

KIV/CPP – Programování v jazyce C++ 10b. Ladění kódu

Martin Úbl

KIV ZČU

2023/2024

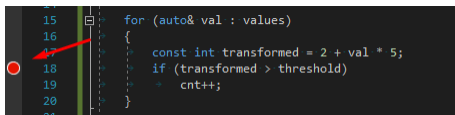
- ladění kódu
- možné použít jakýkoliv debugger nativního kódu
 - MSVS debugger
 - gdb
 - lldb
 - ...
- lépe vysokoúrovňový – podpora C++
 - např. name mangling
 - rozpoznání (polymorfních) typů
 - ...
- v rámci předmětu – **MSVS debugger**

- ladění kódu
- debug informace uloženy v binárce nebo separátně
 - MSVS přibaluje .pdb soubor
 - gcc/clang na GNU/Linuxu vkládají do binárky
- formáty debug informací
 - DWARF
 - PE/COFF
 - stabs
 - OMF
 - ...

- ladění kódu
- debug informace
 - názvy funkcí/metod
 - typy proměnných
 - mapování binárního kódu na zdrojový kód
 - ...

- debugger
- načítá debug informace
- registruje se operačnímu systému pro daný program
- dovoluje:
 - instrukční breakpointy
 - datové breakpointy
 - data watch
 - modifikace paměti
 - a další...

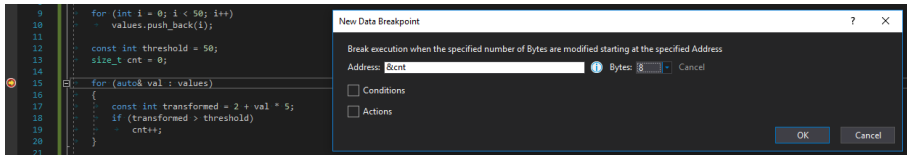
- instrukční breakpoint
- často s podporou hardware – ladicí registry
- nebo čistě softwarové – vkládání instrukcí
- zastaví běh programu na dané instrukci (před ní)
- signalizuje debugger
- může být podmíněný (výrazně pak zpomaluje běh programu)



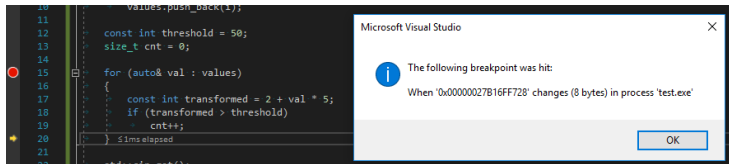
Obrázek: Instrukční breakpoint nastavený v MSVS

Debugging

- datový breakpoint
- dovoluje zastavit provádění programu při změně hodnoty paměti (proměnné)

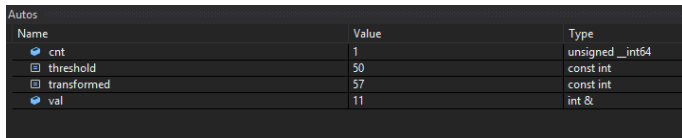


Breakpoints			
Name	Labels	Condition	Hit Count
<input checked="" type="checkbox"/> main.cpp, line 15		break always (currently 1)	
<input checked="" type="checkbox"/> When "0x00000027B16FF728" changes (8 bytes)		break always (currently 0)	



- datový breakpoint
- nevýhoda: adresy se mění, je třeba nastavit instrukční breakpoint v místě, kde už známe adresu a až potom přidat datový breakpoint

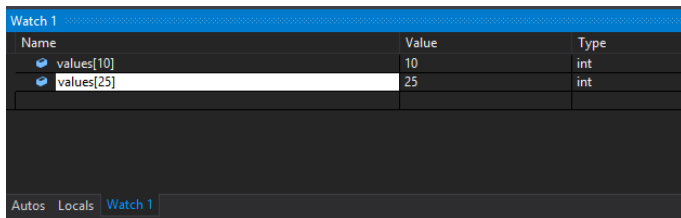
- watch
- pohled na kus paměti při debuggování
- má smysl při krokování
- ukazuje aktuální hodnoty paměti
- automatický watch
 - debugger vydedukuje, co by mohlo zajímat
- ruční watch



The screenshot shows the 'Autos' window in MSVS, which displays variables that the debugger has automatically identified as interesting. The window contains a table with three columns: Name, Value, and Type.

