KIV/CPP – Programování v jazyce C++

12. Překlad vybraných konstrukcí do strojového kódu, optimalizace

Martin Úbl

KIV ZČU

2023/2024

Abstrakce

- C++ je vysokoúrovňový jazyk
- vazba na nízkoúrovňový kód není odstraněna
  - pouze je odstíněna
  - reference
  - STL kontejnery a struktury
  - a další...
- řádově vyšší požadavky na kompilátor za účelem produkování „stejně“ efektivního kódu jako v čistém C
- optimalizace

Abstrakce

- vlastnosti jazyka je třeba převést do nízkoúrovňových rutin (i bez optimalizací)
  - třídy, metody a atributy
  - polymorfismus
  - reference
  - šablony
  - návratové hodnoty
  - ...
- vysokoúrovňové konstrukce lze optimalizovat
  - přístup na prvek array/vektoru
  - STL algoritmy
  - ...

Abstrakce

- třídy, metody a atributy
- nízkoúrovňový kód nezná třídy (ani objekty)
- musíme nějak vyjádřit náležitost metody objektu
- instance třídy (objekt) v nízkoúrovňovém kódu „zdegeneruje“ na instanci struktury
  - z atributů se stanou prvky struktury
- metody jsou transformovány na obyčejné funkce
- jako „nultý parametr“ je typicky předáván ukazatel na strukturu (původně objekt)
  - `this`
  - resp. `thiscall` volací konvence

Abstrakce

```
// C++
class Cislo {
  private:
    int cislo;
  public:
    void setCislo(int c);
    int getCislo();
}

// C
struct Cislo {
  int cislo;
}
void setCislo(Cislo* this, int c);
int getCislo(Cislo* this);
```

Abstrakce

- virtuální metody
- opakování
- nízkoúrovňový kód nezná polymorfismus
- musíme nějak vyjádřit náležitost metody konkrétnímu typu
- tabulka virtuálních metod
  - pole ukazatelů na funkce
  - u objektu jako „nultý“ atribut – ukazatel na tabulku
    - např. MSVS vkládá symbol `__vfptr`

Abstrakce

```
class Cislo {
  protected:
    int cislo;
  public:
    Cislo() : cislo{ 0 } {}
    Cislo(int a) : cislo{ a } {}

    virtual int get() {
      return cislo;
    }
    virtual int getDoubled() {
      return cislo * 2;
    }
};
```

Abstrakce

- virtuální metody
- invokace metod výběrem adresy z `vtable`
- zavolání

```
    a->get();
00007FF77985111D  mov         rax,qword ptr [rsi]
00007FF779851120  mov         rcx,rsi
00007FF779851123  call        qword ptr [rax]
    a->getDoubled();
00007FF779851125  mov         rax,qword ptr [rsi]
00007FF779851128  mov         rcx,rsi
00007FF77985112B  call        qword ptr [rax+8]
```

Obrázek: Třída má dvě virtuální metody, v pořadí `get()` a `getDoubled()`

Abstrakce

```
class CisloNaDruhou : public Cislo {
  public:
    CisloNaDruhou() : Cislo{ } {}
    CisloNaDruhou(int a) : Cislo{ a } {}

    virtual int get() override {
      return cislo * cislo;
    }
    virtual int getDoubled() override {
      return cislo * cislo * 2;
    }
};
```

Abstrakce

- virtuální metody
- invokace metod výběrem adresy z vtable
- zavolání

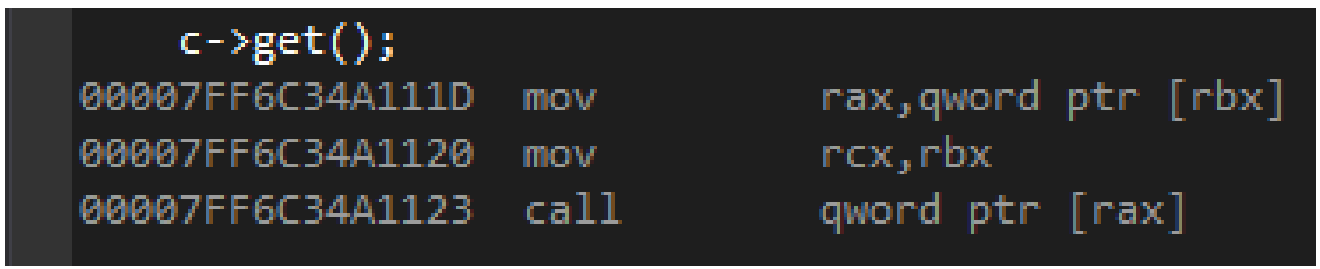
```
    b->get();
00007FF7D7D1111D  mov         rax,qword ptr [rsi]
00007FF7D7D11120  mov         rcx,rsi
00007FF7D7D11123  call        qword ptr [rax]
    b->getDoubled();
00007FF7D7D11125  mov         rax,qword ptr [rsi]
00007FF7D7D11128  mov         rcx,rsi
00007FF7D7D1112B  call        qword ptr [rax+8]
```

