

# Doporučené postupy

Karel Richta a kol.

katedra počítačů FEL ČVUT v Praze

© Karel Richta, 2020

Programování v C++, B6B36PJC

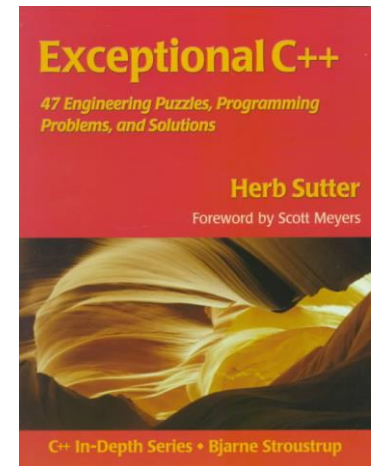
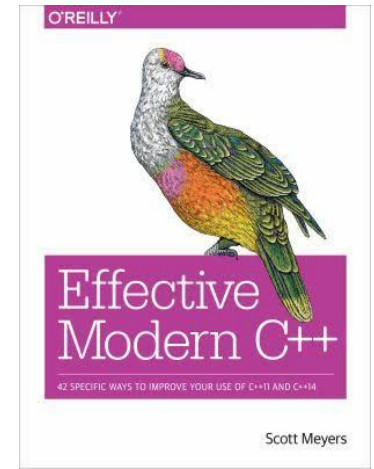
12/2020, Lekce 13

<https://cw.fel.cvut.cz/wiki/courses/b6b36pjc/start>



# Kde najít doporučené postupy?

- Knihy, weby, videa, konference
- Effective Modern C++ od Scotta Meyerse
  - Popřípadě starší verze: (More) Effective C++
- Exceptional C++ od Herba Suttera
  - A navazující More Exceptional C++
  - Jedná se o souhrn „Guru of the Week“ článků
- CppCon
  - Pravidelná konference o C++
  - Mix více i méně pokročilých témat



# Používejte kvalitní nástroje

- Moderní kompilátor s podporou C++11/14
  - Clang, g++, VS2015
- ClangFormat
  - Po nastavení umožňuje automaticky formátovat kód
- ClangTidy
  - Umožňuje hledat a převádět zastaralé konstrukce.
  - Umožňuje hledat podezřelé konstrukce
- Nástroje pro statickou analýzu
  - Clang scan-build
  - PVS-Studio
  - CppCheck
  - ...

# Nástroje pro dynamickou analýzu

- Sanitizéry

- Sada nástrojů pro tzv. dynamickou analýzu, skvělé pro hledání chyb.
- Umožňuje hledat chyby v alokacích, přístupu k paměti, použití vláken, přetékání intů a další.
- Bohužel zatím pouze na Linuxu a OS X.

- Valgrind

- Znáte z upload systému
- Obsahuje i pár dalších nástrojů, například Helgrind (vlákna), Cachegrind (cache profiler)

# Address Sanitizer

- Skvělý nástroj k hledání chyb při práci s pamětí.
  - Memory leaky
  - Use after free
  - Out of bounds access
- Běh programu je ~2x pomalejší.
- Funguje na Linuxu a OS X
- Windows má jiné alternativy

```
int main() {  
    int* p = new int;  
    p = nullptr;  
    return 0;  
}
```

memory-leak.cpp

# Address Sanitizer (výstup)

```
clang++ -fsanitize=address -std=c++11 -g memory-leak.cpp
```

```
horenmar@kepler ~ $ ./a.out
```

```
=====  
=====
```

```
==15307==ERROR: LeakSanitizer: detected memory leaks
```

```
Direct leak of 4 byte(s) in 1 object(s) allocated from:
```

```
  #0 0x4df160 in operator new(unsigned long)  
/var/tmp/portage/sys-devel/llvm-3.7.0-r5/work/llvm-  
3.7.0.src/projects/compiler-rt/lib/asan/asan_new_delete.cc:62
```

```
  #1 0x4e1d9a in main /home/horenmar/memory-leak.cpp:2:14
```

```
  #2 0x7fa617dc261f in __libc_start_main  
(/lib64/libc.so.6+0x2061f)
```

```
SUMMARY: AddressSanitizer: 4 byte(s) leaked in 1  
allocation(s).
```

