KIV/TSI - Seminář C++

09. Další konstrukce jazyka C++, ladění kódu, inline assembly, intrinsics

Martin Úbl

KIV ZČU

2020/2021

Další konstrukce jazyka C++

- přehled doposud neprobíraných konstrukcí
- další speciality z STL
- vlastnosti z C++14 a C++17

Další konstrukce jazyka C++

- *std::filesystem*
- `#include <filesystem>`
- od C++17
- práce se souborovým systémem užitím jednotného rozhraní
  - práce se soubory, s adresáři a odkazy
  - práva
  - časy modifikace, vytvoření, ...
  - obsazení a kapacita oddílu

Další konstrukce jazyka C++

- *std::filesystem::path*
- reprezentuje cestu v souborovém systému
- nemusí nutně identifikovat existující soubor/složku/odkaz
- absolutní/relativní
- převod na absolutní `std::filesystem::absolute`
- převod na relativní `std::filesystem::relative`
- lze implicitně převést na `std::string`

```
std::filesystem::path filepath("input.txt");

std::cout << filepath << std::endl;
std::cout << std::filesystem::absolute(filepath)
          << std::endl;
```

Další konstrukce jazyka C++

- *std::filesystem::path*
- připojení komponenty pomocí append nebo operátoru / a /=

```
std::filesystem::path filepath("C:\\");

filepath.append("slozka");

std::cout << filepath << std::endl;
// C:\slozka

filepath /= "podslozka";

std::cout << filepath << std::endl;
// C:\slozka\podslozka
```

Další konstrukce jazyka C++

- *std::filesystem::path*
- další metody
  - `filename` - vrací název souboru
  - `parent_path` - odebere poslední komponentu (vrací rodičovský adresář)
  - `make_preferred` - převod oddělovačů komponent cesty na preferované pro daný OS
  - `remove_filename` - odstraní z cesty název souboru
  - `replace_filename` - nahradí název souboru jiným názvem
  - `stem` - vrací název souboru bez přípony
  - `extension` - vrací příponu

Další konstrukce jazyka C++

- *std::filesystem*
- další funkce - systémové cesty
  - `std::filesystem::current_path()` - vrací nebo nastavuje pracovní adresář
  - `std::filesystem::temp_directory_path()` - vrací adresář pro dočasné soubory

Další konstrukce jazyka C++

- *std::filesystem*
- další funkce - manipulace se souborovým systémem
  - `std::filesystem::exists()` - existence souboru/složky/odkazu
  - `std::filesystem::copy()` - kopie složky nebo souboru
  - `std::filesystem::rename()` - přejmenování nebo přesun
  - `std::filesystem::remove()` - smazání souboru nebo složky
  - `std::filesystem::status()` - vrací vlastnosti souboru

Další konstrukce jazyka C++

- *std::filesystem*
- `std::filesystem::directory_entry` - třída reprezentující položku v adresáři
  - metoda `path()` vrací fyzickou cestu
- `std::filesystem::directory_iterator` - vrací instanci třídy pro iterování přes položky adresáře

Další konstrukce jazyka C++

- *std::filesystem*
- některé funkce při nezdaru vyhazují výjimku `std::filesystem::filesystem_error()`
  - přístup k souboru který neexistuje
  - systémové soubory
  - práva

Další konstrukce jazyka C++

- *if constexpr*
- podmínka vyhodnotitelná v čase kompilace
- „méně upovídaná“ varianta např. šablonového instancování
- podmínka musí být constexpr

```
template<typename T>
void DoSomething(const T& val) {

  if constexpr (std::is_integral_v<T>)
    DoIntegral(val);
  else
    DoOther(val);

}
```

Další konstrukce jazyka C++

- atributy
- syntaktický doplněk pro další optimalizace
- označení funkce/metody/bloku kódu s určitými vlastnostmi
  - `[[noreturn]]` - z funkce se nebudeme vracet (jediný návrat je buď výjimkou nebo nijak)
  - `[[deprecated("duvod")]]` - funkce je označena jako zastaralá
  - `[[fallthrough]]` - switch case level není zakončen breakem úmyslně
  - `[[maybe_unused]]` - označuje identifikátor, který nemusí být nikde použitý
  - a další...
  - `https://en.cppreference.com/w/cpp/language/attributes`

