KIV/CPP – Programování v jazyce C++

10a. Další konstrukce jazyka C++

Martin Úbl

KIV ZČU

2023/2024

Další konstrukce jazyka C++

- přehled doposud neprobíraných konstrukcí
- další speciality z STL
- vlastnosti z C++14, C++17 a C++20

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 b{"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;
```

Další konstrukce jazyka C++

```
// 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, jehož instance může, ale nemusí být poskytnuta
- 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;
```

Další konstrukce jazyka C++

- `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

Další konstrukce jazyka C++

- 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
```

Další konstrukce jazyka C++

- 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 <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);
```

Další konstrukce jazyka C++

- `std::filesystem`
- `#include <filesystem>`
- od C++17
- práce se souborovým systémem užitím jednotného rozhraní
  - práce se soubory, s adresáři a odkazy
  - práva
  - časy modifikace, vytvoření, ...
  - obsazení a kapacita oddílu

Další konstrukce jazyka C++

- `std::filesystem::path`
- reprezentuje cestu v souborovém systému
- nemusí nutně identifikovat existující soubor/složku/odkaz
- absolutní/relativní
- převod na absolutní `std::filesystem::absolute`
- převod na relativní `std::filesystem::relative`
- lze implicitně převést na `std::string`

```
std::filesystem::path filepath("input.txt");

std::cout << filepath << std::endl;
std::cout << std::filesystem::absolute(filepath)
          << std::endl;
```

Další konstrukce jazyka C++

- `std::filesystem::path`
- připojení komponenty pomocí append nebo operátoru `/` a `/=`

```
std::filesystem::path filepath("C:\\");

filepath.append("slozka");

std::cout << filepath << std::endl;
// C:\slozka

filepath /= "podslozka";

std::cout << filepath << std::endl;
// C:\slozka\podslozka
```

Další konstrukce jazyka C++

- `std::filesystem::path`
- další metody
  - `filename` – vrací název souboru
  - `parent_path` – odebere poslední komponentu (vrací rodičovský adresář)
  - `make_preferred` – převod oddělovačů komponent cesty na preferované pro daný OS
  - `remove_filename` – odstraní z cesty název souboru
  - `replace_filename` – nahradí název souboru jiným názvem
  - `stem` – vrací název souboru bez přípony
  - `extension` – vrací příponu

Další konstrukce jazyka C++

- `std::filesystem`
- další funkce – systémové cesty
  - `std::filesystem::current_path()` – vrací nebo nastavuje pracovní adresář
  - `std::filesystem::temp_directory_path()` – vrací adresář pro dočasné soubory

Další konstrukce jazyka C++

- `std::filesystem`
- další funkce - manipulace se souborovým systémem
  - `std::filesystem::exists()` – existence souboru/složky/odkazu
  - `std::filesystem::copy()` – kopie složky nebo souboru
  - `std::filesystem::rename()` – přejmenování nebo přesun
  - `std::filesystem::remove()` – smazání souboru nebo složky
  - `std::filesystem::status()` – vrací vlastnosti souboru

Další konstrukce jazyka C++

- `std::filesystem`
- `std::filesystem::directory_entry` – třída reprezentující položku v adresáři
  - metoda `path()` vrací fyzickou cestu
- `std::filesystem::directory_iterator` – vrací instanci třídy pro iterování přes položky adresáře

Další konstrukce jazyka C++

- `std::filesystem`
- některé funkce při nezdaru vyhazují výjimku `std::filesystem::filesystem_error()`
  - přístup k souboru, který neexistuje
  - systémové soubory
  - práva

Další konstrukce jazyka C++

- `if constexpr`
- podmínka vyhodnotitelná v čase kompilace
- „méně upovídaná“ varianta např. šablonového instancování
- podmínka musí být constexpr
- lze se tak vyhnout SFINAE a `std::enable_if`

```
template<typename T>
void DoSomething(const T& val) {

  if constexpr (std::is_integral_v<T>)
    DoIntegral(val);
  else
    DoOther(val);