Obrázek: Třída přepisuje obě virtuální metody rodiče

Abstrakce

- virtuální metody
- co když explicitně budeme instancovat potomka?
- kompilátor by *mohl pochopit*, že nemusí použít polymorfismus
  - ale nestane se tak (ne vždy)

```
CisloNaDruhou* c = new CisloNaDruhou(10);
```



Obrázek: Explicitně typovaný potomek

Abstrakce

- virtuální metody
- co když explicitně budeme instancovat potomka?
- musíme tomu pomoci klíčovým slovem `final` (u třídy nebo metody)
- provede se tzv. *devirtualizace*

```
virtual int get() override final { ... }
```

```
    acc += c->get();
00007FF71C9A1114  mov         rcx,rbx
00007FF71C9A1117  call        CisloNaDruhou::get (07FF71C9A1060h)
00007FF71C9A111C  mov         ecx,dword ptr [acc]
00007FF71C9A1120  add         eax,ecx
```

Obrázek: Explicitně typovaný potomek, s `final`

RVO, NRVO

- návratové hodnoty
- z funkce/metody vracíme instanci třídy vykonstruovanou uvnitř
  - inicializace v návratové hodnotě
  - inicializace v průběhu a návrat pojmenovaného objektu
- *(Named) Return Value Optimalization* (RVO, NRVO)
- aby kompilátor zamezil vícenásobné kopii, vydedukuje, že je to v daném kontextu zbytečné a objekt konstruuje jen jednou

```
std::vector<int> gen_fixed_vector(int a) {
    return { a, a * 2, a * 3 };
}
```

RVO, NRVO

- NRVO

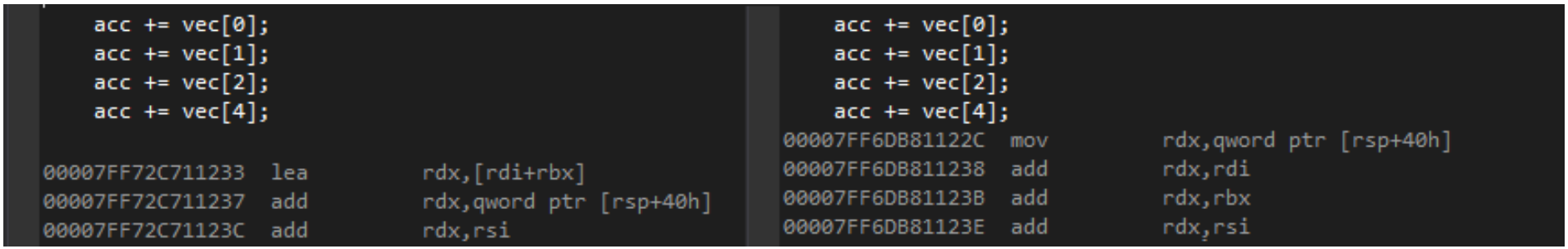
```
std::vector<int> gen_vector(size_t n)
{
  std::vector<int> vec(n);

  // ... generovani ...

  return vec;
}

auto vec = gen_vector(10);
```

Optimalizace STL

- přístupy např. do vektoru a array se dají optimalizovat
- ačkoliv jde o volání metody, díky inlinování je odstíněno
- přístup by pak měl být ekvivalentní s přístupem do C-style pole
  - `std::array` a staticky alokované pole
  - `std::vector` a dynamicky alokované pole

```
// MSVS implementace operatoru [] vectoru
_Ty& operator[](const size_type _Pos) {
  return (this->_Myfirst()[_Pos]);
}
```

