KIV/CPP – Programování v jazyce C++

09. Koncepty, korutiny, moduly

Martin Úbl

KIV ZČU

2023/2024

Koncepty

- rozšíření typového a šablonového systému o možnost pojmenovat požadavky na typ (od C++20)
- definice konceptu
  - klíčové slovo `concept`
  - vždy ve spojení s šablonovou deklarací
  - lze použít již hotové koncepty jako základ
  - lze definovat pokročilejší požadavky
    - např. „má metodu `xyz()` která vrací `double`“
  - lze skládat dohromady jako logické výrazy (`&&`, `||`)
- aplikace konceptu
  - před identifikátorem typu (šablonový typ nebo `auto`)
  - v definici šablonových parametrů

Koncepty

- příklad – jednoduchý koncept, požadavek na typ, že jde o znaménkovou celočíselnou hodnotu
- definice:

```
template<typename T>
concept SignedInteger = std::is_integral_v<T>
                        && std::is_signed_v<T>;
```

- použití 1: „*abbreviated function template*“ (asi nejvíce přehledné):

```
SignedInteger auto Neg(SignedInteger auto in) {
  return -in;
}
```

Koncepty

- použití 2: „constrained template parameter“:

```
template<SignedInteger T>
T Neg(T in) {
  return -in;
}
```

- tyto způsoby zatím neměly tvar tzv. *constraintu* („omezení“), ač o constraint fakticky jde
- *constraints* je princip úzce související s *concepts*
  - dá se říct, že jde o aplikaci *concepts* v praxi

Koncepty

- klíčové slovo `requires` – definuje tzv. *constraint*
- použití 3 (`requires` – constraint na šablonový parametr):

```
template<typename T>
  requires SignedInteger<T>
T Neg(T in) {
  return -in;
}
```

- použití 4 (`requires` – constraint na funkci):

```
template<typename T>
T Neg(T in) requires SignedInteger<T> {
  return -in;
}
```

Koncepty

- pro porovnání – stejná konstrukce pro SFINAE bez konceptů:

```
template<class T,
         class = typename std::enable_if<
                          std::is_integral_v<T>
                       && std::is_signed_v<T>>
                    ::type >
T Negate(T in)
{
  return -in;
}
```

- obtížně čitelný zápis – je třeba znát SFINAE pro pochopení zápisu
- existuje více naprosto odlišných způsobů, jak SFINAE realizovat
- oproti tomu jsou koncepty daleko čistější řešení

Koncepty

- definice konceptu
- (1) složením již existujících konceptů nebo `constexpr` definic převeditelných na `bool`

```
template<typename T>
concept SignedInteger = std::is_integral_v<T>
                     && std::is_signed_v<T>;

template<typename T>
concept SignedLargeInteger = SignedInteger<T>
                          && sizeof(T) > 4;
```

Koncepty

- definice konceptu
- (2) vytvořením sady požadavků, klíčové slovo `requires`

```
template<typename T>
concept HasToString = requires(T t) {
  { t.to_string() }
    -> std::convertible_to<std::to_string>;
};
```

- „typ T má metodu `to_string()`, jejíž návratová hodnota je převeditelná na `std::string`“

Koncepty

- definice konceptu
- (2) vytvořením sady požadavků, klíčové slovo `requires`

```
template<typename T>
concept Addable = requires(T t) {
  t + t;
};
```

- „typ T má definovaný operátor pro sčítání se sebou samým“

```
template<typename T>
concept AddableWithSelf = requires(T t) {
  { t + t } -> std::convertible_to<T>;
};
```

- „typ T má definovaný operátor pro sčítání se sebou samým a výsledkem je typ převeditelný na T“

Koncepty

- definice konceptu
- může být sama šablonově parametrizovaná

```
template<typename T, typename U>
concept MultipliableWith = requires(T t, U u) {
  t * u;
};
```

- „typ T má definovaný operátor pro násobení s typem U“

```
double InchToMeters(MultipliableWith<double>
                                    auto in) {
  return in * 0.0254;
}
```

Koncepty

- koncept definovaný *constraintem* lze chápat i jako „když napíšu tohle, tak se to přeloží“

```
template<typename T>
concept C = requires(const char* u) {
  T{u};
};
```

