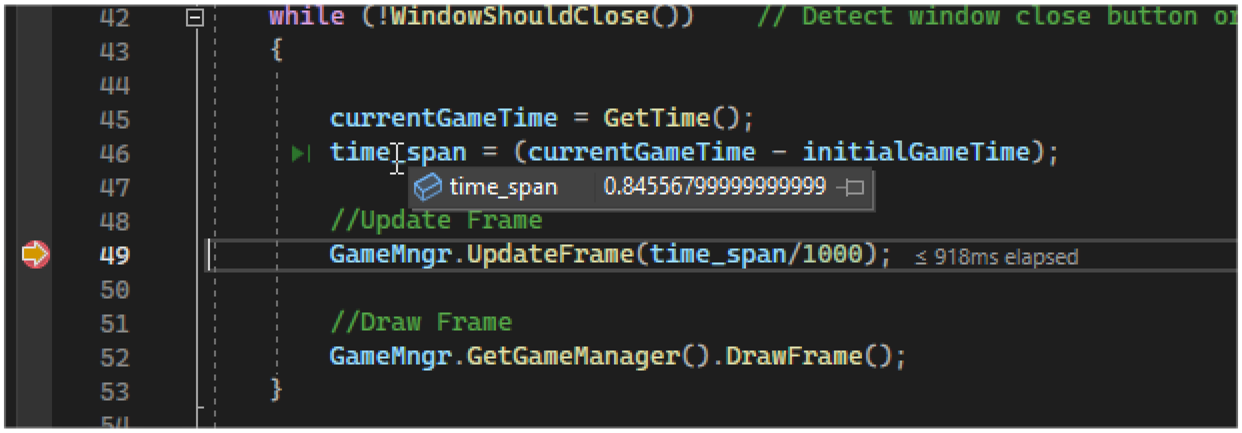
- The continue button will allow us to continue the execution as if nothing had happened.
- Button "Step into" or F11, "Step Over" or F9 and "Step Out" or Shift+F11, these 3 buttons will allow us to control the execution step by step.

  i) "Step into" will allow us to "enter" the function where the execution is at that moment, in our case it will be the Log function.

  ii) "Step over" will allow us to pass over or pass the debug point to the next line of code, in our case std::cin...

  iii) "Step out" will allow us to exit the execution of the function where the debug point is located, and go to the line of code that called this function, in our case, it is main.

When we're debugging we can hover over the variables and examine their contents, for example:

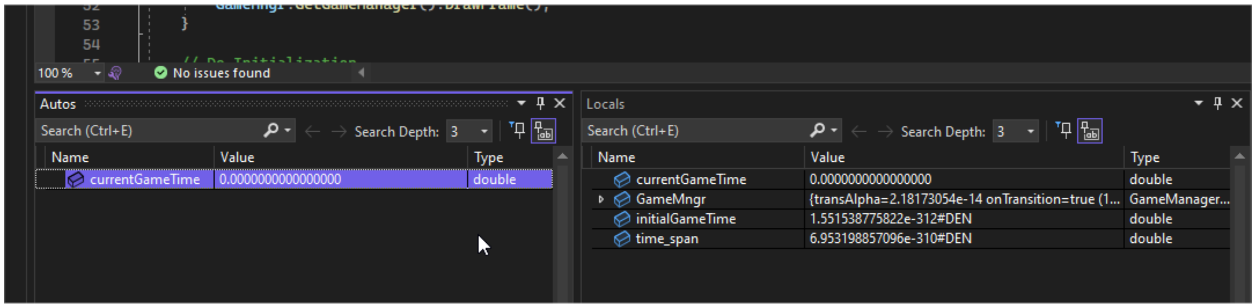


```
while (!WindowShouldClose())    // Detect window close button o
{

    currentGameTime = GetTime();
    time_span = (currentGameTime - initialGameTime);
                time_span    0.84556799999999999
    //Update Frame
    GameMngr.UpdateFrame(time_span/1000);  ≤ 918ms elapsed

    //Draw Frame
    GameMngr.GetGameManager().DrawFrame();
}
```

In this case we can examine the value of time_span, which has just been calculated on this exact line. Here what we are doing is reading the value that this variable has in memory.

