Koncepty: Przyszłość programowania generycznego
lub
Jak projektować dobre koncepty i używać ich dobrze

Abstrakcja

Szybko opiszę **koncepty(wymagania argumentów szablonu)** zdefiniowane przez ISO Technical Specification [C++15] i załadowane (shipped) jako część kompilatora GCC [GCC16]. Dążę do tego żeby zrozumieć cele konceptów, ich zasad projektowych i ich podstawowych zasad użycia:

1. Tło projektowania konceptów
2. Koncepty jako podstawa programowania generycznego
3. Podstawowe użycie konceptów jako wymagań na argumenty szablonu
4. Definicja konceptów jako wartości boolowskie (predykaty)
5. Projektowanie z konceptami
6. Użycie konceptów w celu rozwiązania przeciążeń(The use of concepts to resolve overloads)
7. Krótkie formy (The short-form notations)
8. Pytania dotyczące projektowania języka

Użycie konceptów nie ciągnie za sobą żadnych kosztów czasu wywołania w porównaniu do tradycyjnych nieograniczonych szablonów. Są to tylko mechanizmy selekcji a po selekcji wygenerowany kod jest identyczny to tradycyjnego kodu szablonu.

1. W 1987 próbowano projektować szablony z odpowiednimi interfejsami. Nie udało się. Chcieli trzech właściwości dla szablonów:

- Pełna ogólność / wyrazistość
- zero kosztów('overhead') w porównaniu do kodowania ręcznego
- dobrze określone interfejsy

Potem nikt nie mógł dojść jak otrzymać te 3 cechy, więc dostali

- kompletność Turinga (Turing completeness)
- lepiej niż wydajność kodu pisanego ręcznie
- parszywe (lousy) interfejsy (praktycznie typowanie kaczkowe czasu kompilacji – basically compile-time duck typing)

Brak dobrze określonych interfejsów prowadzi do spektakularnie złych wiadomości błędów, które widziało się na przełomie lat. Dwie pozostałe właściwości uczyniły z szablonów sukces.

Brak dobrze określonych interfejsów wkurzało i inne rzeczy też, bo szablony nie spełniały podstawowych kryteriów C++. Wielu próbowało znaleźć rozwiązanie, szczególnie członkowie komisji standardu C++, dążąca do C++0x [C++09] ale do niedawna nikt nie wymyślił czegoś co spełniłoby wszystkich trzech oryginalnych założeń, wmontowane w C++ i kompilowane rozsądnie szybko.

Zauważ, że cele projektowe dla szablonów są przykładem ogólnych celów projektowych C++:

- ogólność
- zero 'overhead'
- dobrze zdefiniowane interfejsy

Rozwiązaniem problemu specyfikacji interfejsu zostało nazwane „konceptami” przez *Alexa Stepanova*. **Koncept** to zbiór wymagań zbioru argumentów szablonu. Pytanie brzmi: jak uformować (craft) zestaw cech języka by wspierał ten pomysł.

Razem z Gabriel Dos Reis i Andrew Sutton, zaczęli projektować koncepty od zera w 2009 roku. W 2011, Alex Stepanov zwołał spotkanie w Palo Alto, gdzie duża grupa, włączając Sean Parent i Andrew Lumsdaine, zaatakowali problem z punktu widzenia użytkownika: Jak wyglądałby prawidłowo ograniczony STL? Potem, poszli do domu wymyślić mechanizmy języka aby przybliżać do tego idealnego. To ponownie uruchomiło wysiłek standardów oparty na nowym, zasadniczo innym i lepszym podejściu niż wysiłek C++0x. Teraz mamy ISO TS (Technical Specification) dla konceptów [C++15]. Implementacja Andrew Suttona była używana przez 3 lata i teraz jest ładowana jako część GCC[GCC16]

2. Programowanie generyczne

Musimy uprościć programowanie generyczne w C++. Sposób w jaki dzisiaj piszemy generyczny kod jest różny od sposobu w jaki piszemy inny kod. Rozważmy

```
//tradycyjny kod
double sqrt(double d); // C++84: akceptuje d że jest typu double

double d = 7;
double d2 = sqrt(d); // ok: d jest typu double

vector<string> vs = {”Good”, ”old”, ”template”};
double d3 = sqrt(vs); //błąd: vs nie jest double
```

