Koncepty jako sposób ograniczania argumentów szablonu

Maciej Zbierowski

17 września 2017

Spis treści

Wstęp

Pomysł ograniczania argumentów szablonów jest tak stary jak same szablony. Ale dopiero z początkiem dwudziestego pierwszego wieku zaczęły się poważne prace udoskonalające język *C++*, aby zapewnić te możliwości. Te prace ostatecznie dały rezultat w postaci *konceptów C++0x*. Rozwój tych funkcjonalności i ich wdrożenie do biblioteki standardowej *C++* były głównymi tematami *Komisji Standardu C++*[1] dla *C++11*. Te cechy zostały ostatecznie usunięte z powodu istotnych nierozwiązanych kwestii i bardziej rygorystycznego terminu publikacji.

W 2010 roku wznowiono prace nad konceptami (bez udziału komisji). Andrew Sutton i Bjarne Stroustrup opublikowali dokument omawiający jak zminimalizować ilość konceptów potrzebnych do określania części biblioteki standardowej, a grupa z Uniwersytetu w Indianie zainicjowała prace nad nową implementacją. Potem wspólnie z Alexem Stepanovem (twórcą *biblioteki STL*[2]) stworzyli raport, w którym przedstawili w pełni ograniczone algorytmy biblioteki *STL* i zasugerowali projekt języka, który mógłby wyrazić te ograniczenia. Próbowano zaprojektować minimalny zestaw funkcji językowych, które umożliwiłyby użytkownikom ograniczanie szablonów. Z tych prób narodziło się rozszerzenie języka, zwane *Concepts Lite*.

Koncepty są jedną z najbardziej przeanalizowanych, zaproponowanych funkcjonalności *C++*. Powstały by wspierać programowanie generyczne. Proponowano różne ich projekty, ostatni to *Concepts TS*[3]) opublikowany w październiku 2015 jako *ISO C++14 Technical Specification*. Ta specyfikacja została wdrożona i jest dostępna w gałęzi *GNU Compiler Collection (GCC)*[4] od wielu lat i została dostarczana do produkcji jako część oficjalnych wydań *GCC* od czasu wydania *GCC-6.0*. Praca ta jest częściowo finansowana przez *NSF*[5] z wyraźnym celem wprowadzenia programowania generycznego

[1](ang. C++ Standards Committee) znana również pod nazwą "ISO JTC1/SC22/WG21". Składa się z akredytowanych ekspertów z krajów członkowskich, którzy są zainteresowani pracą nad C++

[2](ang. Standard Template Library) biblioteka C++ zawierająca algorytmy, kontenery, iteratory oraz inne konstrukcje w formie szablonów

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

[4]zestaw kompilatorów do różnych języków programowania

[5](ang. The National Science Foundation) amerykańska agencja rządowa wspierająca

do głównego nurtu języku *C++*. Jak każda funkcjonalność języka z potencjałem zmiany skali w jakiej piszemy programy, jak myślimy o programach, jak je organizujemy, zgromadziły wiele opinii i sugestii na przestrzeni lat.

podstawowe badania naukowe i edukacje we wszystkich niemedycznych dziedzinach nauki i sztuki.

1 Szablony - definicja, zastosowania

Szablony są jedną z głównych cech języka *C++*. Dzięki nim możemy dostarczać generyczne typy i funkcje, bez kosztów czasu wykonania. Skupiają się na pisaniu kodu w sposób niezależny od konkretnego typu, dzięki czemu wspierają programowanie generyczne. *C++* to bogaty język wspierający polimorficzne zachowania zarówno w czasie wykonania jak i kompilacji. W czasie wykonania używa on hierarchii klas i wywołań funkcji wirtualnych by wspierać praktyki zorientowane obiektowo, gdzie wywoływana funkcja zależy od typu obiektu docelowego podczas czasu wykonania. Natomiast w czasie kompilacji szablony wspierają programowanie generyczne, gdzie wywoływana funkcja zależy od statycznego typu czasu kompilacji argumentów szablonu.

Polimorfizm czasu kompilacji był w języku od bardzo dawna. Polega na dostarczeniu szablonu, który umożliwia kompilatorowi wygenerowanie kodu w czasie kompilacji.

Szablony grają kluczową rolę w projektowaniu obecnych, znanych i popularnych bibliotek i systemów. Stanowią podstawę technik programowania w różnych dziedzinach, począwszy od konwencjonalnego programowania ogólnego przeznaczenia do oprogramowywania wbudowanych systemów bezpieczeństwa.

Szablon to coś w rodzaju przepisu, z którego translator *C++* generuje deklaracje.

```
template<typename T>
T kwadrat (T x) {
   return x * x;
}
```

Kod ten deklaruje rodzinę funkcji indeksowanych po parametrze typu. Można odnieść się do konkretnego członka tej rodziny przez zastosowanie konstrukcji `kwadrat<int>`. Mówimy wtedy, że żądana jest specjalizacja szablonu dla funkcji `kwadrat` z listą argumentów szablonu `<int>`. Proces tworzenia specjalizacji nosi nazwę inicjalizacji szablonu, potocznie zwany inicjalizacją. Kompilator C++ stworzy stosowny odpowiednik definicji funkcji:

```
int kwadrat(int x) {
    return x * x;
}
```

Argument typu `int` jest podstawiony za parametr typu `T`. Kod wynikowy jest sprawdzany pod względem typu, by zapewnić brak błędów wynikających z podmiany. Inicjalizacja szablonu jest wykonywana tylko raz dla danej specyfikacji nawet jeśli program zawiera jej wielokrotne żądania.

W przeciwieństwie do języków takich jak *Ada* czy *System F*, lista argumentów szablonu może być pominięta z żądania inicjalizacji szablonu funkcji. Zazwyczaj, wartości parametrów szablonu są *dedukowane*[6].

```
double d = kwadrat(2.0);
```

Argument typu jest dedukowany na `double`. Warto zauważyć, że odmiennie niż w językach takich jak *Haskell* czy *System F*, parametry szablonu w *C++* nie są ograniczone względem typów.

Szablonów używa się do zmniejszania kar abstrakcji i zjawiska *code bloat*[7] w systemach wbudowanych w stopniu, który jest niepraktyczny w standardowych systemach obiektowych. Robi się to z dwóch powodów:

- Po pierwsze, inicjalizacja szablonu łączy informacje zarówno z definicji, jak i z kontekstu użycia. To oznacza, że pełna informacja zarówno z definicji jak i z wywołanych kontekstów (włączając w to informacje o typach) jest udostępniana generatorowi kodu. Dzisiejsze generatory kodu dobrze sobie radzą z używaniem tych informacji w celu zminimalizowania czasu wykonania i przestrzeni kodu. Różni się to od zwykłego przypadku w języku obiektowym, gdzie wywołujący i wywoływany są kompletnie oddzieleni przez interfejs, który zakłada pośrednie wywołania funkcji.

[6](ang. deduction) Dedukcja - określenie lub wyliczenie (przez kompilator) argumentu szablonu pominiętego przy wywołaniu funkcji.

[7]Code bloat - produkowanie kodu, który postrzegany jest jako niepotrzebnie długi, spowalniający lub w inny sposób marnujący zasoby

- Po drugie, szablon w *C++* jest zazwyczaj domyślnie tworzony tylko jeśli jest używany w sposób niezbędny dla semantyki programu, automatycznie minimalizując miejsce w pamięci, które wykorzystuje aplikacja. W przeciwieństwie do języka *Ada* czy *System F*, gdzie programista musi wyraźnie zarządzać inicjalizacjami.

1.1 Parametryzacja szablonów

Parametry szablonu są określane na dwa sposoby:

1. *parametry szablonu* – wyraźnie wspomniane jako parametry w deklaracji szablonu

2. *nazwy zależne* - wywnioskowane z użycia parametrów w definicji szablonu

W *C++* nazwa nie może być użyta bez wcześniejszej deklaracji. To wymaga od użytkownika ostrożnego traktowania definicji szablonów. Np. w definicji funkcji `kwadrat` nie ma widocznej deklaracji symbolu *. Jednak, podczas inicjalizacji szablonu `kwadrat<int>` kompilator może sprowadzić symbol * do (wbudowanego) operatora mnożenia dla wartości `int`. Dla wywołania `kwadrat(zespolona(2.0))`, operator * zostałby rozwiązany do (zdefiniowanego przez użytkownika) operatora mnożenia dla wartości `zespolona`. Symbol * jest więc *nazwą zależną* w definicji funkcji `kwadrat`. Oznacza to, że jest to ukryty parametr definicji szablonu. Możemy uczynić z operacji mnożenia formalny parametr:

```
template<typename Multiply, typename T>
T square(T x) {
    return Multiply() (x,x);
}
```

Pod-wyrażenie `Multiply()` tworzy obiekt funkcji, który wprowadza operację mnożenia wartości typu `T`. Pojęcie *nazw zależnych* pomaga utrzymać liczbę jawnych argumentów.

