C++ support

Část 11 - 14: CMake, GDB, Valgrind, Perf

Kennny

srpen 2017

CMake úvod

- nástroj pro dynamické generování konfigurací k sestavení
- generuje MSVS solutions, makefile, Codeblocks projekty, atd...
- vždy CLI utilita
- nyní i GUI

CMakeLists.txt

- root soubor
- zpravidla obsahuje definici min. verze

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.4)
```

- vždy obsahuje název celého projektu (workspace, solution)

```
PROJECT(my-fancy-cpp-app)
```

- pro přehlednost je možné, aby podsložky obsahovaly také CMakeLists.txt, který bude načten odděleně; vyvoláme ho přidáním podsložky

```
ADD_SUBDIRECTORY(src)
```

- nic z toho zatím nemá vliv na strukturu projektu

Spustitelný soubor

- přidání spustitelného souboru k sestavení (resp. projektu pro MSVS v rámci solution)

```
ADD_EXECUTABLE(<target-name> <file list...>)
```

- např.

```
ADD_EXECUTABLE(my-executable main.cpp module.cpp othermodule.cpp)
```

- toto vybuildí soubor `my-executable` (resp. `my-executable.exe` na Windows)
- má další možné parametry, pro základní pochopení nejsou nutné

Knihovny

- přidání knihovny k sestavení (resp. projektu pro MSVS v rámci solution) probíhá analogicky

```
ADD_LIBRARY(<target-name> [library type] <file list...>)
```

- kde `[library type]` může být např.:
  - `SHARED` - dynamicky linkovaná knihovna (.dll nebo .so)
  - `STATIC` - staticky linkovaná knihovna (.lib nebo .a)
- např.

```
ADD_LIBRARY(my-toolset STATIC libmain.cpp a.cpp b.cpp)
```

- toto vybuildí soubor `my-toolset.a` (resp. `my-executable.lib` na Windows)

Include cesty

- include cesty se dají přidat globálně / pro jeden cíl

```
INCLUDE_DIRECTORIES(<dirs>)
TARGET_INCLUDE_DIRECTORIES(<target> <specifier> <dirs>)
```

- hodnota `specifier` může být
  - `INTERFACE`
  - `PUBLIC` - vystačíme si s touto
  - `PRIVATE`
- např. pro přidání složky `include`

```
INCLUDE_DIRECTORIES(${INCLUDE_DIRECTORIES} include/)
TARGET_INCLUDE_DIRECTORIES(my-executable PUBLIC
    ${INCLUDE_DIRECTORIES} include/)
```

Linkování knihoven

- linkování knihoven taktéž globálně / pro jeden cíl
- pozor - globálně je třeba před přidáním cílů

```
LINK_LIBRARIES(<libraries>)
TARGET_LINK_LIBRARIES(<target> <libraries>)
```

- např.

```
LINK_LIBRARIES(my-toolset)
TARGET_LINK_LIBRARIES(my-executable my-toolset)
```

Hledání externích knihoven

- je potřeba modul CMake
- nějaké má CMake přibalené s sebou
- modul je možno vyvolat

```
FIND_PACKAGE(<packagename> [flag])
```

- `flag` nás bude zajímat jen jedna: `REQUIRED` - nelze bez této knihovny pokračovat
- např.

```
FIND_PACKAGE(OpenSSL REQUIRED)
```

- modul nastavuje proměnné
- ostatní silně specifické

Hledání externích knihoven

- interní moduly CMake většinou nastavují nějak standardizované proměnné
- vždy `<packagename>_FOUND` pokud se knihovnu podaří nalézt
- často `<packagename>_INCLUDE_DIR` - kde hledat hlavičkové soubory
- často `<packagename>_LIBRARIES` - seznam knihoven k linkování
- všechny moduly, které s sebou CMake nese, jsou zdokumentovány: `https://cmake.org/cmake/help/latest/manual/cmake-modules.7.html`

Parametrizace překladu

- často je potřeba předat nějaký přepínač pro parametrizaci sestavení
- opět lze globálně i pro target

```
ADD_DEFINITIONS(<defines>)
TARGET_COMPILE_DEFINITIONS(<target> <specifier> <defines>)
```

- pozor, defines musí být včetně prefixu `-D` pro preprocesorové direktivy (aby se chovaly jako `#define`)
- např.

```
ADD_DEFINITIONS(-DDEV_BUILD)
TARGET_COMPILE_DEFINITIONS(my-executable PUBLIC -DDEV_BUILD)
```

Vyhledání souborů

- abychom nemuseli otrocky vypisovat seznam souborů, lze je nechat vyhledat
- po vyhledání jsou uloženy do proměnné
- využijeme příkaz `FILE`, který má širší využití

```
FILE(<operation> <parameters>)
```

- pro vyhledání souborů v dané složce s příponou .cpp a .h např.:

```
FILE(GLOB_RECURSE my_app_sources ./ *.cpp *.h)
```

- operace `GLOB_RECURSE` projde i podsložky
- pokud nechceme, lze použít jen `GLOB`

Řídicí struktury

- CMake podporuje všemožné řídicí struktury
- nejjednodušší je `IF`
- syntaxe se myšlenkou podobá podmínkám např. z bashe

```
IF(<expression>)
    # something
ELSE()
    # something else
ENDIF()
```

- výrazem může být pouhá proměnná - pak se ověřuje její existence
- nebo např. porovnávací výraz

Řídicí struktury

- příklad - podařilo se nalézt OpenSSL knihovnu?