We also have three windows that will help us investigate the value of the memory at that time.

- Auto Window and Local Window: local variables of the function you are currently in will be displayed here.



```
    }

100 %    No issues found
```

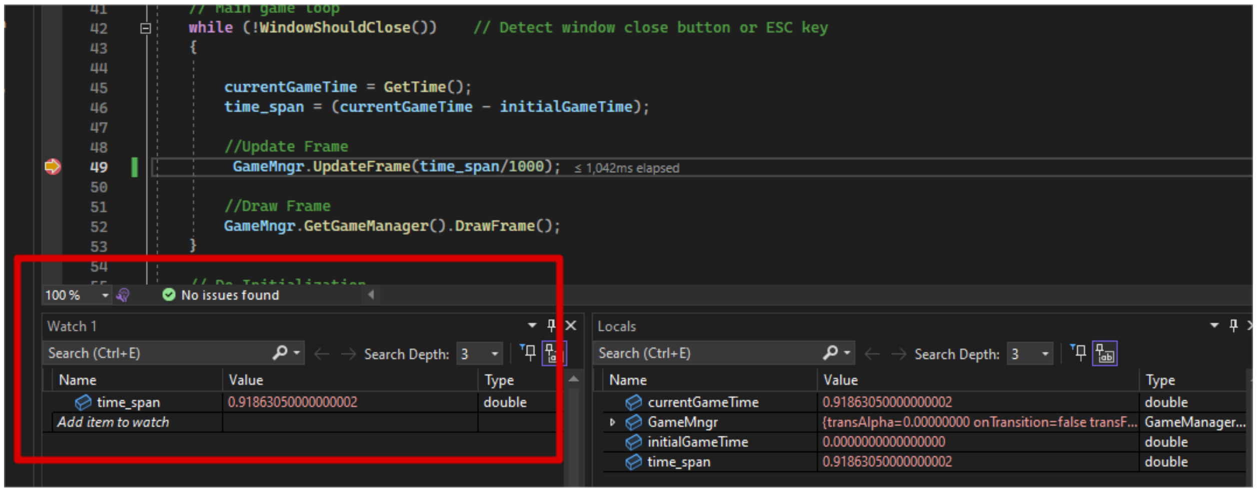
Autos — Search (Ctrl+E) — Search Depth: 3

Name	Value	Type
currentGameTime	0.0000000000000000	double

Locals — Search (Ctrl+E) — Search Depth: 3

Name	Value	Type
currentGameTime	0.0000000000000000	double
GameMngr	{transAlpha=2.18173054e-14 onTransition=true (1...	GameManager...
initialGameTime	1.551538775822e-312#DEN	double
time_span	6.953198857096e-310#DEN	double

- Watch window: this window allows us to monitor the variables that we consider. In the example we are interested in knowing the value of time_span at any time, to achieve this we just have to write the name of the variable in the "Name" column.



```
while (!WindowShouldClose())    // Detect window close button or ESC key
{

    currentGameTime = GetTime();
    time_span = (currentGameTime - initialGameTime);

    //Update Frame
     GameMngr.UpdateFrame(time_span/1000);  ≤ 1,042ms elapsed

    //Draw Frame
    GameMngr.GetGameManager().DrawFrame();
}

100 %    No issues found
```