1.2 Inicjalizacje i sprawdzanie

Minimalne przetwarzanie semantyczne odbywa się, gdy po raz pierwszy pojawia się definicję szablonu lub jego użycie. Pełne przetwarzanie semantyczne jest przesuwane na czas inicjalizacji (tuż przed czasem linkowania), na podstawie każdej instancji. Oznacza to, że założenia dotyczące argumentów szablonu nie są sprawdzane przed czasem inicjalizacji. Np.

```
string x = "testowy tekst";
kwadrat(x);
```

Bezsensowne użycie zmiennej `string` jako argumentu funkcji `kwadrat` nie jest wyłapane w momencie użycia. Dopiero w czasie inicjalizacji kompilator odkryje, że nie ma odpowiedniej deklaracji dla operatora *. To ogromny praktyczny błąd, bo inicjalizacja może być przeprowadzona przez kod napisany przez użytkownika, który nie napisał definicji funkcji `kwadrat` ani definicji `string`. Programista, który nie znał definicji funkcji `kwadrat` ani `string` miałby ogromne trudności w zrozumieniu komunikatów błędów związanych z ich interakcją (np. "illegal operand for *").

Istnienie symbolu operatora * nie jest wystarczające by zapewnić pomyślną kompilację funkcji `kwadrat`. Musi istnieć operator *, który przyjmuje argumenty odpowiednich typów i ten operator * musi być bezkonkurencyjnym dopasowaniem według zasad przeciążania *C++*. Dodatkowo funkcja `kwadrat` przyjmuje argumenty przez wartość i zwraca swój wynik przez wartość. Z tego wynika, że musi być możliwe skopiowanie obiektów dedukowanego typu. Potrzebny jest rygorystyczny framework do opisywania wymagań definicji szablonów na ich argumentach.

Doświadczenie podpowiada nam, że pomyślna kompilacja i linkowanie może nie gwarantować końca problemów. Udana budowa pokazuje tylko, że inicjalizacje szablonów były poprawne pod względem typów, dostając argumenty które przekazaliśmy. Co z typami argumentów szablonu i wartościami, z którymi nie próbowaliśmy użyć naszych szablonów? Definicja szablonu może zawierać przypuszczenia na temat argumentów, które przekazaliśmy ale nie zadziała dla innych, prawdopodobnie rozsądnych argumentów. Uprosz-

czona wersja klasycznego przykładu:

```
template<typename FwdIter>
bool czyJestPalindromem(FwdIter first, FwdIter last){
    if(last <= first) return true;
    if(*first != *last) return false;
    return czyJestPalindromem(++first, --last);
}
```

Testujemy czy sekwencja wyznaczona przez parę iteratorów do jego pierwszego i ostatniego elementu, jest palindromem. Przyjmuje się, że te iteratory są z kategorii *forward iterator*. To znaczy, że powinny wspierać co najmniej operacje takie jak: *, != i ++. Definicja funkcji `czyJestPalindromem` bada czy elementy sekwencji zmierzają z początku i końca do środka. Możemy przetestować tę funkcję używając `vector`, tablicy w stylu `C` i `string`. W każdym przypadku nasz szablon funkcji zainicjalizuje się i wykona się poprawnie. Niestety, umieszczenie tej funkcji w bibliotece byłoby dużym błędem. Nie wszystkie sekwencje wspierają operatory `--` i $\leqslant$. Np. listy pojedyncze nie wspierają. Eksperci używają wyszukanych, regularnych technik by uniknąć takich problemów. Jednakże, fundamentalny problem jest taki, że definicja szablonu nie jest (według siebie) dobrą specyfikacją wymagań na swoje parametry.

1.3 Wydajność

Szablony grają kluczową rolę w programowaniu w *C++* dla wydajnych aplikacji. Ta wydajność ma trzy źródła:

- eliminacja wywołań funkcji na korzyść *inliningu*[8]
- łączenie informacji z różnych kontekstów w celu lepszej optymalizacji
- unikanie generowania kodu dla niewykorzystanych funkcji

Pierwszy punkt nie odnosi się tylko do szablonów ale ogólnie do cech funkcji *inline* w *C++*. Wydajność ta przekłada się zarówno na czas wykonania jak i pamięć. Szablony mogą równocześnie zmniejszyć obie wydajności. Zmniejszenie rozmiaru kodu jest szczególnie ważne, ponieważ w przypadku nowoczesnych procesorów zmniejszenie rozmiaru kodu pociąga za sobą zmniejszenie ruchu w pamięci i poprawienie wydajności pamięci podręcznej.

```
template<typename FwdIter, typename T>
T suma(FwdIter first, FwdIter last, T init){
    for(FwdIter cur = first, cur != last, T init)
        init = init + *cur;
    return init;
}
```

Funkcja `suma` zwraca sumę elementów jej sekwencji wejściowej używając trzeciego argumentu ("akumulatora") jako wartości początkowej

```
vector<zespolona<double>> v;
zespolona<double> z = 0;
z = suma(v.begin(), v.end(), z);
```

By wykonać swoją pracę, `suma` użyje operatorów dodawania i przypisania na elementach typu `zespolona<double>` i dereferencji iteratorów `vector<zespolona<double>>`. Dodanie wartości typu `zespolona<double>`

[8]Optymalizacja kompilatora, która zamienia wywołanie funkcji na jej ciało w czasie kompilacji.

pociąga za sobą dodanie wartości typu `double`. By zrobić to wydajnie wszystkie te operacje muszą być *inline*. Zarówno `vector` jak i `zespolona` są typami zdefiniowanymi przez użytkownika. Oznacza to, że typy te jak i ich operacje są zdefiniowane gdzie indziej w kodzie źródłowym *C++*. Obecne kompilatory *C++* radzą sobie z tym przykładem, dzięki czemu jedyne wygenerowane wywołanie to wywołanie funkcji `suma`. Dostęp do pól zmiennej `vector` staje się prostą operacją maszyny ładującej, dodawanie wartości typu `zespolona` staje się dwiema instrukcjami maszyny dodającej dwa elementy zmiennoprzecinkowe. Aby to osiągnąć, kompilator potrzebuje dostępu do pełnej definicji `vector` i `zespolona`. Jednak wynik jest ogromną poprawą (prawdopodobnie optymalną) w stosunku do naiwnego podejścia generowania wywołania funkcji dla każdego użycia operacji na parametrze szablonu. Oczywiście instrukcja dodawania wykonuje się znacznie szybciej niż wywołanie funkcji zawierającej dodawanie. Poza tym, nie ma żadnego wstępu wywołania funkcji, przekazywanie argumentów itd., więc kod wynikowy jest również wiele mniejszy. Dalsze zmniejszanie rozmiaru generowanego kodu uzyskuje się nie wysyłając kodu niewykorzystywanych funkcji. Klasa szablonu `vector` ma wiele funkcji, które nie są wykorzystywane w tym przykładzie. Podobnie szablon klasy `zespolona` ma wiele funkcji i funkcji nieskładowych (nienależących do funkcji klasy). Standard *C++* gwarantuje, że nie jest emitowany żaden kod dla tych niewykorzystanych funkcji.

