Doporučené postupy

Karel Richta a kol.

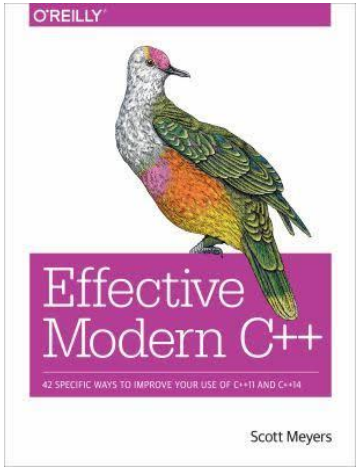
katedra počítačů FEL ČVUT v Praze

Programování v C++, B6B36PJC
01/2019, Lekce 13

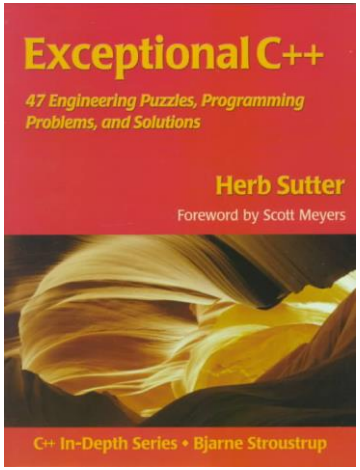
https://cw.fel.cvut.cz/wiki/courses/b6b36pjc/start

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

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 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.

misaligned.cpp

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

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 pamět
  - Čtení z nealokované paměti
  - ...

ClangTidy (modernize)

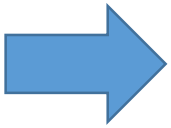
- Umožňuje automaticky modernizovat různé konstrukty.
- Složitější k používání.

```
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;
}
```

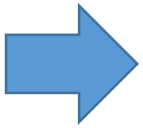
```
std::vector<int> v = {1, 2, 3};
```

```
// 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 vlastnícím ukazatelům

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

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

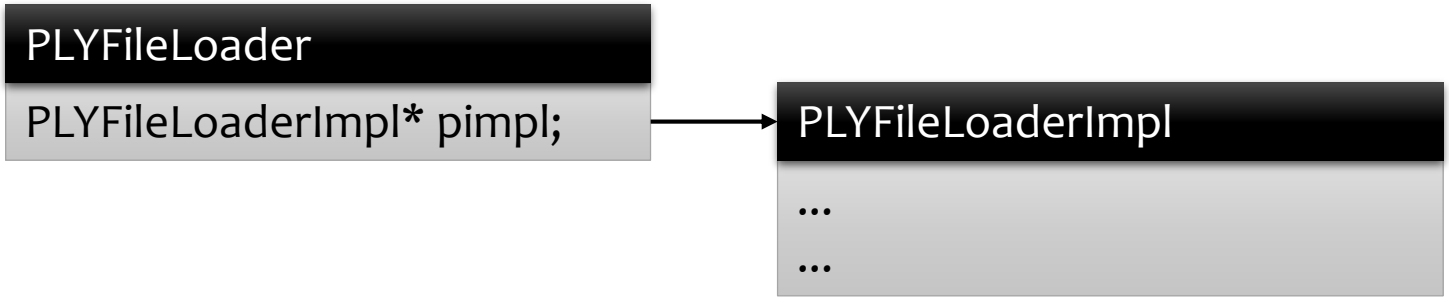
- `std::unique_ptr`

```
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 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

```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, 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!

Jak doopravdy zrychlit běh programu?

- Kromě dříve uvedených technik je důležité implementovat pokud možno nejjednodušší algoritmus pro řešení daného problému.
- Jeho vylepšení na úrovni implementace je pak jen doplňkem tohoto výběru.
- Obecně rozlišujeme časovou a paměťovou složitost – kolik času a paměti k řešení problému potřebujeme.

Příklad: Výpočetní problém

- Problém řazení čísel (Sorting Problem for Numbers).
  - **Předpoklad**: Umíme určit, které číslo je větší (uspořádání $\leq$).