# Memory Sanitizer

- Najde například použití neinicializované paměti.
- Běh programu je ~3x pomalejší

```
int main(int argc, char** argv) {  
    int x[10];  
    x[0] = 1;  
    if (x[argc]) return 1;  
    return 0;  
}
```

uninit-val.cpp

# Memory Sanitizer (výstup)

```
clang++ -fsanitize=memory -fsanitize-memory-track-origins -g
horenmar@kepler ~ $ ./a.out
=====
====
==10199==WARNING: MemorySanitizer: use-of-uninitialized-value
    #0 0x7ff4b7893407 in main /home/horenmar/uninit-
val.cpp:4:9
    #1 0x7ff4b641461f in __libc_start_main
(/lib64/libc.so.6+0x2061f)
    #2 0x7ff4b7813c38 in _start
(/home/horenmar/a.out+0x19c38)

Uninitialized value was created by an allocation of 'x' in
the stack frame of function 'main'
    #0 0x7ff4b7892fc0 in main /home/horenmar/uninit-val.cpp:1

SUMMARY: MemorySanitizer: use-of-uninitialized-value
/home/horenmar/uninit-val.cpp:4:9 in main
Exiting
```



# Undefined Behavior Sanitizer

- Najde použití nedefinovaného chování.
- Běh programu je triviálně pomalejší.

```
#include <iostream>

enum color { RED, GREEN, BLUE };

int main() {
    color c = RED;
    c = static_cast<color>(6);
    std::cout << c << '\n';
}
```

undefined.cpp

```
clang++ -fsanitize=undefined
-g undefined.cpp
```

```
horenmar@kepler ~ $ ./a.out
```

```
undefined.cpp:12:18: runtime error: load of value 6, which is
not a valid value for type 'color'
```

# Undefined Behavior Sanitizer

- Najde použití nedefinovaného chování.
- Běh programu je triviálně pomalejší.

```
#include <iostream>
#include <cstring>
```

misaligned.cpp

```
int main() {
    int* mem = new int[1000];
    memset(mem, 123, 1000 * 4);
    int* mem2 = (int*)((char*)mem + 7);
    std::cout << *mem2 << '\n';
}
```

```
clang++ -fsanitize=undefined -g misaligned.cpp
```

```
horenmar@kepler ~ $ ./a.out
```

```
misaligned.cpp:8:18: runtime error: load of misaligned
address 0x000002f18c21 for type 'int', which requires 4 byte
alignment
```

# Thread Sanitizer

- Hledá tzv. data races – nesynchronizované přístupy k paměti.
- 10x pomalejší běh

```
clang++ -fsanitize=thread -g -lpthread -std=c++14 race.cpp
```

```
horenmar@kepler ~ $ ./a.out
```

# Thread Sanitizer

race.cpp

```
#include <iostream>
#include <thread>

int main() {
    int counter = 0;
    auto thread_func = [&counter]() {
        for (int i = 0; i < 1'000'000; ++i) {
            counter++;
            counter--;
        }
    };

    auto t1 = std::thread(thread_func);
    auto t2 = std::thread(thread_func);

    t1.join(); t2.join();
}
```

```
clang++ -fsanitize=thread -g -lpthread -std=c++14 race.cpp
```

```
horenmar@kepler ~ $ ./a.out
```

# Thread Sanitizer

=====

**WARNING: ThreadSanitizer: data race (pid=12837)**

Write of size 4 at 0x7ffc600a20cc by thread T2:

#0 operator() /home/pjc/scratch/race.cpp:8 (a.out+0x0000004a6715)

...

Previous write of size 4 at 0x7ffc600a20cc by thread T1:

#0 operator() /home/pjc/scratch/race.cpp:8 (a.out+0x0000004a6715)

...

Location is stack of main thread.

Thread T2 (tid=12840, running) created by main thread at:

#0 pthread\_create <null> (a.out+0x000000422e46)

#1 std::thread::\_M\_start\_thread(std::shared\_ptr<std::thread::\_Impl\_base>, void (\*)()) <null> (libstdc++.so.6+0x0000000b8db2)

#2 main /home/pjc/scratch/race.cpp:14 (a.out+0x0000004a4f3d)

Thread T1 (tid=12839, running) created by main thread at:

#0 pthread\_create <null> (a.out+0x000000422e46)

#1 std::thread::\_M\_start\_thread(std::shared\_ptr<std::thread::\_Impl\_base>, void (\*)()) <null> (libstdc++.so.6+0x0000000b8db2)

#2 main /home/pjc/scratch/race.cpp:13 (a.out+0x0000004a4f30)

SUMMARY: ThreadSanitizer: data race /home/pjc/scratch/race.cpp:8 in operator()

=====

# Valgrind

- Narozdíl od sanitizérů funguje i na předkompilovaných binárkách.
- 100x pomalejší běh, než normální kód.
- Kontroluje během běhu spoustu věcí
  - Ztracenou paměť
  - Vícenásobný delete
  - Zápis mimo alokovanou paměť
  - Čtení z nealokované paměti
  - ...

