

KIV/CPP – Programming in C++

10b. Code Debugging

Martin Úbl

KIV ZČU

2025/2026

- code debugging
- any native-code debugger can be used
 - MSVS debugger
 - gdb
 - lldb
 - ...
- preferably a high-level one – C++ support
 - e.g., name mangling
 - recognition of (polymorphic) types
 - ...
- within the course – **MSVS debugger**

- code debugging
- debugging information stored in the binary or separately
 - MSVS bundles a .pdb file
 - gcc/clang on GNU/Linux embed it into the binary
- debugging information formats
 - DWARF
 - PE/COFF
 - stabs
 - OMF
 - ...

- code debugging
- debugging information
 - function/method names
 - variable types
 - mapping of binary code to source code
 - ...

- debugger
- loads debugging information
- registers itself with the operating system for the given program
- allows:
 - instruction breakpoints
 - data breakpoints
 - data watch
 - memory modification
 - and more...

Debugging

- instruction breakpoint
- often with hardware support – debug registers
- or purely software-based – inserting instructions
- stops program execution at the given instruction (before it)
- signals the debugger
- can be conditional (then significantly slows down program execution)

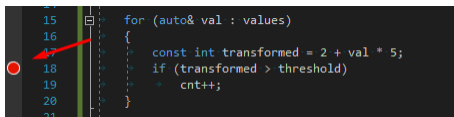
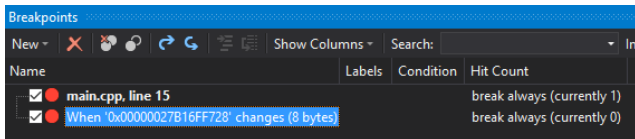
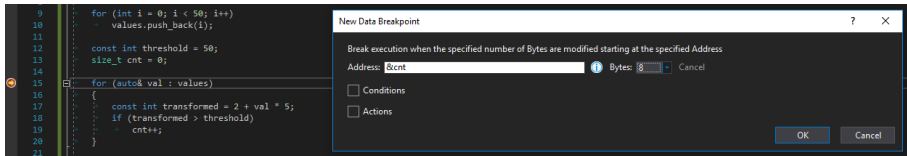
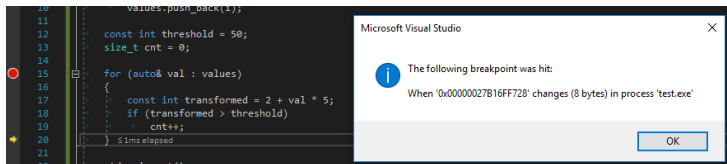


Figure: Instruction breakpoint set in MSVS

Debugging

- data breakpoint
- allows stopping program execution when a memory value (variable) changes

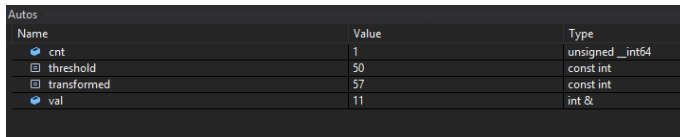




- data breakpoint
- disadvantage: addresses change, so it is necessary to set an instruction breakpoint at a place where we already know the address and only then add a data breakpoint

Debugging

- watch
- view of a piece of memory during debugging
- makes sense during stepping
- shows current memory values
- automatic watch
 - the debugger deduces what might be of interest
- manual watch



The image shows a screenshot of the 'Autos' window in Microsoft Visual Studio. The window has a dark background and contains a table of variables currently in scope. The table has three columns: 'Name', 'Value', and 'Type'. There are four rows of data, each with a small blue icon to the left of the variable name. The variables are 'cnt', 'threshold', 'transformed', and 'val'.

Name	Value	Type
cnt	1	unsigned __int64
threshold	50	const int
transformed	57	const int
val	11	int &

Figure: Automatic watch in MSVS

The screenshot shows the 'Watch 1' window in MSVS. It contains a table with three columns: Name, Value, and Type. Two entries are listed: 'values[10]' with value 10 and type 'int', and 'values[25]' with value 25 and type 'int'. The 'values[25]' row is highlighted. Below the table is a large empty area for comments. At the bottom, there are tabs for 'Autos', 'Locals', and 'Watch 1', with 'Watch 1' being the active tab.

Name	Value	Type
values[10]	10	int
values[25]	25	int

Autos Locals Watch 1

Figure: Manual watch in MSVS

- conditional instruction breakpoint
- a condition can be set as a Boolean expression or a hit count
- when the breakpoint is hit, the condition is checked, and execution breaks if it is satisfied

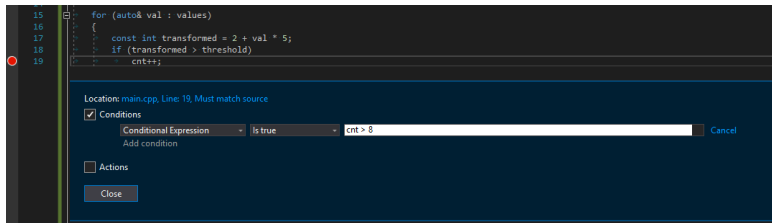


Figure: Conditional instruction breakpoint in MSVS

- memory values can be changed at runtime when the program is paused

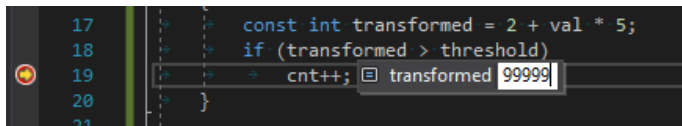


Figure: Editing memory in MSVS

Debugging

- MSVS also recognizes composed types
- their internals can be “expanded” and edited here as well

