



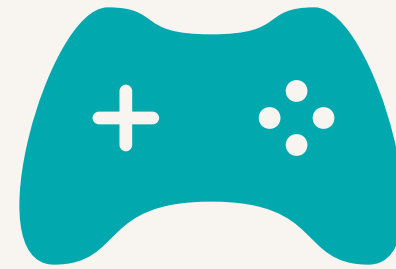
# 30 TIPS AND TRICKS IN 30 MINUTES

OTTO KIVLING, LEAD PROGRAMMER, REDHILL GAMES

# Who am I



Currently Lead Programmer at Redhill Games



Worked at Starbreeze Studios, Guerrilla Games, Remedy Entertainment

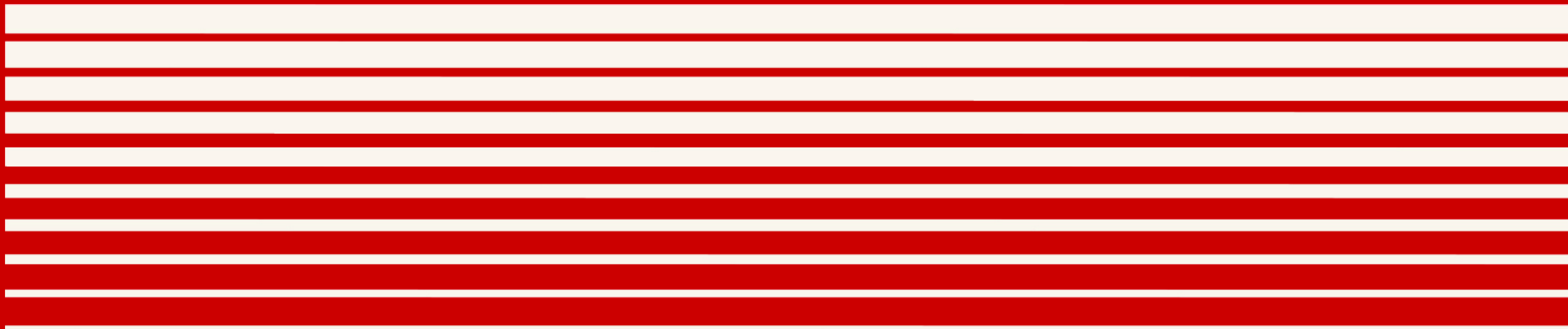


Also have a background in low level and embedded programming

# Agenda

- ▶ Debugging
- ▶ Performance and memory
- ▶ Compiling
- ▶ Generic

# DEBUGGING TRICKS



# Static variables when debugging

```
→ GetRunningState  bool GetRunningState()
1  #include <iostream>
2
3  bool GetRunningState()
4  {
5      static bool debugVar = true;
6      return debugVar;
7  }
8
9  int main()
10 {
11     while (GetRunningState())
12     {
13         std::cout << "Hello world" << std::endl;
14     }
15 }
16
```

- ▶ Use for debugging or testing
  - ▶ Shortcut through logic
- ▶ Examples
  - ▶ Simulate message received
  - ▶ Go straight to high score
  - ▶ Button/key presses

# Recognize magic debug values

- ▶ Debugging/crashes
  - ▶ You'll recognize if it's heap or stack
    - ▶ 0xCC Stack (Windows)
    - ▶ 0xCD Heap (Windows)
- ▶ Different patterns on different platforms
- ▶ More extensive list
  - ▶ [https://en.wikipedia.org/wiki/Magic\\_number\\_\(programming\)#Debug\\_values](https://en.wikipedia.org/wiki/Magic_number_(programming)#Debug_values)

Value	Description
CCCCCCCC	Microsoft's C++ debugging runtime library and many DOS environments to mark uninitialized stack memory.
CDCDCDCD	Microsoft's C/C++ debug malloc() function to mark uninitialized heap memory, usually returned from HeapAlloc()
FDFDFDFD	Microsoft's C/C++ debug malloc() function to mark "no man's land" guard bytes before and after allocated heap
ABABABAB	Microsoft's debug HeapAlloc() to mark "no man's land" guard bytes after allocated heap memory.
BAADF00D	Microsoft's debug HeapAlloc() to mark uninitialized allocated heap memory
DDDDDDDD	Microsoft's C/C++ debug free() function to mark freed heap memory
FEEEFEEE	Microsoft's debug HeapFree() to mark freed heap memory.

# Uninitialized variables release compiled

- ▶ Global scope gets initialized
  - ▶ `globalVariable == 0`
- ▶ Local (stack) doesn't get initialized
  - ▶ Becomes random when release compiled

```
int globalVariable;
int main()
{
    int variable;
    std::cout << "Hello globalVariable: " << globalVariable << "\n";
    std::cout << "Hello variable: " << variable << "\n";

    return 0;
}
```

C:\Users\otto\Documents\Visual Studio 2019\ConsoleApplication1\ConsoleApplication1\Release\ConsoleApplication1.exe

```
Hello globalVariable: 0
Hello variable: 2004933264
```

# Uninitialized variables debug compiled

- ▶ Remember the memory guardians?
- ▶ `variable == -858993460`
  - ▶ Which is `0xCCCC CCCC`
- ▶ You are looking at uninitialized stack memory

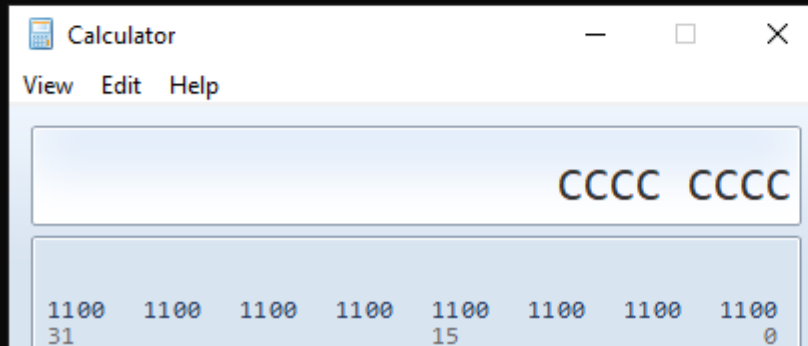
```
int globalVariable;
int main()
{
    int variable;
    std::cout << "Hello globalVariable: " << globalVariable << "\n";
    std::cout << "Hello variable: " << variable << "\n";

    return 0;
}
```

Microsoft Visual Studio Debug Console

```
Hello globalVariable: 0
Hello variable: -858993460

C:\Users\otto\Documents\Visual Studio 2019\ConsoleApplication1\ConsoleApplication1\Debug\
s 29548) exited with code 0.
Press any key to close this window . . .
```



The image shows a screenshot of a Windows Calculator window. The title bar reads "Calculator" and the menu bar includes "View", "Edit", and "Help". The main display area shows the hexadecimal value "CCCC CCCC". Below the display, there is a grid of numbers: 1100, 1100, 1100, 1100, 1100, 1100, 1100, 1100 in the top row, and 31, 15, 0 in the bottom row.



# Use the memory viewer in VStudio

```
22
23 int main()
24 {
25     const char stringVariable[] = "A string!";
26     std::cout << stringVariable;
27 }
```

