

1 Koncepty

Wprowadzenie

W 1987 próbowano projektować szablony z odpowiednimi interfejsami. Chciano by szablony:

- były w pełni ogólne i wyraziste
- by nie wykorzystywały większych zasobów w porównaniu do kodowania ręcznego
- by miały dobrze określone interfejsy

Długo nie dało się osiągnąć tych trzech rzeczy, ale za to osiągnięto:

- *kompletność Turinga*¹
- lepszą wydajność (w porównaniu do kodu pisanego ręcznie)
- kiepskie interfejsy (praktycznie *typowanie kaczkowe czasu kompilacji*)²

Brak dobrze określonych interfejsów prowadzi do spektakularnie złych wiadomości błędów. Dwie pozostałe właściwości uczyniły z szablonów sukces.

Rozwiązanie problemu specyfikacji interfejsu zostało, przez Alexa Stepanova nazwane **konceptami**. **Koncept** to zbiór wymagań argumentów szablonu. Można też go nazwać systemem typów dla szablonów, który obiecuje znacząco ulepszyć diagnostyki błędów i zwiększyć siłę ekspresji, taką jak przeciążanie oparte na konceptach oraz częściowa specjalizacja szablonu funkcji.

Koncepty (*The Concepts TS*³) zostały opublikowane i zaimplementowane w wersji 6.1 kompilatora GCC w kwietniu 2016 roku. Fundamentalnie to predykaty czasu kompilacji typów i wartości. Mogą być łączone zwykłymi operatorami logicznymi (&&, ||, !)

1.1 Ulepszenie programowania generycznego

Specyfikacja konceptów zawiera wiele ulepszeń, by lepiej wspierać programowanie generyczne przez:

¹(ang. Turing Completeness) umiejętność do rozwiązania każdego zadania, czyli udzielenie odpowiedzi na każde zadanie. Program, który jest kompletny według Turinga może być wykorzystany do symulacji jakiegokolwiek 1-taśmowej maszyny Turinga

²(ang. duck typing) rozpoznanie typu obiektu, nie na podstawie deklaracji, ale przez badanie metod udostępnionych przez obiekt

³(ang. The Concepts Technical Specification) Specyfikacja techniczna konceptów

- umożliwienie wyraźnego określenia ograniczeń argumentów szablonu jako części deklaracji szablonów
- wsparcie możliwości przeciążania szablonów funkcji i częściowego określania szablonów klas i zmiennych opartych na tych ograniczeniach
- dostarczenie składni do definiowania konceptów i wymagań narzuconych na argumenty szablonu
- ujednolicenie `auto` i konceptów w celu zapewnienia jednolitej i dostępnej notacji dla programowania ogólnego
- radykalnej poprawy jakości wiadomości błędów wynikających z niewłaściwego wykorzystania szablonów
- osiągnięcie powyższych bez żadnego narzucania jakichkolwiek zasobów ani znacznego wzrostu czasu kompilacji
- bez ograniczania tego, co można wyrazić przy użyciu szablonów

```
double pierwiastek(double d);
double d = 7;
double d2 = pierwiastek(d);
vector<string> v = {"jeden", "dwa"};
double d3 = pierwiastek(v);
```

Mamy funkcję `pierwiastek`, która jako parametr przyjmuje zmienną typu `double`. Jeśli dostarczymy taki typ, wszystko będzie w porządku, ale jeśli damy inny typ od razu otrzymamy pomocną wiadomość błędu.

```
template<class T>
void sortuj(T &c){
    //kod sortowania
}
```

Kod funkcji `sortuj` zależy od różnych właściwości typu `T`, takiej jak posiadanie operatora `[]`

```
vector<string> v = {"jeden", "dwa"};
sortuj(v);
//OK: zmienna v ma wszystkie syntaktyczne właściwości
wymagane przez funkcję sort
```

```
double d = 7;
sortuj(d);
//Błąd: zmienna d nie ma operatora []
```

Mamy kilka problemów:

- wiadomość błędu jest niejednoznaczna i daleko jej do precyzyjnej i pomocnej, tak jak : "Błąd: zmienna d nie ma operatora []"
- aby użyć funkcji `sortuj`, musimy dostarczyć jej definicję, a nie tylko deklarację. Jest to różnica w sposobie pisania zwykłego kodu i zmienia się model organizowania kodu
- wymagania funkcji dotyczące typu argumentu są domniemane w ciałach ich funkcji
- wiadomość błędu funkcji pojawi się tylko podczas inicjalizacji szablonu, a to może się zdarzyć bardzo długo po momencie wywołania
- Notacja `template<typename T>` jest powtarzalna, bardzo nie lubiana.

