Typový systém pro fyzikální jednotky

OPRAVA 1. DÚ Z POKROČILÉHO PROGRAMOVÁNÍ V C++

Podproblémy

- Operace nad statickým vektorem
  - Vytvoření jednotkového, sčítání a odčítání
- Implementace quantity
- Shoda `TEnum` u operátorů
  - **Bonus:** Shoda `TEnum` i u `multiplied_unit` a `divided_unit`
- Kontrola TEnum
- Čitelnost

Operace nad statickým vektorem

- Obecný přístup:
  - Pomocný push_front, rekurzivní volání

```
using type = typename push_front<0,typename create_vector<Size - 1, Index - 1>::type>::type;
```

  - Rekurzivní předávání mezivýsledku v šabloně

```
template<int Head1, int Head2, int ... DiffHead>
struct vector_diff<static_vector<Head1>, static_vector<Head2>, static_vector<DiffHead...>> {
    using type = static_vector<DiffHead..., Head1 - Head2>;
};
```

Operace nad statickým vektorem

- Obecný přístup:
  - Pomocný push_front, rekurzivní volání

```
using type = typename push_front<0,typename create_vector<Size - 1, Index - 1>::type>::type;
```

  - Rekurzivní předávání mezivýsledku v šabloně

```
template<int Head1, int Head2, int ... DiffHead>
struct vector_diff<static_vector<Head1>, static_vector<Head2>, static_vector<DiffHead...>> {
    using type = static_vector<DiffHead..., Head1 - Head2>;
};
```

  - `std::index_sequence` a unpacking (nešel pro všechny operace)

```
template<int ...A, int ...B, size_t ...I>
struct sum_two_vectors<static_vector<A ...>, static_vector<B ...>, std::index_sequence<I ...> > {
    using type = static_vector<(at<static_vector<A ...>, I>::value + at<static_vector<B ...>, I>::value)...>;
};
```

Operace nad statickým vektorem

- Rekurzivní vytvoření jednotkového vektoru, tj. <0,…,1,…,0>
  - Předání délky a indexu
  - Šlo vyřešit parciálními specializacemi, našly se ale i složitější řešení (mohlo by se hodit jinde):

```
template<int Size, int Index, int ... Values>
struct basis_vector{
    constexpr static int value(){
        if constexpr(Index == Size)
            return 1;
        else
            return 0;
    }
    using type = typename basis_vector<Size - 1, Index, value(), Values ...>::type;
};

template<int Index, int ... Values>
struct basis_vector<-1, Index, Values ...>{ using type = static_vector<Values ...>; };
```

Operace nad statickým vektorem

- Sčítání a odčítání
  - Běžné (a doporučené): Dvě samostatné “operace”
  - Možné zobecnění: Šablona pro obecný “zip”:

```
template <typename X, typename Y, typename Result, typename BinaryPredicate>
struct op;

template <typename T>
struct Plus {
  static consteval T op(T x, T y) {
    return x + y;
  };
};
```

Operace nad statickým vektorem

- Sčítání a odčítání
  - Elegantní řešení pomocí rozbalení variadických šablon:

```
template<typename LeftVector, typename RightVector>
struct add;

template<int ... LeftValues, int ... RightValues>
struct add<static_vector<LeftValues ...>, static_vector<RightValues ... >> {
    using type = static_vector<(LeftValues + RightValues) ... >;
};
```

Implementace quantity

- Field bylo mírně lepší dát privátní (immutable sémantika)
- Konstruktor explicitní, aby tam nešel omylem přiřadit skalár
- Operátory bylo možné definovat ve třídě i globálně
  - U první varianty nebylo potřeba psát tolik šablon

Shoda `TEnum` u operátorů

- Možnosti:
  - Navrhnout vhodně šablonu pro operátory:

```
template <typename TEnum, typename TPowers1,
          typename TPowers2, typename TValue>
auto operator*(quantity<unit<TEnum,TPowers1>,TValue> &a,
               quantity<unit<TEnum,TPowers2>,TValue> &b)
```

  - Použít requires/koncepty:

```
template <typename TUnit1, typename TUnit2, typename T>
requires std::is_same_v<typename TUnit1::enum_type,
                        typename TUnit2::enum_type>
```

Shoda `TEnum` u `multiplied_unit` a `divided_unit`

- Nebylo v zadání, ale v praxi by se mohlo hodit
  - Tj. nešlo by vůbec neplatný typ zkonstruovat, nejen od něj vytvořit `quantity`
- Zajímavější je `multiplied_unit` – variadická šablona
- Možné řešení: (šlo by i bez `requires`?)

```
template <typename TFirstUnit, typename... TOtherUnits>
requires (... && std::is_same_v<typename TFirstUnit::enum_type,
                                typename TOtherUnits::enum_type>)
using multiplied_unit = ...
```

Kontrola TEnum

- Šlo použít koncept:

```
 template<typename TEnum>
concept HasCount = requires (TEnum t) {
    std::is_enum_v<TEnum>;
    TEnum::_count;
};

template <HasCount TEnum, typename TPowers>
struct unit;
```

Čitelnost

- Vhodné odsazování složitějších instanciací šablon
- Pomocné typy (usingy)
- Zanoření implementací operací do vlastních typů
- Koncepty