Inaczej sprawa wygląda, gdy argumenty są dostępne za pośrednictwem interfejsów zdefiniowanych jako wywołania funkcji pośrednich. Każda operacja staje się wtedy wywołaniem funkcji w pliku wykonywalnym generowanym dla kodu użytkownika, takiego jak `suma`. Co więcej, byłoby wyraźnie nietypowe unikać odkładania kodu nieużywanych (wirtualnych) funkcji składowych. Jest to poza zdolnością obecnych kompilatorów *C++* i prawdopodobnie pozostanie takie dla głównych programów *C++*, gdzie oddzielna kompilacja i łączenie dynamiczne jest normą. Ten problem nie jest wyjątkowy dla *C++*. Opiera się on na podstawowej trudności w ocenieniu, która część kodu źródłowego jest używana, a która nie, gdy jakakolwiek forma procesu *run-time dispatch*[9] ma miejsce. Szablony nie cierpią na ten problem

[9]Zwany również *dynamic dispatch* proces wybierania, implementacji polimorficznej operacji (metody lub funkcji) do wywołania w czasie uruchomienia.

bo ich specjalizacje są rozwiązywane w czasie kompilacji.

```
vector<int> v;
zespolona<double> s = 0;
s = suma(v.begin(), v.end(), s);
```

W powyższej funkcji dodawanie wykonywane jest przez konwertowanie wartości `int` do wartości `double` i potem dodawanie tego do akumulatora `s`, używając operatora `+` typu `zespolona<double>` i `double`. To podstawowe dodawanie zmiennoprzecinkowe. Kwestia jest taka, że operator `+` w funkcji `suma` zależy od dwóch parametrów szablonu i leży to w kwestii kompilatora by wybrać bardziej odpowiedni operator `+` bazując na informacji o tych dwóch argumentach. Byłoby możliwe utrzymanie lepszego rozdzielenia między różnymi kontekstami przez przekształcanie typu elementu w typ akumulatora. W takim przypadku spowodowałoby to powstanie dodatkowego `zespolona<double>` dla każdego elementu i dodania dwóch wartości typu `zespolona`. Rozmiar kodu i czas wykonywania byłyby większe niż dwukrotnie.

Duże ilości prawdziwego oprogramowania zależą od optymalizacji. W konsekwencji udoskonalone sprawdzanie typu, co zostało obiecane przy użyciu konceptów, nie może kosztować tych optymalizacji.

2 Koncepty

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*[10]
- lepszą wydajność (w porównaniu do kodu pisanego ręcznie)
- kiepskie interfejsy (praktycznie *typowanie kaczkowe czasu kompilacji*)[11]

Brak dobrze określonych interfejsów prowadzi do szczególnie 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 konceptowe oraz częściową specjalizację szablonu funkcji.

Koncepty (*The Concepts TS* 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 (`&&`, `||`, `!`)

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

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

2.1 Ulepszenie programowania generycznego

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

- umożliwienie wyraźnego określania 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`[12] i konceptów w celu zapewnienia jednolitej i dostępnej notacji dla programowania ogólnego
- radykalną poprawę jakości wiadomości błędów wynikających z niewłaściwego wykorzystania szablonów
- osiągnięcie tego wszystkiego bez żadnego narzucania jakichkolwiek dodatkowych 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);
```

Funkcja `pierwiastek`, która jako parametr przyjmuje zmienną typu `double`. Jeśli zostanie jej dostarczony taki typ, wszystko będzie w porządku, ale jeśli inny, od razu kompilator wyprodukuje pomocną wiadomość błędu.

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

[12]Słowo kluczowe wg standardu *C++11*, oznaczające zastępczy typ zmiennej, który zostanie wydedukowany na podstawie wartości za pomocą której zmienna zostanie zainicjalizowana.

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

```
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 []
```

Pojawiło się 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 deklaracje. 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 nielubiana.

Używając konceptu, możemy dotrzeć do źródła problemu, poprzez poprawne określenie 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 typu `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 jako 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 konceptów to, zrobienie:

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

2.2 System konceptów

Reprezentacja definicji szablonu w C++ to zazwyczaj *drzewo wyprowadzania*[13]. Używając identycznych technik kompilatora, możemy przekonwerto-

[13](ang. Parse Tree) - uporządkowane, zakorzenione drzewo, które reprezentuje strukturę składniową łańcucha znakowego zgodnie z gramatyką bezkontekstową. Zwane również drzewem składniowym

wać koncepty do takich drzew. Posiadając to, sprawdzanie konceptów możemy zaimplementować jako *abstrakcyjne drzewo dopasowań*. 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*[14] 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 z powodu sprawdzania konceptów. Szablon będzie w takim wypadu "prawie ograniczony". Odwrotnie, jeśli zestaw konceptów definicji szablonu określa więcej operacji niż potrzeba, niektóre inne uzasadnione użycia mogą również zawieźć sprawdzanie konceptów. Szablon będzie wtedy "nad ograniczony". Przez "inne uzasadnione" rozumie się, że sprawdzanie typów udałoby się w przypadku braku sprawdzania konceptów.

2.3 Definicja konceptu

Rozróżniamy dwa rodzaje konceptów:

[14](ang. Abstract Syntax Tree) Drzewo składniowe, drzewo składni abstrakcyjnej - drzewo etykietowane, wynik przeprowadzenia analizy składniowej zzdania (słowa) zgodnie z pewną gramatyką.

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

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";
}
```

2.4 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){
    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

2.5 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.

2.6 Definiowanie konceptów

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

dzielnie:

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

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 zwykłe *aliasy szablonu*[15]. 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 taki sam jak jej wartość typu jej iteratora. `Input_iterator` i `Same_type` to koncepty z biblioteki.

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 zdefiniwoany 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`.

[15] nazwa odwołująca się do rodziny typów.

3 Przeciążanie

Koncepty są użyteczne nie tylko w poprawianiu wiadomości błędów i precyzyjnej specyfikacji interfejsów. Zwiększają również ekspresyjność. Używane są do skracania kodu, robieniu go generycznym i zwiększania wydajności. Wyjątkowo potężną cechą jest ich rola w przeciążaniu funkcji.

W kwietniu 2016 został wydany kompilator *GCC 6.2*. Ta wersja zawierała główne unowocześnienie dwóch komponentów implementacji konceptów. Jeden z nich to generator diagnostyki, który został znacznie odnowiony, aby zapewnić dokładną diagnostykę niepowodzeń konceptu przy sprawdzaniu czy jest spełniony. Drugi to wsparcie dla przeciążania ograniczeń, które zostało całkowicie przepisane, aby zapewnić znaczne zwiększenie wydajności. W *GCC* można teraz używać konceptów do projektów o znacznej wielkości i złożoności.

Niektórzy twierdzą, że wyrażenia takie jak `SFINAE`[16], `constexpr if`[17], `static_assert`[18] i mądre techniki metaprogramowania w zupełności wystarczą do przeciążania. To oczywiście poprawne myślenie, lecz jest to obniżanie poziomu abstrakcji, co skutkuje tym, że programuje się w sposób żeby było zrobione a nie jak powinno być. Wynikiem jest więcej pracy dla programisty, zwiększona ilość błędów i mniej szans optymalizacyjnych. *C++* nie jest przeznaczony do metaprogramowania szablonów. Koncepty pomagają nam podnieść poziom programowania i ułatwić kod, bez dodawania kosztów czasu wykonania.

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

[16](ang. Substitution failure is not an error) sytuacja w *C++* gdzie nieprawidłowe zastąpienie parametrów szablonu nie jest samo w sobie błędem

[17]Wyrażenie, którego wartość warunku musi być kontekstowo konwertowanym stałym wyrażeniem typu bool.

[18]Wykonuje sprawdzanie porównania w czasie kompilacji

```
  return false;
}
```

Funkcja `czyIstnieje` przyjmuje sekwencję typu `Sequence` jako pierwszy argument i wartość `Equality comparable` jako drugi. Algorytm ma trzy ograniczenia:

- `seq` musi być typu `Sequence`
- `value` musi być typu `Equality_comparable`
- typ `value` musi być taki sam jak element typu `seq`

Wyrażenie `value_type_t` to alias typu, który odnosi się do zdeklarowanego lub wydedukowanego typu wartości `R`. Definicje konceptów `Sequence` i `Range` potrzebne do tego algorytmu wyglądają tak:

```
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>&;
  };
```

```
}
```

Większość sekwencji posiada operacje `front()` i `back()`, które zwracają pierwszy i ostatni element przedziału. To nie jest w pełni rozwinięta specyfikacja sekwencji. Możemy użyć algorytmu do określenia, czy element znajduje się w dowolnej sekwencji. Niestety, algorytm nie działa w przypadku niektórych kolekcji:

```
std::set<int> testSet { ... };
if (czyIstnieje(testSet, 42)) // (1)
  ...
```

(1) - błąd: brak operacji `front()` lub `back()`

Potrzebny jest sposób, żeby jasno określić, czy klucz znajduję się w zbiorze.