To jest rodzaj kodu z jakim zapoznaliśmy się na początku nauki programowania. Mamy fukcję **sqrt** określoną aby wymagała typu **double.** Jeśli damy typ **double** (tak jak w **sqrt(d)**) wszystko będzie w porządku i jeśli damy coś innego niż **double** (tak jak w **sqrt(vs)**) od razu otrzymamy pomocną wiadomość błędu, taką jak **”vector<string>** nie jest **double”**

W porównaniu:

```
//1990 – styl kodu generycznego
template<class T> void sort(T& c) // C++98: akceptuje zmienną c jakiegokolwiek typu T
{
        //kod sortowania zależący od różnych właściwości T, takich jak posiadanie [] i wartości typu
        //z <
}

vector<string> vs = {”Good”, ”old”, ”templates”};
sort(vs); // ok, vs ma wszystkie semantyczne właściwości wymagane przez sort

double d = 7;
sort(d);
```

Mamy problemy:

- Jak już zapewne wiesz, wiadomość błędu którą dostaliśmy z **sort(d)** jest wielomówny i nigdzie bliski precyzyjnemu i pomocnemu
- Żeby użyć **sort** musimy dostarczyć jego definicję, zamiast tylko deklaracji, to odróżnia od zwykłego kodu i zmienia model organizowanego kodu
- Wymagania argumentów **sortu** są domniemane(”ukryte”) w ich ciałach funkcji
- Wiadomość błędu dla **sort(d)** pojawi się tylko gdy szablon będzie inicjowany, a to może być długo po punkcie wywołania
- Notacja **template<typename T>** jest unikalna, dużo mówiąca, powtarzająca się i szeroko

nielubiąca (disliked)

Używając konceptu, możemy dostać się do korzenia problemu przez prawidłowe określanie wymagań argumentów szablonu:

```
//Generyczny kod używający konceptu (Sortable)
void sort(Sortable& c); // Koncepty: akceptuj jakąkolwiek zmienną c która jest Sortable

vector<string> vs = {”one”, ”two”, ”three”};
sort(vs); // ok: vs jest kontenerem Sortable

double d = 7;
sort(d);  //error: d nie jest Sortable (double nie dostarcza [], itd.)
```

Kod jest analogiczny do przykładu **sort.** Jedyna prawdziwa różnica to

- dla **double,** projektant języka (Dennis Ritchie) wbudował to w kompilator jako konkretny typ ze znaczeniem określonym w dokumentacji
- dla **Sortable** użytkownik określił co znaczy w kodzie.Krótko, typ jest **Sortable** jeśli posiada **begin()** i **end()** dostarczające randomowy dostęp do sekwencji z elementami które mogą być porównane używając operatora <

Teraz mamy wiadomość błędu ”d nie jest Sortable (double nie dostarcza [], itd.)”. Wiadomość jest generowana natychmiast w momencie gdzie kompilator widzi błędne wywołanie (**sort(d)**)

Celem jest zrobienie:

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

Koncepty same w sobie nie kierują organizacji kodu w inną stronę (Concepts by themselves do not adddress the code organization difference). Wciąż musimy wsadzać szablony w nagłówki (headers). Jednakże, jest do kierowane (addressed) przez moduły [Rei16]. W module szablon jest reprezentowany przez abstrakcyjnie typowany graf (typed abstract graph) i żeby sprawdzić wywołanie funkcji szablonu używającej konceptów potrzebne są tylko ich interfejsy (deklaracje).

3. Używanie konceptów

Koncept to predykat czasu kompilacji (czyli coś co daje wartość logiczną). Np. argument typu szablonu **T,** mógłby być wymaganym żeby być

- iteratorem **Iterator<T>**
- iteratorem randomowego dostępu: **Random_access_iterator<T>**
- liczbą: **Number<T>**

Notacja **C<T>** gdzie **C** to koncept a **T** to typ, to wyrażenie mające wartość: **true** jeśli **T** spełnia wszystkie wymagania konceptu **C** i **false** w przeciwnym wypadku.