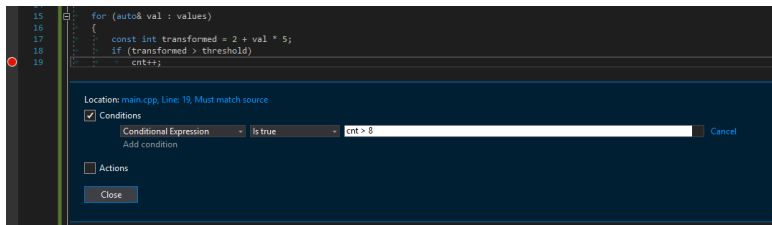
Name	Value	Type
cnt	1	unsigned __int64
threshold	50	const int
transformed	57	const int
val	11	int &

Obrázek: Automatický watch v MSVS



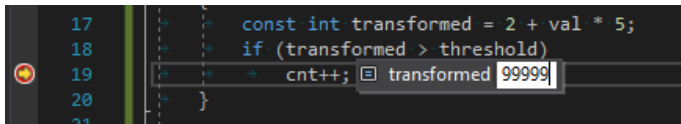
Obrázek: Ruční watch v MSVS

- podmíněný instrukční breakpoint
- lze nastavit podmínku v podobě logického výrazu nebo počtu průchodů (hit count)
- při průchodu breakpointem se podmínka ověří a při splnění breakne



Obrázek: Podmíněný instrukční breakpoint v MSVS

- hodnoty paměti lze za běhu měnit, když je program pozastaven



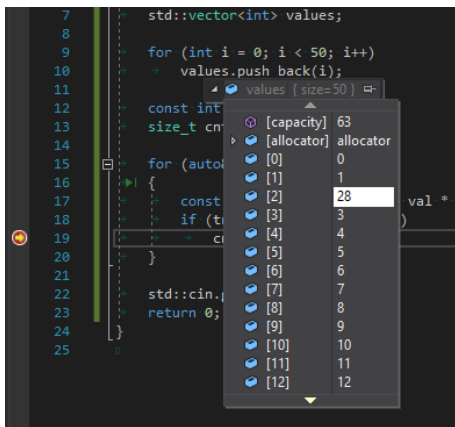
The screenshot shows a code editor with the following C++ code:

```
17     const int transformed = 2 + val * 5;
18     if (transformed > threshold)
19         cnt++;
20     }
21 }
```

A red arrow icon is on the left margin of line 19. A tooltip is displayed over the code, showing the variable `transformed` with a value of `99999`. The tooltip has a small icon on the left and a text input field containing the value `99999`.

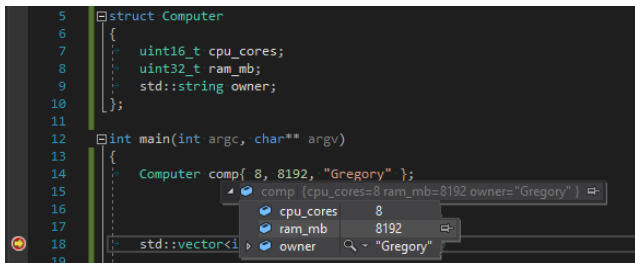
Obrázek: Editace paměti v MSVS

- MSVS rozpoznává i komponované typy
- lze jejich vnitřnosti „rozbalit“ a zde i editovat



Obrázek: Editace paměti v MSVS

- lze inspektovat i vlastní objekty



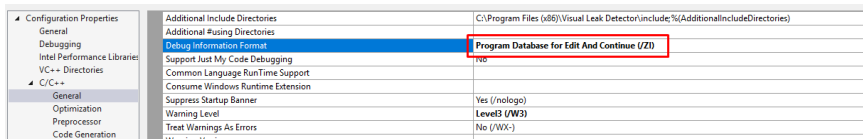
```
5 struct Computer
6 {
7     uint16_t cpu_cores;
8     uint32_t ram_mb;
9     std::string owner;
10 };
11
12 int main(int argc, char** argv)
13 {
14     Computer comp{ 8, 8192, "Gregory" };
15
16     std::vector<i>
```