}
```

Další konstrukce jazyka C++

- atributy
- syntaktický doplněk pro další optimalizace
- označení funkce/metody/bloku kódu s určitými vlastnostmi
  - `[[noreturn]]` – z funkce se nebudeme vracet (jediný návrat je buď výjimkou nebo nijak)
  - `[[deprecated("duvod")]]` – funkce je označena jako zastaralá
  - `[[fallthrough]]` – switch case level není zakončen breakem úmyslně
  - `[[maybe_unused]]` – označuje identifikátor, který nemusí být nikde použitý
  - `[[likely]]`, `[[unlikely]]` – např. pro podmínky - když víme, že podmínka bude velmi často pravdivá, označíme blok příkazů pod ní za `likely` (optimalizace)

Další konstrukce jazyka C++

- označení funkce/metody/bloku kódu s určitými vlastnostmi
  - `[[assume(...)]]` – předpoklad, že se výraz vyhodnotí na true (optimalizace)
  - další mohou být specifické pro kompilátor, např. `[[gnu::always_inline]]`
  - a další...
  - `en.cppreference.com/w/cpp/language/attributes`

Další konstrukce jazyka C++

- *structured binding*
- lze svázat vícenásobnou inicializaci s nějakým objektem
- např. rozkopírovat pole do více proměnných
- nebo nahradit `std::tie` u `std::tuple`
- syntaxe s `auto[...] = ...`
- lze svázat hodnotou nebo referencí

```
std::array<int, 2> arr{ 5, 10 };
auto[a, b] = arr;

std::tuple<int, double> tup{ 5, 12.5 };
auto&[i, d] = tup;
```

Další konstrukce jazyka C++

- *structured binding*
- možno např. ověřovat, zda se vložení prvku do kontejneru povedlo

```
std::map<int, std::string> mp;

if (auto[itr, succ]
        = mp.insert({ 42, "meaning" }); succ)
  std::cout << "OK" << std::endl;
else
  std::cout << "FAIL" << std::endl;
```

Další konstrukce jazyka C++

- *structured binding*
- lze svázat hodnotou, l- nebo r-value referencí

```
std::array<int, 2> arr{ 5, 10 };
auto[a, b] = arr;
auto&[c, d] = arr;
auto&&[e, f] = std::make_tuple(5, 6);
```

Další konstrukce jazyka C++

- `std::numeric_limits`
- `#include <limits>`
- šablonová standardní varianta numerických konstant různých typů
  - maximální a minimální reprezentovatelná hodnota
  - hodnota nekonečna a Not-a-Number
  - různé HW-related vlastnosti

```
double nejvic =
    std::numeric_limits<double>::max();

bool presne =
    std::numeric_limits<double>::is_exact();
    // false :(
```

Další konstrukce jazyka C++

- Regulární výrazy
- `#include <regex>`
- klasická forma regulárních výrazů
- sada tříd a algoritmů

Další konstrukce jazyka C++

- Regulární výrazy
- `std::regex`
- reprezentuje právě jeden regulární výraz
- konstruktor přebírá text výrazu a přepínače
- přepínače (1 či více), lze skládat operátorem |
  - `std::regex::icase` – ignoruje zda jde o velké/malé písmeno
  - `std::regex::ECMAScript` – použije syntaxi regexů z ECMAScript specifikace
  - `std::regex::awk` – použije awk syntaxi regexů
  - `std::regex::multiline` – matching přes více řádek
  - atd...
  - `https://en.cppreference.com/w/cpp/regex/basic_regex`

- `std::regex_match` – match celého vstupu

```
std::string valid_email{ "fanda@domena.cz" };
std::string invalid_email{ "aa:bb@domena.cz" };

const std::regex r(
  "[a-z0-9\\.]+@[a-z0-9\\.]+\\.[a-z]+",
  std::regex::icase | std::regex::ECMAScript
);

std::regex_match(valid_email, r);   // true
std::regex_match(invalid_email, r); // false
```

- pozn.: regex pro e-maily je pochopitelně složitější

Další konstrukce jazyka C++

- `std::regex_match` – match celého vstupu + sub-matches
- `std::smatch`

```
const std::string dom{ "www.zcu.cz" };
const std::regex r(
  "([a-z]+)\\.([a-z]+)\\.([a-z]+)"
);

std::smatch sm;
if (std::regex_match(dom, sm, r)) {
  for (auto& s : sm)
    std::cout << s.str() << std::endl;
}
```

- submatch objekt vždy na první pozici obsahuje celý matchnutý řetězec a až pak sub-matches

- `std::regex_search` – match části vstupu

```
const std::string info{
  "Name: Gordon, Surname: Freeman, Age: 44"
};
const std::regex r("(Surname: )([a-z]+)",
  std::regex::icase | std::regex::ECMAScript
);

std::smatch sm;
if (std::regex_search(info, sm, r)
    && sm.size() == 3) {
  // sm[2].str() == "Freeman"
}
```

- `std::regex_replace` – match a replace částí vstupu

