

VYBRANÉ FUNKCIONALITY STANDARDU C++20

Studijní materiály

OBSAH

1	ÚVOD	3
2	POPIS VYBRANÝCH ZMĚN	4
2.1	KONCEPTY	4
2.2	KNIHOVNA RANGES	5
2.3	MODULY	5
2.4	KORUTINY	6
3	ZADÁNÍ SAMOSTATNÉ PRÁCE	8

1 ÚVOD

Přestože komise dříve zamýšlela střídat “velké” a “malé” standardy, a dle toho měl tedy po C++17 následovat spíše standard opravující chyby podobně jako standard C++14, komise od tohoto upustila a standard C++20 přináší více funkcionalit než předchozí standard C++17.

Mezi nejvýznamnější nové funkcionality patří korutiny, moduly, koncepty a ranges, označované jako “Velká čtyřka” (The Big Four). Tyto funkcionality byly plánovány již pro minulé standardy, ovšem komise potřebovala více času pro jejich dokončení, a tak je konečně dostáváme ve standardu C++20.

2 POPIS VYBRANÝCH ZMĚN

2.1 Koncepty

Koncepty jsou novým C++20 nástrojem generického programování. Umožňují k parametrům šablon nadefinovat jejich omezení, která kontroluje překladač před vytvořením instance. Díky konceptům se požadavky na parametry šablon stanou členy veřejného rozhraní a dostáváme lepší chybové zprávy.

Koncepty mají následující zápis:

```
template < seznam-parametru >
```

```
concept NazevOmezeni = omezeni;
```

Pokud chceme omezit šablonu pomocí konceptu, můžeme použít jeden z několika zápisů:

1. *template<NazevOmezeni T>*
void Foo(T t);
2. *template<typename T> requires NazevOmezeni <T>*
void Foo(T t);
3. *template<typename T>*
void Foo(T t) requires NazevOmezeni <T>;
4. *void Foo(NazevOmezeni auto t);*

Koncepty lze také mezi sebou dvěma způsoby kombinovat:

1. *template<typename T> requires NazevOmezeni<T> && DalsiOmezeni<T>*
void Foo(T t);
2. *template<typename T>*
concept C = NazevOmezeni<T> && DalsiOmezeni<T>;
...
void Foo(C auto t);

2.2 Knihovna Ranges

Ranges jsou objekty, které reprezentují sekvenci prvků. Ranges knihovna rozšiřuje knihovny algoritmů a iterátorů, a umožňuje zavolat funkci nad celým rozsahem bez nutnosti specifikovat začátek a konec sekvence. Můžeme také modifikovat prvky kontejneru před tím, než je pošleme do algoritmu.

Porovnání zápisů:

```
std::sort(begin(vector), end(vector));           // dřívější zápis
```

```
std::ranges::sort(vector);                     // nový zápis
```

2.3 Moduly

Moduly představují alternativu k hlavičkovým souborům a poskytují možnost sdílet deklarace a definice napříč soubory. Rozhraní a implementace modulu můžou ale nemusí být v oddělených souborech, ovšem exportovat modul může pouze jeden soubor.

U modulů musíme pomocí slova `export` explicitně uvést, které funkcionality chceme zdostupnit uživatelům.

Exportování funkce:

```
void func() {}                                //private funkce
```

```
export void func2() {}                        //veřejná funkce dostupná uživateli
```

Modul importujeme pomocí klíčového slova *import*:

```
import NazevModulu;
```

Výhodami modulů je, že jsou importovány pouze jednou, na pořadí jejich importování nezáleží, nemusí obsahovat unikátní jména, mohou být strukturovány do částí a podmodulů, jsou zpracovány pouze jednou a mohou tak urychlit překlad, makra preprocesoru vně modulu nemají na modul vliv, a makra preprocesoru v konkrétním modulu nemají vliv na program vně modulu.

2.4 Korutiny

Korutiny jsou funkce, jejichž běh můžeme pozastavit a obnovit, přičemž si funkce zachovává svůj stav uložený na haldě. Korutiny pomáhají k tvorbě aplikací řízených událostmi, jako jsou simulace, hry, uživatelská prostředí atd, a umožňují implementovat lazy evaluation.

C++20 neobsahuje přímo konkrétní korutiny, ale poskytuje framework pro implementaci korutin, nadefinovaný v hlavičkovém souboru `<coroutine>`. Knihovna pro jednodušší práci s korutinami by měla přijít s C++23.

Korutinou nesmí být konstruktory a destruktory, funkce označené jako *constexpr*, funkce s proměnným počtem argumentů, funkce vracející *auto* nebo *concept*, a funkce *main*.

Aby se funkce stala korutinou, musí používat jedno z klíčových slov *co_await*, *co_yield* nebo *co_return*.

- *co_await* pozastavuje běh korutiny
- *co_yield* pozastavuje funkci na příkazu, který vrací hodnotu z funkce, dá si ji představit jako ykratku pro *co_await promise.yield_value(value)*
- *co_return* dokončuje provedení příkazu vracejícího hodnotu.

Návratový typ korutiny si musíme nadefinovat sami. Typ musí obsahovat *promise_type* - objekt který vrací hodnotu z korutiny a ovládá stav korutiny.

Příklad korutiny:

```
struct task {  
  
    struct promise_type {  
  
        task get_return_object() { return {}; }  
  
        std::suspend_never initial_suspend() { return {}; }  
  
        std::suspend_never final_suspend() noexcept { return {}; }  
  
        void return_void() {}  
  
        void unhandled_exception() {}  
  
    };  
  
};
```

```
};
```

```
};
```

```
task coroutine()
```

```
{
```

```
    //...
```

```
    co_return;
```

```
}
```

```
int main(){
```

```
    task t = coroutine();
```

```
}
```

3 ZADÁNÍ SAMOSTATNÉ PRÁCE

- Vytvořte strukturu *generator* pro generování čísel. Korutina bude obsahovat strukturu *promise_type*, obsahující
 - hodnotu *current_value*
 - funkci *yield_value*, díky které bude generátor moci využívat funkci *co_yield*
- Implementujte korutinu *Fibonacci* typu *generator*. Korutina bude přebírat parametr *n*, a bude generovat Fibonacciho posloupnost pro *n* prvků.
- Předved'te správné fungování korutiny.