3.1 Rozszerzanie algorytmów

Rozwiązaniem jest dodanie kolejnego przeciążenia, które jako parametr przyjmuje kontener asocjacyjny[19].

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

Ta wersja funkcji `czyIstnieje()` ma tylko dwa ograniczenia: `A` musi być typu `Associative_container`, a typ `T` musi być taki sam jak typ klucza `A` (`key_type_t<A>`). W przypadku kontenerów asocjacyjnych po prostu wyszukujemy wartość przy użyciu `find()`, a następnie sprawdzamy, czy znaleźliśmy ją przez porównanie z `end()`. To prawdopodobnie szybsze rozwiązanie

[19](ang. associative container) grupa szablonów klas w standardowej bibliotece *C++*, która implementuje uporządkowane tablice asocjacyjne. Kontenery zdefiniowane w obecnej wersji standardu: set, map, multiset, multimap, unordered set, unordered multiset, unordered map, unordered multimap.

niż wyszukiwanie sekwencyjne. W przeciwieństwie do wersji `Sequence`, `T` nie musi być typu `Equality_comparable`. Wynika to z faktu, że dokładne wymagania `T` są określone przez kontener asocjacyjny, a wymogi te są zwykle określane przez osobny komparator lub funkcję haszującą.

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;
    };
}
```

Kontener asocjacyjny jest typu `Regular`, definiuje `Range` elementów, ma `key_type` (który może różnić się od wartości `value_type`), a także zestaw operacji, w tym `find()`, itd.

Podobnie jak poprzednio w przypadku `Sequence`, nie jest to wyczerpująca lista wymagań dla kontenera asocjacyjnego. Nie dotyczy wstawiania i usuwania, a także wyklucza szczególne wymagania dotyczące iteratorów `const`. Ponadto nie opisaliśmy dokładnie tego, jak oczekujemy, że zachowają się funkcje `size()`, `empty()`, `find()` i `count()`.

Ten koncept dotyczy wszystkich kontenerów asocjacyjnych z biblioteki standardowej *C++* (`set`, `map`, `unordered_multiset`, itp.). Obejmuje również te niestandardowe, zakładając, że narażają interfejs. Na przykład przeciążenie to będzie działało dla wszystkich kontenerów asocjacyjnych typu `Q(QSet<T>, QHash<T>)`.

Aby używać konceptów do rozwijania algorytmów, należy zrozumieć, jak kompilator wybiera pomiędzy wersją `Sequence` a `Associative_container`. Innymi słowy, co się dzieje gdy wywoływana jest funkcja `czyIstnieje()`

```
std::vector<int> v { ... };
std::set<int> s { ... };

if (czyIstnieje(v, 42)) // (1)
    //...
if (czyIstnieje(s, 42)) // (2)
    //...
```

(1) - wywołuje przeciążenie `Sequence`

(2) - wywołuje przeciążenie `Associative_container`

Dla każdego wywołania `czyIstnieje` kompilator określa, która funkcja jest wywoływana na podstawie podanych argumentów. Nazywa się to *rozwiązaniem przeciążenia*[20]. Jest to algorytm, który próbuje znaleźć jedną najlepszą funkcję (wśród jednego lub więcej kandydatów), aby ją wywołać na podstawie podanych argumentów. Oba wywołania funkcji odnoszą się do szablonów, więc kompilator wykonuje dedukcję argumentów szablonu, a potem formuje specjalizację deklaracji w oparciu o wyniki. W obydwu przypadkach dedukcja i zastąpienie powiodą się w zwykły i przewidywalny sposób, dlatego w każdym punkcie wywołania musimy wybrać jedną z dwóch specjalizacji. W tym miejscu ograniczenia wchodzą w grę. Tylko funkcje których ograniczenia są spełnione mogą być wybrane przez rozwiązanie przeciążenia. Aby określić, czy ograniczenia funkcji są spełnione, zastępujemy dedukowane argumenty szablonu powiązanymi ograniczeniami deklaracji szablonu funkcji, a następnie oceniamy wynikowe wyrażenie. Ograniczenia są spełnione, gdy substytucja się powiedzie, a wyrażenie okaże się prawdziwe.

W pierwszym wywołaniu, dedukowane argumenty szablonu to `vector<int>` i `int`. Argumenty te spełniają ograniczenia `Sequence`, ale nie tych `Asociative_container`, ponieważ `vector` nie ma `find()` lub `count()`. Dlatego kandydat `Asociative_container` zostaje odrzucony, pozostawiając

[20](ang. overload resolution)

tylko kandydata `Sequence`. W drugim wywołaniu, dedukowane argumenty to `set<int>` i `int`. Rozwiązanie jest odwrotne do poprzedniego: `set` nigdy nie jest `Sequence`, ponieważ brakuje mu operacji `front()` i `back()`, tak więc kandydat jest odrzucany, a rozwiązanie przeciążenia wybiera kandydata `Asociative_container`. To działa, ponieważ ograniczenia obu przeciążeń są wystarczająco wyczerpujące, aby zapewnić, że kontener spełnia ograniczenia jednego szablonu lub drugiego, ale nie obu. Sytuacja jest nieco bardziej interesująca, jeśli chcemy dodać więcej przeciążeń tego algorytmu. Możemy rozszerzyć algorytm dla konkretnych typów lub szablonów, tak jak mogliśmy to zrobić bez konceptów. Zasadniczo możemy określić prawidłowe definicje funkcji dla tych typów. Jeśli będziemy mieli szczęście, wiele z tych nowych przeładowań będzie miało identyczne definicje.

Ogólnie rzecz biorąc, możemy kontynuować rozszerzanie definicji algorytmu generycznego przez dodanie przeciążeń, które różnią się tylko ich ograniczeniami. Są trzy przypadki, które trzeba wziąć pod uwagę podczas przeładowywania z konceptami:

1. Rozszerzać definicję poprzez dostarczenie przeciążenia, które działa dla zupełnie innego zestawu typów. Ograniczenia tych nowych przeładowań byłyby wzajemnie wykluczające lub miałyby minimalną ilość nakładania się na istniejące ograniczenia.

2. Dostarczać zoptymalizowaną wersję istniejącego przeciążenia, specjalizując ją w podzbiorze swoich argumentów. Wymaga to utworzenia nowego przeciążenia, które ma silniejsze ograniczenia niż jego bardziej ogólna forma.

3. Dostarczać uogólnioną wersję, która jest zdefiniowana w kategoriach ograniczeń współużytkowanych przez jedno lub więcej istniejących przeładowań.

Jeśli ograniczenia nie są rozłączne z wieloma kandydatami, mogą być opłacalne. Kompilator musi określić najlepszego kandydata na wywołanie. Jeśli jednak kompilator nie może określić najlepszego kandydata, rozwiązanie jest niejednoznaczne. Gdy w pierwszym algorytmie `czyIstnieje()`

zmieni się wymaganie `Sequence` zamiast tylko `Range`. To zminimalizuje ilość nakładania się, a zatem i prawdopodobieństwo dwuznaczności.

Ograniczenia rozłączne nie gwarantują, że połączenie będzie niedwuznaczne. Możemy na przykład spróbować zdefiniować kontener, który spełnia wymagania zarówno `Sequence` i `Associative_container`. W tym przypadku oba przeciążenia byłyby opłacalne, ale przeciążenie nie jest z natury lepsze od innych. Chyba że dodamy nowe przeciążenia, aby dostosować się do tego rodzaju struktury danych, wynik byłby niejednoznacznym rozwiązaniem.

`Sequence` i `Associative_container` tak naprawdę mają pokrywające się ograniczenia. Oba wymagają konceptu `Range`. Możemy rozważyć te przeciążenia jako przykład trzeciego przypadku. To wskazuje, że może istnieć algorytm, który można zdefiniować w odniesieniu do wymagań przecinających. Ale to nie jest takie proste.

Drugi przypadek jest ważną cechą programowania generycznego w języku *C++* i jest podstawą optymalizacji typów w bibliotekach generycznych. Ograniczenie subsumpcji pozwala na optymalizację generycznych algorytmów opartych na interfejsach dostarczonych przez ich argumenty.

3.2 Specjalizacja algorytmów

W niektórych przypadkach możemy definiować struktury danych z rozszerzonym zestawem właściwości lub operacji, które mogą być wykorzystane do definiowania bardziej dopuszczalnych lub bardziej wydajnych wersji algorytmu. Ten pomysł jest realizowany przez hierarchię iteratorów biblioteki standardowej.

Iteratory forward mogą być użyte do przechodzenia przez sekwencję w jednym kierunku (do przodu) poprzez przesuwanie się po jednym elemencie naraz, używając operatora `++`.

Prosty koncept iteratora forward:

```
template<typename I>
concept bool Forward_iterator() {
```

```
  return Regular<I>() && requires (I i) {
    typename value_type_t<I>;
    { *i } -> const value_type_t<I>&;
    { ++i } -> I&;
  };
}
```

Opierając się na tym koncepcie, możemy zdefiniować dwa użyteczne algorytmy. Jeden, który przechodzi przez iterator wielokrotnymi krokami używając pętli i drugi, który oblicza liczbę kroków między dwoma iteratorami.