Podobnie, możemy określić, że zestaw argumentów szablonu musi spełniać predykat, np. **Mergeable<In1, In2, Out>.** Takie predykaty **multi-type** są ważne do opisania STL i większość domen aplikacji. Są bardzo wyraziste i ładnie tanie (nicely cheap) do kompilowania (tańsze niż metaprogramowanie szablonów). Studenci mogą używać tego po wykładzie czy dwóch. Możesz oczywiście, zdefiniować swoje własne koncepty i możemy mieć biblioteki konceptów. Koncepty umożliwiają przeciążanie i eliminowanie potrzeby na mnóstwo metaprogramowania na poczekaniu (ad hoc) i wielu programowania kodu rusztowania (metaprogramming scaffolding code) a tym samym znacznie upraszczają metaprogramowanie oraz programowanie generyczne.

3.1 Określanie interfejsów szablonu

Najpierw zobaczmy jak możemy użyć koncepty to określenia algorytmów. Rozważ wariant **std::find** który bierze sekwencję zamiast pary iteratorów

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

- To jest szablon który bierze dwa argumenty typu szablonu
- Pierwszy argument szablonu musi być sekwencją (**Sequence<S>**) i musimy być w stanie porównać elementy sekwencji do **value** używając operatora == (**Equality_comparable<Value_type<S>, T>**)
- **find()** bierze swoją sekwencję przez referencję i **value** do znalezienia jako referencję const. Zwraca iterator

Sekwecja to coś co zawiera **begin()** i **end(),** ale żeby zrozumieć deklarację **find()** to nieistotne.

Użyto aliasu szablonu by móc powiedzieć **Value_type<S>** i **Iterator_of<S>.** Najprostsze definicje byłyby to:

```
template<typename X> using Value_type<X> = X::value_type;
template<typename X> using Iterator_of<X> = X::iterator;
```

Alias szablonu nie ma nic szczególnie wspólnego z konceptami. Są po prostu użyteczne to wyrażenia generycznego kodu. Spodziewaj się takich aliasów w bibliotekach.

Koncept **Equality_comparable** jest proponowany jako koncept standardowej biblioteki. Wymaga że jego argumenty dostarczają operatory == i !=. Zauważ, że **Equality_comparable** bierze dwa argumenty. Wiele konceptów bierze więcej niż jeden argument: koncepty mogą opisywać nie tylko typy, ale również zależności wśród typów. To jest kluczowe dla wielu STL (dla których te relacje są opisywane w standardowej bibliotece) i większość innych bibliotek. Koncept jest nie tylko ”typen typu”.

Możemy spróbować użyć **find():**

```
void use(vector<string>& vs, list<double>& lstd){
        auto p0 = find(vs, ”Waldo”); // OK
        auto p1 = find(vs, 0.54); // error: can't compare a string and a double
        auto p2 = find(lstd, 0.54); // OK
        auto p3 = find(lstd, ”Waldo”); //erro: can't compare a double and a string

        if(p0 != vs.end()) { /* znaleziono Waldo */}
}
```

To tylko jeden przykład i całkiem prosty ale używanie tylko tych technik z tego przykładu, opisaliśmy wszystkie algorytmy STL które są znane jako ”The Palo Alto TM”. Możesz znaleźć wiele więcej przykładów użycia konceptów w Ranges TS[Niel5] (który spodziewamy się że przekształci się w STL2) i [Sut15, Sut16, Sut17]. The Palo Alto TM zostało dokumentem projektowym, ale Ranges TS jest skompilowanym i wytestowanym kodem.

3.2 Notacja 'shorthand'

Gdy wymagamy argument szablonu żeby był sekwencją, możemy powiedzieć
template<typename Seq> requires Sequence<Seq>
void algo(Seq& s);

Potrzebujemy argumentu typu **Seq** który musi być **Sequence.** Zrobiono to przez powiedzenie: ”szablon bierze argument typu; ten typ musi być **Sequence**”. To wiele mówi.
To nie tak mówimy kiedy rozmawiamy o takim kodzie. Mówimy: ”Szablon bierze argument **Sequence**” i możemy napisać:

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

To oznacza dokładnie to samo co dłuższa wersja wyżej, ale jest krótsza i pasuje do naszego myślenia lepiej. Podobnie, nie mówimy ”Jest zwierzę i to jest kurczak”, mówimy ”Jest kurczak”. Przerobiona zasada jest prosta i ogólna dla konceptu **C.**

template<C T>

znaczy

template <typename T> requires C<T>

Używamy tego prostego skrótu dla konceptów pojedynczego argumentu. Tak jest kiedy wymagamy coś pojedynczego typu. Np. możemy uprościć

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

na

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

Uważam, że krótsza forma to znaczące udoskonalenie w przejrzystości nad starą wersją. Użyto konkretną **requires** klauzulę głównie do konceptów multi-type (np. **Equality_comparable)** i gdzie typ argumentu szablonu musi być referred to repeatedly (np. find). Odnosi się to do częstych i trwałych skarg, że składnia szablonów C++ jest pełna (verbose) i brzydka. Zgadzam się z tymi krytykami. Uczynienie zbyt rozwlekłej składni zbędną, jest zgodne z ogólnym celem projektowania ”rób proste rzeczy prostymi”.

4. Definiowanie konceptów

Często, znajdziesz pomocne koncepty, takie jak **Equality_comparable** w bibliotekach (np. the Ranges TS[Nie15]) i mamy nadzieję zobaczyć zestaw konceptów standardowej biblioteki, ale zobaczyć jak koncepty mogą być definiowane. Rozważ:

template<typename T>
concept bool Equality_comparable =
requires (T a, T b){
{ a == b} → boo; // porównuje a i b operatorem ==

```
        { a != b } → bool; // porównuje a i b operatorem !=
}
```

Koncept Equality_comparable jest zdefiniowany jako szablon zmiennej. Żeby być **Equality_comparable** typ **T** musi dostarczać operacje == i != takie że każda musi zwracać wartość logiczną (**bool**) (technicznie: ”coś zamienialnego na **bool**”). Wyrażenie **requires** pozwala nam na bezpośrednie wyrażenie jak typ może być użyty:

- **{ a == b }** mówi że dwa **T** powinny być porównywalne używając ==
- **{ a == b} → bool** mówi, że wynik takiego porówania musi być typu **bool** (technicznie, ”coś porównywalnego do **bool**”)

Wyrażenie **requires** jest nigdy w zasadzie wykonywane. Zamiast tego, kompilator patrzy na wszystkie wymagania wylistowane i zwraca **true** jeśli one się skompilują i **false** jeśli nie. To jest oczywiście bardzo potężne ułatwienie. Nauczyć się szczegołów, polecam notatki Asndrew Suttona. Oto przykład:

```
template <typename T>
concept bool Sequence = requires(T t) {
        typename Value_type<T> // musi mieć typ wartości
        typename Iterator_of<T> // musi mieć typ iteratora

        { begin(t) } → Iterator_of<T>; //musi mieć begin() i end()
        { end(t) } → Iterator_of<T>;

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

To znaczy żeby być **Sequence:**

- typ **T** musi mieć twa powiązane typy **Value_type<T>** i **Iterator_of<T>. Value_type<T>** i **Iterator<T>** są po prostu zwykłymi szablonami aliasów (alias templates). Listowanie tych typów w wyrażeniu **requires** oznacza że typ **T** musi je mieć żeby był **Sequence.**
- typ T musi mieć operacje **begin()** i **end(),** które muszą zwracać odpowiednie iteratory.
- Przez ”poprawny iterator” mamy na myśli że typ iteratora **T** musi być **Input_iterator** i typ wartości **T** (value type) musi być takie same jak typ wartości jego iteratora. **Input_iterator i Same_type** są konceptami z biblioteki, ale mógłbyś łatwo napisać je sam.

Teraz w końcu, możemy zrobić **Sortable.** Żeby być **Sortable,** typ musi być sekwencją oferującą randomowy dostęp i musi być z typem wartości (value type), który wspiera < porównania:

```
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, chcemy zrobić wymagania na związek pomiędzy konceptami. Np. zdefiniowano koncept Equality_comparable by wymagał pojedynczego typu, ale to jest zazwyczaj definiowane by radziło sobie z dwoma typami:

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

To pozwala porównywać **inty** z **double'ami** i **stringi** z **char*'ami,** ale nie **inty** z **stringami.**

5. Projektowanie z konceptami

Co się składa na dobry koncept? Idealnie, koncept reprezentuje podstawową koncepcję w jakiejś domenie, stąd nazwa. Koncept ma semantykę: to znaczy coś, nie tylko niepowiązane operacje i typy. Bez pomysłu na to co operacje znaczą i jak są powiązane ze sobą nie możemy pisać kodu generycznego który działa dla wszystkich prawidłowych typów.

