

VYBRANÉ FUNKCIONALITY STANDARDU C++17

Studijní materiály

OBSAH

1	ÚVOD	3
2	POPIS VYBRANÝCH ZMĚN	4
2.1	ODEBRANÉ FUNKCE.....	4
2.1.1	auto_ptr	4
2.1.2	register.....	4
2.1.3	bool operator++	4
2.2	NOVÉ FUNKCIONALITY A MODIFIKACE JAZYKA	4
2.2.1	Structured bindings	4
2.2.2	Inicializace <i>if</i> a <i>switch</i>	6
2.2.3	Inline proměnné	6
2.2.4	Vnořený (nested) namespace	8
2.2.5	Dedukce argumentů šablony třídy (Class template argument deduction, CTAD).....	8
2.2.6	Paralelní algoritmy	9
3	ZADÁNÍ SAMOSTATNÉ PRÁCE	10

1 ÚVOD

Jelikož standard C++14 byl v porovnání se standardem C++11 spíše “bug fix“, standard C++17 měl větší ambice. Několik funkcionalit nebylo včas dokončeno a bylo přesunuto do standardu C++20, ale C++17 přesto přináší mnoho nových funkcí. Mezi nejhlavnější patří např. paralelní algoritmy, structured binding nebo dedukce argumentů šablony třídy.

2 POPIS VYBRANÝCH ZMĚN

2.1 Odebrané funkce

Přestože C++ má za cíl být se svými předchozími verzemi kompatibilní, některé funkce jazyka mohou být málokdy používané nebo dokonce chybné. Standard C++17 proto některé funkce odstraňuje.

2.1.1 `auto_ptr`

V C++11 byl `auto_ptr` nahrazen chytrými ukazateli a move semantics, ovšem odstraněn byl až v C++17.

2.1.2 `register`

V jazyku C je *register* klíčové slovo, které říká že proměnná by se preferovaně měla uložit do registru místo do RAM (ovšem zda se tak opravdu stane záleží na implementaci překladače; některé *register* zcela ignorují.) Klíčové slovo přestalo být podporováno už od C++11, ovšem od C++17 je konečně odstraněno. Slovo *register* je stále zarezervováno, a je tedy možné že v budoucnu bude využito pro jinou funkcionalitu.

2.1.3 `bool operator++`

Používání post-increment operátoru `++` na typu *bool* bylo považováno za zastaralé již v roce 1998 když byl odsouhlasen první C++ standard, ale přesto bylo v jazyce ponecháno. Od standardu C++17 je již tento operátor odstraněn.

2.2 Nové funkcionality a modifikace jazyka

2.2.1 Structured bindings

C++17 umožňuje nadefinovat množinu proměnných navázáním na jednotlivé členy předávaného objektu. Tomuto říkáme *structured binding*, neboli strukturované vazby. Obecný zápis je následovný:

auto [identifier-list] = expression ;

Structured binding nadeklaruje všechny proměnné v *identifier-list* v obalujícím jmenném prostoru, a naváže je postupně na objekty obsažené v *expression*. *Identifier-list* musí obsahovat přesně počet proměnných které vrací *expression*.

Expression může být těchto typů:

1. Pole
2. Struktury a třídy. Všechny nestatické členy ovšem musí být členy stejné rodičovské třídy.
3. N-tice, která implementuje funkce `get<N>()`, `std::tuple_size<typ>` a `std::tuple_element<N, type>`.

Structured binding na pozadí funguje tak, že do skryté proměnné uloží celý objekt, který vrací *expression*. Proměnné v *inicializer-list* se poté stanou aliasy (tedy nové jména pro již existující proměnné) na členy tohoto skrytého objektu.

```
auto temp = expression;
```

```
using x = temp.first;
```

```
using y = temp.second;
```

```
using z = temp.third;
```

V následujícím kódu se tedy nainicializuje x jako 0, a y jako prázdný řetězec. Pokud by *Struktura* obsahovala pouze jednu proměnnou, nebo by v hranatých závorkách za *auto* bylo pouze x, program by se nepřeložil.

```
struct Struktura {
```

```
    int i = 0;
```

```
    std::string s;
```

```
};
```

```
Struktura str;
```

```
auto [x,y] = str;    // x=0; y=""
```

2.2.1.1 Modifikátory

Na structured binding můžeme používat modifikátor `const`, nebo získat reference. Nemůžeme ovšem použít `constexpr`.

```
const auto [a, b, c, ...] = ex;
```

```
auto& [a, b, c, ...] = ex;
```

```
auto&& [a, b, c, ...] = ex;
```

```
constexpr auto [x, y] = std::pair(0, 0);    //error
```

2.2.2 Inicializace *if* a *switch*

C++17 přináší nový styl zápisu podmínek *if* a *switch*, kdy v závorce můžeme uvést nejen podmínku, ale nově i inicializovat proměnné. Inicializované hodnoty musí být vždy přiřazeny k proměnné, jinak budou ihned zahozeny.

```
if (inicializace; podminka){};
```

```
switch (inicializace; podminka){};
```

Pro podmínku *if* inicializace platí do konce *then*, popřípadě *else* bloku. Pro podmínku *switch* platí inicializace do konce *switch* bloku.

```
if (int x = geValue(); x > 1) {}
```

```
else {}
```

```
// x již není nadefinováno
```

2.2.3 Inline proměnné

V C++ není možno inicializovat členské proměnné třídy v místě jejich definice, pokud jsou

- nekonstantní a statické
- nebo pokud je jejich definice v hlavičkovém souboru.