132 % No issues found

Memory 1

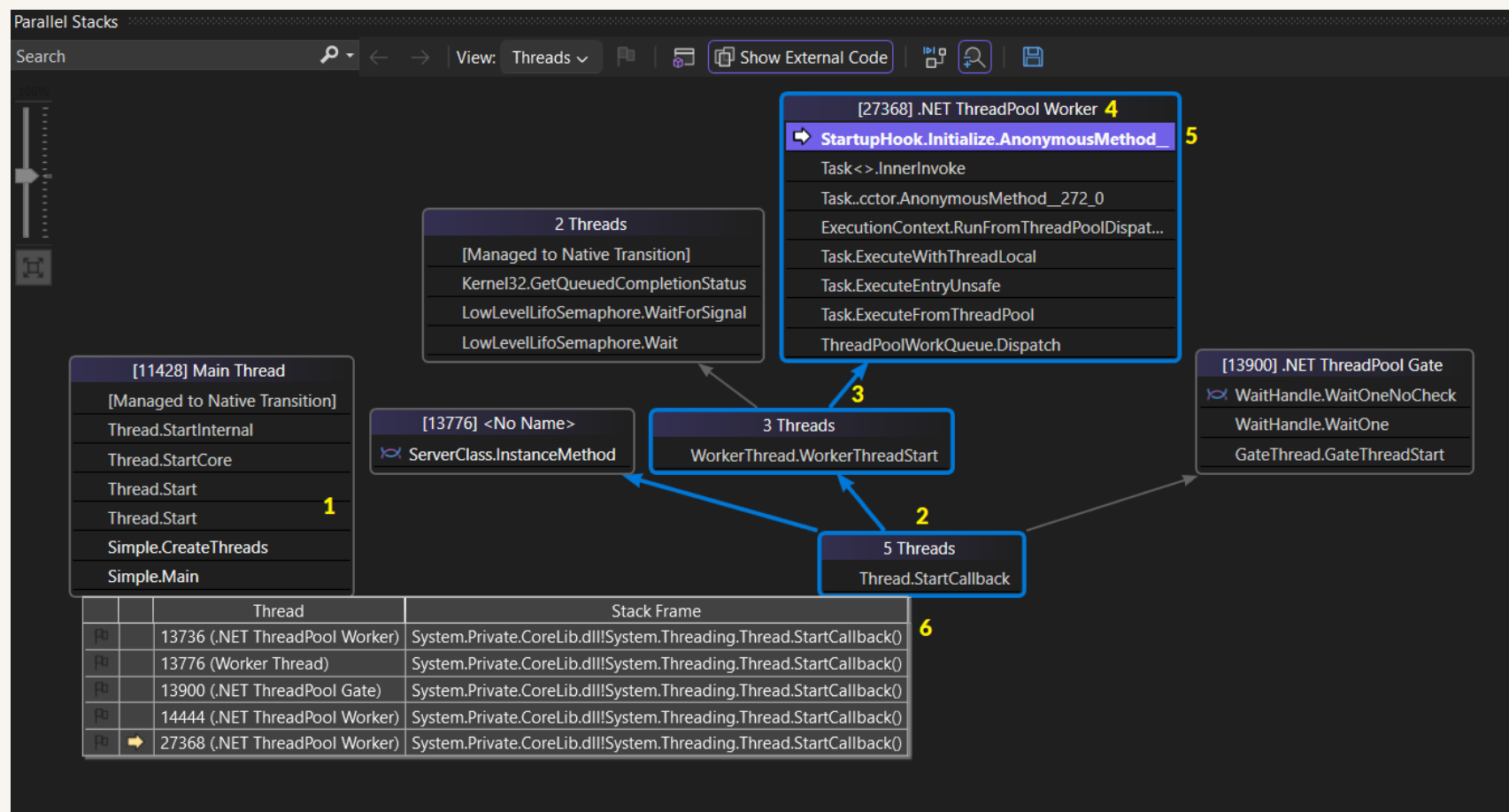
Address: 0x003CFA88 Columns: Auto

0x003CFA88	41 20 73 74 72 69 6e 67 21 00 cc cc cc cc cc cc 85 e7 33 97 bc fa 3c	A string!.iiiiii.ç3-.ú<
0x003CFA9F	00 83 37 01 01 01 00 00 00 88 1a 76 00 20 23 76 00 01 00 00 00 88 1a	.f7.....^v. #v.....^.
0x003CFAB6	76 00 20 23 76 00 18 fb 3c 00 d7 35 01 01 01 e6 33 97 90 da 80 77 e8	v. #v..û<.x5...æ3-.Ú€wè
0x003CFACD	26 81 77 a5 01 01 00 00 00 00 00 00 00 00 90 da 80 77 00 00 00 00	&.w¥.....Ú€w....
0x003CFAE4	a5 01 01 00 00 00 42 a0 04 fb 3c 00 57 29 ef 75 7c c5 01 01 88 c5 01	¥.....B .û<.W)iu Á..^Á.
0x003CFAFB	01 00 00 00 00 c4 fa 3c 00 38 fb 3c 00 84 fb 3c 00 30 59 01 01 69 af	.....Äú<.8ú<..ú<.0Y..i~
0x003CFB12	0e 96 00 00 00 00 20 fb 3c 00 6d 34 01 01 28 fb 3c 00 08 38 01 01 38	.-.... û<.m4..(û<..8..8
0x003CFB29	fb 3c 00 c9 00 5f 76 00 60 5c 00 b0 00 5f 76 94 fb 3c 00 1e 7b 76 77	û<.É._v.`\..°._v"û<..{vw
0x003CFB40	00 60 5c 00 91 86 09 55 00 00 00 00 00 00 00 00 60 5c 00 00 00 00	..\.f..U.....\....
0x003CFB57	00 00	.....

- ▶ Can give hints as to what's causing a memory stomp, overflow, etc
- ▶ Example below is read from heap(0xCD) to stack (0xCC)

```
0:000> dd esp 00000084`67b9fd30
00000084`67b96cc0 67b96d00 00000084 89194aa8 00007ff7
00000084`67b96cd0 cdcddcdcd cdcddcdcd ccccccccc ccccccccc
00000084`67b96ce0 ccccccccc ccccccccc ccccccccc ccccccccc
00000084`67b96cf0 ccccccccc ccccccccc ccccccccc ccccccccc
00000084`67b96d00 67b96d40 00000084 89194978 00007ff7
00000084`67b96d10 cdcddcdcd cdcddcdcd ccccccccc ccccccccc
00000084`67b96d20 ccccccccc ccccccccc ccccccccc ccccccccc
00000084`67b96d30 ccccccccc ccccccccc ccccccccc ccccccccc
00000084`67b96d40 67b96e10 00000084 891a942f 00007ff7
00000084`67b96d50 cdcddcdcd cdcddcdcd 67b96dd8 00000084
```

# Parallel stacks in Visual Studio

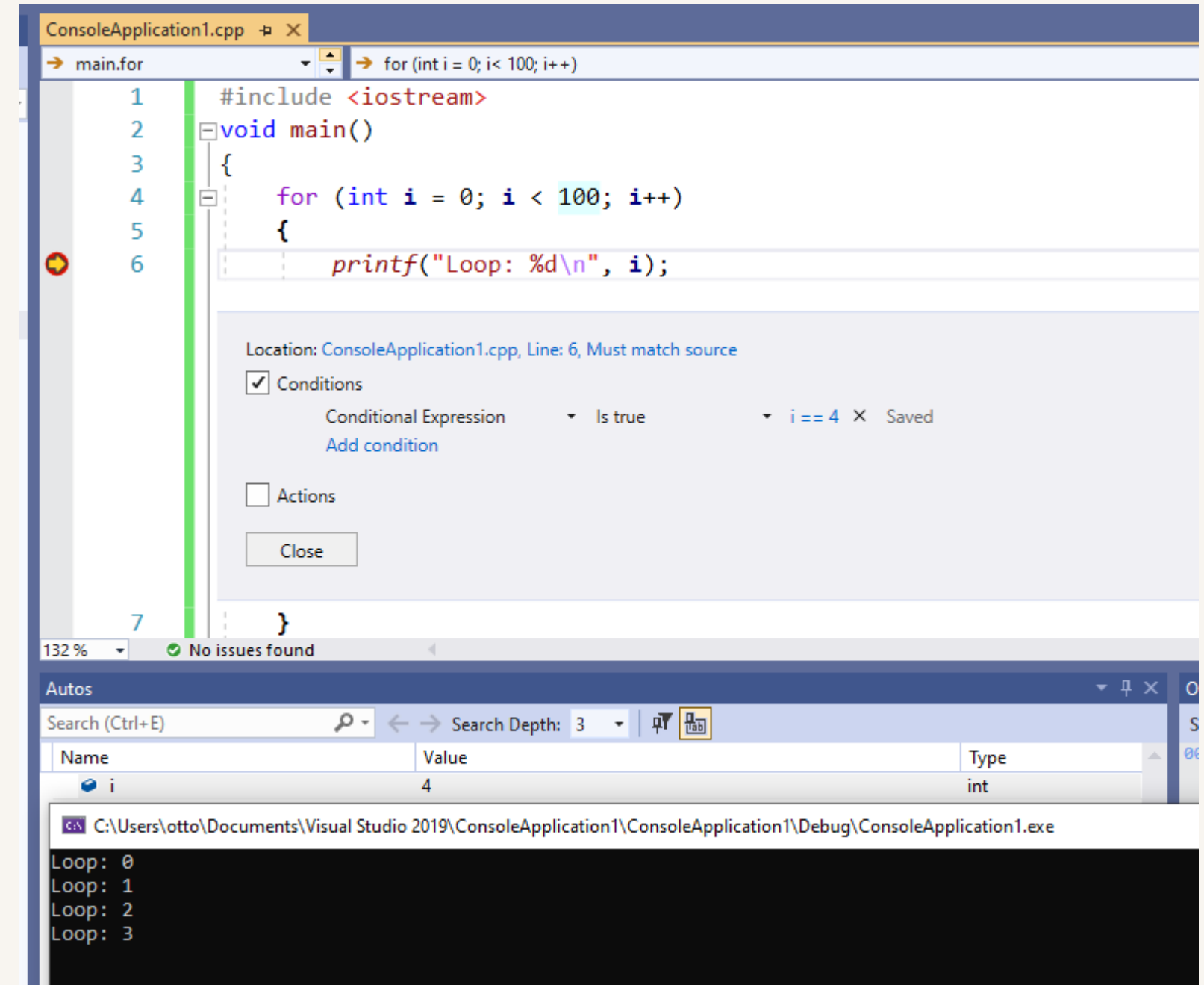


- ▶ See what other threads are doing during a breakpoint
  - ▶ Can be useful to see what else is getting processed
- ▶ Can be a timesaver when debugging deadlocks
  - ▶ Could very well see cause of deadlock

# Conditional breakpoints

- ▶ Stop when a condition is met
  - ▶ A file is open
  - ▶ A socket is open
  - ▶ Iteration reached a certain value
- ▶ Poor-man conditional breakpoint
  - ▶ Use if-statement and `DebugBreak()`

```
#include "debugapi.h"  
...  
if(i == 5) DebugBreak();
```



# Data Breakpoints

- ▶ Good for finding e.g memory stomps
- ▶ Must first hit any breakpoint before it can be set
- ▶ Will break when X bytes at memory location changes
  - ▶ Can use offset if watching a struct
- ▶ Memory address change between sessions