# ClangTidy (modernize)

- Umožňuje automaticky modernizovat různé konstrukty.
- Složitější k používání.

```
const int N = 5;
int arr[] = { 1,2,3,4,5 };

// safe conversion
for (int i = 0; i < N; ++i) {
    std::cout << arr[i];
}
```

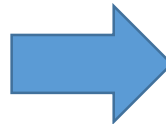


```
const int N = 5;
int arr[] = { 1,2,3,4,5 };

// safe conversion
for (auto & elem : arr) {
    std::cout << elem;
}
```

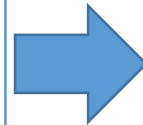
```
std::vector<int> v = {1, 2, 3};
```

```
// reasonable conversion
for (int i = 0; i < v.size(); ++i){
    std::cout << v[i];
}
```



```
// reasonable conversion
for (auto & elem : v) {
    std::cout << elem;
}
```

```
// reasonable conversion
for (std::vector<int>::iterator it =
    v.begin(); it != v.end(); ++it) {
    std::cout << *it;
}
```



```
// reasonable conversion
for (auto & elem : v) {
    std::cout << elem;
}
```

# Jak nejlépe spravovat prostředky?

- Velmi často narazíme na kód, který používá ukazatele k vlastnění alokované paměti (a jiných prostředků).
- Ukazatele přinášejí řadu problémů:
  - Není zřejmé, jestli máme ukazatel smazat, až dokončíme práci (nebo jestli to má na starost někdo jiný).
  - Není zřejmé, jestli se jedná o jeden objekt nebo pole objektů. Nevíme, zda použít `delete` nebo `delete [ ]`.
  - Není zřejmé, zda ukazatel vůbec někam ukazuje. Může být neinicializovaný, příp. objekt, na který ukazuje, už může být smazaný.
  - Není zřejmé, zda jsme ukazatel smazali vždy, když opouštíme naši funkci. Když někdo mezi `new` a `delete` zavolá `return`, objekt nebude smazán. To samé platí, když někdo vyhodí vyjímku.



# Problémy s ukazateli

```
int main() {  
    GraphNode* gn = createGraph();  
  
    // Co teď? Jak se vypořádat s gn?  
    // Záleží, co je uvnitř createGraph()...  
}
```

```
GraphNode* createGraph() {  
    GraphNode* nodes = new GraphNode[20];  
    // ...  
    return nodes;  
}
```

delete[] gn;

```
GraphNode* createGraph() {  
    GraphNode* node = new GraphNode;  
    // ...  
    return node;  
}
```

delete gn;

```
GraphNode* createGraph() {  
    static GraphNode nodes[20];  
    // ...  
    return nodes;  
}
```

// nic!

# Alternativy k vlastním ukazatelům

- Ke správě prostředků je radno využívat RAII.

- `std::vector`

```
std::vector<GraphNode> createGraph() {  
    std::vector<GraphNode> nodes;  
    // ...  
    return nodes;  
}
```

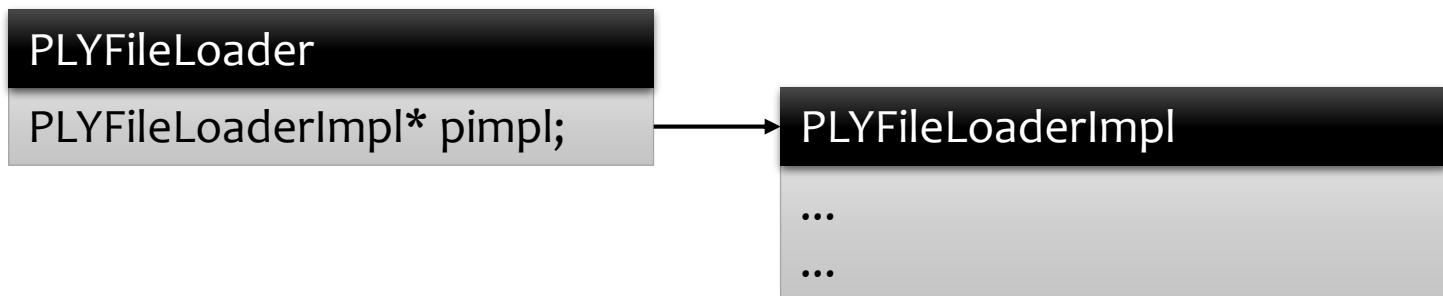
- `std::unique_ptr`

```
std::unique_ptr<GraphNode> createGraph() {  
    auto node = std::make_unique<GraphNode>();  
    // ...  
    return node;  
}
```

- ...a další (`std::shared_ptr`, `std::weak_ptr`).
- Nyní se o správný úklid prostředků postarají destruktory.

