C++; delete Java;

Část 10: speciality

Kennny

srpen 2017

constexpr

- klíčové slovo `constexpr`
- umožňuje vyhodnocení v čase kompilace, pokud to je možné
- pokud ne, vyhodnotí za běhu jako normálně
- `constexpr` proměnné a funkce
- hodí se např. na vyhodnocení magic constant nebo obecně sady hodnot, o které víme, že pro daný argument nikdy jiná nebude

constexpr

- `constexpr` funkce funguje pouze, pokud je její návratová hodnota přiřazována do `constexpr` konstanty, jinak je vyhodnocena v čase běhu

```
constexpr int magicTransform(const int arg)
{
    return arg*5 + 10;
}

// provede se v case kompilace
constexpr int arg1 = magicTransform(15);
// provede se az za behu
int arg2 = magicTransform(99);
```

Příklad

- Prostor pro příklad 10_a_constexpr

Lambda funkce

- `#include <functional>`
- anonymní funkce
- šablonový typ `std::function`
- lze definovat vztah s vnější scope pomocí *capture* bloku
  - předávání hodnotou (kopií) (`=`) nebo referencí (`&`)
  - předávání selektivně nebo všeho
- parametry funkce standardně
- návratová hodnota určena typem v šabloně, dedukována, popř. určena

Příklady

- bezparametrická, bez capture, bez návratové hodnoty

```
std::function<void()> fn = []() { };
```

- bezparametrická, capture všeho hodnotou, vrací integer

```
std::function<int()> fn = [=]() -> int { return 5; };
auto fn = [=]() { return (int)5; };
```

- parametry, capture specifický, návratová hodnota dedukovaná

```
auto fn = [a, b, &c, this](int p1, float p2) {
    return p1*a + b + c.x*p2 + getSomething();
};
```

Příklady

- často použití v konkrétním kontextu
- vláknová funkce

```
std::thread thr1([&]() {
    while (running)
        doSomething();
});
```

- predikát pro řazení

```
std::sort(a.begin(), a.end(), [](int x, int y) {
    return x < y;
});
```

- a další...

Vsuvka: std::bind

- v STL modulu *functional* dále možnost svázat volání funkce s parametrem
- hodí se např. pokud víme, že budeme určitou funkci volat stále se stejným parametrem
- pro ostatní parametry musíme při vytváření bindingu použít tzv. placeholder
- výsledný typ pak přetěžuje operátor() a odstiňuje funkční volání

Vsuvka: std::bind

- např. bind parametru funkce powf pro druhou mocninu - exponent vždy 2
- funkce `powf` má dva argumenty: základ (ten chceme proměnný) a exponent (ten chceme vždy 2)
- proto za základ vložíme placeholder a za exponent konstantu

```
auto druha_mocnina =
     std::bind(powf, std::placeholders::_1, 2.0f);
```

- nyní lze volat

```
// ekvivalent k powf(15.0f, 2.0f);
float kv = druha_mocnina(15.0f);
```

Vsuvka: std::bind

- nelze spolehlivě říct, kdy je lepší bind, a kdy funkce / lambda
- většinou jde o konkrétní případy
- hraje roli pouze čistota kódu a styl, než výkon nebo možnosti

Příklad

- Prostor pro příklad 10_b_lambda

Automatická dedukce typu

- klíčové slovo `auto`
- lze dedukovat typ proměnné, návratový typ funkce, šablonový typ, a další...
- nám bude stačit typ proměnné a návratový typ funkce
- někdy se velmi hodí - šetří psaní a pamatování
- např. u `std::bind` je to více než vhodné

Automatická dedukce typu

- dedukce typu proměnné
  - podle přiřazované hodnoty
  - podle návratové hodnoty funkce
- dedukce návratové hodnoty
  - podle `return` statementu
  - explicitním určením syntaxí s `->` (např. částečné dourčení typu, vhodné u lambda funkcí)

Automatická dedukce typu

- dedukce u proměnných je většinou úspěšná

```
auto val = 5; // dedukovan integer
auto valf = 1.2f // dedukovan float
auto str = "abcd"; // dedukovan const char*
auto kv = powf(5.0f, 2.0f); // dedukovan float
```

- u funkcí existují případy jednoduché (jeden return, není jiná možnost jak dedukovat)

```
auto getPositionX()
{
    return mPositionX;
}
```

Automatická dedukce typu

- ...i případy složitější

```
auto getSomething()
{
    if (condition)
        return nullptr;
    if (otherCondition)
        return mMember;
    return std::get<0>(sometuple);
}
```

- syntaxi lze doplnit o dedukci typu

```
auto getSomething() -> MyType*
```

- má smysl např. u lambda funkcí - šetří psaní std::function<..>

```
auto myLambda = []() -> int { return a ? 2 : 9.0; };
```