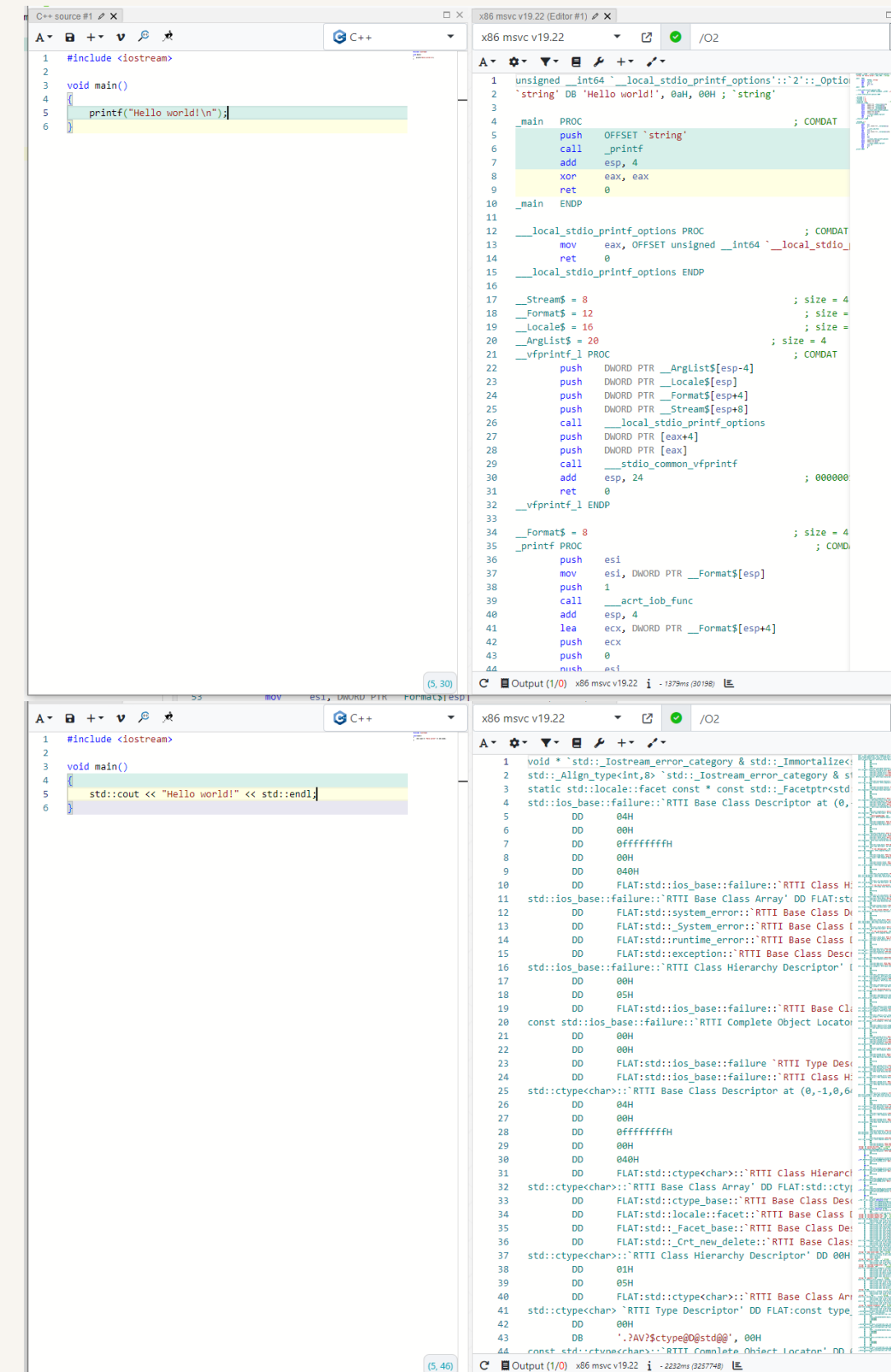
The screenshot displays the Visual Studio IDE with a C++ project named 'ConsoleApplication1.cpp'. The code is as follows:

```
1 #include <iostream>
2
3 struct dataStruct
4 {
5     char counter, x, y, z, direction;
6     dataStruct() : x(0), y(0), z(0), direction(0), counter(0) {}
7 };
8
9 void main()
10 {
11     dataStruct aStruct;
12     for (int i = 1; i < 100; i++)
13     {
14         aStruct.counter = i;
15         // Here we do a memory stomp
16         *(&aStruct.counter + 1) = 0xff;
17     }
18 }
19
```

A 'Data Breakpoint Hit' dialog box is open, stating: 'The following breakpoint was hit: When 0x012FFBFC changes (4 bytes) in process'. The breakpoint is set on line 16. The 'Parallel Stacks' window shows the main thread and several worker threads. The 'Watch' window shows '&aStruct' as the current watch item. The 'Output' window shows the application's output.

# Check snippet in Compiler Explorer

- ▶ Put your snippet in, select compiler, check output
  - ▶ <https://godbolt.org/>
- ▶ You can also use disassembly view in VStudio
- ▶ Compiler errors on cross-platform code e.g Linux vs Windows
  - ▶ ...or verify that it is a compiler bugs
- ▶ Visualize bloat in functions
  - ▶ printf vs std::cout on length



# #pragma optimize("", off)

## Parameters of the optimize Pragma

Parameter(s)	Type of optimization
<code>g</code>	Enable global optimizations. Deprecated. For more information, see <a href="#">/Og (Global optimizations)</a> .
<code>s</code> or <code>t</code>	Specify short or fast sequences of machine code.
<code>y</code>	Generate frame pointers on the program stack.

These parameters are the same letters used with the `/O` compiler options. For example, the following pragma is equivalent to the `/Os` compiler option:

```
C++ Copy
#pragma optimize( "s", on )
```

Using the `optimize` pragma with the empty string (`""`) is a special form of the directive:

When you use the `off` parameter, it turns all the optimizations, `g`, `s`, `t`, and `y`, off.

When you use the `on` parameter, it resets the optimizations to the ones that you specified using the `/O` compiler option.

```
C++ Copy
#pragma optimize( "", off )
/* unoptimized code section */
#pragma optimize( "", on )
```

- ▶ Makes release/optimized code easier to debug
- ▶ #pragma are compiler specific
- ▶ Don't forget it in shipping code
  - ▶ Grep before shipping!
  - ▶ Could be checked for in your build system

# #pragma inline\_depth(0)

- ▶ Used when debugging optimized code
  - ▶ inlined code can be hard to debug
- ▶ Value of 0 inhibits inlining

```
C++ x86 msvc v19.22 /O2
1 #include <iostream>
2
3 // #pragma inline_depth(0)
4
5 inline void printSum(int num1,int num2)
6 {
7     printf("%d\n", num1 + num2);
8 }
9
10 void main()
11 {
12     printSum(10, 20);
13     printSum(2, 5);
14     printSum(100, 400);
15 }
```

