Koncepty

Maciej Zbierowski

6 września 2017

Spis treści

Wstęp

Pomysł ograniczania argumentów szablonów jest tak stary jak stare są same szablony. Ale dopiero z początkiem dwudziestego pierwszego wieku zaczęły się poważne prace nad projektem języka 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 którymi 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 nie zostały włączone w *C++17*. Niektórzy członkowie komisji uważali że nie minęło wystarczająco dużo czasu od publikacji specyfikacji technicznej, żeby sprawdzić czy projekt jest wystarczająco dobry, a wielu z nich było niezdecydowanych.

[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)

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 tym pierwszym używa 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 on na dostarczeniu szablonu, który umożliwia kompilatorowi wygenerowanie kodu w czasie kompilacji.

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.

```
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* 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.

- 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 specyfikowane 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 operacje 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 widzi 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`, tablicę 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 `--` 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 jego wymagań na jego 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*
- 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++*. Jakkolwiek, *inlining* jest istotny dla drobno-granularnej parametryzacji, którą powszechnie stosuje się w bibliotece *STL* i innych bibliotekach bazujących na generycznych technikach programowania. 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>` 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.

Aby kontrastować, rozważ bardziej konwencjonalny przypadek, w którym 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 wysyłki czasu wykonania ma miejsce. Szablony nie cierpią na ten problem bo ich specjalizacje są rozwiązywane w

czasie kompilacji.

Przykład funkcji `suma` nie jest idealny do zilustrowania subtelności generowania kodu obiektu z kodu źródłowego znalezionego w różnych częściach programu. Nie polega na niejawnych konwersjach lub nietypowych przeciążaniach. Jednak, rozważ wariant gdzie wartości int są sumowane w obiekcie `zespolona<double>`:

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

Tu dodawanie jest wykonane 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 zawsze 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.

Nie spodziewalibyśmy się zobaczyć tego ostatniego przykładu bezpośrednio w kodzie źródłowym. Gdybyśmy go zobaczyli, uznaliśmy, że jest on źle napisany. Jednakże, równoważny kod jest powszechny w wyniku zagnieżdżonych abstrakcji. Jest to szczególnie ważne by generować dobry kod w takich przypadkach ponieważ nie robienie tego byłoby zniechęcające dla abstrakcji.

Warto zauważyć, że te optymalizacje są wspólnym miejscem. Duże ilości prawdziwego oprogramowania zależą od nich. W konsekwencji udoskonalone sprawdzanie typu, co zostało obiecane przy użyciu konceptów, nie może kosztować tych optymalizacji.

2 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*[3]
- lepszą wydajność (w porównaniu do kodu pisanego ręcznie)
- kiepskie interfejsy (praktycznie *typowanie kaczkowe czasu kompilacji*)[4]

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*[5]) 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 (`&&`, `||`, `!`)

[3](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

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

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

2.1 Ulepszenie programowania generycznego

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

- 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:

- wiadomosc 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 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

2.2 System konceptów

Reprezentacja definicji szablonu w C++ to zazwyczaj drzewa wyprowadzania[6]. Używając identycznych technik kompilatora, możemy przekonwertować koncepty do drzew wyprowadzania. Posiadając to możemy zaimplementować sprawdzanie konceptów jako abstrakcyjne drzewo dopasowań[7]. Wygodnym sposobem implementowania takiego dopasowywania jest generowanie

[6](ang. Parse Trees)
[7](ang. Abstract Tree Matching)

i porównywanie zestawów wymaganych funkcji i typów (zwane *zestawami ograniczeń*) z definicji szablonów i konceptów.

Definicja konceptu to zestaw równań *drzewa AST*[8] 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.

2.3 Definicja konceptu

Rozróżniamy dwa rodzaje konceptów:

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;
```

[8](ang. Abstract Syntax Tree)

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 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*[9] z metaprogramowania, co znacznie upraszcza metaprogramowanie, a także programowanie generyczne.

2.5 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);
```

[9]metaprogramistyczna 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

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

2.6 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.

2.7 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 zwykłe *aliasy szablonu*[10]. 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.

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 `<`:

[10] ALIAS SZABLONU

```
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`.

2.8 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*[11] 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*[12], 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);

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

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

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

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*[13]. Kompilator przetwarza hierarchię dla użytkownika. To jest prostsze, bardziej elastyczne i mniej podatne na błędy.

Przeciążanie oparte na konceptach eliminuje znaczącą ilość *boiler-plate*[14] z programowania generycznego i kodu meta programowania (użycia `enable_if` [15]).

Funkcja `czyZnaleziono` ocenia czy element znaduje się w sekwencji

```
template<Sequence S, Equality_comparable T>
    requires Same_as<T, value_type_t<S>>
```

[13] klasy traits
[14] BP
[15] EI

```
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>;
        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;
    };
};
```

3 Rozdział 3

Zawartość rozdziału 3

4 Rozdział 4

Zawartość rozdziału 4

5 Rozdział 5

Zawartość rozdziału 5

6 Rozdział 6

Zawartość rozdziału 6

7 Bibliografia

Literatura