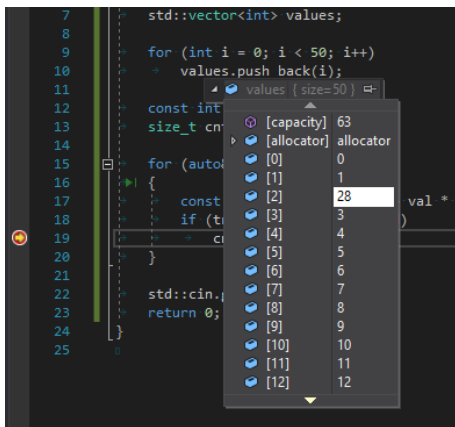


Figure: Editing memory in MSVS

- custom objects can also be inspected

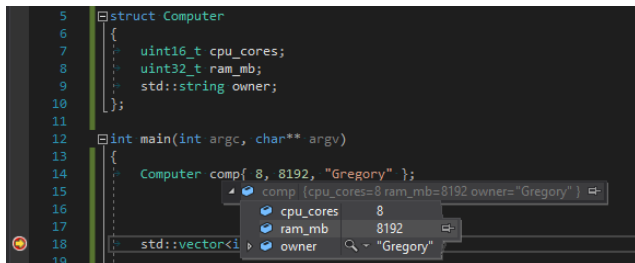


Figure: Editing memory in MSVS

- Edit and continue
- under certain circumstances, a paused program can be modified, recompiled, and its code replaced at runtime
- it is necessary to switch the debugging information format to /ZI

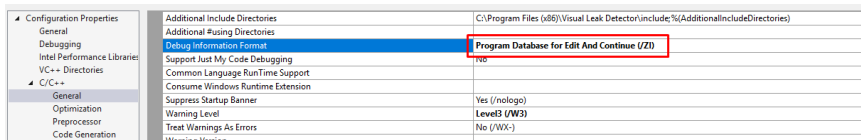
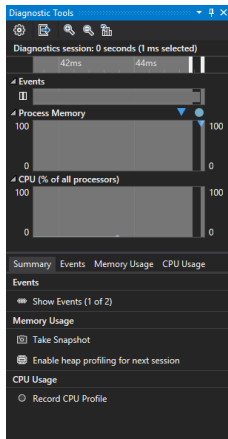


Figure: Setting the debugging information format in MSVS

Debugging

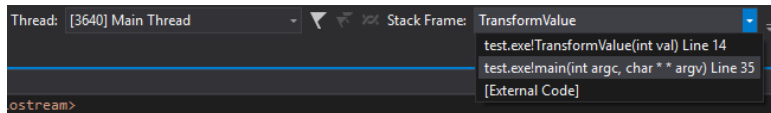
- diagnostic tools
- heap profiler



- *continue* – continue execution
- *pause* – pause the program at the given moment
- *stop* – stop the program
- *restart*
- stepping
 - *step into* – enter a function
 - *step over* – step to the next line of code
 - *step out* – continue until returning from a function

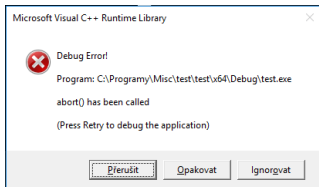


- inspection of the call stack state
 - it is possible to switch to any nesting level within a function call
- switching thread contexts
 - multithreaded programs allow observing different thread contexts as well



Debugging

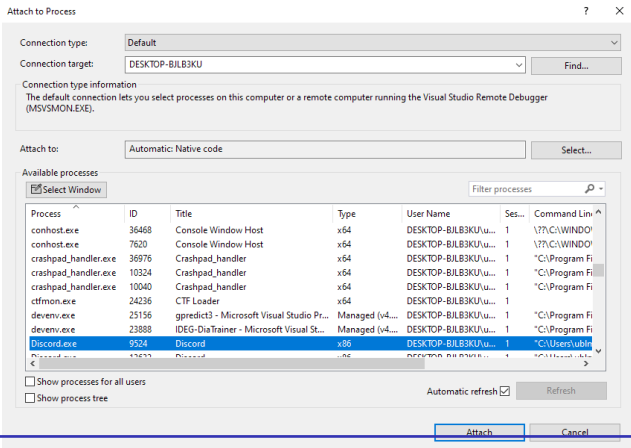
- runtime assertion
- `#include <assert.h>`
- `assert(...)` macro
- it contains a condition; if it is not satisfied, the operating system is signaled
- Windows – after choosing “Retry”, the signal can be passed to the debugger



```
assert(count == 5 && "Count_is_incorrect");
```

Debugging

- a debugger can also be attached to another process
- it does not have to be a process started from Visual Studio (or gdb, ...)
- successful debugging requires debug symbols (PDB, ...)



- a debugger can also be attached to a “remote” process
 - over the network on another PC (SSH, ...)
 - in WSL
 - in the cloud (Azure)
 - interpreting another supported language (Javascript, Python, ...)
 - to other environments (Unity, Unreal Engine, qemu, ...)