Używając konceptu, możemy dotrzeć do źródła problemu, poprzez poprawne określanie wymagań argumentów szablonu. Fragment kodu używającego konceptu `Sortable`:

```
void sortuj(Sortable &c); //(1)
vector<string> v = {"jeden", "dwa"};
sortuj(v); //(2)
double d = 7;
sortuj(d); //(3)
```

- (1) - akceptuj jakąkolwiek zmienną c, która jest `Sortable`
- (2) - OK: v jest kontenerem typu `Sortable`
- (3) - Błąd: d nie jest `Sortable` (`double` nie dostarcza operatora [], itd.

Kod jest analogiczny do przykładu *pierwiastek*. Jedyna różnica polega na tym, że:

- w przypadku typu `double`, projektant języka wbudował go do kompilatora jak określony typ, gdzie jego znaczenie zostało określone w dokumentacji
- zaś w przypadku `Sortable`, użytkownik określił co on oznacza w kodzie. Typ jest `Sortable` jeśli posiada właściwości `begin()` i `end()` dostarczające losowy dostęp do sekwencji zawierającej elementy, które mogą być porównywane używając operatora `<`

Teraz otrzymujemy bardziej jasny komunikat błędu. Jest on generowany natychmiast w momencie gdzie kompilator widzi błędne wywołanie `(sortuj(d);)`

Cele to zrobienie:

- kodu generycznego tak prostym jak nie-generyczny
- bardziej zaawansowanego kodu generycznego tak łatwym do użycia i nie tak trudnym do pisania

1.2 System konceptów

Reprezentacja definicji szablonu w C++ to zazwyczaj drzewa wyprowadzania⁴. Używając identycznych technik kompilatora, możemy przekonwertować koncepty do drzew wyprowadzania. Posiadając to możemy zaimplementować sprawdzanie konceptów jako abstrakcyjne drzewo dopasowań⁵. Wygodnym sposobem implementowania takiego dopasowywania jest generowanie i porównywanie zestawów wymaganych funkcji i typów (zwane *zestawami ograniczeń*) z definicji szablonów i konceptów.

Definicja konceptu to zestaw równań *drzewa AST*⁶ z założeniami typu. Koncepty dają dwa zamysły:

1. w *definicjach szablonu*, koncepty działają jak reguły osądzania typowania. Jeśli *drzewo AST* zależy od parametrów szablonu i nie może być rozwiązane przez otaczające środowisko typowania, wtedy musi się pojawić w strzegących ciałach konceptów. Takie zależne *drzewa AST* są domniemanymi parametrami konceptów i zostaną rozwiązane przez sprawdzanie konceptów w momentach użycia.
2. w *użyciach szablonów*, koncepty działają jak zestawy predykatów, które argumenty szablonu muszą spełniać. Sprawdzanie konceptów rozwiązuje domniemane parametry w momentach inicjalizacji.

Jeśli zestaw konceptów definicji szablonu określa zbyt mało operacji, kompilacja szablonu nie powiedzie się przez sprawdzanie konceptów. Szablon jest prawie ograniczony. Odwrotnie, jeśli zestaw konceptów definicji szablonu określa więcej operacji niż potrzeba, niektóre inne uzasadnione użycia mogą również zawieźć sprawdzanie konceptów. Szablon jest nad ograniczony. Przez "inne uzasadnione" rozumie się, że sprawdzanie typów udałoby się w przypadku braku sprawdzania konceptów.

1.3 Definicja konceptu

Rozróżniamy dwa rodzaje konceptów:

Zmienna konceptowa - jest typem czasu kompilacji i nie niesie za sobą żadnych kosztów czasu wykonania.

⁴(ang. Parse Trees)

⁵(ang. Abstract Tree Matching)

⁶(ang. Abstract Syntax Tree)

Najprostsza forma zmiennej konceptowej:

```
template<template T>
concept bool zmienna_konceptowa = true;
```

Taka zmienna nie może być zadeklarowana z jakimkolwiek innym typem niż `bool` oraz bez inicjalizatora. Błąd pojawi się też, gdy inicjalizatorem nie będzie ograniczone wyrażenie.