- „když budu mít typ `T` a budu ho chtít takhle vykonstruovat z `const char*`, tak to půjde“

Koncepty

- více požadavků (*constraintů*) v jednom konceptu
- odděleno středníky

```
template<typename T>
concept AddableMultipliable =
  std::convertible_to<T, double>
  && requires(T t) {

    { t + t } -> std::convertible_to<T>;
    { t * t } -> std::convertible_to<T>;
};
```

- „typ T je převeditelný na double a má definované operátory sčítání a násobení, jejichž výsledek je převeditelný na T“

Koncepty

- proč je potřeba říci „je převeditelný“?
- potřebujeme úplné sémantické vyjádření – za šipkou je uvedeno něco, co se musí v čase kompilace přepsat na `true` nebo `false`

```
{ t + t } -> T  // nefunguje
{ t + t } -> std::same_as<T>;
{ t + t } -> std::convertible_to<T>;
```

Koncepty

- řetězení konceptů a přetěžování funkcí
- mějme koncepty A a B
  - koncept B rozšiřuje koncept A
  - koncept B nepřidává konfliktní požadavky
  - pak je koncept B tzv. *silnější* než koncept A
- můžeme pak definovat přetížené funkce pro koncepty A a B zároveň
- kompilátor pak vybere nejsilnější dostupný koncept

Koncepty

```
template<typename T>
concept SignedInteger = std::is_integral_v<T> && std::is_signed_v<T>;

template<typename T>
concept SignedLargeInteger = SignedInteger<T> && sizeof(T) > 4;

void Test1(SignedInteger auto a) {
  std::cout << "SignedInteger: " << a << std::endl;
}

void Test1(SignedLargeInteger auto a) {
  std::cout << "SignedLargeInteger: " << a << std::endl;
}

// ...

Test1(55);
Test1(99LL);

// SignedInteger: 55
// SignedLargeInteger: 99
```

Koncepty

- koncept může pro některé případy suplovat prvek *rozhraní*
  - např. pokud bychom měli rozhraní kvůli jedné nebo dvěma konkrétním implementacím
- ten by se v C++ jinak suploval abstraktními třídami
- abstraktní třídy jsou ale prvek dynamického polymorfismu
- koncepty (a šablony) jsou čistě záležitost kompilace
- o něco srozumitelnější, než statický polymorfismus/CRTP
- lze kombinovat genericky s variadickými šablonami

Korutiny

- korutiny (od C++20)
- částečně podpora v jazyce, částečně ve standardní knihovně
- korutiny jsou takové funkce, které lze pozastavit na domluveném místě
- generují výsledky průběžně
- nová klíčová slova – syntakticky jde o operátory
  - `co_yield` – pozastaví korutinu a vrátí výsledek
  - `co_await` – předá řízení jiné korutině s úmyslem v budoucnu pokračovat s vykonáváním současné
  - `co_return` – ukončí korutinu a vrací výsledek
- bohužel implementace logiky je čistě stylem „udělej si sám“
  - ve standardu C++23 se snad dočkáme i konkrétní podpory

Korutiny

- korutiny
- „ta hezčí část“ I. – generátor hodnot

```
generator<int> counter()
{
  for (int i = 0; i < 3; i++) {
    std::cout << "Yielding:␣" << i << std::endl;
    co_yield i;
  }
}

auto gen = counter();
while (gen)
  std::cout << "Receiving:␣" << gen() << std::endl;

// Yielding: 0
// Received: 0
// Yielding: 1
// Received: 1
// Yielding: 2
// Received: 2
```

Korutiny

- korutiny
- „ta hezčí část“ II. – odložený výpočet/task

```
lazy<int> lazy_task() {

   int result;

   // ...

   co_return result;
}
```

Korutiny

- korutiny
- „ta hezčí část“ III. – uspatelná funkce nějaké zpracovací smyčky

```
task<> lazy_task() {
  Buffer buf;
  while (running) {
    // precti ze vstupu
    size_t count = co_await ReadFile(file, buf);
    // zpracuj
    Process(buf);
    // zapis na vystup
    co_await WriteFile(file, buf);
  }
}
```