Další konstrukce jazyka C++

- structured binding
- lze svázat vícenásobnou inicializaci s nějakým objektem
- např. rozkopírovat pole do více proměnných
- nebo nahradit `std::tie` u tuple
- syntaxe s `auto[...] = ...`
- lze svázat hodnotou nebo referencí

```
std::array<int, 2> arr{ 5, 10 };
auto[a, b] = arr;

std::tuple<int,double> tup{5, 12.5};
auto&[i, d] = tup;
```

Další konstrukce jazyka C++

- structured binding
- možno např. ověřovat, zda se vložení prvku do kontejneru povedlo

```
std::map<int, std::string> mp;

if (auto[itr, succ]
          = mp.insert({ 42, "meaning" }); succ)
  std::cout << "OK" << std::endl;
else
  std::cout << "FAIL" << std::endl;
```

Další konstrukce jazyka C++

- structured binding
- lze svázat hodnotou, l- nebo r-value referencí

```
std::array<int, 2> arr{ 5, 10 };
auto[a, b] = arr;
auto&[c, d] = arr;
auto&&[e, f] = std::make_tuple(5, 6);
```

Debugging

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

Debugging

- 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
  - ...

Debugging

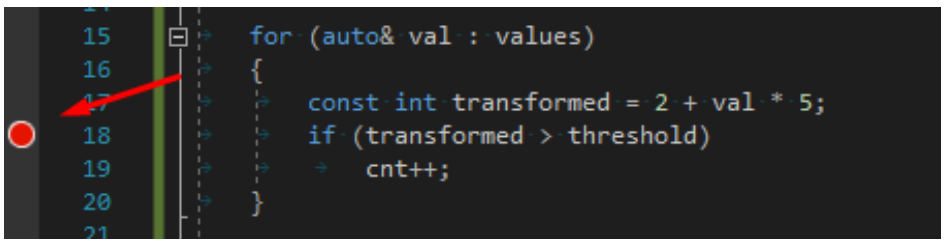
- ladění kódu
- debug informace
  - názvy funkcí/metod
  - typy proměnných
  - mapování binárního kódu na zdrojový kód
  - ...

Debugging

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

Debugging

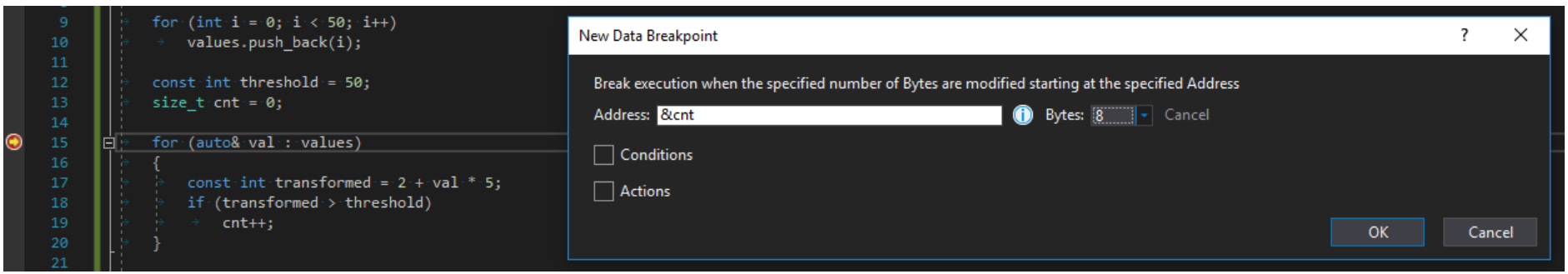
- instrukční breakpoint
- často s podporou hardware - ladicí registry
- nebo čistě softwarové - vkládání instrukcí
- zastaví běh programu na dané instrukci (před ní)
- signalizuje debugger
- může být podmíněný (výrazně pak zpomaluje běh programu)