```
const std::string in{ "miniaturni␣dikobraz" };
const std::regex r("i",
  std::regex::icase | std::regex::ECMAScript);

auto res = std::regex_replace(in, r, "y");
// "mynyaturny dykobraz"

auto res2 = std::regex_replace(in, r, "_$&_");
// "m_i_n_i_aturn_i_ d_i_kobraz"
```

Další konstrukce jazyka C++

- Regulární výrazy
- lze různě kombinovat a řetězit
- podpora regexů v C++ by měla umět to, na co jsme zvyklí z jiných jazyků

Další konstrukce jazyka C++

- Value array
- `#include <valarray>`
- kontejner pro čísla podprující hromadné matematické operace nad prvky
- interně souvislá paměť
  - chová se jako omezený `std::vector`
- doplněné matematické funkce pro tento typ
  - např. `std::pow`, `std::sqrt` a jiné
- podporuje slicing (řezání v intervalu se zadanou střídou)
- dává velký prostor optimalizacím
  - využití cache
  - vektorizace (SMP)
- sortiment operací omezený
  - daleko více možností např. knihovna Eigen3

- např. řešení 6 kvadratických rovnic zároveň

```
// ax^2 + bx + c = 0
std::valarray<double> a{ 1, 1, 2, 2, 3, 3 };
std::valarray<double> b{ 1, 2, 1, 2, 1, 2 };
std::valarray<double> c{ 0, 1, 0, 1, 0, 1 };

// ( -b +- sqrt(b*b - 4ac) ) / 2a
auto x1 = (-b + std::sqrt(b * b - 4 * a * c))
          / 2 * a;
auto x2 = (-b - std::sqrt(b * b - 4 * a * c))
          / 2 * a;
```

Další konstrukce jazyka C++

- *complex*
- `#include <complex>`
- podpůrná knihovna pro práci s komplexními čísly
- vlastně jen kontejner nad dvojicí `double` čísel
- jmenný prostor `std::complex_literals`
  - definuje literál pro komplexní jedničku `i`
- šablonový typ `std::complex`
- zápis klasicky např. `2.0 + 3.0i`
- rozšíření standardních matematických funkcí
- nebudeme nijak více rozebírat, jen je dobré vědět, že to C++ umí

- příklad

```
using namespace std::complex_literals;

auto c1 = 2.0 + 1.0i;
auto c2 = 1.0 - 2.0i;

auto res = c1 * std::pow(c2, 2);
std::cout << res.real() << " + "
          << res.imag() << "i"
          << std::endl;

// -38 + 41i
```

Další konstrukce jazyka C++

- *feature testing* (od C++20)
- ukládá povinnost kompilátoru deklarovat, co vše ze standardu jazyka a knihovny již umí
- lze tak zvýšit zpětnou kompatibilitu
- kompilátor definuje makra vč. „verze“ podpory (pokud se standard upravoval)
- lze pak v kódu ověřit klasickým `#ifdef`
- seznam maker
  - `https://en.cppreference.com/w/cpp/feature_test`

Další konstrukce jazyka C++

- co třeba `constexpr` ve VS2019?

```
std::cout << __cpp_constexpr << std::endl;
// 201907
```

`__cpp_constexpr`	constexpr	`200704L`	(C++11)
	Relaxed constexpr, non-const constexpr methods	`201304L`	(C++14)
	Constexpr lambda	`201603L`	(C++17)
	Trivial default initialization and asm-declaration in constexpr functions	`201907L`	(C++20)

- `std::span`
- obaluje rozsah (statický či dynamický)
- „surovější“ varianta views – menší omezení
- může ukazovat na rozsah prvků v kontejneru, ale i v obyčejném poli

```
int* arr = new int[4];

// staticky
std::span<int, 4> sp1 = arr;
// dynamicky
std::span<int> sp1 = arr;
```

Další konstrukce jazyka C++

- `#include <bit>`
- hlavičková knihovna pro práci s daty na úrovni bitů
  - `bit_cast` – čistější varianta dereference reinterpret cast výsledku
  - `popcount` – vrátí počet jedniček
  - `rotl`, `rotr` – levá či pravá bitová rotace
  - `countl_...` – vrací počet leading/trailing jedniček/nul
  - a další...
- `https://en.cppreference.com/w/cpp/header/bit`

Další konstrukce jazyka C++

- `#include <numbers>`
- číselné konstanty
  - `std::numbers::pi` – Ludolfovo číslo π
  - `std::numbers::e` – Eulerova konstanta e
  - `std::numbers::sqrt2` – odmocnina ze dvou $\sqrt{2}$
  - a další...
- `https://en.cppreference.com/w/cpp/header/bit`

Další konstrukce jazyka C++

- další, nezařazené
  - `std::midpoint` – výpočet střední hodnoty bez rizika přetečení
  - `std::to_array` – převod kontejneru na pole
  - `contains()` nad mapou a množinou – náhrada `find(x) == end()`
  - `starts_with()`, `ends_with()` nad řetězcem
  - `std::byte` – typ reprezentující bajt (není znakovým typem)
  - a spousty dalších...