The screenshot shows the Visual Studio debugger's 'Locals' window. It displays the variable 'comp' of type 'Computer' with the following values:

Field	Value
cpu_cores	8
ram_mb	8192
owner	"Gregory"

Obrázek: Editace paměti v MSVS

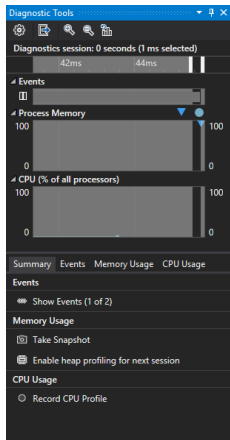
- Edit and continue
- pozastavený program lze za určitých okolností modifikovat, překompilovat a nahradit kód za běhu
- nutno přepnout formát debug informace na /ZI



Obrázek: Nastavení formátu debug informací v MSVS

Debugging

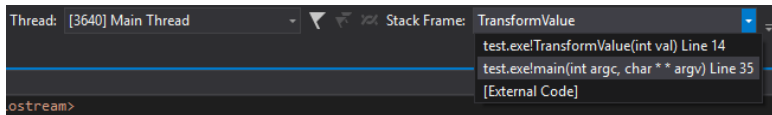
- diagnostic tools
- heap profiler



- *continue* – pokračování v provádění
- *pause* – pozastavení programu v daném momentě
- *stop* – zastavení programu
- *restart*
- krokování
 - *step into* – zanoření do funkce
 - *step over* – překročení na další řádku kódu
 - *step out* – pokračování do návratu z funkce



- inspekce stavu zásobníku volání
 - lze se přepnout na libovolnou z úrovní zanoření při funkčním volání
- přepínání kontextů vláken
 - vícevláknové programy dovolují pozorovat i odlišné kontexty vláken



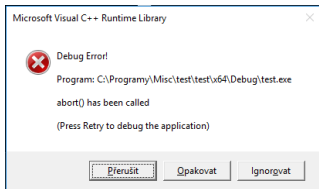
The screenshot shows a debugger's stack window. At the top, it indicates the current thread is "[3640] Main Thread". Below this, the stack frames are listed:

- Stack Frame: TransformValue (highlighted in blue)
- test.exe!TransformValue(int val) Line 14
- test.exe!main(int argc, char ** argv) Line 35
- [External Code]

At the bottom left of the window, the text ".ostream>" is visible.

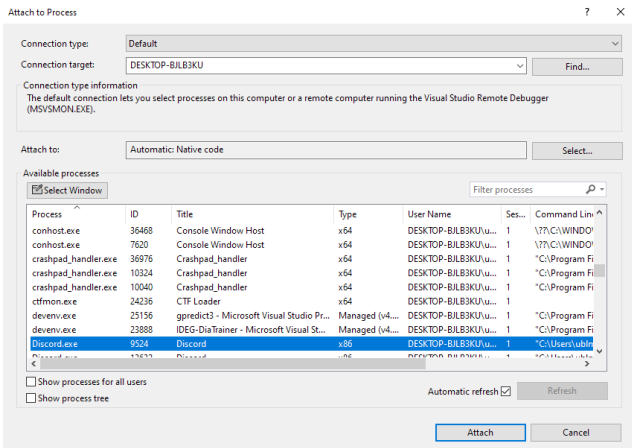
Debugging

- runtime assertion
- `#include <assert.h>`
- makro `assert(...)`
- uvnitř je podmínka, při nesplnění je signalizován operační systém
- Windows – po zvolení „Opakovat“ lze předat signál do debuggeru



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

- debugger lze připojit i k jinému procesu
- nemusí jít o proces spuštěný z Visual Studio (popř. gdb, ...)
- pro úspěšné ladění je třeba mít ladicí symboly (PDB, ...)



- debugger lze připojit i na „vzdálený“ proces
 - přes síť na jiném PC (SSH, ...)
 - ve WSL
 - v cloudu (Azure)
 - interpretující jiný podporovaný jazyk (Javascript, Python, ...)
 - na jiná prostředí (Unity, Unreal Engine, qemu, ...)