KIV/TSI - Seminář C++

08. Další konstrukce jazyka C++, práce s knihovnami

Martin Úbl

KIV ZČU

2020/2021

Další konstrukce jazyka C++

- přehled doposud neprobíraných konstrukcí
- další speciality z STL
- vlastnosti z C++14 a C++17

Další konstrukce jazyka C++

- *small string optimizations*
- optimalizace `std::string` pro malé řetězce
- normálně by byla potřeba dynamická alokace
- `std::string` od C++17 dovoluje uchovávat krátké řetězce přímo v sobě
  - 15 znaků pro MSVS
  - 23 znaků pro GCC/clang
- jistá podobnost s fast symbolic link ve VFS/ext

```
// bez dynamicke alokace
std::string a{"Hello"};

// s dynamickou alokaci
std::string a{"Could␣somebody␣tell␣me␣what␣the␣f
```

Další konstrukce jazyka C++

- *string views*
- `#include <string_view>`
- třída `std::string_view` poskytuje „pohled“ na jinou řetězcovou paměť
- sama neuchovává řetězec
- lze např. poznat rozdíl při volání `substr`
  - `std::string` vrací `std::string` - $O(n)$
    - vrací „kopii“ podřetězce
  - `std::string_view` vrací `std::string_view` - $O(1)$
    - vrací pouze pohled na podřetězec (v podstatě dva ukazatele)
- hodí se např. při parsování vstupů

```
std::string s{ "retezec" };
std::string_view sv(s.c_str(), 3);
std::cout << sv << std::endl;
```

```
// 2 instance std::string
// dvakrat rozkopirovany retezec
std::string s{ "nejaky␣velmi␣dlouhy␣retezec␣co␣s
auto s2 = s.substr(7);

// 2 instance std::string_view
// ale jen jedna instance retezce
std::string_view sv(s);
auto sv2 = sv.substr(7);
```

Další konstrukce jazyka C++

- `#include <optional>`
- třída `std::optional` obaluje jiný typ, který může, ale nemusí být poskytnut
- šablonový typ
- konstruktor s instancí daného typu nebo implicitní
- `std::nullopt` lze použít pro vytvoření prázdného

```
std::optional<int> GetValue(size_t idx) {
  if (myMap.find(idx) != myMap.end())
    return myMap[idx];
  return std::nullopt;
}
```

Další konstrukce jazyka C++

- `std::optional`
- ověření, zda má hodnotu
  - metoda `has_value()`
  - přetížený operátor `bool()`
- vyzvednutí hodnoty
  - metoda `value()`
  - přetížený operátor dereference

Další konstrukce jazyka C++

- `#include <variant>`
- třída `std::variant` obaluje jednu hodnotu různých typů
- typově bezpečný `union` z C
- šablonový typ (variadický), v šabloně všechny typy, které může potenciálně ukládat
- vždy ukládá právě jeden z nich (přiřazení, konstruktor)
- vyzvednutí funkcí `std::get<T>`
- get špatného typu vyhodí výjimku `std::bad_variant_access`

```
std::variant<int, double, long long> var;

var = 1;
var = 15.4;
var = 99999LL;
```

- `std::variant`
- visitor pattern

```
std::variant<int, double, long long> var = ...;

std::visit([&](auto&& value) {
  // "value" ma spravny typ
}, var);
```

- pro daný variant se vydedukuje správný typ hodnoty a tedy se i specializuje příslušná lambda (`auto` parametr)
- může se např. hodit pro procházení vektoru variantů
- pozn. `std::variant` používá pouze zásobník (nikdy haldu)

Další konstrukce jazyka C++

- `#include <any>`
- třída `std::any` obaluje jednu hodnotu libovolného typu
- typově bezpečný `void*` z C
- není šablonový typ
- alokace vždy na haldě
- vyzvednutí přetypováním `std::any_cast<T>`
- cast na špatný typ vyhodí výjimku `std::bad_any_cast`
- typ musí být kopírovatelný

```
std::any v;

v = 42;
v = "retezec";
```

Další konstrukce jazyka C++

- `std::any`

```
std::any v = ...;

try {
  int ival = std::any_cast<int>(v);
}
catch (std::bad_any_cast& bac) {
  // ...
}
```

- pomalejší než `std::variant`
  - dynamická dedukce za běhu
  - alokace na haldě

- binární literály

```
uint8_t bin = 0b00010011;
```

- oddělovače cifer

```
uint32_t i1 = 1'005'999;
uint16_t i2 = 0b00001111'10110110;
```