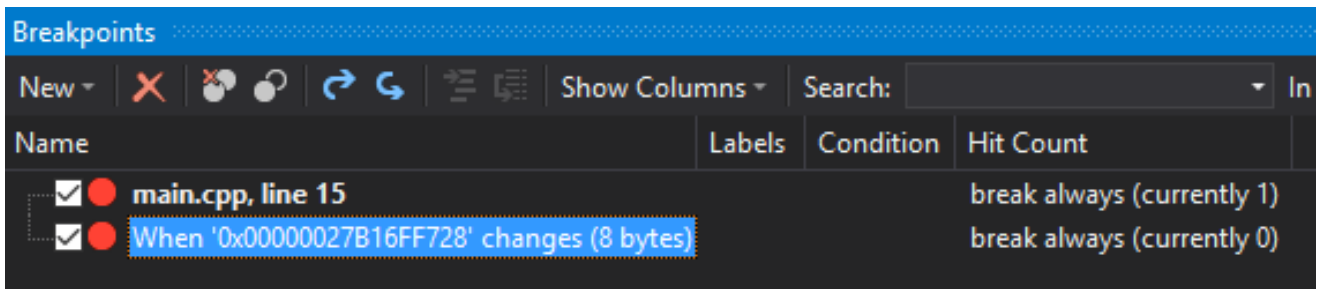


Obrázek: Instrukční breakpoint nastavený v MSVS

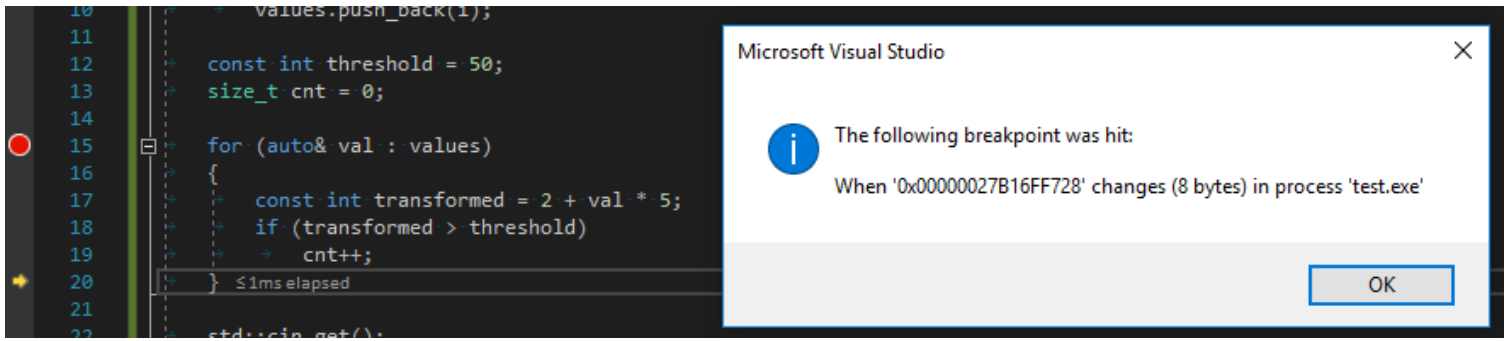
Debugging

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





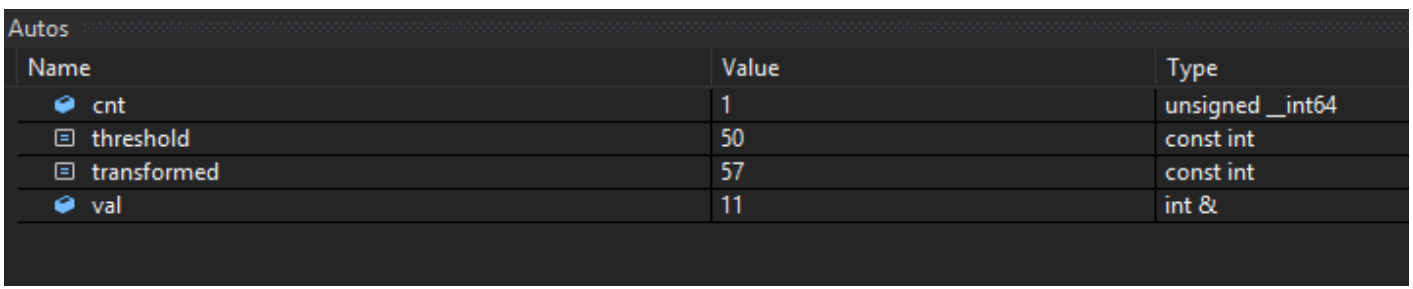
Debugging