# Jak zrychlit kompilaci velkých projektů?

- U obrovských projektů musíme minimalizovat množství kódu v hlavičkových souborech. Oblíbený způsob, jak to provést, je tzv. *pimpl* (pointer to implementation).
- Každou rozsáhlejší třídu rozdělíme na dvě: rozhraní a implementaci (data). Implementační část se nebude v hlavičkových souborech vyskytovat, budeme pouze používat její ukazatel.
- Výsledkem je, že bude v hlavičkových souborech méně příkazů `#include` a změny v našem kódu budou způsobovat kompilaci méně CPP souborů.



# Pimpl – předtím

```
#ifndef PLY_FILE_LOADER_HPP
#define PLY_FILE_LOADER_HPP

#include <string>
#include <vector>
#include <fstream>
#include "encoding.hpp"
#include "buffer.hpp"

struct Vertex { float x, y, z; };
struct Triangle { uint32_t u, v, w; };

struct PlyFileInfo {
    std::string filename;
    FileEncoding encoding;
    enum class Format {
        UNKNOWN, BINARY, ASCII
    } format = Format::UNKNOWN;
    std::size_t numVerts = 0, numTris = 0;
};

struct PlyFile {
    std::ifstream in;
    InputBuffer buffer;
};

struct PlyData {
    std::vector<Vertex> vertices;
    std::vector<Triangle> triangles;
};
```

```
class PlyFileLoader {
public:
    ~PlyFileLoader();
    PlyFileLoader(const std::string& filename);
    PlyFileLoader(PlyFileLoader&&);
    PlyFileLoader& operator=(PlyFileLoader&&);
    std::vector<Vertex>& getVertices();
    std::vector<Triangle>& getTriangles();
private:
    PlyFile file;
    PlyFileInfo info;
    PlyData data;
};

#endif
```

# Pimpl – potom

```
#ifndef PLY_FILE_LOADER_HPP
#define PLY_FILE_LOADER_HPP

#include <string>
#include <vector>
#include <memory>

struct Vertex { float x, y, z; };
struct Triangle { uint32_t u, v, w; };

class PlyFileLoader {
public:
    ~PlyFileLoader();
    PlyFileLoader(const std::string& filename);
    PlyFileLoader(PlyFileLoader&&);
    PlyFileLoader& operator=(PlyFileLoader&&);
    std::vector<Vertex>& getVertices();
    std::vector<Triangle>& getTriangles();
private:
    struct Impl;
    std::unique_ptr<Impl> pimpl;
};

#endif
```

"encoding.hpp" a  
"buffer.hpp" se přestanou  
šířit projektem skrze tento  
hlavičkový soubor

také je teď v tomto  
souboru méně deklarací,  
které jsou nepotřebné pro  
ostatní součásti programu

```
// někde v CPP souboru...
struct PlyFileLoader::Impl {
    PlyFile file;
    PlyFileInfo info;
    PlyData data;
};
```

# Jak zrychlit běh programu?

Existuje řada technik, jak program zrychlit...

- Odstranění zbytečných alokací a dealokací paměti
- Paralelní běh (`std::async`)
- Paralelismus na úrovni instrukcí (instrukční sada SSE, AVX)
- Eliminace virtuálních volání
- Agresivní inlining (`__forceinline`)
- Implementace v kódu nižší úrovně (assembly)

# Jak zrychlit běh programu?

Existuje řada technik, jak program zrychlit...

- ~~Odstranění zbytečných alokací a dealokací paměti~~
- ~~Paralelní běh (`std::async`)~~
- ~~Paralelismus na úrovni instrukcí (instrukční sada SSE, AVX)~~
- ~~Eliminace virtuálních volání~~
- ~~Agresivní inlining (`__forceinline`)~~
- ~~Implementace v kódu nižší úrovně (assembly)~~

Jenže!

- Každá z těchto technik může výkonu naopak ublížit.
- **Pokud chceme program doopravdy zrychlit, musíme umět výkon změřit.**

# Jak změřit výkon?

- Můžeme získat i hodnotnější informace, než je samotný čas běhu. Nástroje pro tzv. *profiling* nám umožňují nalézt místo v programu, kde trávíme většinu času.
- Pokud aplikace běží pomalu, typicky za to může jen malý kousek programu, tzv. *bottleneck* ("úzké hrdlo"). Profiler jej nalezne a my víme, kde hledat problém.
- Jak získat profiler:
  - Nástroje na profiling jsou součástí Visual Studia.
  - Na Windows: Very Sleepy, UIforETW.
  - Na Linux a Mac: Perf.



# Jak doopravdy zrychlit běh programu?

- Profiluj!
- Odstraň zbytečné alokace a dealokace paměti
- Profiluj!
- Použij paralelní běh (`std::async`)
- Profiluj!
- Použij paralelismus na úrovni instrukcí (instrukční sada SSE, AVX)
- Profiluj!
- Eliminuj virtuální volání
- Profiluj!
- Použij agresivní inlining (`__forceinline`)
- Profiluj!

Děkuji za pozornost.