Optimalizace STL



```
    acc += vec[0];
    acc += vec[1];
    acc += vec[2];
    acc += vec[4];

00007FF72C711233  lea         rdx,[rdi+rbx]
00007FF72C711237  add         rdx,qword ptr [rsp+40h]
00007FF72C71123C  add         rdx,rsi
```

```
    acc += vec[0];
    acc += vec[1];
    acc += vec[2];
    acc += vec[4];
00007FF6DB81122C  mov         rdx,qword ptr [rsp+40h]
00007FF6DB811238  add         rdx,rdi
00007FF6DB81123B  add         rdx,rbx
00007FF6DB81123E  add         rdx,rsi
```

Obrázek: `std::array` (vlevo) vs. statické pole (vpravo)

- pozn.: prvky pole jsou v obou případech cachované v registrech, proto se sčítají `rdi`, `rbx`, `rsi`, ...
- pozn. 2: obsah pole je na zásobníku, proto do toho vstupuje ještě `rsp`

Optimalizace STL

```
    acc += vec[0];
    acc += vec[1];
    acc += vec[2];
    acc += vec[4];
00007FF73ACE15CB  mov         rbx,qword ptr [vec]
00007FF73ACE15D0  mov         rdx,qword ptr [rbx+20h]
00007FF73ACE15D4  add         rdx,qword ptr [rbx+10h]
00007FF73ACE15D8  add         rdx,qword ptr [rbx+8]
00007FF73ACE15DC  add         rdx,qword ptr [rbx]
```

```
    acc += vec[0];
    acc += vec[1];
    acc += vec[2];
    acc += vec[4];
00007FF6A158165A  mov         rdx,qword ptr [rax+20h]
00007FF6A158165E  add         rdx,qword ptr [rax+10h]
00007FF6A1581662  add         rdx,qword ptr [rax+8]
00007FF6A1581666  add         rdx,qword ptr [rax]
```

Obrázek: `std::vector` (vlevo) vs. dynamicky alokované pole (vpravo)

- pozn.: obsah vectoru/dyn. alokovaného pole je na haldě – v případě vectoru se nejprve načte bázová adresa do `rbx`, v druhém případě již z nějakého důvodu bázová adresa je v `rax`

Strength reduction

- kompilátor se snaží použít rychlejší operace k dosažení ekvivalentního výsledku
- např. násobení je náročnější než sčítání, a proto

```
for (int i = 0; i < 1000; i++)
  func(i * 9999);
```

- nejspíš kompilace změní na

```
for (int i = 0; i < 9999000; i += 9999)
  func(i);
```

Strength reduction

- velmi častá je snaha vyhnout se dělení za každou cenu
- celočíselné dělení je jednou z nejnáročnějších operací v moderních CPU
- dá se často nahradit bitovými operacemi (and, or, bitshift, xor)
  - ALE... pouze za předpokladu, že kompilátor zná dělitele předem
  - obecně je tedy lepší dělit compile-time konstantou
- např. dělení `unsigned long` čísel na 64-bit ALU (změnu udělá sám kompilátor):

```
// cca 96 cyklu
y = x / 3;

// cca 12 cyklu
y = (x * 0xAAAAAAAB) >> 33;
```

Lookup table

- časté je využití lookup tables
- např. operace s omezenou doménou
- často se obětuje pár bajtů paměti navíc pro tabulku s předpočítanými výsledky
- příkladem může být alternativa `popcnt` instrukce na 1B
  - instrukce procesoru, která spočítá jednotkové bity ve vstupu
  - když tuto instrukci nemáme, musíme suplovat
  - lokální proměnná, shifting a ruční počítání jednotek? moc náročné (cca 18 instrukcí)
  - pojmeme hodnotu jako index do tabulky
  - tabulka obsahuje předpočítané počty jednotek na všech 1B číslech (256 hodnot) (cca 1-2 instrukce)

Inlining

- princip známý i mimo C++
- vložení těla funkce do místa, kde je volána
- efektivní pro malé funkce
  - odebere overhead funkčního volání
    - machinace se zásobníkem
    - předání parametrů
    - návratové hodnoty
    - překryvu registrů, ...
  - dovoluje optimalizovat inlinovanou funkci v kontextu volajícího
    - nebylo by možné bez inlinování kvůli sdílení kódu napříč různými kontexty
- klíčové slovo `inline`
- kompilátor často dedukuje sám, ale je dobré funkce takto značit