Watch 1 — Search (Ctrl+E) — Search Depth: 3

Name	Value	Type
time_span	0.91863050000000002	double
Add item to watch		

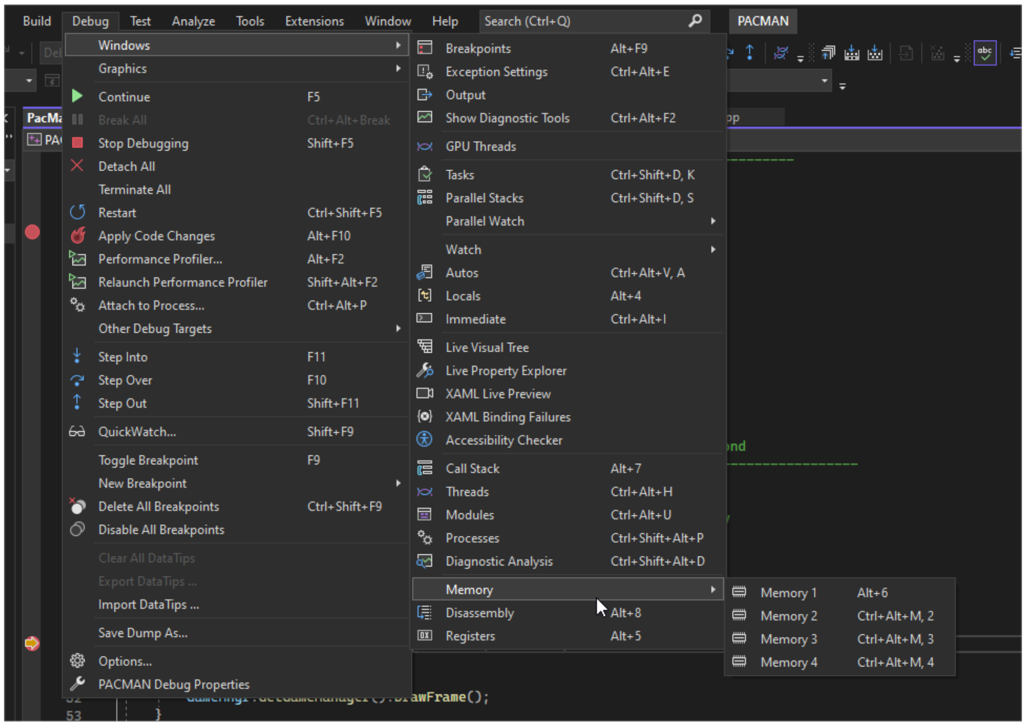
Locals — Search (Ctrl+E) — Search Depth: 3

Name	Value	Type
currentGameTime	0.91863050000000002	double
GameMngr	{transAlpha=0.00000000 onTransition=false transF...	GameManager...
initialGameTime	0.0000000000000000	double
time_span	0.91863050000000002	double

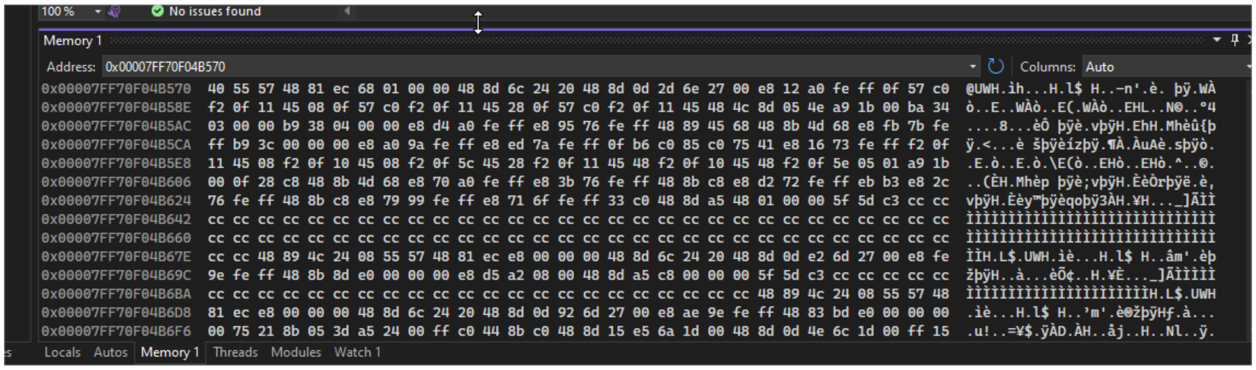
-

Visualization of the memory status

In VS we have a window, very similar to the ones explained above, which allows us to see the state of all the memory. To open this window we will have to go to Debug->Windows->Memory (**Important: This option only appears when we are running our code**)



A window like this will appear showing us all the memory of our program.