Przykład użycia:

```
template<template T>
requires zmienna_konceptowa<T>
void f(T t){
    std::cout << t << "\n";
}
```

Funkcja konceptowa - wygląda i zachowuje się jak zwykła funkcja.

```
template<template T>
concept bool funkcja_konceptowa(){
    return true;
}
```

Funkcja konceptowa nie może:

- być zadeklarowana z żadnym specyfikatorem funkcji w deklaracji
- zwracać żadnego innego typu niż `bool`
- mieć żadnych elementów w liście parametrów
- mieć innego ciała niż `{ return E; }`, gdzie `E` to wyrażenie ograniczone

Przykład użycia:

```
template<template T>
requires funkcja_konceptowa<T>()
void f(T t){
    std::cout << t << "\n";
}
```

1.4 Używanie konceptów

Koncept to predykat czasu kompilacji (coś co zwraca wartość boolowską). Np. argument typu szablonu `T` mógłby mieć wymagania żeby być:

- iteratorem `Iterator<T>`

- iteratorem losowego dostępu `Random_access_iterator<T>`
- liczbą: `Number<T>`

Notacja `C<T>`, gdzie `C` to koncept a `T` to typ, to wyrażenie znaczące ”prawda jeśli `T` spełnia wszystkie wymagania `C`, a nieprawda w przeciwnym wypadku.”

Podobnie, możemy określić, że zestaw argumentów szablonu musi spełniać predykat, np. `Mergeable<In1, In2, Out>`. Taki predykaty wielu typów są niezbędne do opisywania biblioteki STL i wielu innych. Są bardzo ekspresywne i łatwo kompilowalne (tańsze niż obejścia metaprogramowania szablonów). Można oczywiście definiować własne koncepty i można tworzyć biblioteki konceptów. Koncepty pozwalają na przeciążanie i eliminują potrzebę wielokrotnego doraźnego metaprogramowania i kodu *scaffoldingu*⁷ z metaprogramowania, co znacznie upraszcza metaprogramowanie, a także programowanie generyczne.

1.5 Określanie interfejsu szablonu

```
template<typename S, typename T>
    requires Sequence<S> &&
    Equality_comparable<Value_type<S>, T>
    Iterator_of<S> szukaj(S &seq, const T &value);
```

Powyższy szablon przyjmuje dwa argumenty typu szablonu. Pierwszy argument typu musi być typu `Sequence` i musimy być w stanie porównywać elementy sekwencji ze zmienną `value` używając operatora `==` (stąd `Equality_comparable<Value_type<S>, T>`). Funkcja `szukaj` przyjmuje sekwencję przez referencję i `value` do znalezienia jako referencję `const`. Zwraca iterator.

Sekwencja musi posiadać `begin()` i `end()`. Koncept `Equality_comparable` jest zaproponowany jako koncept standardowej biblioteki. Wymaga by jego argument dostarczał operatory `==` i `!=`. Ten koncept przyjmuje dwa argumenty. Wiele konceptów przyjmuje więcej niż jeden argument. Koncepty mogą opisywać nie tylko typy, ale również związki między typami.

Użycie funkcji `szukaj`:

```
void test(vector<string> &v, list<double> &list){
```

⁷metaprogramistyczna metoda budowania aplikacji bazodanowych. To technika wspierana przez niektóre frameworki MVC, w których programista może napisać specyfikację opisującą sposób wykorzystania bazy danych aplikacji. Kompilator używa tej specyfikacji, aby wygenerować kod, który aplikacja może wykorzystać do odczytu, tworzenia, aktualizacji i usuwania wpisów bazy danych

```

    auto a0 = szukaj(v, "test");(1)
    auto p1 = szukaj(v, 0.7);(2)
    auto p2 = szukaj(list, 0.7);(3)
    auto p3 = szukaj(list, "test");(4)

    if(a0 != v.end()){
        //Znaleziono "test"
    }
}

```

1) OK 2) Błąd: nie można porównać string do double 3) OK 4) Błąd: nie można porównać double ze string

1.6 Notacja skrótowa

Gdy chcemy podkreślić, że argument szablonu ma być sekwencją, piszemy:

```

template<typename Seq>
    requires Sequence<Seq>
void algo(Seq &s);

```

To oznacza, że potrzebujemy argumentu typu `Seq`, który musi być typu `Sequence`, lub innymi słowy: Szablon przyjmuje argument typu, który musi być typu `Sequence`. Możemy to uprościć:

```

template<Sequence Seq>
void algo(Seq &s);

```

To znaczy dokładnie to samo co dłuższa wersja, ale jest krótsza i lepiej wygląda. Używamy tej notacji dla konceptów z jednym argumentem. Np. moglibyśmy uprościć funkcję `szukaj`:

```

template<Sequence S, typename T>
    requires Equality_comparable<Value_type<S>, T>
Iterator_of<S> szukaj(S &seq, const T &value);

```

Upraszcza to składnię języka. Sprawia, że nie jest zbyt zagmatwana.