Omitting frame pointer

- *-fomit-frame-pointer*
- nevyužití frame pointeru (registr BP/EBP/RBP na x86)
- efektivní pro malé funkce s malým množstvím lokálních proměnných
- dovoluje využít frame pointer pro něco jiného (registr namísto reference paměti)
- urychlení běhu, přístup do registru je vždy rychlejší
- občas ale komplikuje život některým diagnostickým nástrojům
  - debugger
  - profiler

Konstanty

- *constant folding*, *constant propagation*
- kompilátor detekuje, které hodnoty se nemění
- některé případně dopočte v čase kompilace (folding)
- pak lze optimalizovat lépe
  - počet lokálních proměnných (hodnotu dá přímo do instrukce)
  - charakter operací (např. celočíselné násobení 2 je bitshift)
  - a jiné...
- nelze ale vždy, proto...
  - používat `constexpr` a `consteval`
  - pokud jsme si vědomi, že něco konstantní bude, můžeme to rovnou nahradit konstantou

Konstanty

- *omitting constant variable*
- přesto, že konstantu přiřazujeme do proměnné a tu až někam do paměti, kompilátor detekuje neměnnost a hodnotu vloží přímo do instrukce

```
int nasobek = 5;
*cil = nasobek;

mov   dword ptr [memory (07FF68FDD3628h)],5
```

Konstanty

- *operation substitution*
- např.
  - násobení dvěma $\rightarrow$ sečtení se sebou samým
  - násobení pěti $\rightarrow$ sečtení hodnoty s jejím čtyřnásobkem (bitshift)

```
int nasobek = argc * 5;
promenna = nasobek * 2;

lea    eax,[rcx+rcx*4]
add    eax,eax
```

Konstanty

- *operation substitution*
- násobení mocninou dvou → levý posun o exponent

```
int nasobek = argc * 5;
memory = nasobek * 1024;

lea    eax,[rcx+rcx*4]
shl    eax,0Ah
```

Konstanty

- *operation substitution*
- ale ...
- násobení mocninou dvou uloženou v proměnné → skutečné násobení

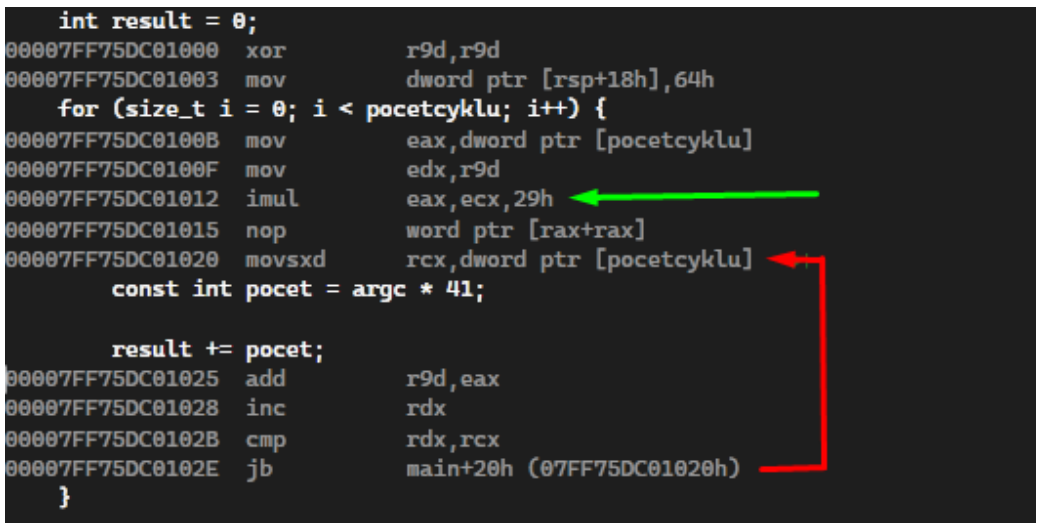
```
int nasobek = argc * 5;
volatile int nasobic = 1024;
memory = nasobek * nasobic;
```

```
mov    dword ptr [rsp+8],400h
lea    edx,[rcx+rcx*4]
mov    eax,dword ptr [nasobic]
imul   eax,edx
```