Toto chování vychází z C++ Pravidla jedné definice (One Definition Rule, ODR), které říká, že objekty a proměnné mohou definovány pouze na jednom místě. Pokud by dva různé

soubory includovaly hlavičkový soubor, oba by definovaly stejné proměnné třídy, a pravidlo by bylo porušeno.

Tento problém lze vyřešit pomocí nových *inline proměnných*. Pokud členskou proměnnou třídy označíme jako *inline*, můžeme mít v hlavičkovém souboru nadefinovaný objekt, který lze globálně použít ve všech CPP souborech. Inicializace je provedena při překladu první instance definice.

```
class ExampleClass {  
  
    static inline int x = 1;  
  
};  
  
inline ExampleClass globalniObjekt;
```

2.2.3.1 constexpr a inline

Pro statické členské proměnné nově platí, že modifikátor *constexpr* v sobě zahrnuje i *inline*. Následující dvě definice budou tedy totožné:

```
struct S {  
  
    static constexpr int x = 0;  
  
};  
  
struct S {  
  
    inline static constexpr int x = 0;  
  
};
```

2.2.4 Vnořený (nested) namespace

Vnořené namespaces byly poprvé navrženy v roce 2003, ovšem nadefinovány byly až v C++17. Umožňují nadefinovat namespace uvnitř jiného namespace, ovšem vnořený namespace nesmí být *inline*. Vnořené namespacey mohou být nadefinovány následujícími dvěma ekvivalentními způsoby:

```
namespace A {  
    namespace B {  
        namespace C {  
            int i;  
        }  
    }  
}
```

```
namespace A::B::C {  
    int i;  
}
```

2.2.5 Dedukce argumentů šablony třídy (Class template argument deduction, CTAD)

Dříve pokud jsme chtěli vytvořit instanci šablony třídy, museli jsme specifikovat datové typy všech argumentů. Od C++17 je překladač schopen odvodit si argumenty podle datového typu inicializátoru. Nelze ovšem použít pouze částečnou dedukci - pokud chceme, aby si překladač odvodil argumenty za nás, nesmíme tedy specifikovat žádný argument.

```
std::tuple<int,int> x(1, 2);           // Před C++17  
std::tuple y(1, 2);                   // Od C++17  
std::tuple<int> y(1, 2);               // Error: částečná dedukce
```


2.2.6 Paralelní algoritmy

C++17 umožňuje většinu algoritmů ve standardní knihovně paralelizovat – tedy pro svůj běh budou využívat více vláken. Algoritmu můžeme přidat nový parametr nazvaný *execution policy*, který říká, jakým způsobem se má algoritmus spustit.

```
std::sort(std::execution::par, v.begin(), v.end());
```

K dispozici máme:

- `std::execution::seq` – sekvenční běh, algoritmus provádí operace krok po kroku na jednom vlákne
- `std::execution::par` – paralelní sekvenční běh, několik vláken může provádět operace nad různými prvky. Nad celým kontejnerem prvků může pracovat více vláken, ale jedno vlákno zpracovává jeden prvek.
- `std::execution::par_unseq` - paralelní nesequenční běh je podobný jako paralelní sekvenční, ale vlákno nemusí pro prvek dodělat všechny potřebné operace než začne pracovat nad jiným prvkem

Paralelní algoritmy mohou být rychlejší než jejich sekvenční alternativy, ovšem není tomu tak vždy. Je to kvůli synchronizaci na sdíleném objektu, sdílení paměťové sběrnice mezi jádru CPU, a tomu že paralelní algoritmus musí data pro paralelismus připravit.

Překladač Clang paralelní algoritmy prozatím nepodporuje, je nutno použít GCC nebo MSVC.

3 ZADÁNÍ SAMOSTATNÉ PRÁCE

- Vyberte si algoritmus a porovnejte rychlosti jeho sekvenční, paralelní sekvenční a paralelní nesequenční verze.
- Proveďte statistiku – každou verzi algoritmu několikrát spustěte pro různé velikosti dat, na konzoli vypište jednotlivé časy a průměrný čas každé verze algoritmu.
- Pro časování můžete využít knihovny <chrono>