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

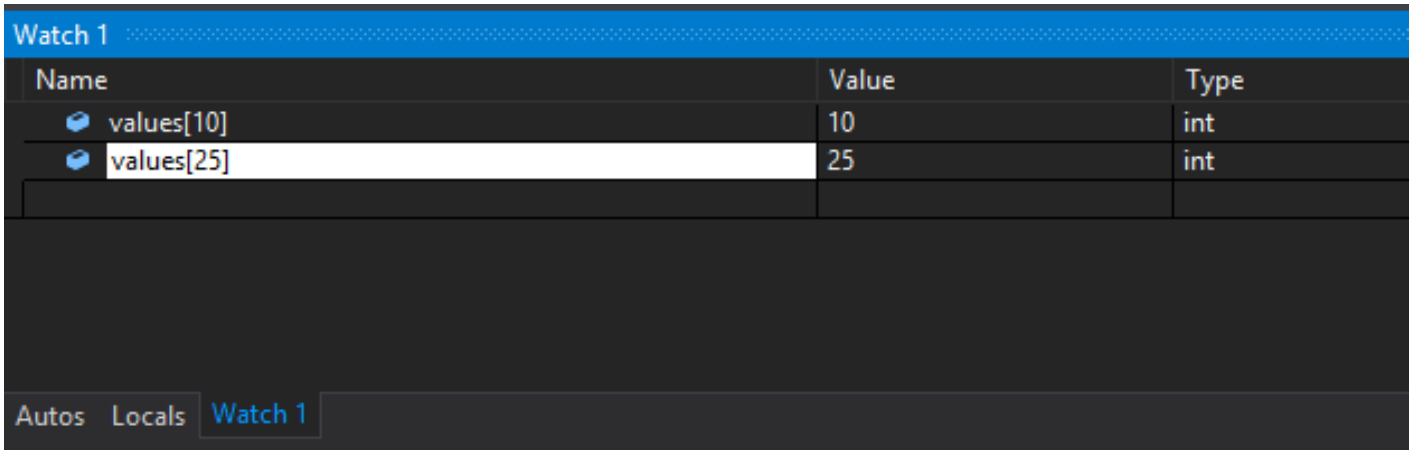
Debugging

- 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



Obrázek: Automatický watch v MSVS

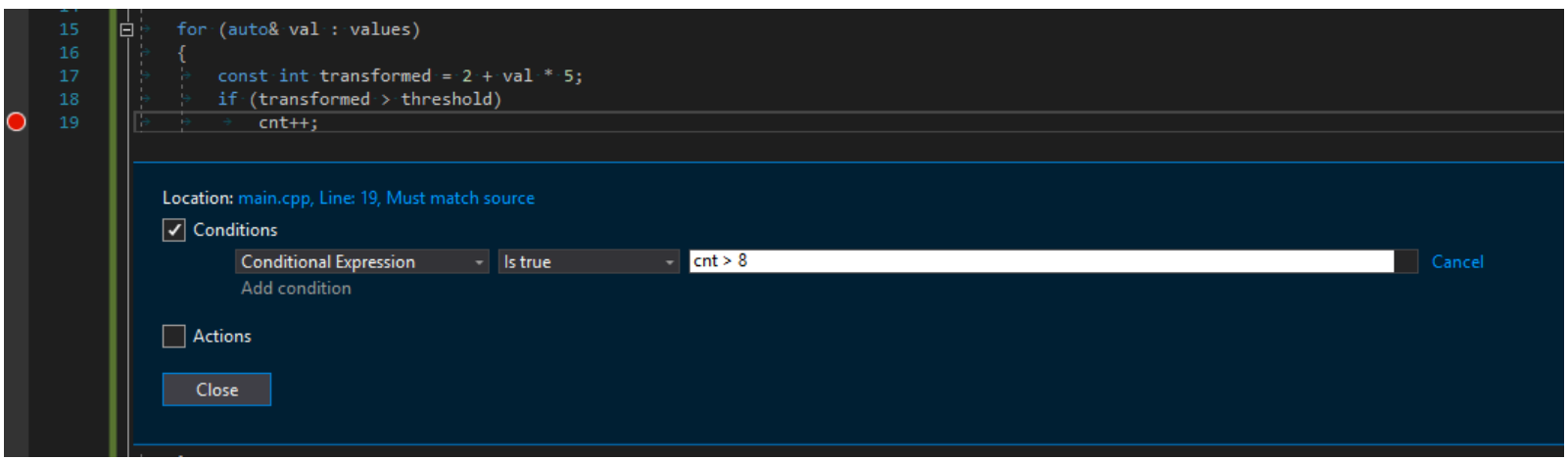
Debugging