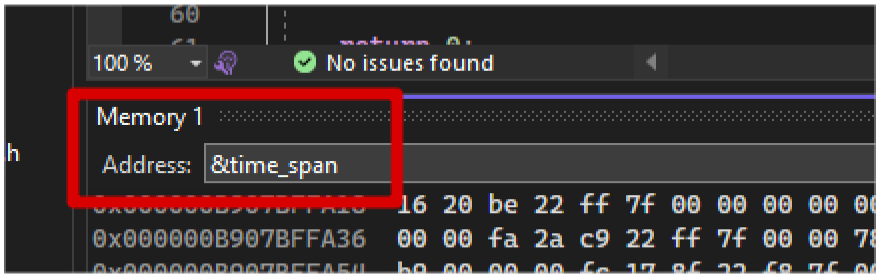


Address: 0x00007FF70F04B570

```
0x00007FF70F04B570  40 55 57 48 81 ec 68 01 00 00 48 8d 6c 24 20 48 8d 0d 2d 6e 27 00 e8 12 a0 fe ff 0f 57 c0  @UWH.ìh...H.l$ H..-n'.è. þÿ.WÀ
0x00007FF70F04B58E  f2 0f 11 45 08 0f 57 c0 f2 0f 11 45 28 0f 57 c0 f2 0f 11 45 48 4c 8d 05 4e a9 1b 00 ba 34  ò..E..WÀò..E(.WÀò..EHL..N®..º4
0x00007FF70F04B5AC  03 00 00 b9 38 04 00 00 e8 d4 a0 fe ff e8 95 76 fe ff 48 89 45 68 48 8b 4d 68 e8 fb 7b fe  ....8...èÔ þÿè.vþÿH.EhH.Mhèû{þ
0x00007FF70F04B5CA  ff b9 3c 00 00 00 e8 a0 9a fe ff e8 ed 7a fe ff 0f b6 c0 85 c0 75 41 e8 16 73 fe ff f2 0f  ÿ.<...è šþÿèízþÿ.¶À.ÀuAè.sþÿò.
0x00007FF70F04B5E8  11 45 08 f2 0f 10 45 08 f2 0f 5c 45 28 f2 0f 11 45 48 f2 0f 10 45 48 f2 0f 5e 05 01 a9 1b  .E.ò..E.ò.\E(ò..EHò..EHò.^..©.
0x00007FF70F04B606  00 0f 28 c8 48 8b 4d 68 e8 70 a0 fe ff e8 3b 76 fe ff 48 8b c8 e8 d2 72 fe ff eb b3 e8 2c  ..(ÈH.Mhèp þÿè;vþÿH.ÈèÒrþÿë.è,
0x00007FF70F04B624  76 fe ff 48 8b c8 e8 79 99 fe ff e8 71 6f fe ff 33 c0 48 8d a5 48 01 00 00 5f 5d c3 cc cc  vþÿH.Èèy™þÿèqoþÿ3ÀH.¥H..._]ÃÌÌ
0x00007FF70F04B642  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc  ÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌ
0x00007FF70F04B660  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc  ÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌ
0x00007FF70F04B67E  cc cc 48 89 4c 24 08 55 57 48 81 ec e8 00 00 00 48 8d 6c 24 20 48 8d 0d e2 6d 27 00 e8 fe  ÌÌH.L$.UWH.ìè...H.l$ H..âm'.èþ
0x00007FF70F04B69C  9e fe ff 48 8b 8d e0 00 00 00 e8 d5 a2 08 00 48 8d a5 c8 00 00 00 5f 5d c3 cc cc cc cc cc  žþÿH..à...èÕ¢..H.¥È..._]ÃÌÌÌÌÌ
0x00007FF70F04B6BA  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc 48 89 4c 24 08 55 57 48  ÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌH.L$.UWH
0x00007FF70F04B6D8  81 ec e8 00 00 00 48 8d 6c 24 20 48 8d 0d 92 6d 27 00 e8 ae 9e fe ff 48 83 bd e0 00 00 00  .ìè...H.l$ H..’m'.è®žþÿHƒ.à...
0x00007FF70F04B6F6  00 75 21 8b 05 3d a5 24 00 ff c0 44 8b c0 48 8d 15 e5 6a 1d 00 48 8d 0d 4e 6c 1d 00 ff 15  .u!..=¥$.ÿÀD.ÀH..åj..H..Nl..ÿ.
```