Niestety, nie możemy określić semantyki konceptu w kodzie. Wynika z tego, że gwarancja, że wszystkie typy akceptowane przez sprawdzanie konceptu będą działać, jest niemożliwe: mogą mieć dokładnie te wymagane syntaktyczne właściwości ale mogą mieć nieprawidłową składnię. To nic nowego: podobnie, funkcja przyjmująca double może interpretować go inaczej niż użytkownik oczekuje. Rozważmy **set_speed(4.5).** Co to znaczy? Czy **4.5** ma być w jednostce metry na sekundę czy mile na godzinę? Czy **4.5** to stała wartość, delta do podanej szybkości czy czynnik zmiany?

Podejrzewam że *idealne* sprawdzanie dla całego kodu zawsze będzie nas omijać. Jak dostaniemy lepsze narzędzia, deweloperzy stworzą bardziej delikatne błędy, ale są techniki by uczynić błędy mniej podatne do ucieczki lub zauważenia.

5.1 Przypadkowe dopasowanie typu / konceptu (Type/concept accidental match)

Po pierwsze, pozwól mi zrobić powszechny błąd projektowy. Ułatwi to zilustrowanie dobrego projektu. Ostatnio widziałem wersję konceptu starego błędu obiektowego:

```
template<typename T>
concept bool Drawable = requires(T t) { t.draw(); };
class Shape {
	// ...
	void draw(); // zapal wybrane piksele na ekranie
}

class Cowboy {
	//...
	void draw(); //Pociągnij śmiertelną broń z kabury
}

template<Drawable D>
void draw_all(vector<D*>& v) {
	for(auto x : v) v->draw();
}
```

To **draw_all** mogłaby, jak jej obiektowy odpowiednik, akceptować **vector<Cowboy*>** z

zaskakującym i potencjalnie szkodliwym skutkiem. To problem ”nieoczekiwanego dopasowania” (w przeciążaniu i hierarchii klas) jest szeroko budzący obawę, rzadki w realnym kodowaniu, i łatwo dający się uniknąć dla konceptów.

Zapytaj siebie: Co podstawowy koncept który ”posiada funkcję draw() biorącą żadne argumenty” reprezentuje? Nie ma na to dobrej odpowiedzi. Cowboy może zrobić dobry koncept w kontekście gier a rysowalny kształt w kontekście grafiki, ale nigdy byśmy tego nie pomylili. Kształt ma wiele więcej ważnych właściwości niż tylko ”może być rysowany” (np. ”może jechać na koniu”, ”lubić wódeczkę”, ”może umrzeć”). Koncept który wymaga całego zestawu ostrożnie określonych ważnych właściwości jest mało prawdopodobny żeby był pomylony z jakimś innym.

Moja zasadą jest unikanie ”konceptów pojedynczej właściwości”. Z tego powodu, **Drawable** jest natychmiast podejrzany. Jest to dobry przykład czegoś co nie powinno być narażone dla budowniczych aplikacji. Żeby być bardziej realistycznym, ludzie czasami wpadają w kłopoty definiując coś takiego:

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

Są często zaskoczeni dowiadując się że **std::string** jest **Addable (std::string** dostarcza operator **+,** ale ten + konkatenuje bardziej niż dodaje**). Addable** nie jest odpowiednim konceptem do ogólnego użytku, nie reprezentuje podstawowego konceptu z poziomu użytkownika. Jeśli **Addable,** dlaczego nie **Substractable? (std::string** nie jest **Substractable** ale **int*** jest**).** Niespodzianki są powszechne dla ”prostych, konceptów pojedynczej właściwości”. Zamiast tego, zdefiniuj coś jak **Number:**

```
template<typename T>
concept bool Number = requires(T a, T b){
        { a + b } → T,
        { a - b } → T,
        { a * b } → T,
        { a / b } → T,
        { -a } → T,

        { T{0} }; // potrafi konstruować T z zera
        // ...
}
```

To jest mało prawdopodobne żeby być przypadkowo dopasowane.