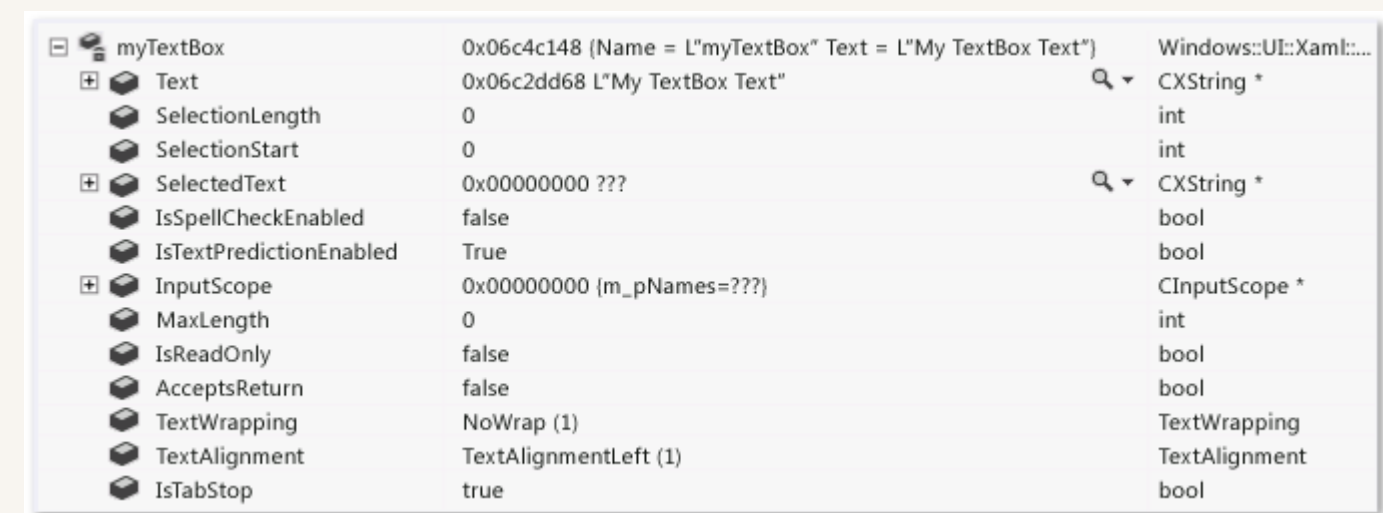
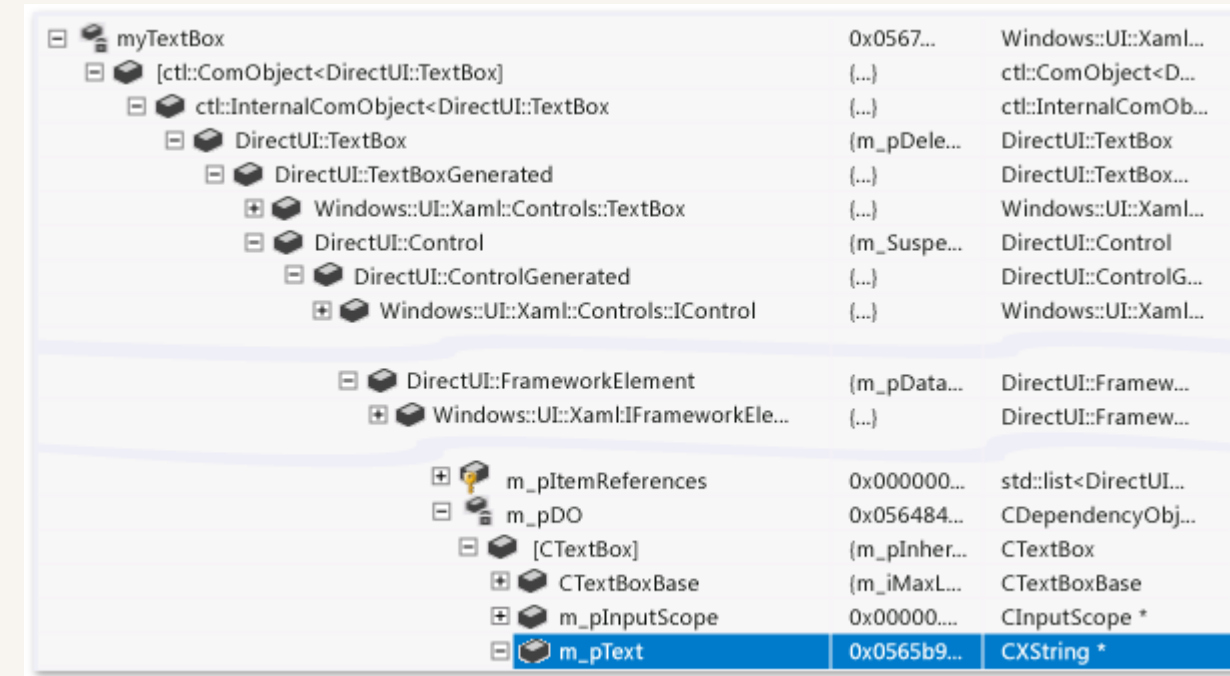
```
x86 msvc v19.22 /O2
1 unsigned __int64 __local_stdio_printf_options::`2'::_OptionsStorage DQ 01H
2 `string' DB '%d', 0aH, 00H ; `string'
3
4 _main PROC ; COMDAT
5     push 30 ; 0000001eH
6     push OFFSET `string'
7     call _printf
8     push 7
9     push OFFSET `string'
10    call _printf
11    push 500 ; 000001f4H
12    push OFFSET `string'
13    call _printf
14    add esp, 24 ; 00000018H
15    xor eax, eax
16    ret 0
```

```
C++ x86 msvc v19.22 /O2
1 #include <iostream>
2
3 #pragma inline_depth(0)
4
5 inline void printSum(int num1,int num2)
6 {
7     printf("%d\n", num1 + num2);
8 }
9
10 void main()
11 {
12     printSum(10, 20);
13     printSum(2, 5);
14     printSum(100, 400);
15 }
```

```
x86 msvc v19.22 /O2
6 void printSum(int,int) PROC ; printSum, COMDAT
7     mov eax, DWORD PTR _num1$[esp-4]
8     add DWORD PTR _num2$[esp-4], eax
9     mov DWORD PTR _num1$[esp-4], OFFSET `string'
10    jmp _printf
11 void printSum(int,int) ENDP ; printSum
12
13 _main PROC ; COMDAT
14     push 20 ; 00000014H
15     push 10 ; 0000000aH
16     call void printSum(int,int) ; printSum
17     push 5
18     push 2
19     call void printSum(int,int) ; printSum
20     push 400 ; 00000190H
21     push 100 ; 00000064H
22     call void printSum(int,int) ; printSum
23     add esp, 24 ; 00000018H
24     xor eax, eax
25     ret 0
26 _main ENDP
```

# Use natvis in Visual Studio

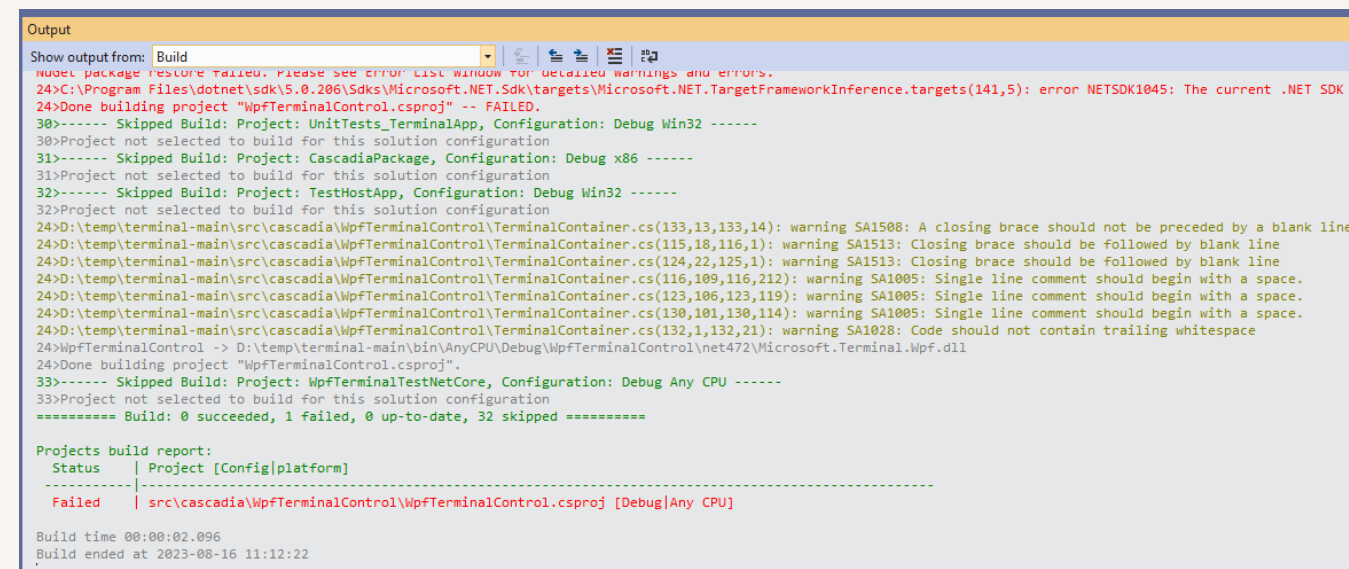
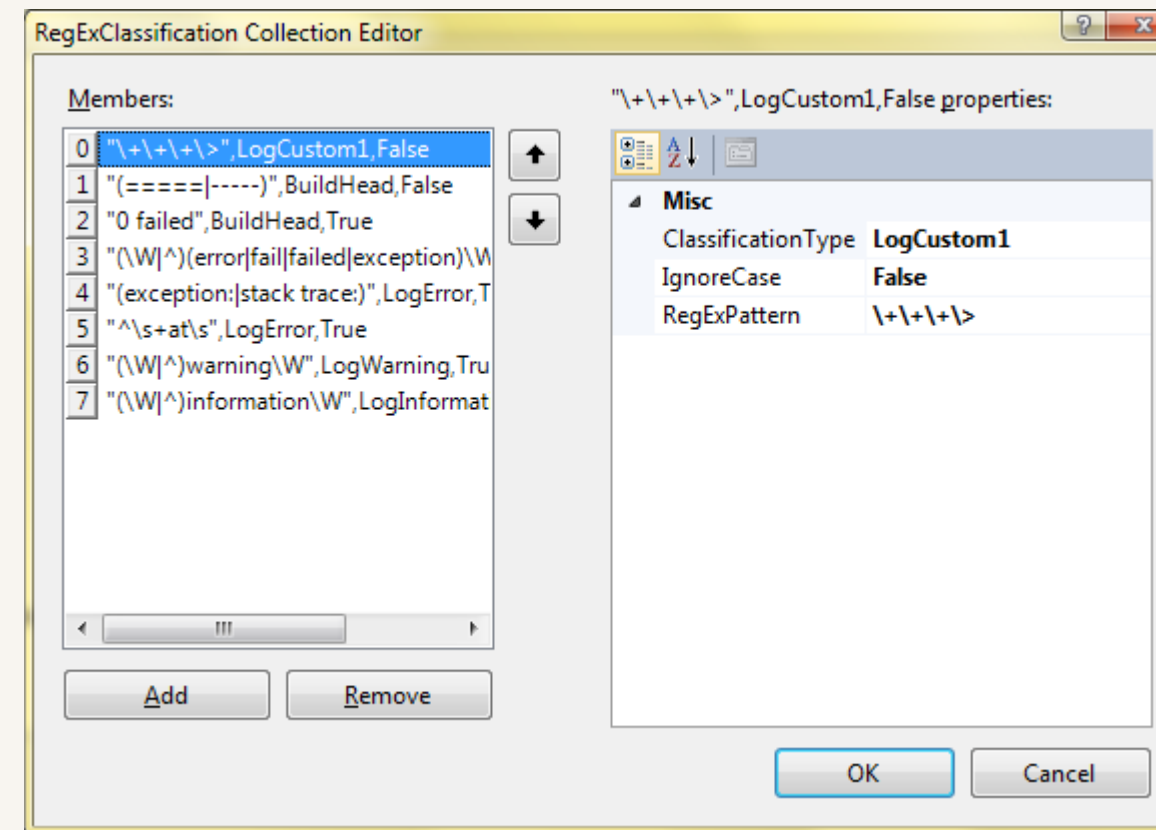
- ▶ NATive type VISualization
- ▶ Helps with debugging
- ▶ Premade exists for major engines
- ▶ Make your own for proprietary engine





# VSColorOutput for Visual Studio

- ▶ Makes sifting through text much easier
- ▶ Create your own regexp for highlight
- ▶ Compiler output
- ▶ Log output



# Use Internal speakers

## Syntax

```
C++ Copy  
  
BOOL Beep(  
    [in] DWORD dwFreq,  
    [in] DWORD dwDuration  
);
```

## Parameters

[in] dwFreq

The frequency of the sound, in hertz. This parameter must be in the range 37 through 32,767 (0x25 through 0x7FFF).