  - **Vstup**: Posloupnost n čísel $Inp = \langle a_1, \ldots, a_n \rangle$.
  - **Výstup**: Taková permutace $Out = \langle a'_1, \ldots, a'_n \rangle$ stejných čísel z Inp, pro kterou platí, že $a'_1 \leq \ldots \leq a'_n$.
- Instance problému řazení je např. zadání: seřaďte vzestupně posloupnost:

  $Inp = \langle 75, 11, 34, 176, 59, 6, 54 \rangle$.

  Správným výstupem řazení je pak posloupnost:

  $Out = \langle 6, 11, 34, 54, 59, 75, 176 \rangle$.

Algoritmus

- Neformálně: algoritmus je jakýkoli dostatečně přesný popis postupu výpočtu – dostatečně přesný znamená:
  - v každém okamžiku postupu víme, který krok bude následovat (postup je deterministický),
  - pro libovolná vstupní data postup vždy skončí (je konečný) a
  - lze jej spustit pro všechny instance problému (je hromadný).
- Mezi algoritmy a výpočetními problémy nemusí být vztah jedna ku jedné – stejný problém lze zpravidla řešit různými algoritmy.
- Algoritmus není program.
  - Program může být realizací algoritmu v určitém výpočetním prostředí.
- Různé algoritmy pro stejný problém mohou mít různou složitost.

Program

- Program *P* je zápis (implementace) algoritmu *A* v nějakém programovacím jazyku, spustitelný (proveditelný) na nějakém počítači s nějakým OS a SW prostředím.
- U návrhu a implementace programu nás zajímají SW inženýrské otázky, jako např.:
  - modularita,
  - ošetření výjimek,
  - datová abstrakce a ošetření vstupu a výstupu,
  - dokumentace.
- Jednomu algoritmu *A* mohou odpovídat různé programy P_1, P_2, . . . dokonce v různých programovacích jazycích.
- Složitosti provedení jednotlivých programů P_i téhož algoritmu se mohou lišit v závislosti na architektuře počítače — nikoli ale řádově.

Jak měřit a určovat složitost algoritmu?

- Analýza složitosti algoritmu: výpočet/odhad/predikce požadavků na výpočetní prostředky, které provedení algoritmu bude vyžadovat v závislosti na velikosti vstupních dat.
- Velikost (složitost) vstupních dat závisí na specifikaci daného problému, může se jednat např. o:
  - počet bitů reprezentace vstupního čísla,
  - délka vstupní posloupnosti čísel,
  - rozměry vstupní matice,
  - počet znaků vstupního textu, . . .
- Pro jeden problém *V* mohou existovat různé algoritmy. Jejich složitosti se mohou výrazně (= řádově) lišit svými nároky na výpočetní prostředky.
- Zvláště se mohou lišit svojí rychlostí (operační složitostí).
- Nebo nároky na použitou paměť (paměťovou složitostí).

Výpočetní (operační) složitost

- Varianty měření operační složitosti:
  - doba běhu algoritmu,
  - doba výpočtu,
  - počet provedených operací,
  - počet provedených instrukcí (aritmetických, logických, paměťových),
  - počet transakcí nad databází.
- Případová analýza složitosti:
  - Typický algoritmus obsahuje podmíněné příkazy nebo cykly $\Rightarrow$ složitost algoritmu může obecně záviset nejen na velikosti vstupních dat, ale také na jejich hodnotách (říkáme pak, že algoritmus je citlivý na vstupní data).
- Proto je obecně potřebné vyjádřit složitost algoritmu:
  - v nejlepším případě,
  - v nejhorším případě,
  - v průměrném případě.

Umění analýzy složitosti algoritmu

- Odhad složitosti v průměrném případě je nejobtížnější, protože není vždy zřejmé, co to jsou vstupní data v "průměrném" případě.
- Typicky náhodně vygenerovaná data, ale to nemusí být v praxi splněno.
- Platí pro časovou, paměťovou, komunikační, transakční,. . . .
