Doporučené postupy

Karel Richta a kol.

katedra počítačů FEL ČVUT v Praze

Přednášky byly připraveny s pomocí materiálů, které vyrobili Ladislav Vágner, Pavel Strnad, Martin Hořeňovský, Aleš Hrabalík a Martin Mazanec

Programování v C++, A7B36PJC

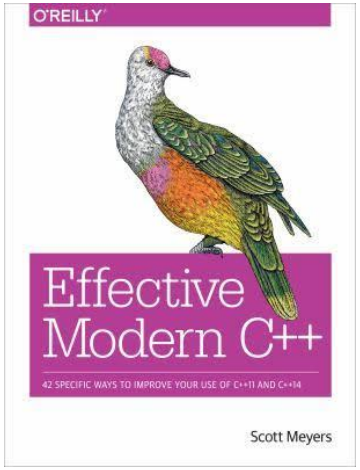
09/2015, Lekce 12

https://cw.fel.cvut.cz/wiki/courses/a7b36pjc/start

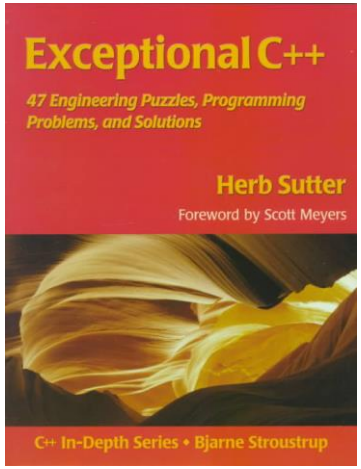
Kde najít doporučené postupy?

- 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

cppcon
the c++ conference

Používejte kvalitní nástroje

- Moderní kompilátor s podporou C++11/14
  - Clang, g++, VS2015
- ClangFormat
  - Po nastavení umožňuje automaticky formátovat kód
- ClangTidy
  - Umožňuje automaticky hledat a převádět zastaralé konstrukce.
- 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.

Address Sanitizer

- Skvělý nástroj k hledání chyb při práci s pamětí.
- Funguje na Linuxu a OS X
- Windows má jiné alternativy

memory-leak.cpp

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

```
clang++ -fsanitize=address -std=c++11 -g -fno-omit-frame-pointer \
        memory-leak.cpp
```

```
martin@hmkii:~/tests$ ./a.out

=================================================================
==15887==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 4 byte(s) in 1 object(s) allocated from:
    #0 0x4dc382 in operator new(unsigned long)
(/home/xarn/tests/a.out+0x4dc382)
    #1 0x4dd2b9 in main /home/xarn/tests/memory-leak.cpp:2:11
    #2 0x7fa274c08ec4 in __libc_start_main /build/buildd/eglibc-2.19/csu/libc-
start.c:287

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

ClangTidy

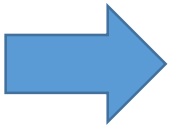
- Umožňuje automaticky modernizovat různé konstrukty.
- Složitější k používání.

```
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;
}
```

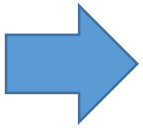
```
std::vector<int> v = {1, 2, 3};
```

```
// 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;
}
```

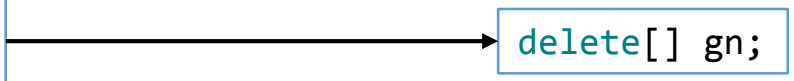
Jak nejlépe spravovat prostředky?

- 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

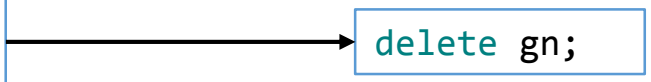
```
int main() {
    GraphNode* gn = createGraph();

    // 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 vlastnícím ukazatelům

- Ke správě prostředků je radno využívat RAII.
- `std::vector`

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

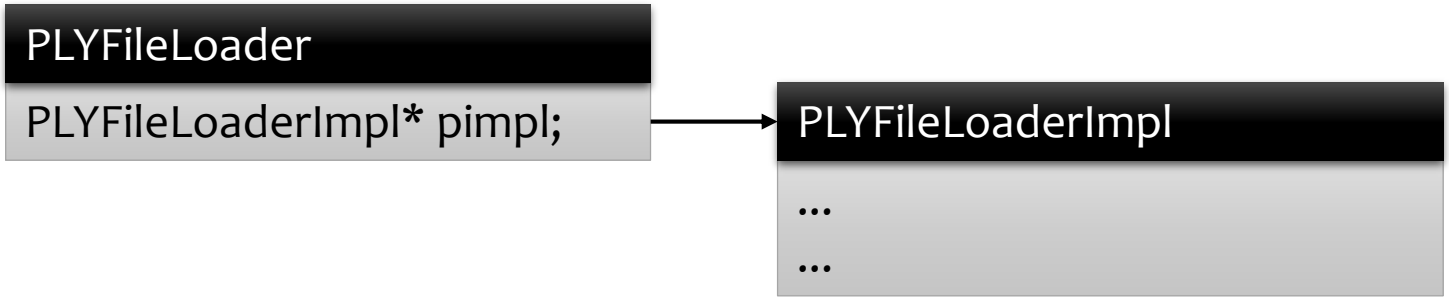
- `std::unique_ptr`

```
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 vekých 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

```cpp
#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

```cpp
// 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.
  - Na Linux a Mac: Rotateright Zoom.

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.