[in] dwDuration

The duration of the sound, in milliseconds.

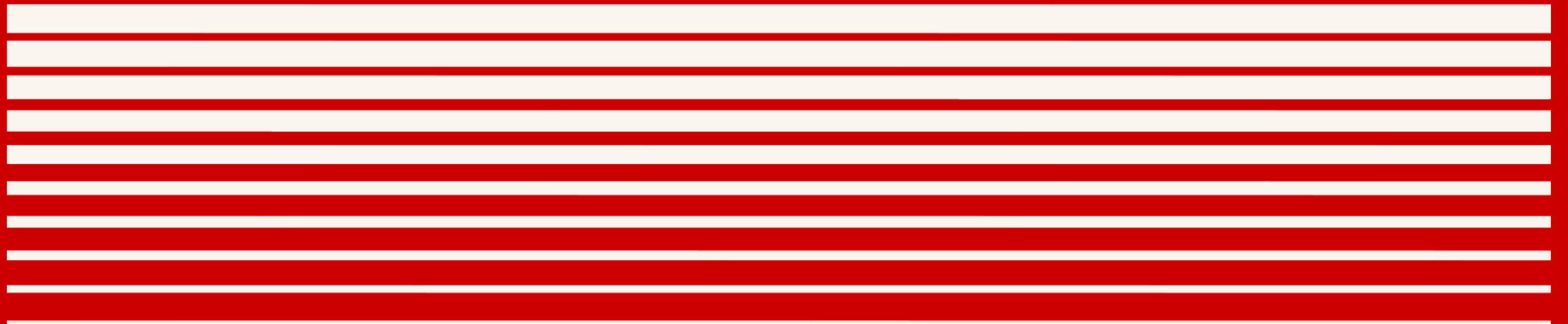
## Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

- ▶ Used to debug threading issues
  - ▶ Listen to the sequence of beeps and hear something out of sync
  
- ▶ One frequency per thread
  - ▶ Thread1: Beep( 750, 50);
  - ▶ Thread2: Beep( 850, 50);
  - ▶ ...
  
- ▶ `utilapi.h`
  
- ▶ <https://learn.microsoft.com/en-us/windows/win32/api/utilapiet/nf-utilapiet-beep>

# PERFORMANCE AND MEMORY



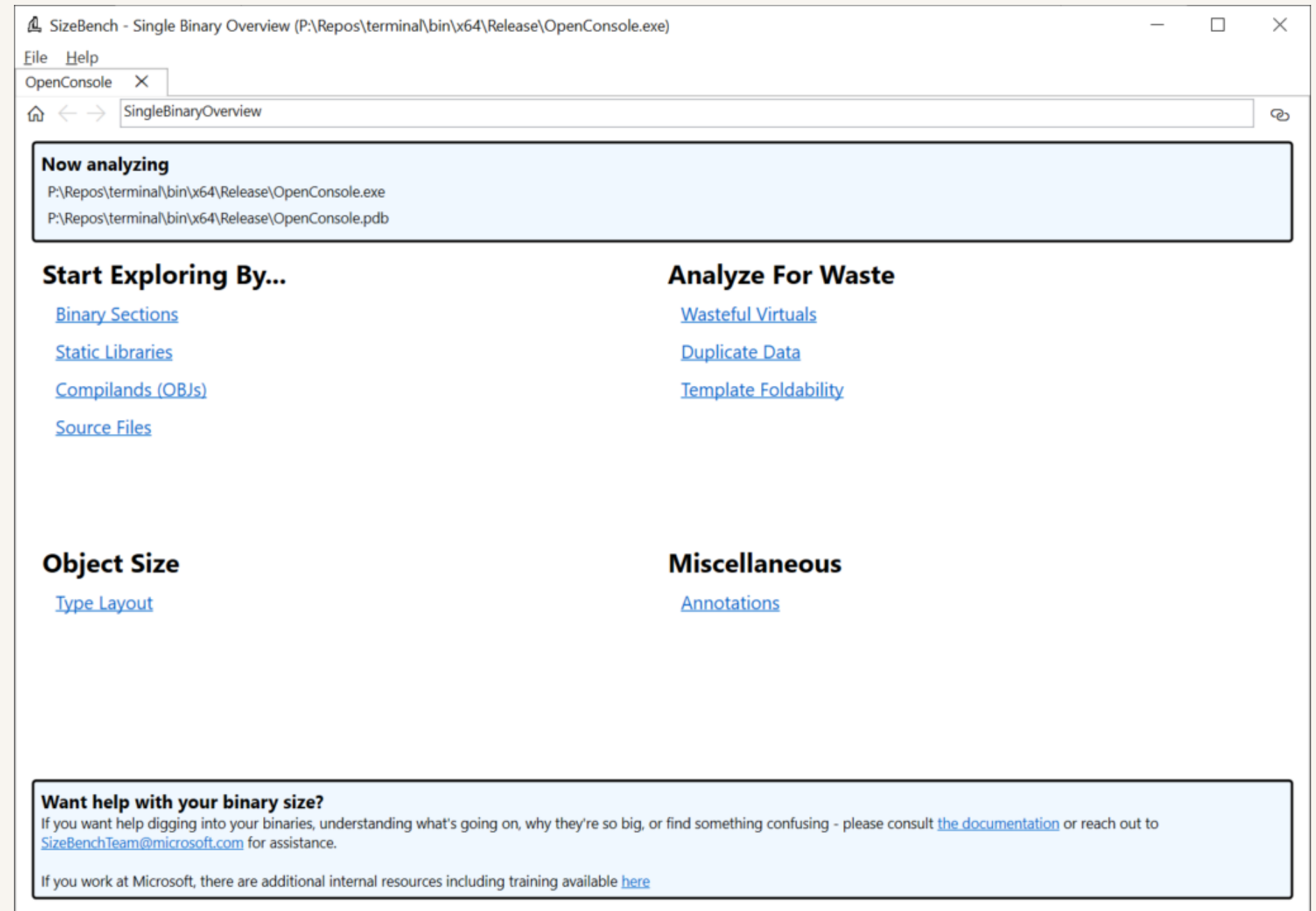
# Measure before and after optimization

- ▶ Can't be sure the optimization was an optimization
- ▶ Must have a baseline to compare against
  - ▶ Compilers and CPUs may behave unexpectedly



# Optimize binary size for DLL & EXE

- ▶ Smaller binary size equals...
  - ▶ Smaller patches/updates
  - ▶ Faster loading
  - ▶ Less memory when loaded
- ▶ Sizebench best tool I've found
  - ▶ Check imports
  - ▶ Binary sections in executable
  - ▶ Template, virtual, padding waste
  - ▶ <https://aka.ms/SizeBench>



# Wasteful virtuals

- ▶ Virtual functions create slots in vtable
- ▶ Having a large hierarchy can add up
- ▶ Wasteful
  - ▶ Virtuals with no overrides
  - ▶ Pure virtual with exactly one override
- ▶ Fix by removing virtual keyword

SizeBench - All Wasteful Virtuals (P:\Repos\terminal\bin\x64\Release\OpenConsole.exe)

File Help

OpenConsole X

AllWastefulVirtuals#ExcludeCOMTypes

These functions are marked virtual, but they have only zero or one overrides - this can cost a lot in terms of vtables in the binary, reloc entries, and so on. It's often much more efficient to devirtualize these.