- *pozn. klíčovým slovem volatile zde nahrazujeme „skutečný kontext“, ve kterém předem není známá hodnota proměnné*

Expression elimination

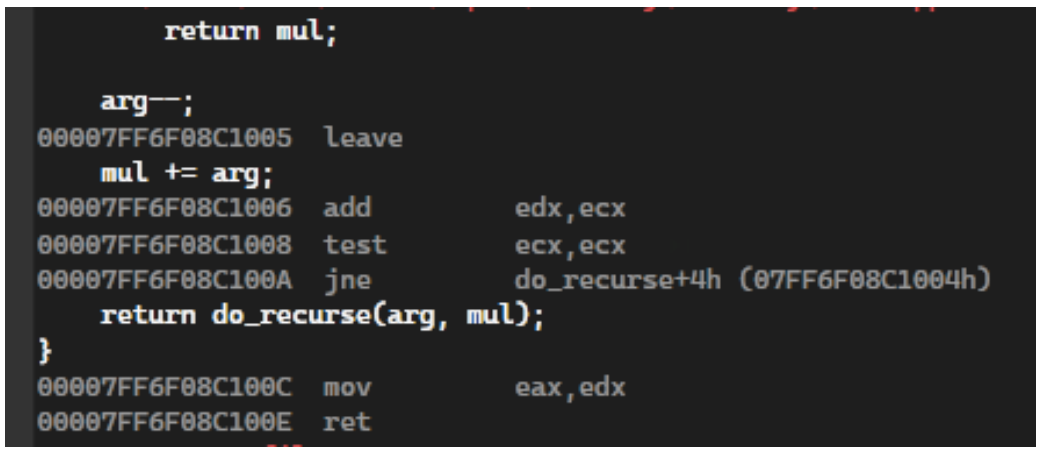
- *expression elimination*
- kompilátor detekuje, které výrazy lze vyhodnotit jen jednou
- např. v cyklu mezivýpočet, který je ale invariantní vůči iteracím cyklu
- takový výpočet je proveden jen jednou a výsledek uložen do lokální proměnné



Obrázek: násobení provedeno jen jednou (zelená šipka), cyklus se opakuje bez něj (červená)

Recursion elision

- *recursion elision*
- za jistých okolností lze transformovat rekurzi na cyklus
- např. tzv. *tail recursion*
  - tělo funkce končí rekurzivním voláním sebe samotné
- opět lze výrazně šetřit overhead funkčního volání
- pochopitelně i zásobník



Obrázek: funkční volání nahrazeno za skok (spolu s dalšími optimalizacemi), fakticky nahrazeno cyklem

Optimalizace cyklů

- optimalizace cyklů tvoří velkou část optimalizačního procesu
- s příchodem vektorových instrukcí nabývá na významu ještě více
- často vyžadují známý počet iterací
- např.
  - *loop unrolling* – znásobení těla `for` cyklu pro redukci počtu ověření podmínky
  - *loop reversal* – obrácení směru inkrementace `for` cyklu na dekrementaci
  - *loop fission* – rozdělení jednoho cyklu na více podcyklů kvůli lokalitě v cache
  - *loop fusion* – sloučení více cyklů se stejným počtem iterací do jednoho (např. vektorizace)
  - a další...

Optimalizace cyklů

- *loop unrolling*
- snížení počtu ověření podmínky, popř. vektorizace
- původní smyčka

```
for (size_t i = 0; i < 32; i++)
  acc += vec[i];
```

- „odrolovaná“ smyčka – dělá automaticky kompilátor

```
for (size_t i = 0; i < 8; i++) {
  acc += vec[i*4 + 0];
  acc += vec[i*4 + 1];
  acc += vec[i*4 + 2];
  acc += vec[i*4 + 3];
}
```

Loop unrolling

```
    for (size_t i = 0; i < VectorSize; i++)
        acc += vec[i];
00007FF6469B1287  mov          rax,qword ptr [vec]
00007FF6469B128C  mov          rdx,qword ptr [rax+18h]
00007FF6469B1290  add          rdx,qword ptr [rax+10h]
00007FF6469B1294  add          rdx,qword ptr [rax+8]
00007FF6469B1298  add          rdx,qword ptr [rax]
```