- Umění analyzovat složitost algoritmu vyžaduje řadu znalostí a schopností:
  - znalosti odhadu řádu rychlosti růstu funkcí,
  - schopnost abstrakce výpočetního modelu,
  - matematické znalosti z diskrétní matematiky, kombinatoriky,
  - znalosti sčítání řad a řešení rekurentních rovnic,
  - znalosti základu teorie pravděpodobnosti,
  - a další.

RAM model

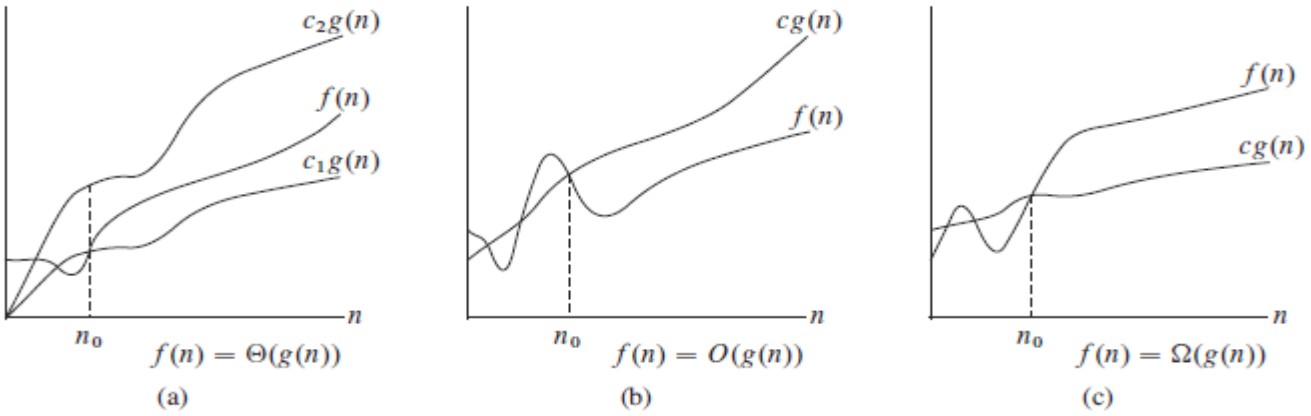
- Pro analýzu složitosti algoritmu potřebujeme model výpočetního systému, na kterém bude algoritmus hypoteticky implementován.
- Nejčastěji se používá matematická abstrakce stroje s přímým přístupem do paměti - jednoprocesorový model RAM (Random Access Machine):
  - data jsou uložena v adresovatelné paměti,
  - stroj umí aritmeticko-logické, řídicí, paměťové instrukce,
  - časová složitost instrukcí je jednotková nebo konstantní,
  - instrukce jsou prováděny postupně (sekvenčně).
- Zpracování každé instrukce trvá přesně jeden krok.
- Neumí manipulace s větším množstvím dat naráz a práci s neomezeně velkými čísly (řetězci), nemá „keše“, disky a vstupně/výstupní zařízení.
- Díky své obecnosti přežil tento model desetiletí.

Asymptotická složitost

- Zajímá nás, jak složitost algoritmu závisí na velkosti vstupních dat v limitě, pokud velikost instance problému poroste nade všechny meze.
- Pro první porozumění chování algoritmu nás nemusejí zajímat ani multiplikativní konstanty - říkáme, že nás zajímá asymptotická složitost.
- Je to složitost vyjádřená pro instance problému dostatečně velké, aby se jasně projevil řád růstu funkce složitosti v závislosti na velikosti vstupních dat.
- Asymptoticky lepší algoritmus bude lepší pro všechny instance problému kromě případného konečného počtu menších instancí.
- Asymptotická notace nám umožnuje zjednodušovat výrazy zanedbáním nepodstatných částí.
- Co znamená zápis $f(n) = 4n^3 + \Theta(n^2)$?
- Výraz $\Theta(n^2)$ zastupuje anonymní funkci z množiny $\Theta(n^2)$ (nějaká kvadratická funkce), jejíž konkrétní podobu prozatím zanedbáváme.