Exclude COM Types

Export To Excel

Waste Per Slot	Wasted Size	Virtual Functions	Type Name
24 bytes	2.0 KB	CursorUp CursorDown CursorForward CursorBackward CursorNextLine CursorPrevLine CursorHorizontalPositionAbsolute VerticalLinePositionAbsolute HorizontalPositionRelative VerticalPositionRelative CursorPosition CursorSaveState CursorRestoreState CursorVisibility InsertCharacter DeleteCharacter ScrollUp ScrollDown InsertLine DeleteLine SetColumns SetCursorKeysMode SetKeypadMode EnableWin32InputMode EnableCursorBlinking SetAnsiMode SetScreenMode SetOriginMode SetAutoWrapMode SetTopBottomScrollingMargins WarningBell CarriageReturn LineFeed	Microsoft::Console::VirtualTerminal::TermDispatch

# Padding waste

- ▶ Caused by alignment
  - ▶ Each field in a type must be aligned in some way
- ▶ Is `sizeof(SimpleStruct) == 4`?

```
struct SimpleStruct
{
    char x; // 1 byte
    short y; // 2 byte
    char z; // 1 byte
};
```

- ▶ `sizeof(SimpleStruct)` is 6, waste is 2 bytes

```
struct SimpleStruct
{
    char x; // offset: 0, alignment 1 byte
    [padding] // offset: 1
    short y; // offset: 2, alignment 2 bytes
    char z; // offset: 4
    [padding] // offset: 5, alignment 1 byte
};
// sizeof(SimpleStruct) == 6
```

# Padding waste cont'd

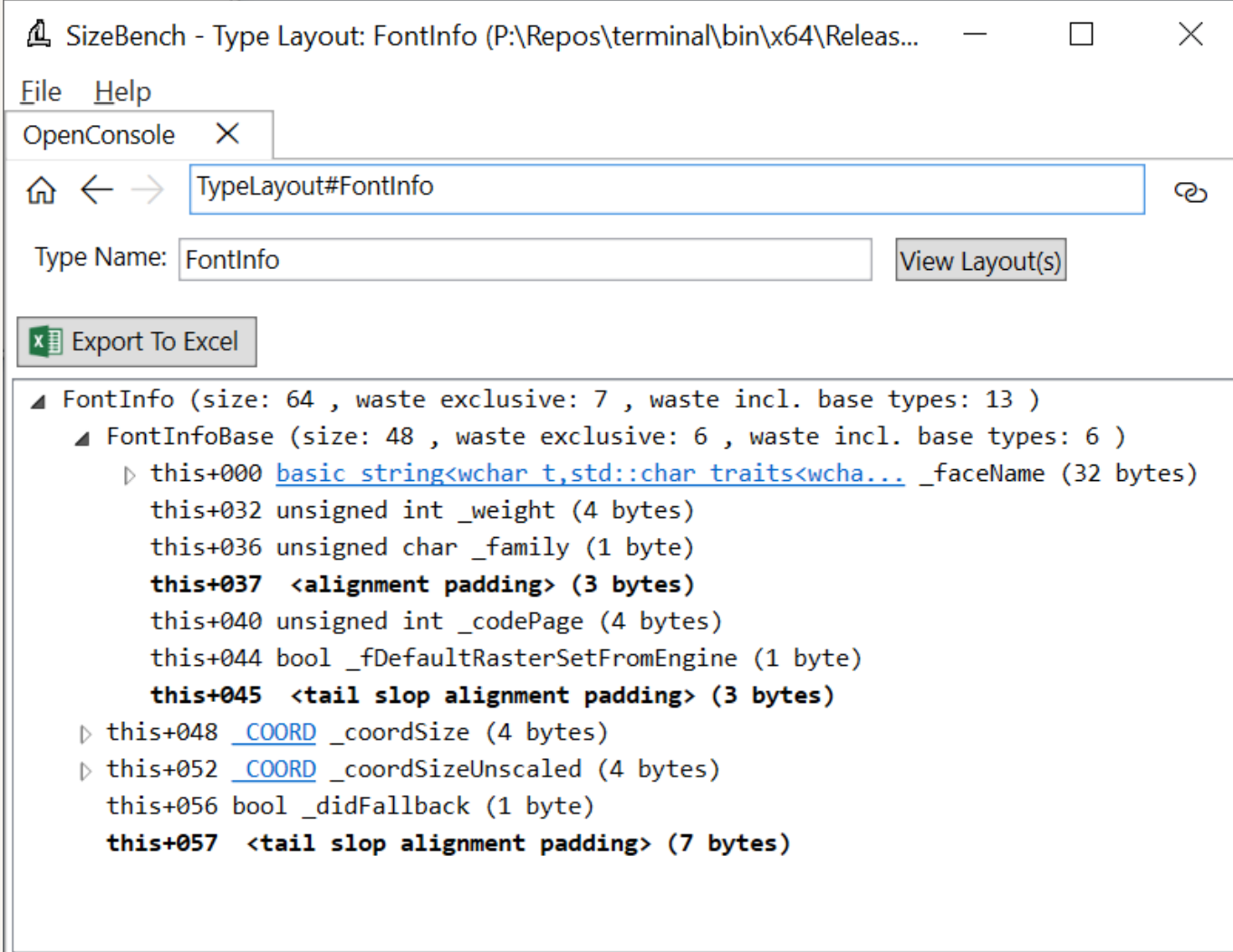
- ▶ Simple fix – restructure your structure

```
struct SimpleStruct
{
    short y; // offset: 0
    char x; // offset: 2
    char z; // offset: 3
};
// Now sizeof(SimpleStruct) == 4
```

- ▶ Fix by rearranging data and using bitfields

- ▶ Note that bitfields get padding if underlying types don't match

```
struct bitfieldStruct
{
    char x : 2; // offset: 0
    [padding] : 30; // offset: 2 bits
    unsigned int y : 30; // offset: 2 bytes
    [padding] : 2; // offset: 2 bytes + 30 bits
};
```



SizeBench - Type Layout: FontInfo (P:\Repos\terminal\bin\x64\Releas... — □ ×

File Help

OpenConsole ×

TypeLayout#FontInfo

Type Name: FontInfo View Layout(s)

Export To Excel

- FontInfo (size: 64 , waste exclusive: 7 , waste incl. base types: 13 )
  - FontInfoBase (size: 48 , waste exclusive: 6 , waste incl. base types: 6 )
    - this+000 [basic\\_string<wchar\\_t, std::char\\_traits<wcha... \\_faceName \(32 bytes\)](#)
    - this+032 unsigned int \_weight (4 bytes)
    - this+036 unsigned char \_family (1 byte)
    - this+037 <alignment padding> (3 bytes)
    - this+040 unsigned int \_codePage (4 bytes)
    - this+044 bool \_fDefaultRasterSetFromEngine (1 byte)
    - this+045 <tail slop alignment padding> (3 bytes)
    - this+048 [\\_COORD \\_coordSize \(4 bytes\)](#)
    - this+052 [\\_COORD \\_coordSizeUnscaled \(4 bytes\)](#)
    - this+056 bool \_didFallback (1 byte)
    - this+057 <tail slop alignment padding> (7 bytes)



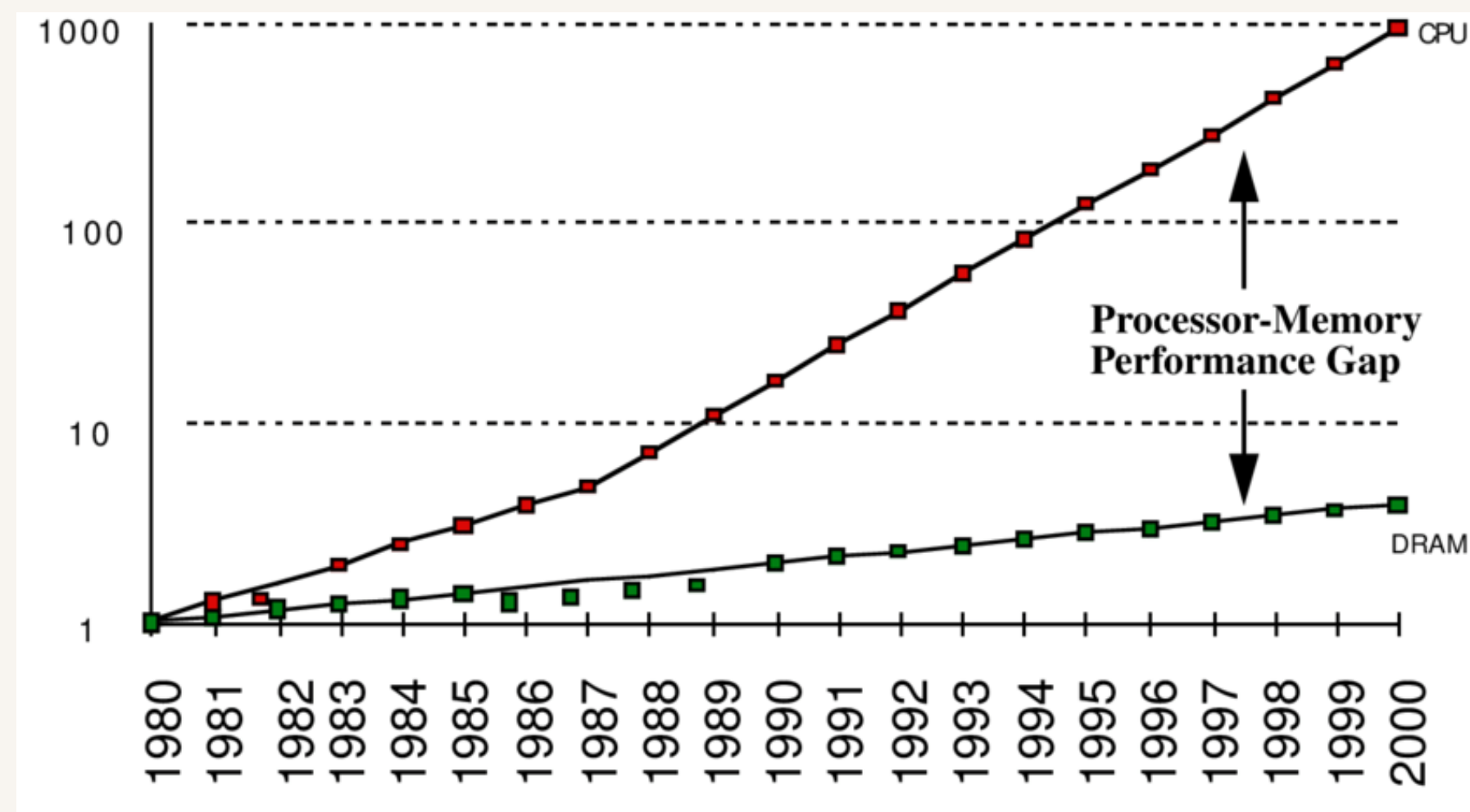
# Unrolling loops when it makes sense

- ▶ It can give gains on stalls
  - ▶ ...but can also get worse
- ▶ Compilers will help unroll
  - ▶ Human needed for more complex cases
- ▶ Compiler+CPU won't behave as you expect
  - ▶ You must benchmark before and after

```
for (int i = 0; i < maxLoop - 3; i += 4)
{
    bigSum += data[i + 0];
    bigSum += data[i + 1];
    bigSum += data[i + 2];
    bigSum += data[i + 3];
}
```