- konstrukce `std::string`
- `using namespace std::string_literals;`

```
auto str = "hello"s;
```

Další konstrukce jazyka C++

- user-defined literály
- možnost definovat si vlastní literály
- např. převody jednotek
- „tváří“ se jako operátor ""
- uživatelské literály by měly začínat podtržítkem
  - bez podtržítka jsou vyhrazené pro standard jazyka

```
// napr. vse na metry
constexpr long double operator"" _cm(
                                long double cm) {
  return cm / 100.0;
}

long double A4_vyska = 29.7_cm; // 0.297
```

- lze definovat jen pro vybrané typy
  - `const char*`
  - `long double`
  - `unsigned long long int`
  - `char`, `wchar_t`, `char16_t`, ...
- vracet však mohou libovolný typ, klidně instanci třídy

Další konstrukce jazyka C++

- `#include <thread>`
- třída `std::thread` reprezentuje instanci vlákna
- vlákna - KIV/PGS, KIV/ZOS, KIV/PPR

```
std::thread thr([size]() {
  for (size_t i = 0; i < size; i++) {
    //
  }
});
```

- získání současného vlákna `std::this_thread`
- vlákna v C++ viz KIV/PPR

Další konstrukce jazyka C++

- další konstrukce souvisí s vlákny
- `std::async` - asynchronní provedení operace
- `std::promise` - provedení operace a možné čekání na výsledek
- `std::future` - synchronizovaná proměnná pro výsledek promise
- `std::mutex` - implementace mutexu
- `std::unique_lock` - zámek nad mutexem
- `std::condition_variable` - podmínková proměnná
- C++20 - parallel execution policy a paralelní algoritmy v STL
- a spousty dalších...
- vlákna v C++ viz KIV/PPR

- `#include <chrono>`
- standardizace časových jednotek a úseků
- práce s časovými hodnotami
- `std::chrono::duration<T>` - časový úsek se zvoleným typem
  - float, double, ...
  - nezávislý na jednotkách (sekundy, minuty, ...)
- `std::chrono::time_point<T>` - hodnota zvoleného časovače

Další konstrukce jazyka C++

- `steady_clock`
  - neklesající časovač, nereprezentuje reálný čas
  - konstantní doba mezi tiky
  - hodí se nejvíce pro měření časových úseků, např. doba výpočtu
- `system_clock`
  - reprezentuje reálný čas
- `high_resolution_clock`
  - časovač s vysokým rozlišením, nemusí reprezentovat reálný čas
- metoda `now()` pro získání hodnoty
- podpora sčítání, odčítání, ...
- převod na požadované jednotky - `std::chrono::duration_cast<T>`

Další konstrukce jazyka C++

- `#include <chrono>`
- `using namespace std::chrono_literals`
- definované literály pro časové úseky
  - `ns`, `us`, `ms`, `s`, `min`, `h`

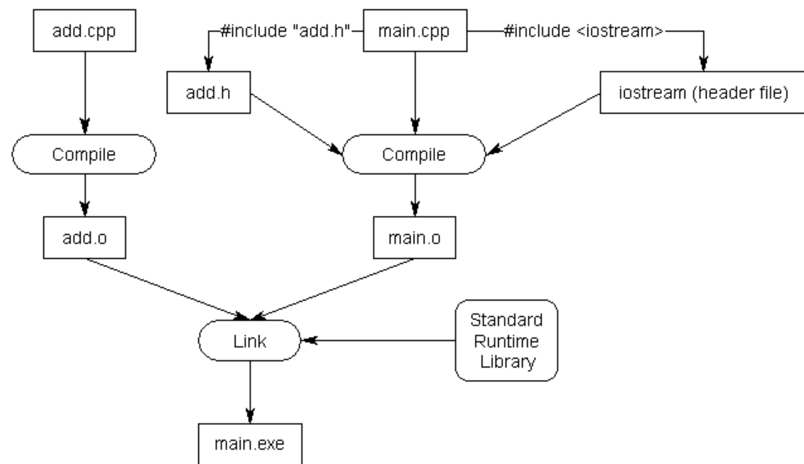
```
auto duration = 150ms;

std::this_thread::sleep_for(250ms);
```