Grafické znázornění pro vysvětlení

- Pokud uvažujeme asymptotickou složitost, zajímá nás zejména chování pro velká vstupní data, pro singulární malá data (menší než n_0) se může chovat trochu jinak. Varianta (a) ukazuje tzv. průměrnou složitost – $f(n) = \Theta(g(n))$ se drží mezi $c_2g(n)$ a $c_1g(n)$ (pro $n \geq n_0$). Varianta (b) ukazuje tzv. asymptotickou horní mez - $f(n) = O(g(n))$ se drží pod $cg(n)$. Naopak varianta (c) ukazuje tzv. asymptotickou dolní mez - $f(n) = \Omega(g(n))$ se drží nad $cg(n)$.



Příklad

- Uvažme problém řazení a dva vývojáře.
- Jeden má rychlý počítač C_1 a navrhl řazení pomocí přímého zatřiďování (INSERTIONSORT) – výstup tvoříme tak, že do prázdné posloupnosti postupně zatřiďujeme jednotlivé prvky vstupní posloupnosti.
- Druhý má pomalý počítač C_2, ale navrhl řazení slučováním (MERGESORT) – vstup rozdělíme na dvě části které (rekurzivně) setřídíme a poté sloučíme do výsledku.
- Lze ukázat, že asymptotická složitost INSERTIONSORT je $\Theta(n^2)$, zatímco složitost MERGESORT je $\Theta(n*\log_2 n)$.
- Pokud budeme třídit 10 milionů (10^7) čísel, počítač C_1 provede 10^{10} instrukcí za vteřinu, počítač C_2 provede 10^7 instrukcí za vteřinu (je 1000x pomalejší), pak:

$$(10^7)^2/(10^{10}) = 10^4 \text{ vteřin} \geq (10^7*\log_2 10^7)/(10^7) = 161 \text{ vteřin}$$

- Tj. rychlejší C_1 počítá téměř 3 hodiny, pomalejší C_2 ani ne 3 minuty.

Srovnání časových složitostí

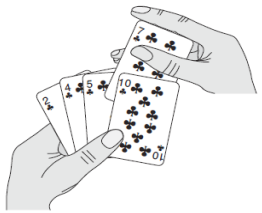
- Předpokládejme, že 1 operace trvá čas 1µs a že počet operací je dán funkcí v prvním sloupci. Pak doba výpočtu bude pro různé hodnoty n následující: (Připomeňme, že počet atomů ve vesmíru se odhaduje na 10^{80} a stáří vesmíru na 14×10^9 let.)

n	10	20	40	80	500	1000
$\log n$	$3,3\ \mu s$	$4,3\mu s$	$5\mu s$	$5,8\mu s$	$9\mu s$	$10\mu s$
n	$10\mu s$	$20\mu s$	$40\mu s$	$60\mu s$	$0,5ms$	$1ms$
$n \log n$	$33\mu s$	$86\mu s$	$0,2ms$	$0,35ms$	$4,5ms$	$10ms$
n^2	$0,1ms$	$0,4ms$	$1,6\ ms$	$3,6ms$	$0,25s$	$1s$
n^3	$1ms$	$8ms$	$64ms$	$0,2s$	$125s$	$17min$
n^4	$10ms$	$160ms$	$2,56s$	$13s$	17 h	11,6 dní
2^n	$1ms$	$1s$	12,7dní	3600 let	10^{137} let	10^{287} let
$n!$	$3,6s$	77100 let	10^{34} let	10^{105} let	10^{1110} let	10^{2554} let

Příklad: Řazení přímým vkládáním (InsertionSort)

Algoritmus: vstup – posloupnost s:

1. Pokud je posloupnost s kratší než 2, skonči, *Out* = *Inp*.
2. Jinak postupně vyber prvek, který je na j-tém místě (j nabývá hodnot od 2 do délky posloupnosti – aktuálně je v čele) a zařaď jej na správné místo v posloupnosti – před nejbližší větší prvek (pokud existuje) v levé části posloupnosti.