On the left we can see the memory addresses, in the center we can see the values saved, represented in hexadecimal, in these addresses, and on the right we can see the ASCII interpretation of these values.

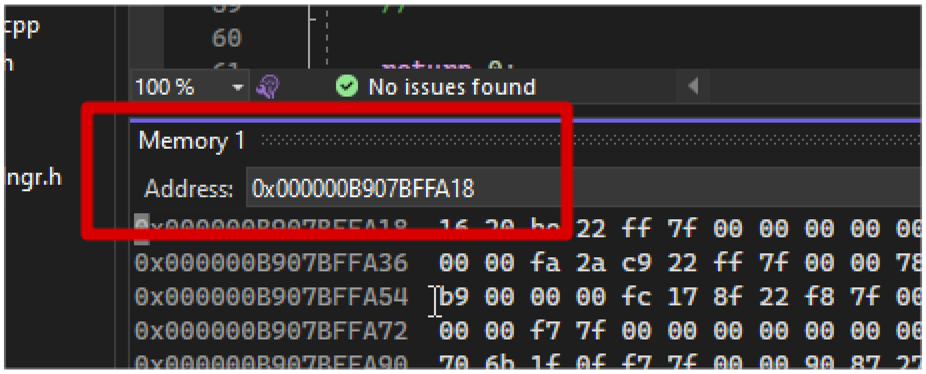
If we want to find the memory position occupied by a specific variable in our code, we will need to know its memory address, and to achieve this basically what we have to do is at the top write the name of our variable with a " &" in front.



Address: &time_span

```
                    16 20 be 22 ff 7f 00 00 00 00 0
0x000000B907BFFA36  00 00 fa 2a c9 22 ff 7f 00 00 78
0x000000B907BFFA54  b9 00 00 00 fc 17 8f 22 f8 7f 0
```

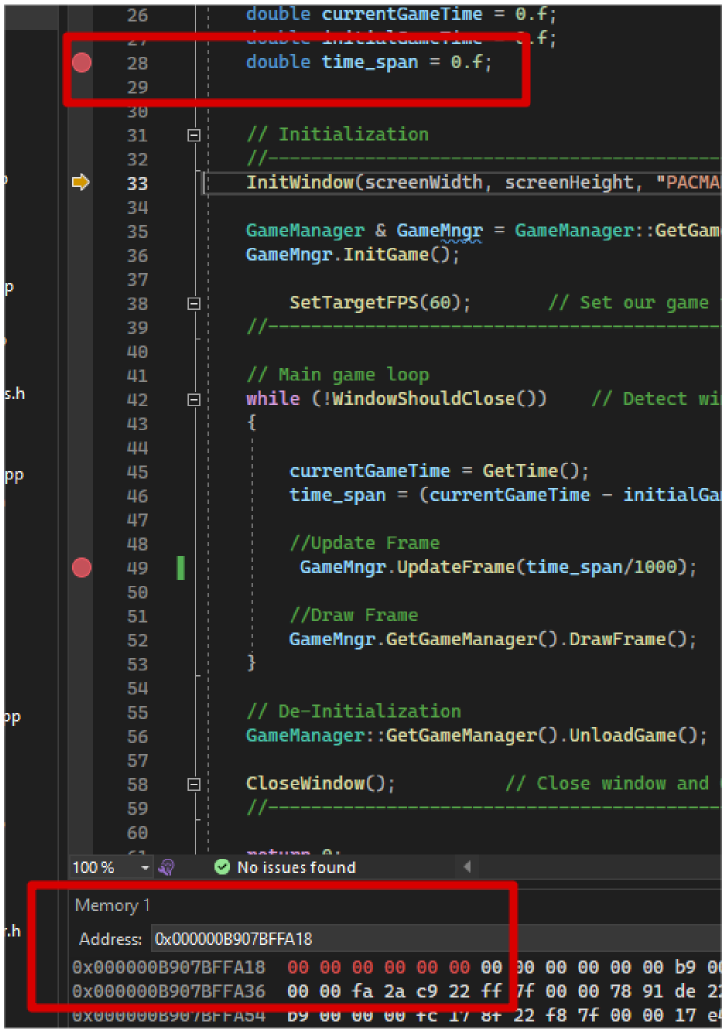
If we click enter, we will see how the value of the memory address corresponding to this variable appears