Práce s knihovnami

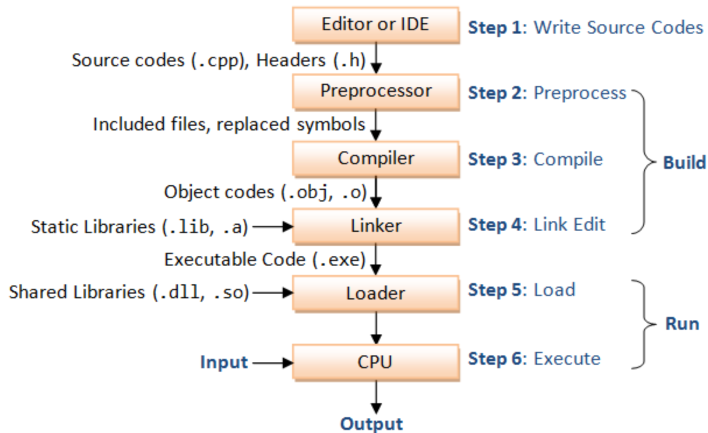
- knihovny
    - zakompilované
        - s kompilovanými moduly
        - „hlavičkové“ - pouze v .h / .hpp souboru
    - statické (staticky linkované)
        - .lib (MS Windows)
        - .a (GNU/Linux, macOS)
    - dynamické (dynamicky linkované)
        - .dll (MS Windows)
        - .so (GNU/Linux)
        - .dylib (macOS)

Práce s knihovnami



Obrázek: Znázornění procesu kompilace a (statického) linkování

Práce s knihovnami



Obrázek: Znázornění procesu kompilace a linkování

Práce s knihovnami

- statické knihovny
- musí být přítomny v čase kompilace
- již jsou zkompilované
- obsahují zatím jen symbolicky adresovaný kód (ne konkrétní adresy)
- linker je v čase kompilace přikompiluje k sestavované aplikaci
- musí být kompatibilní s daným kompilátorem
- dodavatel typicky poskytuje zkompilovanou .lib/.a knihovnu a hlavičkové soubory

Práce s knihovnami

- dynamické knihovny
- musí být přítomny v čase běhu (spouštění)
- načítání
  - automatické (dynamic sekce spustitelného souboru)
  - ruční (dlopen, LoadLibrary, ...)
- knihovny musí symboly tzv. exportovat
  - na MS Windows explicitně
  - na GNU/Linux / macOS jsou exportované všechny

Práce s knihovnami

- dynamické knihovny
- C++ - name mangling
- „serializace“ jména (symbolu)
- často přidá podtržítka, čísla, další znaky
- unikátnost

Práce s knihovnami

```
martin@rattmann:~/test2$ nm a.out
0000000000004058 B __bss_start
0000000000004170 b completed.7389
                 U __cxa_atexit@@GLIBC_2.2.5
                 w __cxa_finalize@@GLIBC_2.2.5
0000000000004040 D __data_start
0000000000004040 W data_start
00000000000010c0 t deregister_tm_clones
0000000000001130 t __do_global_dtors_aux
0000000000003db0 t __do_global_dtors_aux_fini_array_entry
0000000000004048 D __dso_handle
0000000000004050 V DW.ref.__gxx_personality_v0

                                        ...
0000000000004178 B _end
0000000000001354 T _fini
0000000000001175 T main
00000000000010f0 t register_tm_clones
0000000000001090 T _start
0000000000004058 D __TMC_END__
                 U _Unwind_Resume@@GCC_3.0
                 U _ZNSolsEi@@GLIBCXX_3.4
                 U _ZNSolsEPFRSoS_E@@GLIBCXX_3.4
                 U _ZNSt8ios_base4InitC1Ev@@GLIBCXX_3.4
                 U _ZNSt8ios_base4InitD1Ev@@GLIBCXX_3.4
0000000000001264 W _ZN11NejakaTridaC1Ev
0000000000001264 W _ZN11NejakaTridaC2Ev
0000000000001270 W _ZN11NejakaTridaD1Ev
0000000000001270 W _ZN11NejakaTridaD2Ev
000000000000128a W _ZN11NejakaTridapLERKi
000000000000127c W _ZN11NejakaTrida12NejakaMetodaEi
0000000000002004 r _ZStL19piecewise_construct
0000000000004171 b _ZStL8__ioinit
0000000000004060 B _ZSt4cout@@GLIBCXX_3.4
                 U _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_@@GLIBCXX_3.4
0000000000001205 t _Z41__static_initialization_and_destruction_0ii
000000000000129c W _Z9MojePrintIiEvRKT_
```

Obrázek: Name mangling C++ (výpis nástroje nm)