# Lookup tables can lose performance

- ▶ What is it?
  - ▶ Use pre-calculated values
    - ▶ F.e trigonometric values
  - ▶ Quite legacy/old-school
- ▶ CPU + memory gap
  - ▶ CPUs much faster than memory lookup



# Check your compiler flags

- ▶ Set in the beginning of a project
  - ▶ ...then forgotten about
- ▶ Can give you “free” performance boost
- ▶ Check the documentation for your compiler/platform
- ▶ Do this late in a project when needing that extra gain

# Evict log text from memory

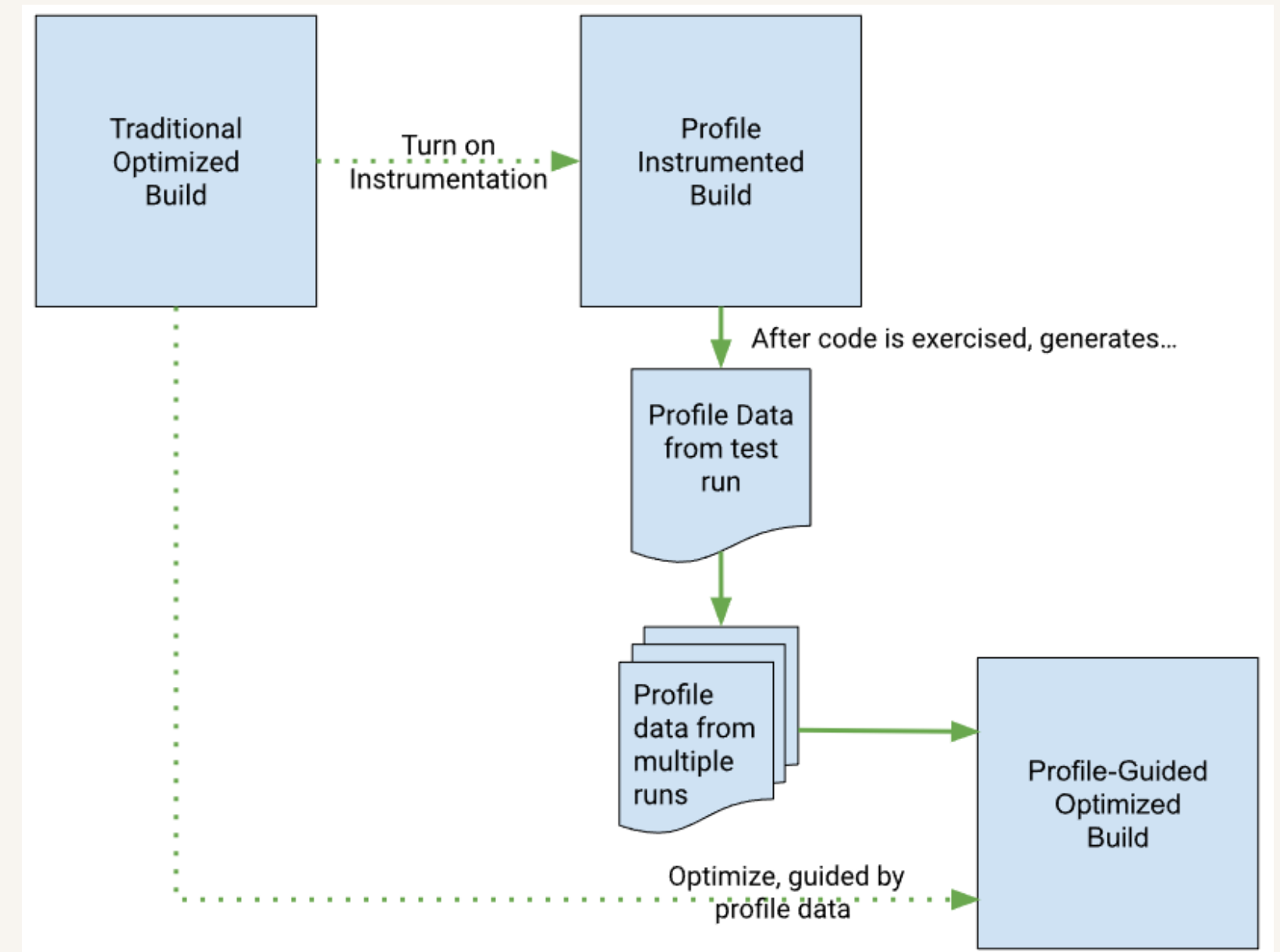
- ▶ Take stock of e.g log windows
- ▶ Serialize to disk, prone amount of text
- ▶ Had a C# tool that kept all text
  - ▶ It did eat up 30GB+ memory when running for a long time

# “Hidden” memory allocation for shipping

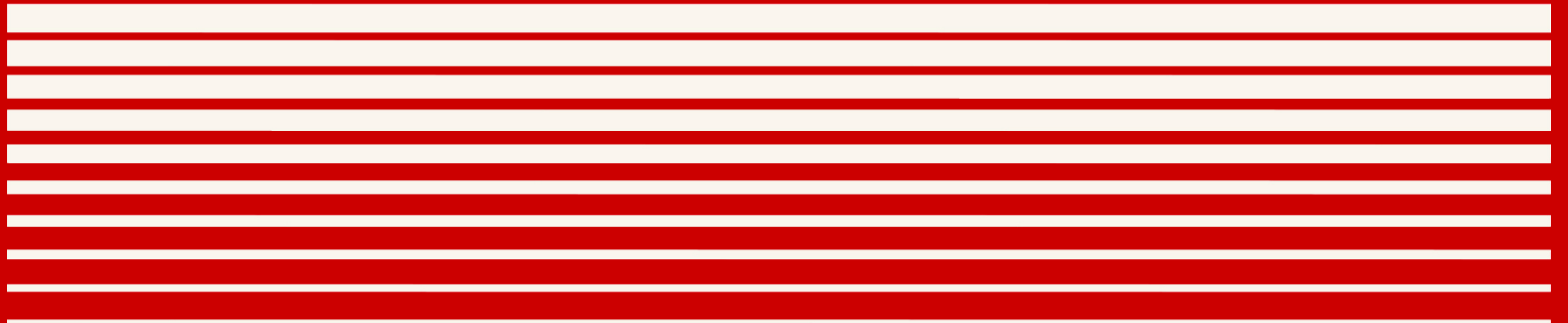
- ▶ Beginning of a project create a huge allocation
- ▶ Don't touch it during production
- ▶ Last thing before cert – release allocation
  - ▶ Can really be a crunchsaver

# PGO – Profile Guided Optimization

- ▶ Better performance, but..
  - ▶ ...extra build step
- ▶ Gets real data from how the game runs
  - ▶ Generates file, feed back to linker
- ▶ Can yield 5% – 10% perf boost



# COMPILING



# Forward declare for faster compiles

- ▶ Reduce the number of files opened by #include
- ▶ Reducing number of the pre-processed files
  - ▶ as the header is not included
- ▶ Reducing recompilation impact...
  - ▶ ...when the forward declared class is modified.

```
class ForwardDeclare;  
.....  
class ForwardDeclare  
{  
...  
};
```



# Use unity build for faster compilation

- ▶ Combines several source files to one bigger
  - ▶ Fewer number of header files for parsing and compiling
- ▶ Easiest implementation is to `#include` cpp files
  - ▶ Other implementations can write and create new (huge) cpp files
- ▶ Potential issues
  - ▶ Memory can be an issue
  - ▶ Symbol clashes going from local to global scopes

# When using unity builds, disable them occasionally

- ▶ If not checked regularly...
  - ▶ ...can yield quite weird compiling and linking errors
  - ▶ If compiling cleanly with Unity-build turned off, it wont break with it turned on
- ▶ Disable and start a compilation
  - ▶ In Unreal `<bUseUnityBuild>>false</bUseUnityBuild>`
  - ▶ Issues are usually forgotten `#includes` / `forward declares`

# Include-what-you-use

- ▶ Clang-based
- ▶ Faster compile / link
  - ▶ Remove redundant #includes
- ▶ Cleaner view of dependencies
  - ▶ An #include is a dependency