Obrázek: Ruční watch v MSVS

Debugging

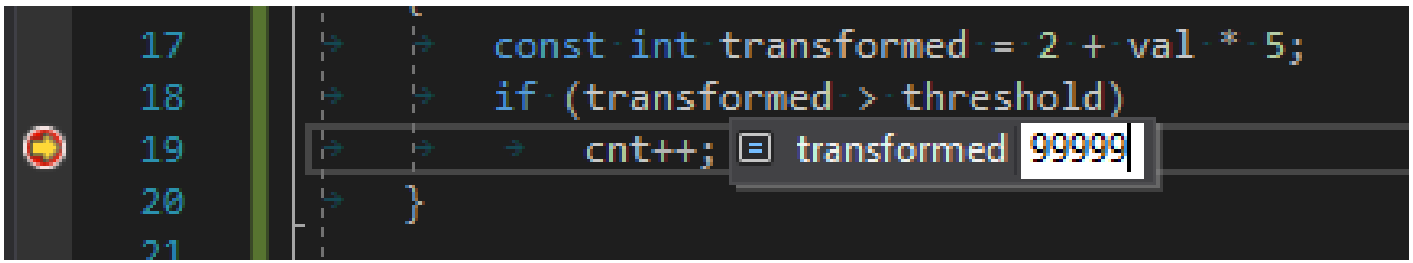
- podmíněný instrukční breakpoint
- lze nastavit podmínku v podobě logického výrazu nebo počtu průchodů (hit count)
- při průchodu breakpointem se podmínka ověří a při splnění breakne



