0.1 Przeciążanie funkcji przy użyciu konceptów

Głowna idea programowania generycznego polega na używaniu tej samej nazwy dla równoważnych operacji używających różnych typów. A zatem, w grę wchodzi przeciążanie. Jest bardzo często przeoczaną, źle rozumianą ale niezwykle potężną cechą konceptów. Koncepty pozwalają na wybieranie spośród funkcji opierając się na właściwościach danych argumentów. Są przydatne nie tylko do poprawiania komunikatów o błędach i dokładnej specyfikacji interfejsów. Zwiększają również ekspresywność. Mogą być użyte do skracania kodu, robienia go bardziej ogólnym i zwiększania wydajności.

C++ jest językiem nie tylko assemblerowym wykorzystywanym do metaprogramowania szablonów. Koncepty pozwalają na podnoszenie poziomu programowania i upraszczają kod, bez angażowania dodatkowych zasobów czasu wykonania.

Przykład algorytmu *advance*[1] ze standardowej biblioteki

```
template<typename Iter> void advance(Iter p, int n);
```

Potrzeba różnych wersji tego algorytmu, m.in.

- prostej, dla iteratorów *Forward*, przechodzących przez sekwencję element po elemencie
- szybkiej, dla iteratorów *RandomAccess*, by wykorzystać umiejętność do zwiększania iteratora do arbitralnej pozycji w sekwencji używając jednej operacji.

Taka selekcja czasu kompilacji jest istotna dla wykonania kodu generycznego. Tradycyjnie, da się to zaimplementować używając funkcji pomocniczych lub techniki *Tag Dispatching*[2], lecz z konceptami rozwiązanie jest proste i oczywiste:

```
template<Forward_iterator F, int n> void advance(F f, int n){
    while(n--) ++f;
}
```

```
template<Random_access_iterator R, int n> void advance(R r, int n){
    r += n;
}
```

```
void test(vector<string> &v, list<string> &l){
    auto pv = find(v, "test"); //(1)
    advance(pv, 2);
```

[1] Algorytm *advance(it, n);* inkrementuje otrzymany iterator *it* o *n* elementów.

[2] Technika programowania generycznego polegająca na wykorzystaniu przeciążania funkcji w celu wybrania, którą implementację funkcji wywołać w czasie wykonania

```
    auto pl = find(l, "test"); //(2)
    advance(pl, 2);
}
```

1) użycie szybkiego `advance` 2) użycie wolnego `advance`

Skąd kompilator wie kiedy wywołać odpowiednią wersję `advance`? Rozwiązanie przeciążania bazującego na konceptach jest zasadniczo proste:

- jeśli funkcja spełnia wymagania tylko jednego konceptu - wywołaj ją
- jeśli funkcja nie spełnia wymagań żadnego konceptu wywołanie - błąd
- sprawdź czy funkcja spełnia wymagania dwóch konceptów - zobacz czy wymagania jednego konceptu są podzbiorem wymagań drugiego
  - jeśli tak - wywołaj funkcję z największą liczbą wymagań (najściślejszych wymagań)
  - jeśli nie - błąd (dwuznaczność)

W funkcji `test`, `Random_access_iterator` ma więcej wymagań niż `Forward_iterator`, więc wywołuje się szybka wersja `advance` dla iteratora zmiennej `vector`. Dla ietratora zmiennej `list`, pasuje tylko iterator *Forward*, więc używamy wolnej wersji `advance`.

`Random_access_iterator` jest bardziej określony niż `Forward_iterator` bo wymaga wszystkiego co `Forward_iterator` i dodatkowo operatorów takich jak `[]` i `+`.

Ważne jest to że nie musimy wyraźnie określać "hierarchii dziedziczenia" pośród konceptami czy definiować *klas traits*[3]. Kompilator przetwarza hierarchię dla użytkownika. To jest prostsze, bardziej elastyczne i mniej podatne na błędy.

Przeciążanie oparte na konceptach eliminuje znaczącą ilość *boiler-plate*[4] z programowania generycznego i kodu meta programowania (użycia `enable_if`[5]).

Funkcja `czyZnaleziono` ocenia czy element znaduje się w sekwencji

```
template<Sequence S, Equality_comparable T>
    requires Same_as<T, value_type_t<S>>
bool czyZnaleziono(const S& seq, const T& value){
    for(const S& seq, const T& value)
        if(x == value)
            return true;
```

[3] klasy traits
[4] BP
[5] EI

```
    return false;
}
```

Funkcja przyjmuje jako parametr sekwencję i wartość typu `Equality_comparable`. Algorytm ma 3 ograniczenia:

- typ parametru `seq` musi być typu `Sequence`
- typ parametru `value` musi być typu `Equality_comparable`
- typ wartości typu `S` musi być taki sam jak typ elementu zmiennej `seq`

Definicje konceptów `Range` i `Sequence` potrzebne do tego algorytmu

```
template<typename R>
concept bool Range() {
    return requires (R range){
        typename value_type_t<R>;
        typename iterator_t<R>;
        { begin(range) } -> iterator_t<R>;
        { end(range) } -> iterator_t<R>;
        requires Input_iterator<iterator_t<R>>();
        requires Same_as<value_type_t<R>,
            value_type_t<iterator_t<R>>>();
    };
};

template<typename S>
concept bool Sequence() {
    return Range<R> && requires (S seq) {
        { seq.front() } -> const value_type<S>&;
        { seq.back() } -> const value_type<S>&;
    };
};
```

Specyfikacja wymaga by typ `Range` miał:

- dwa powiązane typy nazwane `value_type_t` i `iterator_t`
- dwa poprawne operacje `begin()` i `end()`, które zwracają iteratory
- typ wartości typu `R` jest taki sam jak typ wartości iteratora tego typu.

Wydaje się w porządku. Możemy użyć tego algorytmu, żeby sprawdzić czy element jest w sekwencji. Niestety to nie działa dla wszystkich kolekcji:

```
std::set<int> x { ... };
if(czyZnaleziono(x, 42)){
// błąd: brak operatora front() lub back()
}
```

Rozwiązaniem jest dodanie przeciążenia, które przyjmuje kontenery asocjacyjne

```
template<Associative_container A, Same_as<key_type_t<T>> T>
bool czyZnaleziono(const A& a, const T& value){
    return a.find(value) != s.end();
}
```

Ta wersja funkcji `czyZnaleziono` ma tylko dwa ograniczenia: typ `A` musi być `Associative_container` i typ `T` musi być taki sam jak typ klucza `A` (`key_type_t<A>`). Dla kontenerów asocjacyjnych, szukamy wartości używając funkcji `find()` a potem sprawdzamy czy się udało przez porównanie z `end()`. W przeciwieństwie do wersji `Sequence`, typ `T` nie musi być `Equality_comparable`. To dlatego, że precyzyjne wymagania typu `T` są ustalone przez kontener asocjacyjny (te wymagania są ustalane przez oddzielny komparator lub funkcję haszującą.

Zdefiniowany koncept `Associative_container`

```
template<typename S>
concept bool Associative_container() {
    return Regular<S> && Range<S>() && requires {
        typename key_type_t<S>;
        requires Object_type<key_type_t<S>>;
    } && requires (S s, key_type_t<S> k){
        { s.empty() } -> bool;
        { s.size() } -> int;
        { s.find(k) } -> iterator_t<S>;
        { s.count(k) } -> int;
    };
};
```