1.7 Definiowanie konceptów

Koncepty, takie jak `Equality_comparable` często można znaleźć w bibliotekach (np. w `The Ranges TS`), ale koncepty można też definiować samodzielnie:

```

template<typename T>
concept bool Equality_comparable = requires (T a, T b){

```

```

    { a == b } -> bool; //(1)
    { a != b } -> bool; //(2)
};

```

Koncept ten został zdefiniowany jako szablonowa zmienna. Typ musi dostarczać operacje `==` i `!=`, z których każda musi zwracać wartość `bool`, żeby być `Equality_comparable`. Wyrażenie `requires` pozwala na bezpośrednie wyrażenie jak typ może być użyty:

- `{ a == b }`, oznajmia, że dwie zmienne typu `T` powinny być porównywalne używając operatora `==`
- `{ a == b } -> bool` mówi że wynik takiego porównania musi być typu `bool`

Wyrażenie `requires` jest właściwie nigdy nie wykonywane. Zamiast tego kompilator patrzy na wymagania i zwraca `true` jeśli się skompilują a `false` jeśli nie. To bardzo potężne ułatwienie.

```

template<typename T>
concept bool Sequence = requires (T t) {
    typename Value_type<T>;
    typename Iterator_of<T>;

    { begin(t) } -> Iterator_of<T>;
    { end(t) } -> Iterator_of<T>;

    requires Input_iterator<Iterator_of<T>>;
    requires Same_type<Value_type<T>,
        Value_type<Iterator_of<T>>>;
};

```

Żeby być typu `Sequence`:

- typ `T` musi mieć dwa powiązane typy: `Value_type<T>` i `Iterator_of<T>`. Oba typy to zwykle *aliasy szablonu*⁸. Podanie tych typów w wyrażeniu `requires` oznacza, że typ `T` musi je posiadać żeby być `Sequence`.
- typ `T` musi mieć operacje `begin()` i `end()`, które zwracają odpowiednie iteratory.
- odpowiedni iterator oznacza to, że typ iteratora typu `T` musi być typu `Input_iterator` i typ wartości typu `T` musi być taka sama jak jej wartość typu jej iteratora. `Input_iterator` i `Same_type` to koncepty z biblioteki.

⁸ALIAS SZABLONU

Teraz w końcu możemy napisać koncept `Sortable`. Żeby typ był `Sortable`, powinien być sekwencją oferującą losowy dostęp i posiadać typ wartości, który wspiera porównania używające operatora `<`:

```
template<typename T>
concept bool Sortable = Sequence<T> &&
    Random_access_iterator<Iterator_of<T>> &&
    Less_than_comparable<Value_type<T>>;
```

`Random_access_iterator` i `Less_than_comparable` są zdefiniowane analogicznie do `Equality_comparable`

Często, wymagane są relacje pomiędzy konceptami. Np. koncept `Equality_comparable` jest zdefiniowany by wymagał jeden typ. Można zdefiniować ten koncept by radził sobie z dwoma typami:

```
template<typename T, typename U>
concept bool Equality_comparable = requires (T a, U b) {
    { a == b } -> bool;
    { a != b } -> bool;
    { b == a } -> bool;
    { b != a } -> bool;
};
```

To pozwala na porównywanie zmiennych typu `int` z `double` i `string` z `char*`, ale nie `int` z `string`.

1.8 Przeciążanie funkcji przy użyciu konceptów

Główna 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*⁹ ze standardowej biblioteki

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

Potrzeba różnych wersji tego algorytmu, m.in.

⁹Algorytm *advance(it, n)*; inkrementuje otrzymany iterator *it* o *n* elementów.

- 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*¹⁰, lecz z konceptami rozwiązanie jest proste i oczywiste:

```
template<Forward_iterator F, int n> void advance(F f, int n){
    while(n-->0) ++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);

    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)

¹⁰Technika programowania generycznego polegająca na wykorzystaniu przeciążania funkcji w celu wybrania, którą implementację funkcji wywołać w czasie wykonania

W funkcji `test`, `Random_access_iterator` ma więcej wymagań niż `Forward_iterator`, więc wywołuje się szybka wersja `advance` dla iteratora zmiennej `vector`. Dla iteratora 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*¹¹. Kompilator przetwarza hierarchię dla użytkownika. To jest prostsze, bardziej elastyczne i mniej podatne na błędy.

Przeciążanie oparte na konceptach eliminuje znaczącą ilość *boiler-plate*¹² z programowania generycznego i kodu meta programowania (użycia `enable_if`¹³).

Funkcja `czyZnaleziono` ocenia czy element znajduje 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;
    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>;
    };
}
```

¹¹klasy traits

¹²BP

¹³EI

```

        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;
    }
};

```