Dobry, użyteczny koncept wspiera podstawowy koncept (zamierzał zamierzać) poprzez dostarczanie zestawu właściwości – takich jak operacje i typy (member types) – tak przewidziałby ekspert od domeny. Błąd zrobiony dla **Drawable** i **Addable** polegał na użyciu cech języka naiwnie bez odniesienia do zasad projektowych.

Proszę zauważ że **Number** jest tylko ilustracyjnym przykładem. Jeśli miałbym opisać arytmetyczne typy C++, potrzebowałbym odwołać się do problemów signed / unsigned i jak radzić sobie z arytmetyką trybu mieszanego (mixed-mode arithmetic). Jeśli chciałbym opisać komputerową algebrę prawdopodobnie zacząłbym od teorii grupy: monoid, pół grupa, grupa

5.2 Semantyka

Jak nam się podoba taki użyteczny zestaw właściwości do projektowania użytecznego konceptu?

Większość terenów aplikacji już je ma. Przykłady:

- koncepty typów wbudowanych C / C++: arytmetyczne, całkowe i rzeczywiste (tak, C ma koncepty)
- koncepty STL jak iteratory i kontenery
- Koncepty matematyczne jak monoid, grupa, pierścień i pole
- Koncepty grafowe jak krawędzie i wierzchołki grafu, DAG

Zauważ że te istniejące wcześniej koncepty wszystkie mają semantykę powiązaną z nimi(Alex Stepanov raz powiedział: ”koncepty są oparte o semantykę”). Uznaliśmy staranie się określić semantykę nowego konceptu za bezcenną pomoc w projektowaniu użytecznych i stałych interfejsów. Częste pytanie ”czy możemy określić aksjomat?” prowadzi do znaczących ulepszeń szkicu konceptu.

Pierwszy krok do zaprojektowania dobrego konceptu to rozważenie co jest kompletnym (potrzebnym i wystarczającym) zestawem właściwości (operacji, typów, itd.) by dopasować domenę konceptu, biorąc pod uwagę semantykę tej domeny konceptu(domain concept).

5.3 Ideały do projektowania konceptów

Co sprawia że jeden koncept jest lepszy od drugiego? Zasadniczo, koncepty istnieją by pozwolić nam do określania dość abstrakcyjnych pomysłów w kodzie, tak żebyśmy mogli pisać lepszy generyczny kod. Jednym sposobem do spojrzenia na to jest że koncepty pomagają nam algorytmów i typów ”kompatybilnych z wtyczką”:

- chcemy pisać algorytmy które mogą być użyte do szerokiej różnorodności typów i
- chcemy definiować typy które mogą być użyte z szeroką różnorodnością algorytmów

Np. chcemy zdefiniować typy liczb które mogą być użyte do naszych algorytmów numerycznych i algorytmów które mogą być użyte z wszystkim naszymi kontenerami.

To ma ważną implikację opartą na tym co stało się standardem techniki programowania generycznego: Często staramy się wyspecyfikować wymagania algorytmu żeby był absolutnym minimum. To nie jest to, co robimy by zaprojektować użyteczne koncepty. Jako przykład, rozważ ułatwioną wersję **std::accumulate:**

```
template<typename Iter, typename Val>
Val sum (Iter first, Iter last, Val acc) {
        while(first!=last) {
                acc += *first;
                ++first;
        }
        return acc;
}
```

”Klasyczny projekt programowania generycznego” kusiłoby nas do wymuszenia **sum** minimalnie jak to

```
template<Forward_iterator Iter, typename Val>
requires Incrementable<Val, Value_type<Iter>>
Val sum(Iter first, Iter last, Val acc){
        while(first != last) {
                acc += *first;
```

```
            ++first;
        }
}
```

Incrementable był konceptem który po prostu wymagał operatora += by był obecnym. To (widocznie) zminimalizowałoby pracę potrzebną/wymaganą(needed) przez kogoś projektującego typy które mogłyby być użyte jako argumenty **sum** i zmaksymalizowałoby to użyteczność algorytmu **sum.** Jednakże