Obrázek: Loop unrolling, VectorSize = 4

```
        acc += vec[i];
00007FF6C8BE12A0  vpaddq       ymm1,ymm1,ymmword ptr [rax+rbx*8]
00007FF6C8BE12A5  vpaddq       ymm2,ymm2,ymmword ptr [rax+rbx*8+20h]
    for (size_t i = 0; i < VectorSize; i++)
00007FF6C8BE12AB  add          rbx,8
00007FF6C8BE12AF  cmp          rbx,80h
00007FF6C8BE12B6  jb           main+50h (07FF6C8BE12A0h)
    int64_t acc = 0;
00007FF6C8BE12B8  vpaddq       ymm2,ymm1,ymm2
00007FF6C8BE12BC  vextracti128 xmm1,ymm2,1
00007FF6C8BE12C2  vpsrldq      xmm0,xmm1,8
00007FF6C8BE12C7  vpaddq       xmm3,xmm1,xmm0
00007FF6C8BE12CB  vextracti128 xmm2,ymm2,0
00007FF6C8BE12D1  vpsrldq      xmm0,xmm2,8
00007FF6C8BE12D6  vpaddq       xmm0,xmm2,xmm0
00007FF6C8BE12DA  vpaddq       xmm1,xmm0,xmm3
00007FF6C8BE12DE  vmovq        rdx,xmm1
```

Obrázek: Loop unrolling, VectorSize = 128

Optimalizace cyklů

- *loop reversal*
- obrácení směru iterace
- původní smyčka

```
for (size_t i = 0; i < VectorSize; i++)
  acc += vec[i];
```

- obrácená smyčka

```
for (size_t i = VectorSize - 1; i--; )
  acc += vec[i];
```

- pozn. zde se využívá toho, že `i--` vrátí 0 na konci cyklu, a tedy se podmínka vyhodnotí negativně a cyklus skončí

Loop reversal

```
    for (size_t i = 0; i < vec.size(); i++)
00007FF7D9A01289  mov         eax,ebx
00007FF7D9A0128B  mov         rcx,qword ptr [rsp+30h]
00007FF7D9A01290  mov         rdx,qword ptr [vec]
00007FF7D9A01295  sub         rcx,rdx
00007FF7D9A01298  sar         rcx,3
00007FF7D9A0129C  test        rcx,rcx
00007FF7D9A0129F  je          main+5Dh (07FF7D9A012ADh)
        acc += vec[i];
00007FF7D9A012A1  add         rbx,qword ptr [rdx+rax*8]
    for (size_t i = 0; i < vec.size(); i++)
00007FF7D9A012A5  inc         rax
00007FF7D9A012A8  cmp         rax,rcx
00007FF7D9A012AB  jb          main+51h (07FF7D9A012A1h)
```

```
    for (size_t i = vec.size()-1; i--; )
00007FF605AA128E  mov         rcx,qword ptr [vec]
00007FF605AA1293  sub         rax,rcx
00007FF605AA1296  sar         rax,3
00007FF605AA129A  sub         rax,1
00007FF605AA129E  je          main+5Ch (07FF605AA12ACh)
00007FF605AA12A0  dec         rax
        acc += vec[i];
00007FF605AA12A3  add         rbx,qword ptr [rcx+rax*8]
    for (size_t i = vec.size()-1; i--; )
00007FF605AA12A7  test        rax,rax
00007FF605AA12AA  jne         main+50h (07FF605AA12A0h)
```

Obrázek: Loop reversal s neznámou velikostí; vlevo původní cyklus, vpravo obrácený