```
template<Forward_iterator I>
void advance(I& iter, int n) {
  //(1)
  while (n != 0) { ++iter; --n; }
}
template<Forward_iterator I>
int distance(I first, I limit) {
  (2)
  for (int n = 0; first != limit; ++first, ++n);
  return n;
}
```

(1) - warunek wstępny: `n >= 0` (2) - warunek wstępny: `limit` jest osiągalny z `first`

Parametr `n` funkcji `advance` musi być nieujemny bo *iteratory forward* nie mogą iść do tyłu. Ale *iterator bidirectional* może być użyty do wędrowania po sekwencji w oba kierunki (do przodu i do tyłu) poprzez przechodzenie po elementach naraz używając operatorów `++` lub `--`.

```
template<typename I>
  concept bool Bidirectional_iterator() {
    return Forward_iterator<I>() && requires (I i)
    {
      { --i } -> I&;
    };
```

```
  }
```

Koncept `Bidirectional_iterator` jest zbudowany na podstawie `Forward_iterator`. Czyli *iterator bidirectional* jest *iteratorem forward*, który również może poruszać się do tyłu. Zestaw wymagań konceptu `Bidirectional_iterator` całkowicie zalicza się do zestawu konceptu `Forward_iterator`. W wyniku czego, za każdym razem gdy `Bidirectional_iterator<X>` jest prawdziwe (dla wszystkich `X`), `Forward_iterator<X>` musi tez być prawdziwy. W tym przypadku mówimy, że koncept `Bidirectional_iterator` *udoskonala*[21] koncept `Forward_iterator`.

To *udoskonalenie* pozwala nam zdefiniować nową wersję `advance()`, która może poruszać się w oba kierunki.

```
template<Bidirectional_iterator I>
  void advance(I& iter, int n) {
    if (n > 0)
      while (n != 0) { ++iter; --n; }
    else if (n < 0)
      while (n != 0) { --iter; ++n; }
  }
```

Koncept `Bidirectional_iterator` pozwala nam uspokoić warunek wstępny funkcji `advance()`, dzięki czemu możemy użyć ujemnych wartości `n`. Z drugiej strony `Bidirectional_iterator` nie zawiera żadnych nowych informacji, które mogłyby pomóc nam ulepszyć `distance()`. Możemy jednak zapewnić optymalizację zarówno `advance()` jak i `distance()` dla *iteratorów random access*. Te iteratory mogą być użyte do przebycia sekwencji w dwóch kierunkach, ale mogą posuwać się do wielu elementów w jednym kroku używając operatorów + = lub - =. Możemy również policzyć odległość między dwoma iteratorami, odejmując je.

```
template<typename I>
  concept bool Random_access_iterator() {
    return Bidirectional_iterator<I>() &&
      requires (I i, int n) {
```

[21] (ang. refine)

```
        { i += n } -> I&;
        { i -= n } -> I&;
        { i - i } -> int;
    };
  }
```

Koncept `Random_access_iterator` udoskonala koncept `Bidirectional_iterator`. Dodaje trzy nowe wymagane operacje. Dzięki tym operacjom możemy konstruować zoptymalizowane wersje `advance()` i `distance()`, które nie wymagają pętli.

```
template<Random_access_iterator I>
void advance(I& iter, int n) {
    iter += n;
}
template<Random_access_iterator I>
int distance(I first, I limit) {
    return limit - first;
}
```

Te algorytmy można używać do zdefiniowania dużej liczby użytecznych operacji.

```
template<Forward_iterator I, Ordered T>
  requires Same_as<T, value_type_t<I>>()
bool binary_search(I first, I limit, T const& value) {
  if (first == limit)
    return false;
  auto mid = first;
  advance(mid, distance(first, limit) / 2);
  if (value < *mid)
    return search(first, mid, value);
  else if (*mid < value)
    return search(++mid, limit, value);
  else
    return true;
```

```
}
```

Algorytm jest definiowany dla iteratorów *forward*, ale oczywiście może być używany również do *bidirectional* i *random access*. Wersje `advance()` i `distance()`, które są używane, zależą od typu iteratora przekazanego do algorytmu. W przypadku *iteratorów forward* i *bidirectional*, algorytm jest liniowy w zakresie wielkości wejściowych. W przypadku *iteratorów random access* algorytm jest znacznie szybszy, ponieważ `distance()` i `advance()` nie wymagają dodatkowych przejazdów sekwencji wejściowej.

Zdolność do specjalizacji algorytmów według ograniczeń i typów ma decydujące znaczenie dla wydajności bibliotek generycznych języka *C++*. Koncepty znacznie ułatwiają definiowanie i wykorzystywanie tych specjalizacji. Ale jak kompilator wie, które przeciążenie wybrać?

W poprzednich przykładach wykorzystujących sekwencje i kontenery asocjacyjne, tylko jedno przeciążenie funkcji `czyIstnieje()` było zawsze opłacalne, ponieważ argumenty były jednego lub drugiego, ale nie obu. Jeśli jednak wywołamy `binarny_search()` z *iteratorami random access*, powiedzmy, że są to wskaźniki do tablicy, wszystkie trzy przeciążenia `advance()` i oba przeciążenia `distance()` będą opłacalne. To ma sens. Każda implementacja tych funkcji jest doskonale zdefiniowana dla wskaźników.

W takim przypadku kompilator musi wybrać najlepszego spośród potencjalnych kandydatów. Ogólnie rzecz ujmując, *C++* uważa, że jedna z funkcji jest lepsza od innej za pomocą następujących reguł:

1. Funkcje wymagające mniejszych lub "tańszych" konwersji argumentów są lepsze niż te wymagające większych lub bardziej kosztownych konwersji.

2. Funkcje nieszablonowe są lepsze niż specjalizacje szablonów funkcji.

3. Jedna specjalizacja szablonu funkcji jest lepsza od innej, jej typy parametrów są bardziej wyspecjalizowane. Na przykład `T*` jest bardziej wyspecjalizowany niż `T`, i tak samo `vector<T>`, ale `T*` nie jest bardziej wyspecjalizowany niż `vector<T>`, ani też nie jest przeciwnie.

   Specyfikacja techniczna konceptów dodaje jeszcze jedną zasadę:

4. Jeśli dwie funkcje nie mogą być sortowane, ponieważ mają równoważne konwersje lub są specjalizacjami szablonów funkcji o równoważnych typach parametrów, tym lepsza jest bardziej ograniczona. Są to najmniej ograniczone funkcje nie ograniczone.

Innymi słowy, ograniczenia działają jako łącznik dla zwykłych reguł przeciążania w *C++*. Kolejność ograniczeń (bardziej ograniczona) zależy zasadniczo od porównania zestawów wymagań dla każdego szablonu w celu określenia, czy jest to ścisły nadzbiór drugiego. W celu porównania ograniczeń, kompilator najpierw analizuje powiązane ograniczenia funkcji w celu zbudowania zestawu tak zwanych ograniczeń atomowych. Są *atomowe*, ponieważ nie mogą być podzielone na mniejsze części. Ograniczenia atomowe zawierają wyrażenia stałe *C++* (np. *type traits*) i wymagania w wyrażeniu `requires`.

Na przykład, w rozwiązaniu `advance()`, gdy jest wywołany z *iteratorem random access*, zestaw ograniczeń dla każdego przeciążenia to:

Koncept	Atomowe wymagania

`Forward_iterator`	`value_type_t<I>`
	`{ *i } -> value_type_t<I> const&`
	`{ ++i } -> I&`

`Bidirectional_iterator`	`value_type_t<I>`
	`{ *i } -> value_type_t<I> const&`
	`{ ++i } -> I&`
	`{ --i } -> I&`

`Random_access_iterator`	`value_type_t<I>`
	`{ *i } -> value_type_t<I> const&`
	`{ ++i } -> I&`
	`{ --i } -> I&`
	`{ i += n } -> I&`
	`{ i -= n } -> I&`
	`{ i - j } -> int`

Dla zwięzłości wyłączyłem ograniczenie `Regular<I>` pojawiające się w `Forward_iterator`, ponieważ on(i jego wymagania) są wspólne dla wszystkich konceptów iterujących. Porównując powyższe stwierdzimy, że `Bidirectional_iterator` ma ścisły nadzbiór wymagań `Forward_iterator`, a `Rand-`

`om_access_iterator` ma ścisły nadzbiór wymagań `Bidirectional_iterator`. Z tego względu `Random_access_iterator` jest najbardziej ograniczony i to przeciążenie zostało wybrane. Nowa reguła przeciążania nie gwarantuje, że rozwiązanie przeciążenia odniesie sukces. W szczególności, jeśli dwóch realnych kandydatów ma nakładające się lub logicznie równoważne ograniczenia, rozwiązanie będzie niejednoznaczna. Jest kilka powodów, dla których to miałoby się zdarzyć.

3.3 Semantyczne udoskonalanie

W niektórych przypadkach udoskonalenia są czysto semantyczne. Nie dostarczają operacji, które kompilator może wykorzystać do odróżnienia przeciążeń. W rzeczywistości ten problem pojawia się w standardowej hierarchii iteratorów: *iteratory input* i *iteratory forward* dzielą dokładnie te same zestawy operacji.

Pojęciowo *iterator input* jest iteratorem reprezentującym pozycję w strumieniu wejściowym. Ponieważ jest zwiększany, poprzednie elementy są konsumowane. Oznacza to, że wcześniej dostępne elementy nie są już dostępne przez iterator lub dowolną jego kopię. W przeciwieństwie do tego, *iterator forward* nie konsumuje elementów przy zwiększaniu. Wcześniej dostępne elementy mogą być uzyskane dzięki kopiom. Jest to zazwyczaj określane mianem właściwości *multipass*. Jest to czysto semantyczna własność.