Address: 0x000000B907BFFA18

```
0x000000B907BFFA18  16 20 be 22 ff 7f 00 00 00 00 00
0x000000B907BFFA36  00 00 fa 2a c9 22 ff 7f 00 00 78
0x000000B907BFFA54  b9 00 00 00 fc 17 8f 22 f8 7f 00
0x000000B907BFFA72  00 00 f7 7f 00 00 00 00 00 00 00
0x000000B907BFFA90  70 6b 1f 0f f7 7f 00 00 90 87 27
```

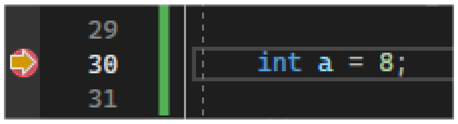
And as we can see, when we put a breakpoint in the initialization of this variable we can see how the memory content is updated:



```
    double currentGameTime = 0.f;
    double initialGameTime = 0.f;
    double time_span = 0.f;


    // Initialization
    //--------------------------------------------
    InitWindow(screenWidth, screenHeight, "PACMA

    GameManager & GameMngr = GameManager::GetGam
    GameMngr.InitGame();

        SetTargetFPS(60);       // Set our game
    //--------------------------------------------

    // Main game loop
    while (!WindowShouldClose())    // Detect wi
    {

        currentGameTime = GetTime();
        time_span = (currentGameTime - initialGa

        //Update Frame
         GameMngr.UpdateFrame(time_span/1000);

        //Draw Frame
        GameMngr.GetGameManager().DrawFrame();
    }

    // De-Initialization
    GameManager::GetGameManager().UnloadGame();

    CloseWindow();          // Close window and
    //--------------------------------------------

    return 0;
```

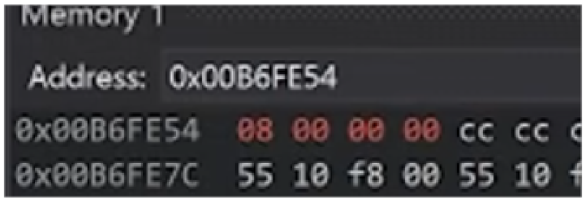
```
Address: 0x000000B907BFFA18
0x000000B907BFFA18  00 00 00 00 00 00 00 00 00 00 00 00 b9 00
0x000000B907BFFA36  00 00 fa 2a c9 22 ff 7f 00 00 78 91 de 22
0x000000B907BFFA54  b9 00 00 00 fc 17 8f 22 f8 7f 00 00 17 e4
```

Example: We will add a new variable (int) initialized with the value 8.



```
    int a = 8;
```

We see that since the variable "a" is an int, 4 bytes have been modified and the value 8 saved. (Remember that 2 digits represent a byte)



```
Address: 0x00B6FE54
0x00B6FE54  08 00 00 00 cc cc c
0x00B6FE7C  55 10 f8 00 55 10 f
```