Loop reversal

```
    for (size_t i = VectorSize - 1; i--; )
        acc += vec[i];
00007FF7A6231342  add         rdx,qword ptr [rax+320h]
00007FF7A6231349  add         rdx,qword ptr [rax+318h]
00007FF7A6231350  add         rdx,qword ptr [rax+310h]
00007FF7A6231357  add         rdx,qword ptr [rax+308h]
00007FF7A623135E  add         rdx,qword ptr [rax+300h]
00007FF7A6231365  add         rdx,qword ptr [rax+2F8h]
00007FF7A623136C  add         rdx,qword ptr [rax+2F0h]
00007FF7A6231373  add         rdx,qword ptr [rax+2E8h]
00007FF7A623137A  add         rdx,qword ptr [rax+2E0h]
00007FF7A6231381  add         rdx,qword ptr [rax+2D8h]
00007FF7A6231388  add         rdx,qword ptr [rax+2D0h]
00007FF7A623138F  add         rdx,qword ptr [rax+2C8h]
00007FF7A6231396  add         rdx,qword ptr [rax+2C0h]
00007FF7A623139D  add         rdx,qword ptr [rax+2B8h]
00007FF7A62313A4  add         rdx,qword ptr [rax+2B0h]
00007FF7A62313AB  add         rdx,qword ptr [rax+2A8h]
00007FF7A62313B2  add         rdx,qword ptr [rax+2A0h]
00007FF7A62313B9  add         rdx,qword ptr [rax+298h]
00007FF7A62313C0  add         rdx,qword ptr [rax+290h]
00007FF7A62313C7  add         rdx,qword ptr [rax+288h]
00007FF7A62313CE  add         rdx,qword ptr [rax+280h]
00007FF7A62313D5  add         rdx,qword ptr [rax+278h]
00007FF7A62313DC  add         rdx,qword ptr [rax+270h]
00007FF7A62313E3  add         rdx,qword ptr [rax+268h]
00007FF7A62313EA  add         rdx,qword ptr [rax+260h]
00007FF7A62313F1  add         rdx,qword ptr [rax+258h]
00007FF7A62313F8  add         rdx,qword ptr [rax+250h]
00007FF7A62313FF  add         rdx,qword ptr [rax+248h]
00007FF7A6231406  add         rdx,qword ptr [rax+240h]
00007FF7A623140D  add         rdx,qword ptr [rax+238h]
00007FF7A6231414  add         rdx,qword ptr [rax+230h]
```

Obrázek: Loop reversal se známou velikostí, VectorSize = 128

Optimalizace cyklů

- *loop fusion*
- spojení více cyklů do jednoho při stejném počtu iterací
- původní smyčky

```
for (size_t i = 0; i < VectorSize; i++)
  acc1 += vec[i];
for (size_t i = 0; i < VectorSize; i++)
  acc2 += vec[i]*2;
```

- spojená smyčka

```
for (size_t i = 0; i < VectorSize; i++) {
  acc1 += vec[i];
  acc2 += vec[i]*2;
}
```

- „opakem“ je loop fission (rozdělení)

Poznámky k psaní efektivního kódu

- závěrem k čistě jazykové části – poznámky k psaní efektivního kódu
- částečně opakování
- používání efektivních struktur
- „pomoc“ kompilátoru při optimalizaci kódu

Poznámky k psaní efektivního kódu

- statická pole
- `std::array`
- přístup k prvkům
  - operátor `[]`
  - metodu `.at()` používat jen tehdy, když si nejste jisti, zda zasahujete do rozsahu – vždy kontroluje hranice a může vyhodit výjimku, ale zpomaluje!

Poznámky k psaní efektivního kódu

- dynamická pole
- `std::vector`
- kontinuální paměť, neefektivní mazání
- vkládání prvků
  - známý počet: `reserve(pocet)` + `push_back()`
    - pro POD lze i `resize(pocet)`
  - neznámý počet: `push_back()`, `emplace_back()`
- přístup k prvkům
  - operátor `[]`
  - metodu `.at()` – stejná implikace jako u array (zpomaluje)
  - iterátor

Poznámky k psaní efektivního kódu

- seznam
- `std::list`
- nezaručuje kontinuální paměť
- vkládání prvků
  - `push_back()`, `emplace_back()`
- přístup k prvkům
  - iterátor
  - range-based for

Poznámky k psaní efektivního kódu

- asociativní mapa
- `std::map`, `std::unordered_map`
- vkládání prvků
  - indexem
  - `insert()`
- přístup k prvkům
  - iterátor, `find()`
  - range-based for

Poznámky k psaní efektivního kódu

- `std::map`
  - menší paměťové nároky
  - uspořádané prvky
  - potenciálně pomalejší
- `std::unordered_map`
  - větší paměťové nároky
  - neuspořádané prvky
  - potenciálně rychlejší
- analogicky platí pro `std::set` a `std::unordered_set`

Poznámky k psaní efektivního kódu

- polymorfismus
- dynamický polymorfismus je dražší (vtable)
- pokud je to možné, vyhnout se
  - statický polymorfismus (CRTP)
  - klíčové slovo `final` u tříd a metod
  - koncepty k nahrazení polymorfismu v kontextu generického programování
- celkově se snažíme omezit nutnost pracovat s typy v čase běhu
  - ne vždy je úplně možné nebo vhodné