- Zapomnieliśmy powiedzieć że **Val** miało być możliwym do kopiowania (copyable) i / lub ruchomy / przesuwalny (movable)
- Nie możemy użyć **sum** dla **Val** który dostarcza operatory + i =, ale nie +=
- Nie możemy zmodyfikować tej **sum** aby używała + i = zamiast += bez zmiany operatorów (część interfejsu funkcji)

To nie jest bardzo ”plug compatible” i raczej zadowalający w danej chwili. Ten rodzaj projektu prowadzi to programów, gdzie

- Każdy algorytm ma jego własne wymagania (różnorodność którą nie możemy łatwo pamiętać)
- Każdy typ musi być zaprojektowany by pasował do nieokreślonego i zmieniającego się zestawu wymagań
- Kiedy ulepszymy implementację algorytmu, musimy jego wymagania (część jego interfejsu), potencjalnie rozwalające kod.

Ten kierunek prowadzi do chaosu. A zatem, ideałem jest nie ”minimalne wymagania” ale ”wymagania wyrażone pod względem fundamentalnych i kompletnych konceptów”. To wymusza ciężar na projektantach typów (by pasowałby do konceptów) ale to prowadzi do lepszych typów i do bardziej elastycznego kodu. Np. lepsza **sum** wyłaby

```
template<Forward_iterator Iter, Number<Value_type<Iter>> Val>
Val sum (Iter first, Iter last, Val acc){
        while(first != last){
                acc += *first;
                ++first;
        }
        return acc;
}
```

Zauważ że poprzez wymaganie **Number** nabyliśmy elastyczności. Również ”straciliśmy” przypadkową zdolność do używania **sum** by konkatenować **std::string**ów i sumowania **vector<int>** w **char*:**

```
void poor_use(vector<string>& vs, vector<int>& vi){
        std::string s;
        s = sum(vs.begin(), vs.end(), s); // error: string nie jest liczbą
        char* p = nullptr;
        p = sum(vi.begin(), vi.end(), p); // error: wskaźnik nie jest liczbą
}
```

Dobrze! Stringi i wskaźniki to nie liczby. Jeśli my naprawdę chcielibyśmy tej funkcjonalności, moglibyśmy łatwo napisać to celowo.

Żeby zaprojektować dobre koncepty i używać je dobrze, musimy pamiętać że implementacja nie jest specyfikacją – kiedyś, ktoś może chcieć usprawnić implementację i najlepiej to zrobić bez wpływu na interfejs. Często nie możemy zmienić interfejsu bo robienie takie by zniszczyło kod. By napisać możliwy do utrzymania i szeroko używany kod dążymy do spójności semantycznej, bardziej niż minimalizmu dla każdego konceptu i algorytmu w izolacji.

5.4 Ograniczenia

Widok konceptów opisany tutaj jest nieco idealistyczny i ma na celu produkowanie ”finalnych” konceptów żeby były używane przez budowniczych aplikacji w dojrzałych domenach aplikacji. Jednakże, niekompletne koncepty mogą być bardzo użyteczne, zwłaszcza podczas wcześniejszych etapów postępu w nowej domenie aplikacji. Np. koncept **Number** powyżej jest niekompletny bo ”zapomniano” wymagać **Number** żeby był copyable i / lub movable. Poważne biblioteki dostarczają pierwszeństwa i wspomagających konceptów by uniknąć niekompletności.

Nawet jeśli tak jest, użycie **Number** ratuje nas od wielu błędów. Wyłapuje wszystkie błędy powiązane z brakującymi operacjami arytmetycznymi. Ale specyfikacja **sum** używająca **Number** nie ratuje nas od błędu jeśli użytkownik wywołuje **sum** z typem który dostarcza wymagane arytmetyczne operacje ale nie może być kopiowany lub ruszany. Jednakże, wciąż dostajemy błąd: dostajemy po prostu jeden z tradycyjnych późnych i bałaganiarskich wiadomości błędów do których byliśmy przyzwyczajeni przez dekady. System jest wciąż bezpieczny pod względem typów. Widzę ”niekompletne koncepty” jako ważny cel do rozwoju i stopniowego wprowadzenia konceptów.