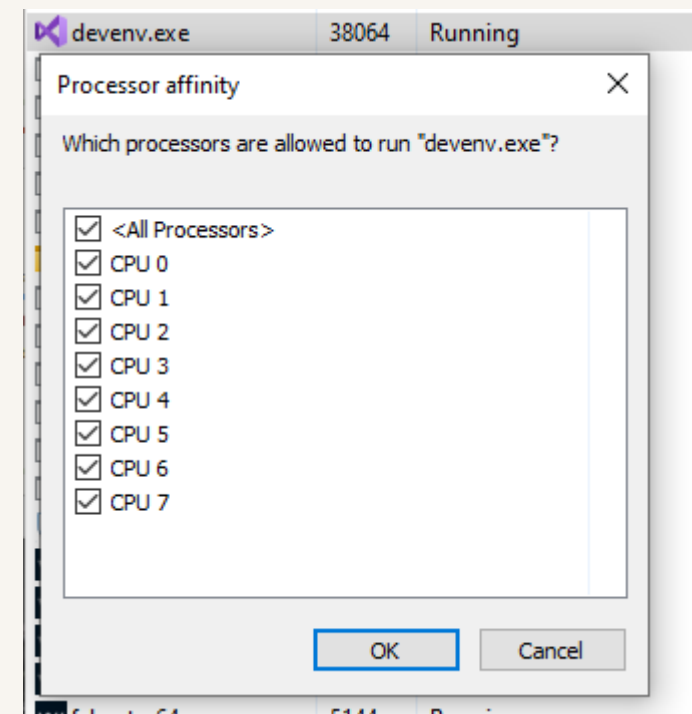
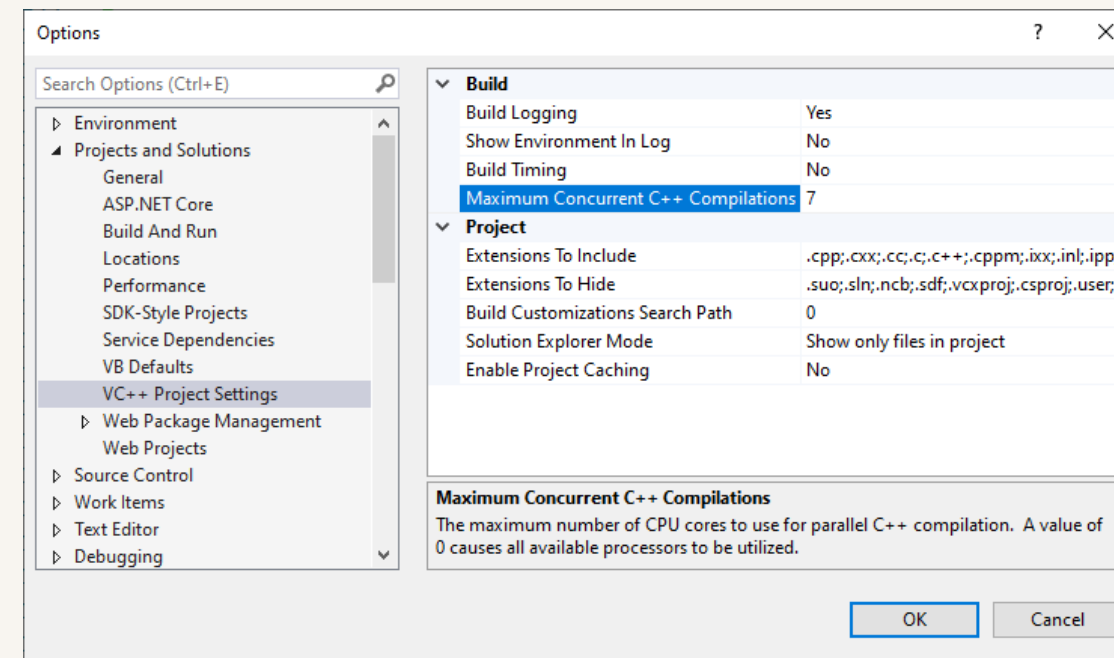
# Use static asserts (compile time asserts)

- ▶ Takes place after preprocessing
  - ▶ When triggered will break compilation
- ▶ Size checks for vectors, sizeof(<datatype>), etc.
- ▶ Error out on dependencies e.g a certain version of a library

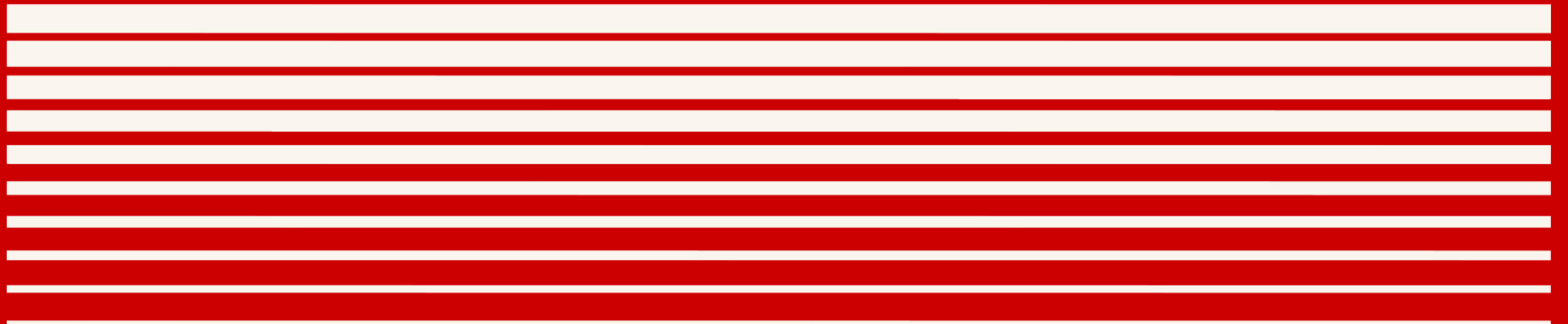
```
static_assert( THIRD_PARTY_LIB_VERSION == 123,  
             "This only works for version 123 of library");
```

# Limit number of cores during compilation

- ▶ In Settings in Visual Studio
  - ▶ Num of cores – 1 works for me
- ▶ Setting affinity also works
  - ▶ Spawned processes inherit affinity
- ▶ VS2022 has better support
  - ▶ Have not tried it myself



# GENERIC TIPS



# Be wary of copy/paste

- ▶ It's easy to miss changing index

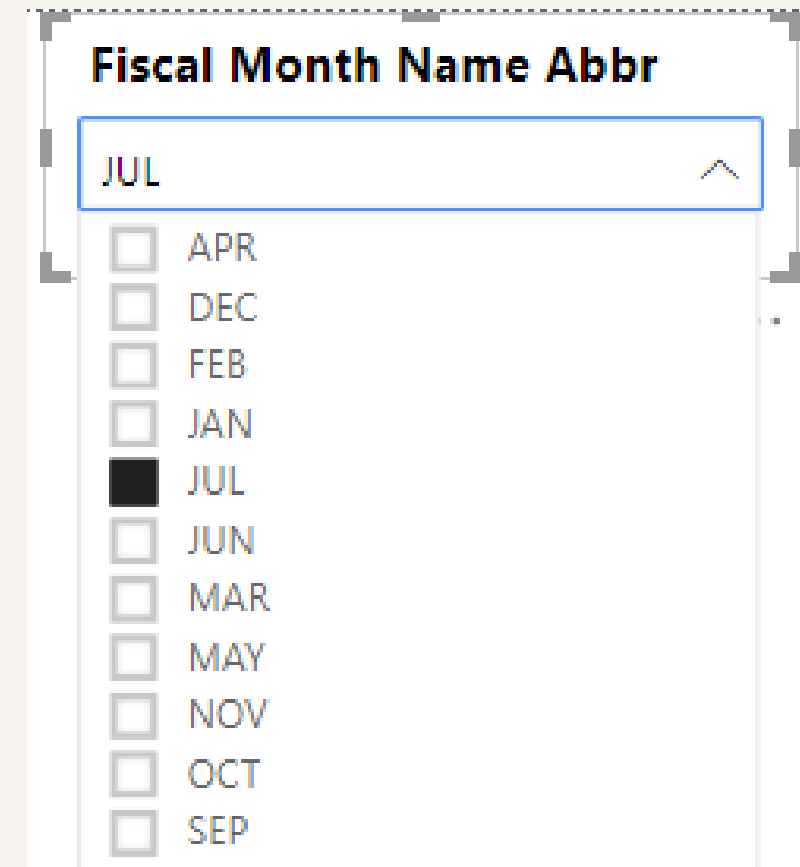
- ▶ `vector[0] = 0.0f;`
  - `vector[1] = 0.0f;`
  - `vector[1] = 0.0f;`

- ▶ Check an extra time

- ▶ If code reviewing, keep an eye on these types of errors

# Sort data – especially when serializing

- ▶ Remove random nature of data
- ▶ Doing patches
  - ▶ Randomized order => big patch
- ▶ UI
  - ▶ Funky ordering
- ▶ Debugging data fields
  - ▶ Easier compare





# Automation

- ▶ Use static code analyzers as build task and part of CI
  - ▶ After initial setup find new bugs
- ▶ Code-complexity charting
  - ▶ Cyclomatic complexity, maintainability index, coupling, ...
- ▶ Submitted bugfix heat mapping in code
  - ▶ Find candidates for rewrite
- ▶ Chart/check compilation times
  - ▶ When reaching a threshold, start optimizing
  - ▶ Build Insights now integrated into Visual Studio 2022
    - ▶ <https://aka.ms/BuildInsightsVS>



**THANK YOU FOR LISTENING!  
ANY QUESTIONS?**

OTTO KIVLING

OTTO@REDHILLGAMES.COM