Korutiny

- korutiny
- síla spočívá v tom, že jde de facto stále o jedno a totéž funkční volání
- např. zmíněná zpracovací smyčka
  - `lazy_task`, `ReadFile` a `WriteFile` jsou také korutiny
  - dá se to brát tak, že všechny funkce jsou zavolány právě jednou
  - jen se pokaždé „posune“ stav provádění
- korutiny nikdy neběží paralelně
  - předávají si řízení naprosto vědomě (kooperativní multitasking)
  - provádění končí, když:
    - skončí platnost příslušného `coroutine_handle`
    - korutina zavolá `co_return`
    - je vyhozena výjimka

Korutiny

- korutiny
- bohužel typ `generator`, `lazy` a `task` je potřeba dodat
- standard definuje požadavky na metody a členské atributy a typy
  - `promise_type` – typ synchronizační promise
    - definuje funkce pro návratovou hodnotu korutiny (generátor)
    - předávání výjimek vzniklých při generování
    - předávání návratové hodnoty (samotná hodnota)
    - a jiné...
- typicky pak chceme nějaké rozhraní pro přístup k prvkům – to si definujeme sami
- některé kompilátory a implementace standardních knihoven už obsahují nějaké pokusné implementace
  - clang/stdc++ – `std::experimental::task`
  - MSVS – `std::experimental::generator`
  - s největší pravděpodobností v nějaké podobě v C++23

Korutiny

- `promise_type` – požadavky na definici
  - `get_return_object()` – definuje návratovou hodnotu korutiny
  - `initial_suspend()` – definuje suspend politiku při spuštění korutiny
  - `final_suspend()` – definuje suspend politiku při ukončení korutiny
  - `yield_value(Typ hodnota)` – volá se v `co_yield` pro předání hodnoty
  - `unhandled_exception()` – volá se když dojde k neošetřené výjimce
  - `return_void()` – pro volání `co_return` bez hodnoty
  - `return_value(Typ hodnota)` – pro volání `co_return` s hodnotou
    - pokud korutina v sobě obsahuje `co_return`, musí být definována právě jedna z metod `return_void` a `return_value`

- třída, nad kterou lze volat `co_await`
  - `await_ready` – definuje suspend politiku v momentě volání `co_await`
  - `await_suspend` – volá se v momentě pozastavení korutiny při volání `co_await`
  - `await_resume` – volá se v momentě obnovení běhu (volaná korutina vrátila hodnotu); návratová hodnota je výsledkem operátoru `co_await`

Korutiny

- příklad generátoru, část I.

```
template<typename T>
struct generator {
  struct promise_type;
  using handle_type = std::coroutine_handle<promise_type>;

  struct promise_type {
    T value_;
    std::exception_ptr exception_;

    generator get_return_object() {
      return generator(handle_type::from_promise(*this));
    }
    std::suspend_always initial_suspend() { return {}; }
    std::suspend_always final_suspend() noexcept { return {}; }
    void unhandled_exception() { exception_ = std::current_exception(); }

    template<std::convertible_to<T> From>
    std::suspend_always yield_value(From&& from) {
      value_ = std::forward<From>(from);
      return {};
    }
    void return_void() {}
  };

  handle_type h_;

  generator(handle_type h) : h_(h) {}
  ~generator() { h_.destroy(); }
```

- příklad generátoru, část II.

```
  explicit operator bool() {
    fill();
    return !h_.done();
  }
  T operator()() {
    fill();
    full_ = false;
    return std::move(h_.promise().value_);
  }

private:
  bool full_ = false;

  void fill() {
    if (!full_) {
      h_();
      if (h_.promise().exception_)
        std::rethrow_exception(h_.promise().exception_);
      full_ = true;
    }
  }
};
```

Korutiny

- vyčerpávající seznam požadavků vč. sémantiky:
  - `https://en.cppreference.com/w/cpp/language/coroutines`
- v tomto předmětu prozatím korutiny více probírat nebudeme
- mj. i proto, že v okamžiku přípravy této prezentace žádný kompilátor neměl kompletní podporu
- a také proto, že standardní knihovna nemá podpůrné struktury pro pohodlnou implementaci

Korutiny

- proč korutiny, a proč ne `std::async` (`std::future`)?
- `std::async` při opakovaném volání přidává velké množství režie
  - korutiny drží právě jeden kontext, který se jen pozastavuje
  - `std::async` vytváří a ničí kontext při každém volání
- korutiny jsou efektivnější
- ovšem alespoň prozatím nejsou přehlednější