```
IF(OpenSSL_FOUND)
    # great!
ENDIF()
```

- příklad - je proměnná nastavena na hodnotu?

```
IF(MY_VARIABLE STREQUAL "windows")
    # great!
ENDIF()
```

- a další, viz `https://cmake.org/cmake/help/latest/command/if.html`

Výstup do konzole

- občas se hodí oznámit něco do konzole

```
MESSAGE([mode] "message")
```

- módy
  - `STATUS` - diagnostický výstup
  - `WARNING`, `AUTHOR_WARNING` - varování, nezastaví provádění
  - `SEND_ERROR` - chyba, nedovolí vygenerovat projekt
  - `FATAL_ERROR` - chyba, okamžitě zastaví provádění
  - `DEPRECATION` - deprecated varování (zastaví, pokud je nastaveno)
- např.

```
MESSAGE(FATAL_ERROR "Could not find OpenSSL!")
```

Příklad

- Prostor pro příklad 11_cmake

Poznámky k memory leak Valgrind příkladům

- byly připraveny příklady na chyby vedoucí k memory leakům, co mohou v C++ vzniknout
- oba zdůrazňují, proč je vhodné používat konstrukce moderního C++ (často založené na RAII)
- v příkladech je návod na sestavení i na spuštění memchecku

Příklad

- Prostor pro příklad 12_a_leak a 12_b_leak

Poznámky k buffer overrun Valgrind příkladu

- byl připraven příklad demonstrující chybu vedoucí k přesažení bufferu (pole, vyhrazené paměti, ..)
- opět zdůrazňuje důležitost např. STL containerů
- v příkladech je návod na sestavení i na spuštění sgchecku

Příklad

- Prostor pro příklad 12_c_range

GDB

- zaměříme se na použití pro diagnostiku segfaultů
- jde samozřejmě o plnohodnotný debugger se spoustou funkcí

Použití

- lze spustit program rovnou pod GDB

```
gdb my-program
```

- to spustí prostředí GDB
- program lze spustit příkazem `run`
- zastavit lze stisknutím ctrl-C
- popř. breakpointem nebo signálem od systému (třeba segfault)

Core dump

- nebo lze dovolit generování tzv. core dumpů

```
ulimit -c <size>
```

- `size` je specifikátor velikosti nebo `unlimited`
- core dumpy jsou 1:1 dumpy paměti s diagnostickými informacemi, které umí GDB interpretovat
- po pádu programu se v konzoli objeví hláška `core dumped` (resp. `obraz paměti uložen`
- lze vyvolat

```
gdb -c <coredump> <program>
```

- výchozí jméno je `core` a ukládá se k aplikaci

Příkazy

- `run` - spuštění programu (od začátku)
- `break [where]` - nastavení breakpointu na místo (např. main.cpp:10)
- `continue` - pokračování po zastavení
- `backtrace [full]` - výstup zanoření v aktuálním vlákně (full = včetně všech kontextových informací)
- `frame [num]` - přesun do jiného frame zanoření v aktuálním kontextu
- `info threads` - informace o běžících vláknech
- `thread [num]` - přesun do kontextu jiného vlákna
- `print [symbol]` - vytištění symbolu (syntaxe jako v kódu)
- `quit` - ukončení GDB
- a samozřejmě spousty, spousty dalších, toto je jen základní sada

Příkazy

- my budeme navíc potřebovat ještě
- `thread apply [thread list / all] command` - provedení příkazu nad více vlákny
- nejčastěji jako:

```
thread apply all backtrace full
```

Příklad

- Prostor pro příklady 13_a_segfault, 13_b_segfault a 13_c_deadlock

Perf

- diagnostický subsystém
- mj. použitelný i jako profiler
- integrovaný do jádra linuxu (perf events)
- neinvazivní metoda diagnostiky aplikace
- bohužel nutný superuser (`root`)
- na Debian-based distribucích balíček `linux-tools`

Práce s perfem

- lze buď aplikaci spustit pod perfem

```
perf record ./aplikace
```

- nebo se lze připojit k již běžícímu procesu podle PIDu

```
perf record -p 1234
```

- je vhodné připojit přepínač `-g` pro zaznamenání hierarchie volání

Eventy

- perf zaznamenává vzorek (zanoření, aktuální adresa) v momentě, kdy přijde tzv. event
- event je propagován interní cestou přes jádro (perf events subsystém), HW eventy navíc pomocí přerušení
- defaultně se zaznamenávají cykly CPU (event `cycles`)
- lze ale profilovat i počet cache-miss, syscallů, instrukcí, branch-miss a mnoho dalšího (celý seznam `perf list`)
- přepínač `-e <event>`

Prohlížení

- perf ukládá soubor `perf.data` do aktuálního adresáře
- v něm jsou zaznamenány i cesty k souborům s kódem, atd. - není proto potřeba být v žádném konkrétním adresáři
- data lze prohlédnout příkazem

```
perf report
```

- popř. formou anotovaného kódu

```
perf annotate
```

- nebo PIVO `https://github.com/ProjectPIVO` :)

Příklad

- Prostor pro příklad 14_a_slow

Závěr

- CMake, GDB, Valgrind a perf jsou obrovské nástroje
- probrali jsme přehled základních prvků

Konec 11., 12., 13. a 14. části

```
std::cout « "Děkuji za pozornost" « std::endl;
exit(0);
```