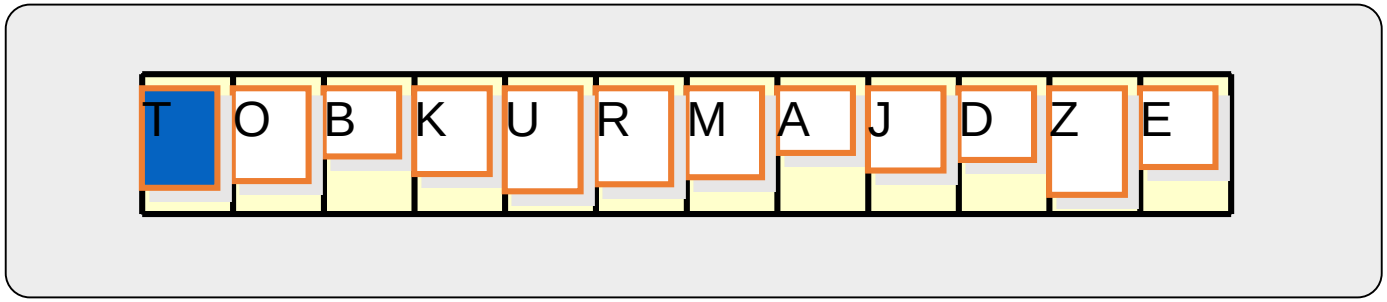


Program (pseudokód):

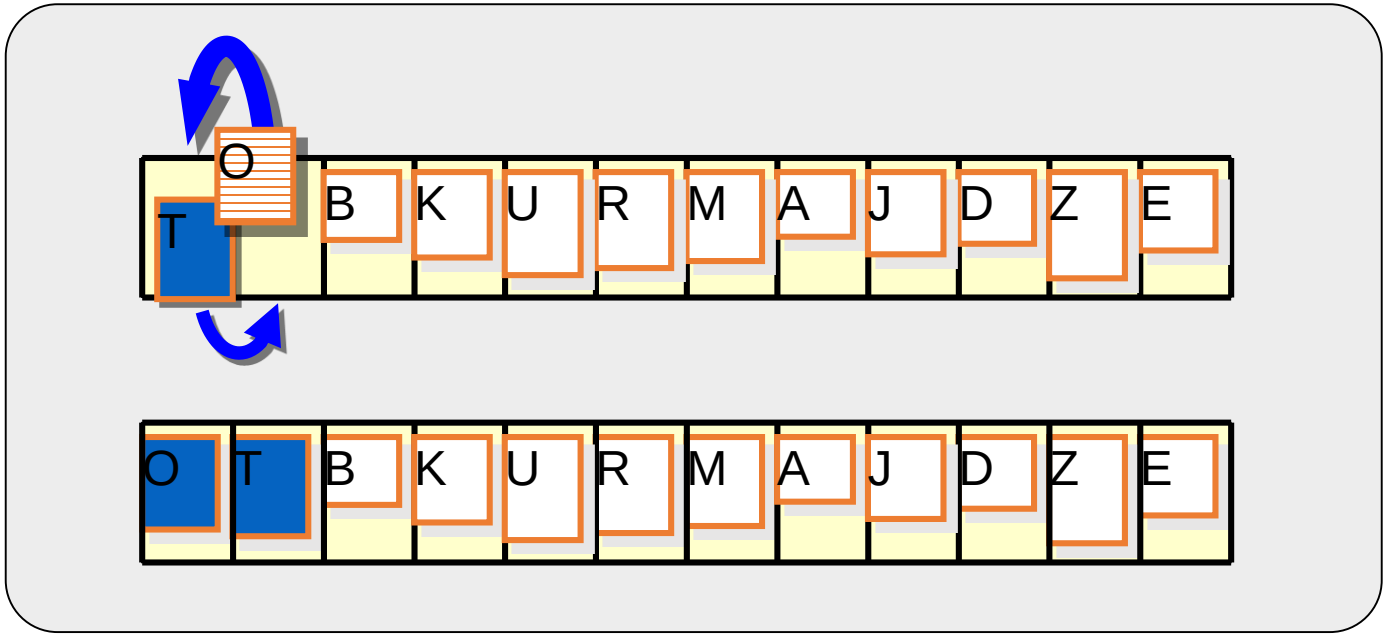
```
for (i = 1; i < n; i++) { // find & make
                          // place for s[i]
  insVal = s[i];
  j = i-1;
  while ((j >= 0) && (s[j] > insVal)) {
    s[j+1] = s[j];
    j--;
  }
  s[j+1] = insVal; // insert s[i]