```
template<typename I>
concept bool Input_iterator() {
  return Regular<I>() && requires (I i) {
    typename value_type_t<I>;
    { *i } -> value_type_t<I> const&;
    { ++i } -> I&;
  };
}

template<typename I>
concept bool Forward_iterator() {
  return Input_iterator<I>();
```

```
}
```

Wszystkie wymagania składniowe są zdefiniowane w koncepcie `Input_iterator`. Koncept `Forward_iterator` zawiera tylko `Input_iterators`. Innymi słowy, zestaw wymagań `Forward_iterator` jest dokładnie taki sam, jak `Input_iterator`. Jeśli próbujemy zdefiniować przeciążenia wymagające tych konceptów, wynik byłby zawsze dwuznaczny (ani lepszy od drugiego). Zróżnicowanie pomiędzy tymi konceptami jest tak naprawdę przydatny. Na przykład jeden z konstruktorów `vector` ma bardziej wydajną implementację *iteratorów forward* niż dla *iteratorów input*.

```
template<Object_type T, Allocator_of<T> A>
class vector {
  template<Input_iterator I>
    requires Same_as<T, value_type_t<I>>()
  vector(I first, I limit) {
    for ( ; first != limit; ++first)
      push_back(*first);
  }

  template<Forward_iterator I>
    requires Same_as<T, value_type_t<I>>()
  vector::vector(I first, I limit) {
    reserve(distance(first, limit));
      // 1 allocation
    insert(begin(), first, limit);
  }
  // ...
```

To nie zadziała, jeśli kompilator nie może odróżnić `Forward_iterator` z `Input_iterator`.

Można to naprawić dodając nowe wymagania składniowe do `Forward_iterator`, które odnoszą się do jego rangi w hierarchii iteratorów. To tradycyjnie zostało zrobione przy użyciu *tag dispatch*. Łączenie *etykiety klasy*[22] z typem

[22](ang. tag class) Pusta klasa w hierarchii dziedziczenia

iteratora w celu wybrania odpowiedniego przeciążenia. Ten skojarzony typ to `iterator_category`. Zmieniony `Forward_iterator` może wyglądać tak:

```
template<typename I>
  concept bool Forward_iterator() {
    return Input_iterator<I>() && requires {
      typename iterator_category_t<I>;
      requires Derived_from<I,
        forward_iterator_tag>();
    };
  }
```

Dzięki tej definicji wymagania `Forward_iterator` zaliczają wymagania `Input_iterator`, a kompilator może rozróżnić powyższe przeciążenia. Jako dodatkowa zaleta, używanie *iteratorów random access* będzie jeszcze bardziej wydajne bo `distance()` wymaga tylko jednej operacji całkowitej.

Jako inny przykład, *C++17* dodaje nową kategorię iteratorów: *iteratory contiguous*. *Iterator contiguous* jest *iteratorem random access*, którego obiekty odwoławcze są przydzielane w sąsiednich obszarach pamięci, których adresy rosną wraz z każdym przyrostem iteratora. Powoduje to otwarcie drzwi na wiele optymalizacji pamięci na niższym poziomie. Jest to oczywiście zupełnie czysta semantyka. Jeśli chcemy zdefiniować nowy koncept, musimy ją odróżnić od `Random_access_iterator`. Na szczęście właśnie zdefiniowaliśmy maszynę, aby to zrobić.

```
template<typename I>
concept bool Contiguous_iterator() {
  return Random_access_iterator<I>() && requires {
    requires Derived_from<I,
    contiguous_iterator_tag>();
  };
}
```

4 Implementacja algorytmu Fleury'ego jako przykład wykorzystujący koncepty

4.1 Omówienie problemu

Algorytm Fleury'ego to algorytm pozwalający na znalezienie *cyklu Eulera* w *grafie eulerowskim.*

Definicja 1. ***Graf*** *- struktura służąca do przedstawiania i badania relacji między obiektami. Jest to zbiór wierzchołków, które mogą być połączone krawędziami, gdzie krawędź zaczyna się i kończy w którymś z wierzchołków.*

Definicja 2. ***Multigraf*** *- graf, w którym mogą występować krawędzie wielokrotne(powtarzające się) oraz pętle (krawędzie, których końcami jest ten sam wierzchołek).*

Definicja 3. ***Graf spójny*** *- graf spełniający warunek, że dla każdej pary wierzchołków istnieje ścieżka, która je łączy.*

Definicja 4. ***Ścieżka*** *- ciąg wierzchołków, połączonych krawędziami.*

Definicja 5. ***Droga*** *- ścieżka, w której wierzchołki są różne.*

Definicja 6. ***Cykl*** *- droga zamknięta czyli ścieżka, w której pierwszy i ostatni wierzchołek są równe.*

Definicja 7. ***Graf eulerowski*** *- spójny multigraf posiadający cykl, który zawiera wszystkie krawędzie.*

Definicja 8. ***Warunek istnienia cyklu Eulera w spójnym multigrafie*** *- stopień każdego wierzchołka musi być liczbą parzystą.*

Data: G = (V,E), G - spójny multigraf, V - zbiór wierzchołków, E - zbiór krawędzi

Result: zbiór wierzchołków reprezentujących cykl Eulera

Zaczynamy od dowolnego wierzchołka ze zbioru V;

while *Dopóki zbiór krawędzi nie jest pusty* **do**

  if *Jeżeli z bieżącego wierzchołka x odchodzi tylko jedna krawędź* **then**

    to przechodzimy wzdłuż tej krawędzi do następnego wierzchołka i usuwamy tę krawędź wraz z wierzchołkiem x;

  else

    wybieramy tę krawędź, której usunięcie nie rozspójnia grafu i przechodzimy wzdłuż tej krawędzi do następnego wierzchołka, a następnie usuwamy tę krawędź z grafu;

  end

end

Algorithm 1: Algorytm Fleury'ego

4.2 Działanie programu

Założeniem programu jest symulacja algorytmu Fleury'ego dla jak największej ilości kontenerów biblioteki STL. Dzięki przeciążaniu funkcji jakie oferują koncepty, w prosty i czytelny sposób, udało się napisać generyczny algorytm.

W programie są dwa kontenery do przechowywania krawędzi i wierzchołków. Krawędź reprezentowana jest przez klasę `Edge`, która przyjmuje dwie wartości typu `int` do konstruktora. Wierzchołek, z kolei reprezentowany jest przez zmienną `int`.

Dane (pary wierzchołków) są wczytywane z pliku, a potem w zależności od rodzaju kontenera i iteratora, sortowane. Dla kontenerów:

- sekwencyjnych z iteratorem `Random access` (`vector`) wywoływana jest funkcja szablonu ograniczonego przez koncepty: `Sequence` i `Random_access_iterator`.