Práce s knihovnami

- zamezení name manglingu
- `extern "C"`

```
extern "C" void perform_magic(int param1);

extern "C" unsigned long long SharedCounter;
```

Práce s knihovnami

- přes rozhraní dynamických knihoven by měly jít pouze POD typy
- dynamické knihovny by měly být nezávislé na jazyce
- nemáme jistotu, že hostitelská aplikace např. psaná v Pascalu bude mít binárně kompatibilní implementaci `std::vector`
- problém na jiné platformě
- problém i s jinou verzí téhož kompilátoru (standardní knihovny)

Práce s knihovnami

- export symbolů
- na GNU/Linux a macOS jsou exportované všechny symboly
- na Windows je nutné explicitně exportovat
  - `_declspec(dllexport)`
  - .def soubor (specifikuje se kompilátoru)
  - a další.. (export přepínač, pragma)

Práce s knihovnami

- `_declspec(dllexport)`

```
extern "C" _declspec(dllexport)
      void perform_magic(int param1);
```

- pokud linkujeme automaticky, může být v definici (hlavičkový soubor) pro hostitelskou aplikaci uveden `_declspec(dllimport)`
- jde o optimalizaci pro linkování

- .def soubor (specifikuje se kompilátoru)

```
LIBRARY magic.dll

EXPORTS
    perform_magic
    SharedCounter
```

Práce s knihovnami

- import symbolů
  - automaticky (generovaná statická knihovna se „spojkami“ + hlavičkový soubor)
  - ručně

Práce s knihovnami

- import symbolů automaticky
- kompilátor při překladu vytvoří malou statickou knihovnu
- tam se nachází resolvery funkcí z knihovny
- hostitelská aplikace přilinkuje statickou knihovnu
- příslušný modul includuje hlavičkový soubor
- dynamická knihovna je načtena při spuštění programu
- programátor se nestará o import symbolu, „prostě volá“ funkce co byly definované v hlavičkovém souboru
    - v malé statické knihovně jsou funkce s těmito jmény
    - nemají ale požadovanou funkci, pouze „návod“, jak najít a zavolat funkci z dynamické knihovny
    - viz GOT/PLT v KIV/OS

Práce s knihovnami

- import symbolů ručně
- je vytvořena jen dynamická knihovna
- sada funkcí pro hostitelskou aplikaci
  - Windows / GNU/Linux+macOS
  - `LoadLibrary / dlopen` - otevření knihovny
  - `GetProcAddress / dlsym` - načtení symbolu
  - `FreeLibrary / dlclose` - uzavření knihovny

Práce s knihovnami

- GetProcAddress / dlsym - načtení symbolu
- vyzvedne pouze adresu
- musíme znát signaturu funkce (např. nějaký hlavičkový soubor knihovny)
- adresa je validní, dokud nezavoláme `FreeLibrary / dlclose`

Práce s knihovnami

```
using MagicFunc = void(*)(int);

auto lib = LoadLibrary("magic.dll");

MagicFunc perform_magic =
    reinterpret_cast<MagicFunc>(
        GetProcAddress(lib, "perform_magic")
    );

perform_magic(42);

FreeLibrary(lib);
```

Práce s knihovnami

- Volací konvence
- „dohoda“ toho, jak se volá funkce/metoda
  - jak se předávají parametry
  - jak se předává návratová hodnota
  - kdo „uklidí“ zásobník
- Například:
  - `cdecl` (x86) - parametry přes zásobník zprava doleva, návratová hodnota v (E)AX/ST0 registru, ...
  - `stdcall` (x86) - podobná, jen volaná funkce uklízí zásobník
  - x64 (x86_64) - parametry částečně registry, částečně zásobníkem, „shadow“ místo v zásobníku před rámcem volání, ... (MS/*nix řeší jinak)
- `https://www.agner.org/optimize/calling_conventions.pdf`
- volací konvenci možno uvést v signatuře funkce

Práce s knihovnami

- objektové knihovní konvence
- např. COM model
- objekty vždy dědí od `IUnknown` rozhraní
  - první tři položky v tabulce virtuálních metod jsou `AddRef`, `Release` a `QueryInterface`
- každá COM třída může podporovat další rozhraní
- rozhraní identifikovány pomocí GUID
- lze zavolat `QueryInterface` nad objektem s daným GUID
  - vrací `S_OK` a přetypovaný pointer
  - nebo vrací `E_NOINTERFACE` pokud objekt rozhraní nepodporuje