Poznámky k psaní efektivního kódu

- zamezení zbytečným kopiím
- někdy udělá sám kompilátor (RVO/NRVO, copy elision)
- někdy mu musíme pomoct
  - move sémantika
  - `emplace_back()` v kontejnerech
  - reference

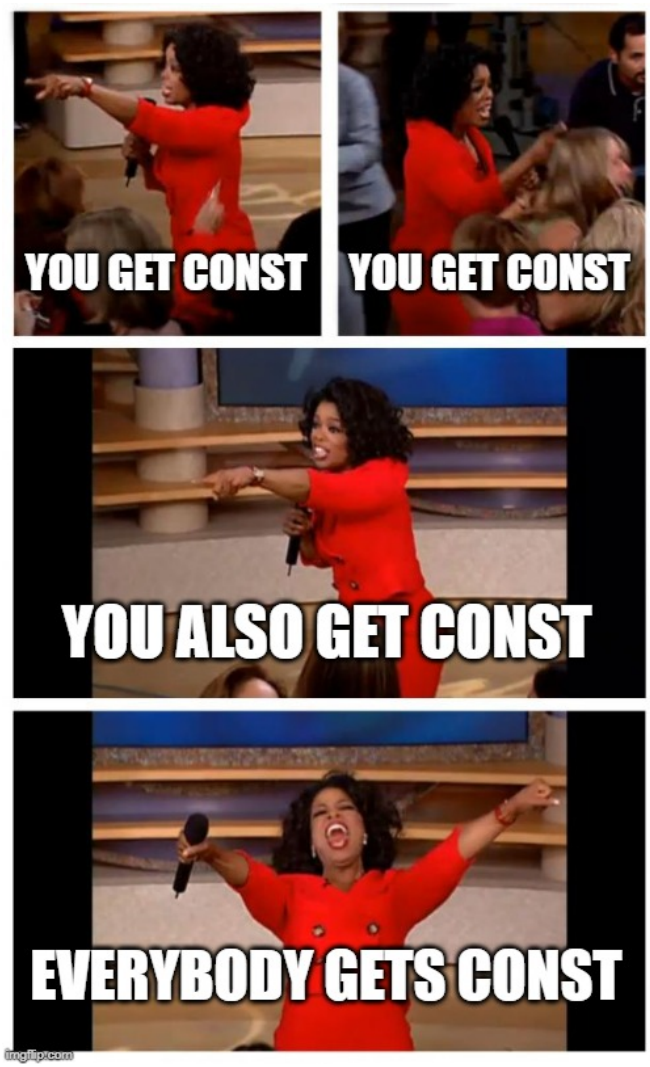
Poznámky k psaní efektivního kódu

- optimalizace alokace
- dynamická alokace je dražší
- upřednostňujeme statickou, pokud je to možné
  - malé, nesdílené třídy/POD
  - deterministická životnost
- dynamickou alokaci bychom měli balit do struktur k tomu určených
  - `unique_ptr` – pro nesdílený ukazatel
  - `shared_ptr` – pro sdílený ukazatel (pozor, má větší režii)
  - STL kontejnery
- minimalizace nebo úplná eliminace použití surových pointerů

Poznámky k psaní efektivního kódu

- použití výjimek
- Zero-Cost model
  - přidaná režie pouze pokud k výjimce dojde
- výjimky používáme jen pro výjimečné situace
  - výhradně chybové řízení
- měli bychom použít třídy odděděné od `std::exception`

Poznámky k psaní efektivního kódu



- `const`
- vyhodnocení v čase kompilace
- `constexpr`, `consteval`
  - konstanty
  - funkce
- šablony

Poznámky k psaní efektivního kódu

- uvážené použití šablon
- příliš mnoho instancí šablon vede k obrovskému kódu
- potenciálně se pak vejde méně do instrukční cache CPU
- najít rovnováhu mezi použitím šablon a běhovým rozlišením
- omezit parametrizace hodnotou

Poznámky k psaní efektivního kódu

- profiler je kámoš
- v případě, že je kód stále pomalejší, než by pocitově být měl
- ale i když ne – profiling by měl být součástí vývojového procesu


Vitamin C++
Nourishing & Renewing
Night Cream
For normal to dry skin