}
```

InsertionSort

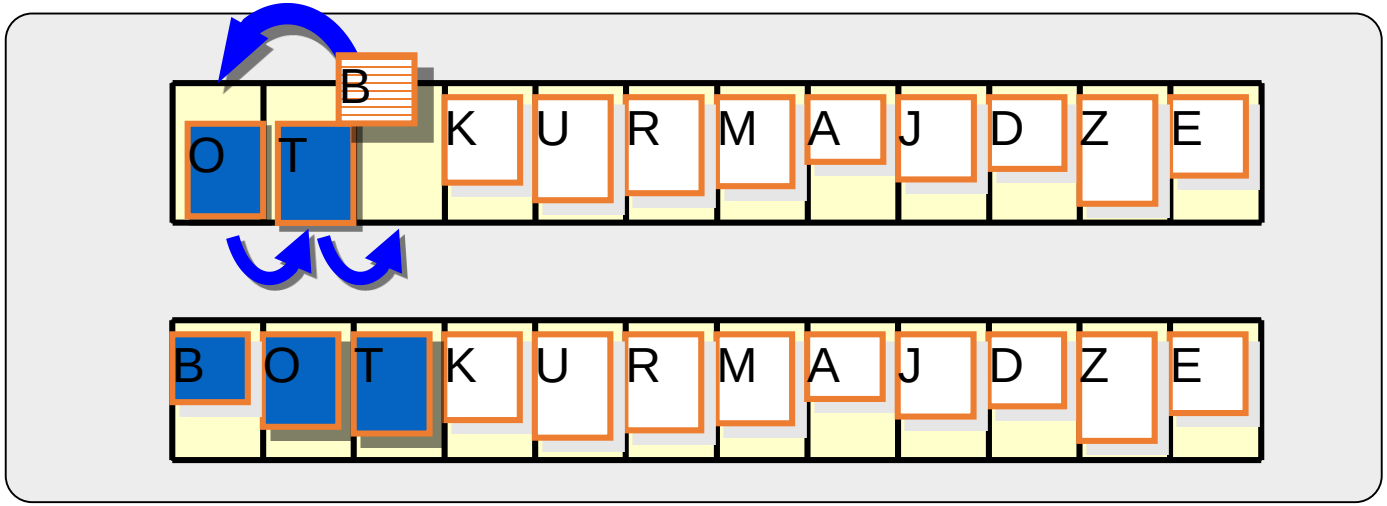
Start



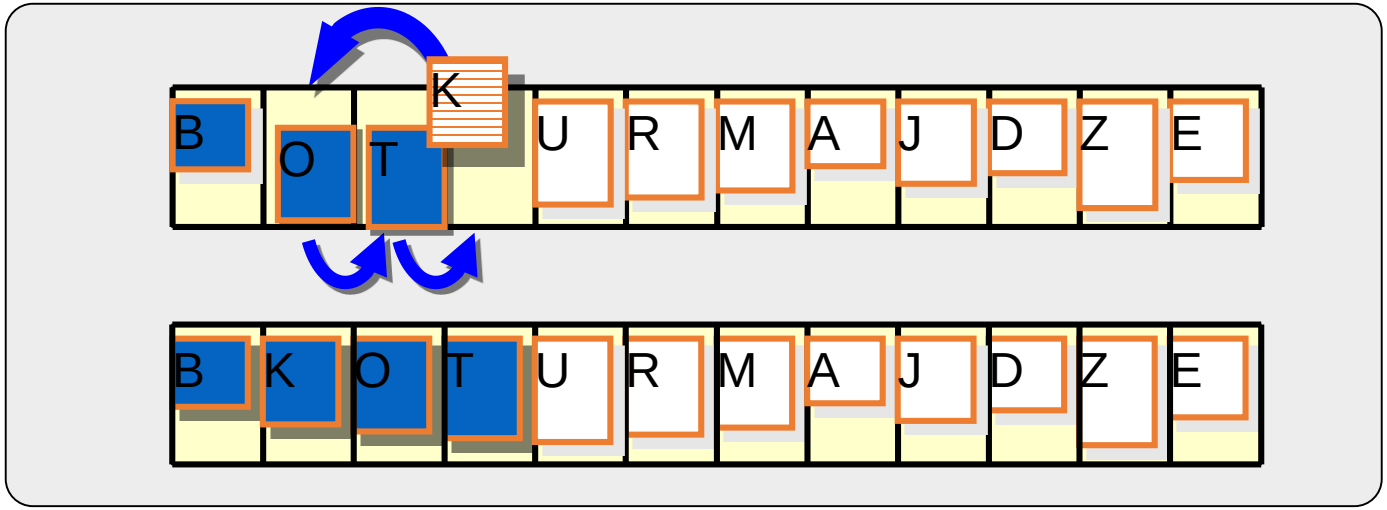
Krok 1

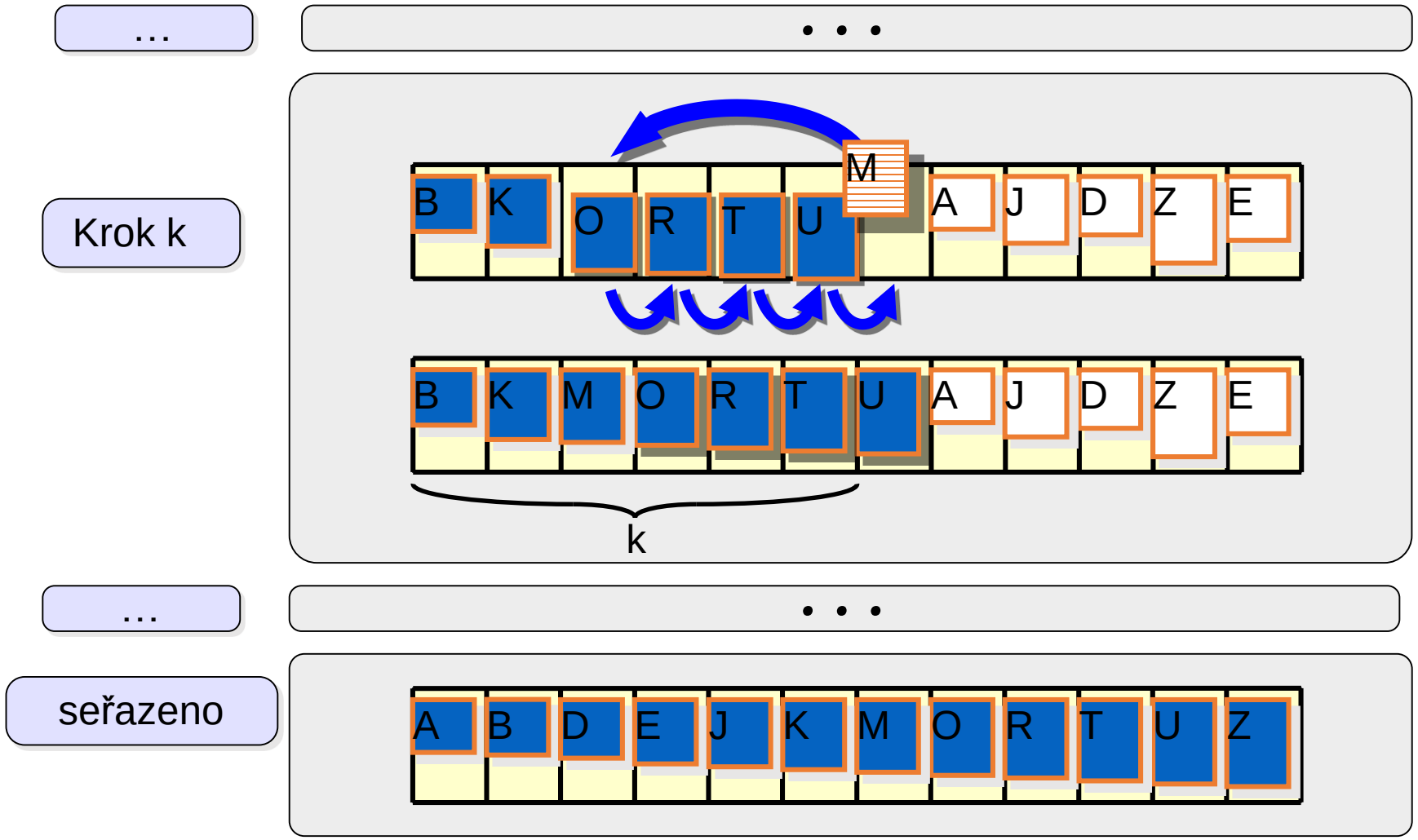


Krok 2



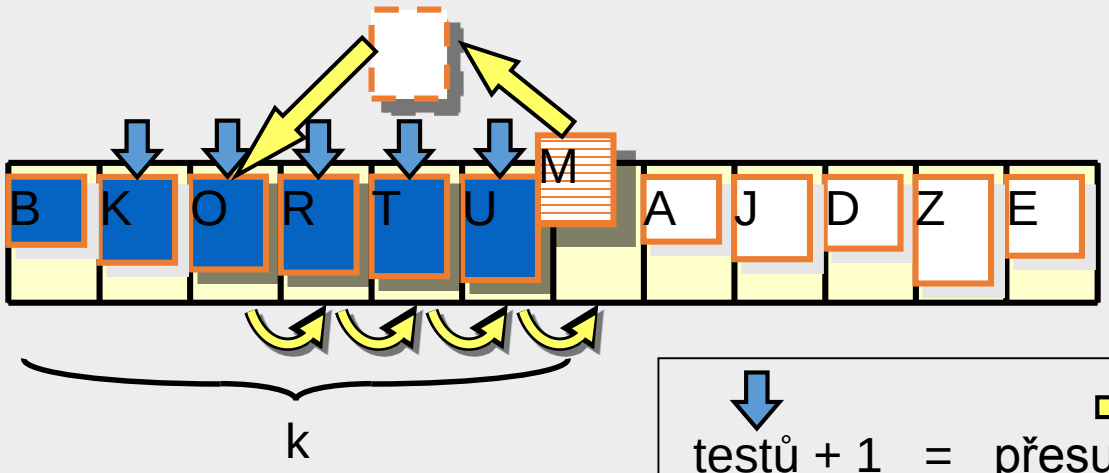
Krok 3




…
• • •
Krok k
M
B K O R T U A J D Z E
B K M O R T U A J D Z E
k
…
• • •
seřazeno
A B D E J K M O R T U Z

Složitost řazení vkládáním (INSERTIONSORT)

Krok k



testů + 1 = přesunů

testů …….	1	nejlepší případ
	k	nejhorší případ
	(k+1)/2	průměrný případ

přesunů …….	2	nejlepší případ
	k+1	nejhorší případ
	(k+3)/2	průměrný případ

Shrnutí

Celkem testů

$n - 1$	$= \Theta(n)$	nejlepší případ
$(n^2 - n)/2$	$= \Theta(n^2)$	nejhorší případ
$(n^2 + n - 2)/4$	$= \Theta(n^2)$	průměrný případ

Celkem přesunů

$2n - 2$	$= \Theta(n)$	nejlepší případ
$(n^2 + n - 2)/2$	$= \Theta(n^2)$	nejhorší případ
$(n^2 + 5n - 6)/4$	$= \Theta(n^2)$	průměrný případ

Asymptotická složitost INSERTIONSORT je $O(n^2)$ (!!)

Děkuji za pozornost.