Obrázek: Podmíněný instrukční breakpoint v MSVS

Debugging

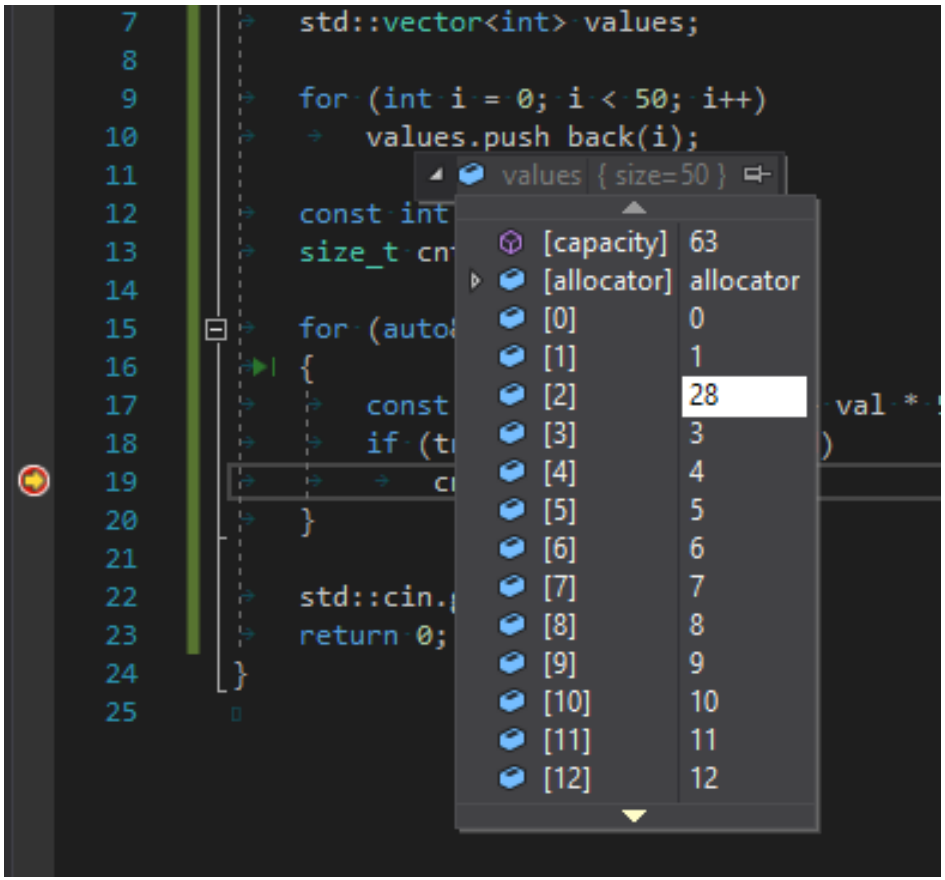
- hodnoty paměti lze za běhu měnit, když je program pozastaven



Obrázek: Editace paměti v MSVS

Debugging

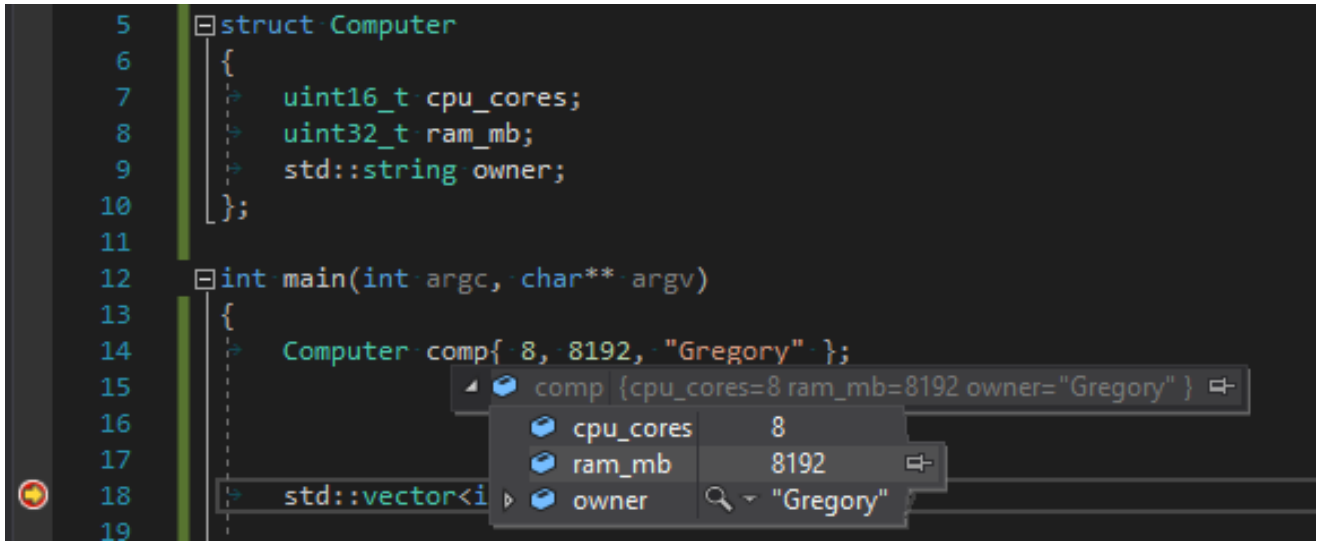
- MSVS rozpoznává i komponované typy
- lze jejich vnitřnosti „rozbalit“ a zde i editovat