Příklad

- Prostor pro příklad 10_c_auto

Algoritmy STL knihovny

- `#include <algorithm>`
- obsahuje základní sadu algoritmů, která se může kdykoliv hodit
- jejich implementace je závislá na dodavateli standardní knihovny (ostatně jako vše ve standardní knihovně)
- můžeme se ale pravděpodobně spolehnout na kvalitní implementaci, co nejblíže optimu

Algoritmy STL knihovny

- algoritmů je přítomno spousty, budou uvedeny jen někteří zástupci
- práce s algoritmy pak odráží jednotné schéma
- často je vstupem rozsah iterátorů, parametr a např. funkce/funktor/lambda

Algoritmy STL knihovny

- `std::generate` - do daného rozsahu nageneruje danou funkcí hodnoty (náhrada za for cyklus)
- `std::shuffle` - náhodné rozmíchání daného rozsahu
- `std::move` - provede přesun obsahu objektu (rozsahu) do druhého (např. přesun vlastnictví unique_ptr
- `std::copy` - kopie objektu (rozsahu) do druhého
- `std::for_each` - nad zadaným rozsahem provede danou funkci
- `std::sort` - seřadí rozsah, můžeme dodat i funkci pro řazení

Algoritmy STL knihovny

- `std::count_if` - spočítá prvky splňující kritérium (funkce)
- `std::min_element`, `std::max_element` - najde minimální/maximální prvek
- `std::minmax_element` - sdružené předchozí funkce
- `std::prev_permutation`, `std::next_permutation` - generování permutací
- a mnoho dalších... `http://en.cppreference.com/w/cpp/header/algorithm`

Příklad

- Prostor pro příklad 10_d_stl

enum class

- klíčové slovo `class` u výčtového typu
- výčtový typ se tímto stane tzv. silně typovaný
- chová se proto jako samostatný typ, ne alias pro integer
- hodnotu je nutné vždy uvodit názvem výčtového typu

```
enum class MyEnum
{
    Value1,
    Value2,
    Value3
};

MyEnum val = MyEnum::Value1;
```

enum class

- lze však explicitně převést typ na integer a nazpátek, jde jen o syntaktickou kontrolu

```
enum class MyEnum
{
    Value1,
    Value2,
    Value3
};

int val = (int)MyEnum::Value1;
MyEnum val2 = (MyEnum)val;
```

Příklad

- Prostor pro příklad 10_e_enum_class

Range-based for

- "chytrý" for cyklus, který si je vědom rozsahů containerů
- nebo lépe.. je si vědom, že se často iteruje "od `begin()` až do `end()`"
- dovoluje tedy projít celý container užitím zkrácené syntaxe, např.

```
for (int &hodnota : intVektor)
{
    // do something
}
```

Range-based for

- zajímat nás budou obměny deklarace prvku (levá část)
- můžeme přejímat hodnoty hodnotou (kopií) a referencí
- některé containery vyžadují, aby byly hodnoty `const` (např. mapa nebo množina)
- lze použít automatickou dedukci typu, zda to bude reference/hodnota ale musíme rozhodnout my

```
for (auto hodnota : container)
{
    // do something
}

for (auto &hodnota : container)
{
    // do something
}
```

Range-based for

- `std::map` takto vrací instanci `std::pair`
- první prvek páru musí být vždy `const`
  - ze zjevných důvodů - přímá modifikace klíče by mohla poškodit integritu kontejneru

```
for (std::pair<const int, int> par : mapa)
{
    // do something
}
```

- je možná i const reference - pak ale nelze měnit ani hodnotu

```
for (std::pair<int, int> const& par : mapa)
{
    // do something
}
```

Range-based for

- `std::set` sice vrací prvek jako takový, ale vždy kopií nebo `const` referencí
- opět ze zjevných důvodů - prvek je klíčován sám sebou

```
for (int par : mapa)
{
    // do something
}

for (int const& par : mapa)
{
    // do something
}
```

Příklad

- Prostor pro příklad 10_f_range_based_for

Konec

- C++ jazyk a STL jsou obří, zdaleka jsme neobsáhli všechno
- další studium, zajímavé věci...

Další studium

- regulární výrazy
- synchronizace vláken - čtenář-písař, různé atomic containery
- další možnosti random enginů
- zbytek `<algorithm>` hlavičky
- šablonové porno (static_assert, variable and variadic templates, ..)
- RTTI, decltype a typeid
- UTF-8
- user-defined literals
- zbytek `std::chrono`

Další studium

- atributy ([[deprecated]], ..)
- multimap, multiset
- C++17
- ...

Konec 10. části

```
std::cout « "Děkuji za pozornost" « std::endl;
exit(0);
```