```
template<Sequence S, Random_access_iterator R>
void sortVertices(S &seq){
    sort(seq.begin(), seq.end());
```

```
}
```

- sekwencyjnych z iteratorem `Bidirectional` (`list`) wywoływana jest funkcja szablonu ograniczonego przez koncepty: `Sequence` i `Bidirectional_iterator`.

```
template<Sequence S, Bidirectional_iterator R>
void sortVertices(S &seq){
    seq.sort();
}
```

- asocjacyjnych (`set`) wywoływana jest funkcja szablonu ograniczonego przez koncept: `Associative_container` .

```
template<Associative_container A>
void sortVertices(A &seq){}
```

Omawiany algorytm wykonuje funkcja `determineEulerCycle`:

```
template<typename E, typename V>
void determineEulerCycle(E &edges, V &vertices){
    int v = 0;
    bool condition = (checkIfGraphConnected(edges,
    vertices, 0, v) && checkIfAllEdgesEvenDegree
    (edges, vertices));

    if(condition){
        cout << "Euler cycle:" << endl << endl << v;
        while(!edges.empty()){
            switch(getNeighboursCount(edges, v)){
                case 1 : {
                    removeEdgeWithOneNeighbour(edges, v);
                    break;
                }
                default: {
                    removeEdgeWithMoreNeighbour(edges,
                    v, vertices);
                    break;
                }
            }
            cout <<" -> "<<v;
        }
        cout << endl << endl;
     } else {
         cout<<"Invalid graph."<<endl;
         if(!checkIfGraphConnected(edges, vertices,
         0, v))
             cout <<"Graph is not connected"<<endl;
         else if(!checkIfAllEdgesEvenDegree(edges,
         vertices))
             cout <<"Not all the edges are even"<<endl;
```

```
    }
}
```

Żeby algorytm się wykonał, muszą zostać spełnione dwa warunki: graf musi być spójny(za to odpowiedzialna jest funkcja `checkIfGraphConnected`) i wszystkie krawędzie muszą być parzystego stopnia (`checkIfAllEdgesEvenDegree`).

`checkIfGraphConnected()`

```
template<typename E, typename V>
bool checkIfGraphConnected(E &ed, V &vertices,
int x, int startVertice) {

    bool *visited = new bool[vertices.size()];
    for (int i = 0; i < vertices.size(); i++)
        visited[i] = false;

    stack<int>stack;
    int vc = 0;

    stack.push(startVertice);
    visited[startVertice] = true;

    while (!stack.empty()) {
        int v = stack.top();
        stack.pop();
        vc++;

        for(typename E::iterator it = ed.begin();
            it != ed.end(); it++){
            if(it->getA() == v && !visited[it->getB()]){
                visited[it->getB()] = true;
                stack.push(it->getB());
            } else if(it->getB() == v &&
            !visited[it->getA()]){
                visited[it->getA()] = true;
```

```
                stack.push(it->getA());
            }
        }
    }

    delete [] visited;

    return (vc == vertices.size()-x);

}
```

Algorytm przechodzi przez graf, po kolei wrzucając odwiedzane wierzchołki na stos, zaznaczając je w tablicy odwiedzonych (`visited`) i zaraz zdejmuje z tego stosu, zwiększając licznik `vc`. Robi to dopóki stos nie jest pusty. Zwraca warunek porównujący licznik `vc` z rozmiarem kontenera wierzchołków (wszystkie wierzchołki zostały odwiedzone, czyli istnieją ścieżki między wierzchołkami, graf jest spójny).

`checkIfAllEdgesEvenDegree`:

```
template<typename E, typename V>
bool checkIfAllEdgesEvenDegree(E &edges, V &vertices){
    int counter = 0, i = 0;
    for(auto v : vertices){
        for(auto e : edges){
            if(e.getA() == v || e.getB() == v)
                counter++;
        }
        if(counter % 2 == 0) i++;
    }
    return (i == vertices.size()) ? true : false;
}
```

Zmienna `i` zwiększa się jeśli ilość wystąpień wierzchołka jest liczbą parzystą. Zwraca wartość `true` jeśli zmienna `i` jest równa liczbie elementów kontenera zawierającego wierzchołki(dla każdego wierzchołka zmienna `i` zwięk-

szała się o 1).

Jeśli warunek nie zostanie spełniony, użytkownik zostaje poinformowany o tym, że graf jest niepoprawny. W odwrotnej sytuacji, w pętli (dopóki kontener krawędzi nie jest pusty), wykonywana jest jedna dwóch operacji. Gdy wierzchołek ma jednego sąsiada, wywołuje się funkcja `removeEdgeWithOneNeighbour()`, a gdy więcej wierzchołków, funkcja `removeEdgeWithMoreNeighbour()`. Pierwsza z nich ma dwa przeciążenia konceptowe:

- Dla kontenera sekwencyjnego:

```
template<Sequence S>
void removeEdgeWithOneNeighbour(S &edges, int &v){

    typename S::iterator it = find(edges.begin(),
    edges.end(), v);

    if(it->getA() == v) v = it->getB();
    else v = it->getA();

    edges.erase(it);
}
```

- Dla kontenera asocjacyjnego:

```
template<Associative_container A>
void removeEdgeWithOneNeighbour(A &edges, int &v){

    typename A::iterator it2;
    for(typename A::iterator it = edges.begin();
    it != edges.end(); it++){
        if(it->getA() == v || it->getB() == v){
            it2 = it;
            if(it->getA() == v) v = it->getB();
            else v = it->getA();
            it = prev(edges.end());
        }
```

```
    }

    edges.erase(it2);
}
```

Funkcja znajduje krawędź, dostając wierzchołek wychodzący. I wierzchołek znalezionej krawędzi przypisuje do tego przekazanego.

Druga `removeEdgeWithMoreNeighbour()` wygląda:

```
template<typename E, typename V>
void removeEdgeWithMoreNeighbour(E &edges, int &v,
V &vertices){
    for(typename E::iterator i = edges.begin();
    i != edges.end(); i++){
        if(i->getA() == v &&
        checkIfStillConnected(edges, *i,
        getZeroDegreeCount(edges, vertices), v,
        vertices)){
            v = i->getB();
            edges.erase(i);
            i = prev(edges.end());
        } else if(i->getB() == v &&
        checkIfStillConnected(edges, *i,
        getZeroDegreeCount(edges, vertices), v,
        vertices)){
            v = i->getA();
            edges.erase(i);
            i = prev(edges.end());
        }
    }
}
```

Jeśli wierzchołek ma więcej sąsiadów, wybiera tego który nie rozspójni grafu. Żeby to sprawdzić używa funkcji `checkIfStillConnected`:

```
template<Associative_container E, typename V>
bool checkIfStillConnected(E &edges, Edge e, int x,
int startVertice, V &vertices){