Obrázek: Editace paměti v MSVS

Debugging

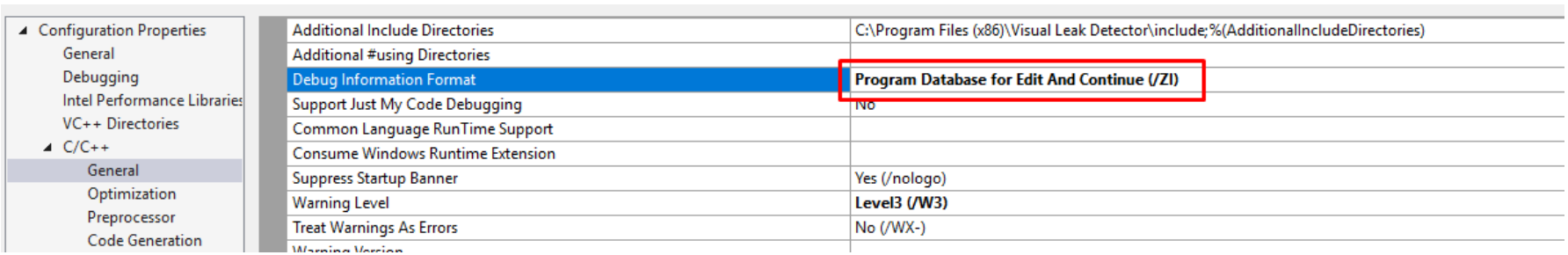
- lze inspektovat i vlastní objekty



Obrázek: Editace paměti v MSVS

Debugging

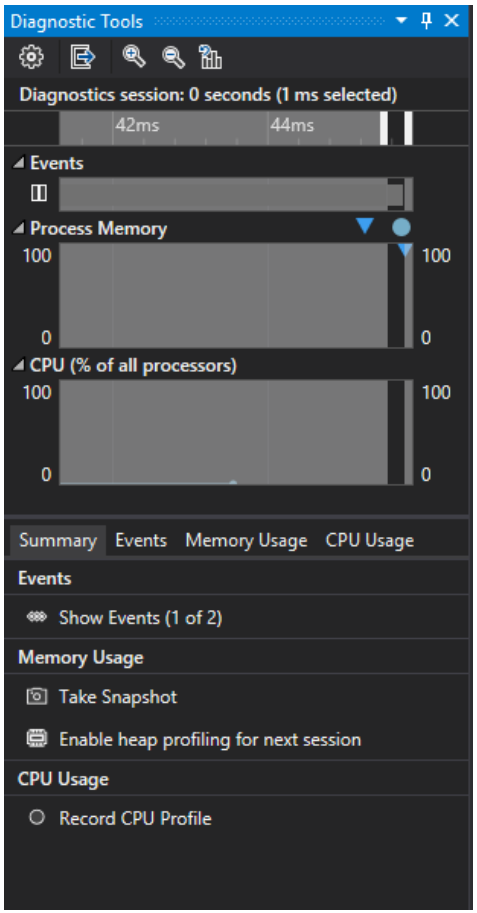
- Edit and continue
- pozastavený program lze za určitých okolností modifikovat, překompilovat a nahradit kód za běhu
- nutno přepnout formát debug informace na `/ZI`