Korutiny

- zpohodlnění práce a suplování toho, co standardní knihovna (zatím) neposkytuje
- `https://github.com/lewissbaker/cppcoro`
  - implementace generátorů, tasků a jiných
  - awaitable mutexy, semafory, ...
  - implementace různých asynchronních I/O operací ve stylu korutin (aby se nad nimi dalo použít `co_await`)

Korutiny

- cvičení:
  - pokus o vlastní generátor

Moduly

- *Modules* (od C++20)
- samostatně kompilovatelné jednotky
- logické oddělení komponent
- jistým způsobem sdružuje do jednoho mechanismu konzistentně to, co by se jinak mohlo definovat kdekoliv a implementovat také kdekoliv
  - deklarace (dopředná) může být vlastně kdekoliv, hlavičkový soubor je jen velmi obvyklé místo, kam ji dát
  - implementace rovněž stačí, aby byla „někde“ (kde ji najde linker)
- moduly mají rozhraní a implementaci

Moduly

- *Modules* (od C++20)
- moduly se kompilují a includují právě jednou
- oproti původnímu způsobu s preprocesorovým makrem
  - `#include <hlavicka.h>`
  - uvnitř buď `#pragma once` nebo makro zabraňující vícenásobnému include
- částečně nahrazují *precompiled headers* mechanismus

Moduly

- *Modules* (od C++20)
- rozhraní modulů
  - nahrazují to, co konvenčně byly hlavičkové soubory
  - obsahují klauzuli `export module Jmeno`
  - mají vlastní „typ“ souboru
    - přípona `.ixx` v MSVS
    - přípona `.cppm` v clang a GCC
- implementace modulů
  - nahrazují to, co konvenčně byl nějaký zdrojový soubor, co se jmenoval stejně, jako hlavičkový
  - obsahují klauzuli `module Jmeno`
  - je již klasicky v `.cpp` souboru

Moduly

- základní příklad modulu
- rozhraní, `calc.ixx`

```
export module calc;

export double pow2(double in);
```

- implementace, `calc.cpp`

```
module calc;

double pow2(double in) {
  return in * in;
}
```

Moduly

- co lze exportovat?
  - funkce
  - třídy
  - jmenné prostory
  - typové aliasy
  - proměnné
  - atd...
- co nelze exportovat?
  - anonymní jmenné prostory
  - statické proměnné
  - cokoliv, co je předem deklarováno jako neexportované

Moduly

- příklad modulu – export třídy
- rozhraní, `calc.ixx`

```
export module calc;

export class calculator {
  public:
    static double pow2(double in);
};
```

- implementace, `calc.cpp`

```
module calc;

double calculator::pow2(double in) {
  return in * in;
}
```

Moduly

- module partitioning
- moduly lze v podstatě hierarchicky rozdělit na části
- struktura importování se dá připodobnit k includování
- jen pochopitelně benefituje z výhod modulů
- *partition* je uvozena dvojtečkou
  - pokud se hlavní modul jmenuje `test`, pak jeho partitions mohou být `test:first`, `test:second`, ...
- neplést s tečkou – tu lze použít pro jmenné oddělení, ne pro partitioning

Moduly

- příklad
- rozhraní modulu, `calc_basic.ixx`

```
export module calc:basic;

export double Add(double a, double b);
export double Sub(double a, double b);
```

- rozhraní modulu, `calc_powers.ixx`

```
export module calc:powers;

export double Pow2(double a);
export double Pow3(double a);
```

Moduly

- příklad
- rozhraní modulu, `calc.ixx`

```
export module calc;

export import :basic;
export import :powers;
```

- implementaci modulů `calc_basic.cpp` a `calc_powers.cpp` si lze snadno domyslet
- syntaxe `export import Jmeno` znamená to, že importujeme modul (nebo partition) s daným názvem a jím exportované rozhraní exportujeme i ze současného modulu

Moduly

- implicitní export
- pokud je exportován jmenný prostor, jsou implicitně exportovány jeho vnitřnosti
- analogicky pokud je exportována některá z vnitřností jmenného prostoru, je implicitně exportován i jmenný prostor

```
export namespace ns {
  void something() { /* ... */ }
}

namespace ns {
  export void something() { /* ... */ }
}
```