    E tmp;

    for(auto e : edges)
        tmp.insert(e);

    for(typename E::iterator it = tmp.begin();
    it != tmp.end(); it++)
        if (it->getA() == e.getA() &&
            it->getB() == e.getB()) {
            tmp.erase(it);
            it = prev(tmp.end());
        }

    return checkIfGraphConnected(tmp, vertices,
    x, startVertice);
}
```

W celu sprawdzenia, czy graf po usunięciu jakiejś krawędzi dalej będzie spójny, potrzebny jest pomocniczy kontener. Zapisujemy do niego aktualne krawędzie, wyszukujemy w nim przekazaną i przekazujemy go do istniejącej już funkcji `checkIfGraphConnected()`.

5 Włączenie konceptów do standardu C++

Koncepty nie zostały włączone do standardu *C++17*. Krótkie uzasadnienie jest takie, że komisja nie osiągnęła porozumienia, że koncepty (określone w specyfikacji technicznej) osiągnęły wystarczające doświadczenie w zakresie wdrożenia i użytkowania, aby być wystarczające do dopuszczenia w obecnym projekcie. Zasadniczo komisja nie powiedziała "nie", ale "jeszcze nie".

Największe zastrzeżenia nie wynikały z problemów technicznych. Powstały następujące obawy:

- specyfikacja konceptów istniała w opublikowanej formie przez mniej niż cztery miesiące
- jedyna znana i dostępna publicznie implementacja konceptów znajduje się w nieopublikowanej wersji *kompilatora GCC*
- implementacja *kompilatora GCC* została opracowana przez tę samą osobę, która napisała specyfikację. W związku z tym implementacja jest dostępna do testowania, ale nie podjęto żadnej próby wprowadzenia w życia specyfikacji. A zatem specyfikacja nie jest przetestowana. Kilku członków komisji wskazało, że posiadanie implementacji wyprodukowanej ze specyfikacji ma decydujące znaczenie dla określenia kwestii specyfikacji.
- najbardziej znaczące i znane użycie konceptów jest dostępne w specyfikacji *Ranges TS*. Jest kilka innych projektów eksperymentujących z konceptami, ale żaden z nich nie zbliża się do skali, której można oczekiwać gdy programiści zaczną korzystać z tej funkcjonalności. Wydajność i problemy związane z obsługą błędów przy użyciu bieżącej implementacji GCC dowodzą, że nie wykonano większej próby używania konceptów.
- specyfikacja konceptów nie dostarcza żadnych definicji. Niektórzy członkowie komisji kwestionują użyteczność konceptów bez dostępności biblioteki definicji konceptów, takiej jak *Ranges TS*. Przyjęcie specyfikacji konceptów do *C++17* bez odpowiedniej biblioteki definicji niesie

ryzyko zablokowania języka bez udowodnienia, że zawiera funkcje potrzebne do wdrożenia biblioteki, które mogłyby być zaprojektowane do konceptualizacji biblioteki standardowej.

Obawy techniczne:

- koncepty zawierają nową składnię do definiowania szablonów funkcji. Skrócona deklaracja szablonu funkcji wygląda podobnie to nieszablonowej deklaracji funkcji z wyjątkiem tego, że co najmniej jeden z jej parametrów zostanie zadeklarowany ze specyfikatorem typu zastępczego `auto` albo nazwą konceptu. Obawa wynika z tego, że taka deklaracja:
  `void f(X x){}`
  definiuje nieszablonową funkcję jeśli `X` jest typem, ale definiuje szablon funkcji jeśli `X` jest konceptem. To ma subtelne konsekwencje dla tego czy funkcja może być zdefiniowana w pliku nagłówkowym, czy słowo kluczowe `typename` jest potrzebne by odnieść się do składowych typów typu `X`, czy istnieje dokładnie jedna zmienna lub brak lub kilka dla każdej deklarowanej zmiennej lokalnej, statycznej. itd.

- specyfikacja konceptów zawiera również składnię szablonów wstępnych, która pozwala ominąć rozwlekłą składnię deklaracji szablonu, do której wszyscy są przyzwyczajeni jednocześnie określając ograniczenia typu. Następujący przykład deklaruje szablon funkcji `f`, przyjmujący dwa parametry `A` i `B`, które spełniają wymagania konceptu `C`:
  `C{ A, B } void f(A a, B b);`
  Ta składnia nie jest lubiana. Wspomniano, że biblioteka *Ranges TS* jej używała w pewnych miejscach a grupa pracująca nad ewolucją biblioteki zażądała żeby ją zmienić i już nigdy nie używać.

- Są dwie formy definiowania konceptów: funkcja i zmienna. Forma funkcji istnieje po to by wspierać przeciążanie definicji konceptów oparte na parametrach szablonu. Forma zmiennej istnieje by wspierać nieco krótsze definicje:

```
//forma funkcji
template<typename T>
```

```
concept bool C(){
   return ...;
}

//forma zmiennej
template<typename T>
concept bool C = ...;
```

Wszystkie koncepty, które można zdefiniować przy użyciu formy zmiennej można zdefiniować za pomocą formy funkcji. Stosowana forma wpływa na składnię wymaganą do oszacowania konceptu, a zatem użycie konceptu wymaga znajomości formy użytej do jego zdefiniowania. Wczesna wersja *Ranges TS* używała zarówno formy zmiennej, jak i funkcji do definiowania konceptów i niespójność spowodowała wiele błędów w specyfikacji. Aktualna *Ranges TS* wykorzystuje tylko formę funkcji do zdefiniowania określonych konceptów. Niektórzy członkowie komitetu uważają, że jedna forma definicji uprości język i uniknie trudności w używaniu i nauczaniu. Zapewnienie odrębnej składni definicji konceptów, a nie określenie ich w kategoriach funkcji lub zmiennych uniknęłoby również dziwnej składni `concept bool`.

- została dodana możliwość używania `auto` jako specyfikatora dla parametrów szablonu bez typu:

```
template<auto V>
constexpr auto v = V*2;
```

Z konceptami można by ograniczyć powyższy szablon tak, że typ `V` spełniałby wymagania konceptu `Integral`:

```
template<Integral V>
constexpr auto v = V*2;
```

Jednak to jest ta sama składnia aktualnie używana przez *Concepts TS*, do deklarowania parametrów typu szablonu ograniczonego. Jeśli *Con-*

cepts TS miały być wprowadzone, potrzebna by była inna składnia aby deklarować ograniczony parametr szablonu bez typu. Prawdopodobnie składnia stosowana przez *Concepts TS* bardziej nadaje się do deklarowania parametrów szablonów bez typu, jak pokazano powyżej, ponieważ pasuje do składni stosowanej dla innych deklaracji zmiennych. To oznacza, że nowa składnia do deklarowania ograniczonych parametrów typu byłaby pożądana ze względu na spójność języka.

- Koncepty były powszechnie oczekiwane w celu uzyskania lepszych komunikatów o błędach niż obecnie są generowane, gdy pojawiają się niepowodzenie podczas tworzenia szablonów. Teoria idzie, ponieważ koncepty pozwalają odrzucić kod oparty na ograniczeniu w punkcie użycia szablonu, kompilator może po prostu zgłosić błąd ograniczenia, a nie błąd w niektórych wyrażeniach w potencjalnie głęboko zagnieżdżonym stosie instancji szablonu. Niestety okazuje się, że nie jest tak proste, a używanie konceptów skutkuje gorszymi komunikatami o błędach. Niepowodzenia ograniczeń często pojawiają się jako błędy w przeciążeniu, co powoduje potencjalnie długą listę kandydatów, z których każda ma własną listę przyczyn odrzucenia. Identyfikacja kandydata, który był przeznaczony do danego użycia, a następnie określenie, dlaczego wystąpiło niepowodzenia ograniczeń, może być gorszym doświadczeniem niż nawigowanie w stosie tworzenia instancji szablonów.

- Wielu członków komisji wyraża zaniepokojenie faktem, czy obecny projekt konceptów wystarcza jako podstawa, na której można w przyszłości wdrożyć sprawdzenie pełnej definicji szablonu. Mimo zapewnień obrońców konceptów, że takie kontrole będą możliwe, wiele pytań pozostaje bez odpowiedzi, a członkowie komitetu pozostają bez przekonania. Wydaje się mało prawdopodobne, że obawy te zostaną rozwiązane w inny sposób niż poprzez wdrożenie sprawdzania definicji.

Wielu wierzy, że koncepty w jakiejś formie zostaną dodane do *C++19/20*.

6 Bibliografia

Literatura

[1] Gabriel Dos Reis, *Generic Programming in C++: The Next Level.*, ACCU, 2002.

[2] Bjarne Stroustrup, *The Design and Evolution of C++*, AddisonWesley, 1994

[3] Bjarne Stroustrup, *Expressing the standard library requirements asconcepts*

[4] Gabriel Dos Reis, Bjarne Stroustrup, *Specifying C++ Concepts*

[5] J. C. Dehnert, A. Stepanov, *Fundamentals of Generic Programming*, Dagstuhl Seminar on Generic Programming.1998. Springer LNCS.

[6] A. Stepanov, Daniel E.Rose, *From Mathematics to Generic Programming*

[7] D. Gregor, J. Jarvi, J. Siek, B. Stroustrup, G. Dos Reis, A. Lumsdaine, *Concepts: Linguistic Support for Generic Programming in C++*, OOPSLA'06.

[8] Scott Meyers, *Effective Modern C++*, O'REILLY 2015