Obrázek: Nastavení formátu debug informací v MSVS

Debugging

- diagnostic tools
- heap profiler

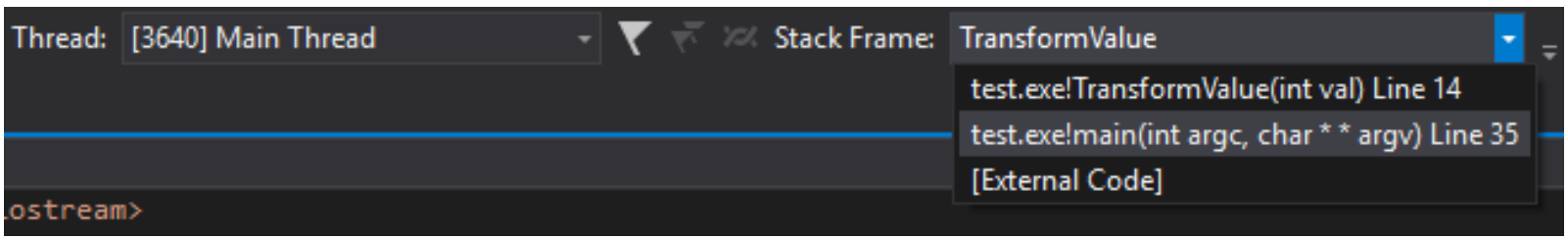


Debugging

- *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

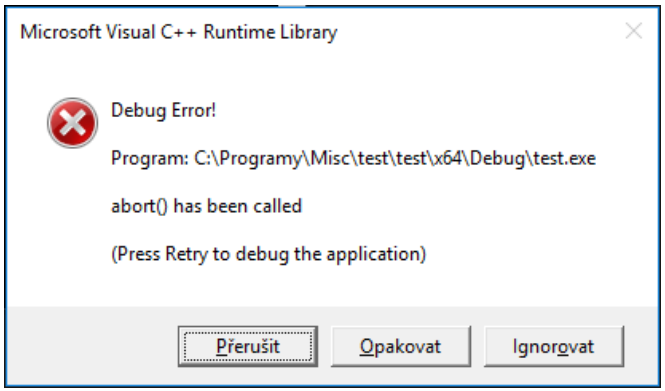
Debugging

- 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



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");
```

Inline assembly

- vložený assembler
- do C/C++ kódu lze vkládat blok assembly
- silně závislé na platformě
- nemusí být dobrý nápad
- uvozen konstrukcí `asm` nebo `_asm` nebo `__asm__`
- specifické pro každý kompilátor
- na x86_64 již povětšinou deprecated
- např. gcc:

```
_asm(
   "movl␣$10,␣%eax;"
   "movl␣$20,␣%ebx;"
   "addl␣%ebx,␣%eax;"
);
```

Inline assembly

- rozšířený vložený assembler
- např. v gcc
- lze specifikovat vstupy a výstupy
- lze dodefinovat použité registry (aby je kompilátor nepoužil pro nic jiného)

```
_asm(
   "...kod␣v␣asm..."
      : vystupni operandy
      : vstupni operandy
      : seznam registru
);
```

Inline assembly

```
int a = 100;
int out;

asm ("movl␣%1,␣%%ebx;"
     "movl␣%%ebx,␣%0;"
   : "=r" (out)
   : "r" (a)
   : "%ebx"
);
```

Intrinsics

- namísto vloženého assembleru se častěji vyplatí použít wrappery
- wrapper pro instrukce - intrinsic function
- volí instrukci dle platformy
- někdy fallback na SW implementaci
- MSVS: `#include <intrin.h>`

```
int a = 0b00001101;

std::cout << __popcnt(a) << std::endl; // 3
```

- např. popcnt - počet jedniček v binární formě

Intrinsics

- hodí se např. pro KIV/PPR, ale i jiné
  - `__popcnt` - počet jedniček v bin. formě
  - `_BitScanForward` - počet leading nul
  - `_inbyte`, `_outbyte` - čte z/zapisuje do I/O portu
  - `_rotl`, `_rotr` - rotace operandu
  - `__rdtsc` - čte hodnotu časovače
  - `_bittest` - je nastaven bit na pozici?
  - a další...
  - `https://msdn.microsoft.com/en-us/library/w5405h95.aspx`