Moduly

- importování
- v modulech může `import` být před jakoukoliv modulovou deklarací
- `import` může být pouze na globální scope (nelze importovat až v těle funkce, apod.)
- `import`ovat lze pouze partitions, které náleží danému modulu
- `import` nemůže importovat tentýž modul, ve kterém se nachází

Moduly

- viditelnost vs. dosažitelnost
- viditelný identifikátor = lze ho najít jeho jménem
- dosažitelný identifikátor = lze použít to, co označuje
- modul exportuje funkci, která vrací instanci třídy, ale neexportuje třídu
  - funkce je viditelná
  - třída je pouze dosažitelná

```
class PrivateClass {
  // ...
};

export PrivateClass create_priv() {
  return PrivateClass{ };
}
```

Moduly

- lze zavolat funkci a manipulovat s vrácenou instancí, jako kdyby byl typ viditelný
- nelze ale instancovat třídu jejím identifikátorem
- trikem však lze vytvořit další instance této třídy

```
// v poradku, 'p' bude typu PrivateClass
auto p = create_priv();

// spatne! PrivateClass nevidime
PrivateClass pp{};

using SecretClass = decltype(p);
// v poradku, 'q' ma stejny typ jako 'p'
SecretClass q{};
```

Moduly

- globální modul
- jakmile v C++ začneme používat moduly, všechno musí být součástí nějakého modulu
- implicitně je definován globální modul
  - nemá jméno
  - nelze ho importovat
  - implicitně do něj patří vše, co je deklarováno mimo modul
  - funkce `main()` smí být pouze zde

Moduly

- header-unit import
- moduly počítají s tím, že budeme chtít „modularizovat“ něco, co modul není
- např. již existující pár hlavičkového souboru a odpovídajícího zdrojového souboru
- pak stačí importovat hlavičkový soubor

```
export module security;

import <openssl/ssl.h>;
```

- hlavičkový soubor je includovaný a všechny jeho interní definice jsou exportovány

Moduly

- header-unit import
- pozor – importy nejsou ovlivněny definovanými makry
- nelze tedy hlavičkový soubor „parametrizovat“ stavem preprocesoru

```
export module winmodule;

#define _UNICODE
import <windows.h>;
```

- moduly (module unit) se linkují odlišně a jinde, makro `_UNICODE` zde tedy nebude mít vliv

Moduly

- od C++23 je v plánu standardní knihovnu modularizovat
- pak pro ni nebude potřeba header-unit import
- kompilace by se měla zrychlit
- MSVS poskytuje část standardní knihovny jako moduly už teď:
  - je potřeba explicitně doinstalovat
  - `import std.core;`
    - `std.threading`, `std.regex`, `std.filesystem`, ...
  - `https://docs.microsoft.com/en-us/cpp/cpp/modules-cpp`

Moduly

- moduly jsou součástí snahy eliminovat potřebu preprocesoru
- mj. do této snahy patří:
  - `constexpr`, `enum class` – náhrada `#define` pro konstanty
  - `if constexpr` – náhrada `#if` a `#ifdef`
  - šablony, koncepty – náhrada textové substituce kódu
- preprocesor má „vlastní jazyk“ nad samotným C/C++
- potenciálně komplikuje překlad, orientaci v kódu, škálovatelnost a bezpečnost
  - např. MSVS a implicitně definované makro `min` (`max`)
  - kompilátor vyhodí silně nespecifickou chybu, pokud někde v kódu použijete `std::min`
  - dojde k nahrazení `min` za obsah makra, tedy vlastně vložení ternárního operátoru
  - kompilace selže, protože `std::(x>y?y:x)` nelze zkompilovat

Moduly

- obecné problémy preprocesoru:
  - nemá sémantiku
  - nezná datové typy
  - mezery a bílé znaky mohou rozbít kód
  - neúčastní se kompilace, neumí „komunikovat“ s kódem, který generuje
- stále je ale potřeba
  - před kompletní podporou modules
  - pro verzování vybraných multitargeting knihoven
    - např. `#define CL_TARGET_OPENCL_VERSION 120`
  - kompatibilita se zdrojovým kódem jazyka C při sdílení kódu
  - určitá část debugging procesu
    - `__FILE__`, `__LINE__`, ...