Koncepty które są za proste do ogólnego użycia i / lub braku czystej semantyki może również być użyty jako bloki budowlane dla bardziej kompletnych konceptów.

Czasami, nazywamy takie zbyt proste i niekompletne koncepty ”ograniczeniami” do rozróżniania ich od ”prawdziwych konceptów”.

5.5 Dopasowywanie typów do konceptów

Jak pisarz nowego typu może być pewnym że pasuje do konceptu? To (zaskakująco ?) proste: Po prostu używamy **static_assert** do pożądanego konceptu. Np.

```
class MyNumber {/* ... */}
static_assert(Number<MyNumber>);
static_assert(Group<MyNumber>);
static_assert(Someone_elses_number<MyNumber>);

class My_container { /* … */}
static_assert(Random_access_iterator<My_container::iterator>);
```

Po wszystkim, koncepty to po prostu predykaty, więc możemy je testować. Są to predykaty *czasu kompilacji,* więc możemy je testować w czasie kompilacji. Zauważ, że nie musimy budować zestawu konceptów żeby były dopasowane do definicji typu. To nie jest jakiś projekt hierarchii który wymaga idealnej przezorności lub refaktoryzacji za każdym razem gdy nowe użycie jest odkrywane. To jest istotne do zachowania benefitów kompozycji generycznego kodu jako porównane do obiektowych hierarchii. **static_assert**y nie mają nawet być w typie kodu projektanta. Użytkownik może chcieć dodać takie testy do wyłapania niedopasowań wcześnie i w specyficznych miejscach w kodzie. Jeśli zrobione, robienie tak bez modyfikowania kodu biblioteki jest zasadniczy.

5.6 Stopniowe wprowadzenie konceptów

Jak możemy zacząć używać konceptów? W systemach prawdziwego świata, istotnie nigdy nie mamy luksusu zaczynania od zera. Polegamy na kodzie innych ludzi (np. biblioteki) i jeśli ulepszamy istniejący bazę kodu (np. standardową bibliotekę vendor lub popularne biblioteki sieci) i przez lata takie biblioteki mogą nie używać konceptów. Więc znajdujemy nasz kod wywołujący szablony który nie używa konceptów. Np.

```
template<Sortable S>
void sort(S& s) {
        std::sort(s.begin(), s.end());
}
```

Nasza **sort** wymaga żeby **s** była **Sortable** ale co **std::sort** wymaga? W tym przypadku, możemy spojrzeć w standard, ale ogólnie szczegóły dotyczące jakiej implementacji używa nie jest dokładnie określone. Ponadto, implementacja może zawierać ”kod scaffoldu (scaffolding code)” dla zbierania statystyk, logowania, celów debugowania, porównać, odwołań do bibliotek specyficznych dla platformy i ułatwień. W innych słowach, w momencie wywołania, nie możemy być pewni że implementacja nie używa ułatwień których nie wymagaliśmy od naszych wywoływaczy. Ponadto, implementacja takich szablonów zmienia się z czasem. Implementacja może nawet różnić się na podstawie opcji budowy.

Jednakże, to nie ma znaczenia za dużego bo dostajemy kompletne sprawdzanie typów, jak zawsze, tylko że późno (czas inicjowania) i straszną wiadomością błędu. Ważna sprawa to że możemy uaktualnić nasz kod by używał konceptów bez robienia (typically niemożliwe) kompletnej aktualizacji całego kodu na którym polegamy. Tak jak nasi dostawcy aktualizują ich interfejsy szablonów, nasze sprawdzania błędów przesuwa się do punktu wywołania i jakości ulepszeń wiadomości błędów.

Jeśli potrzebujesz być zdolnym do kompilowania kompilatorami które wspierają koncepty i kompilatorami które nie, potrzeba trochę obejścia. Jedna oczywista technika to użycie makr. Klauzule requires mogą być obsługiwane przez komentowanie ich w starszych kompilatorachL

```
#ifdef GOOD_COMPILER
#define REQUIRES requires
#elseif
#define REQUIRES //
#endif
```

Jeśli notacja short-hand jest używana, koncepty muszą być wylistowane

```
#ifdef GOOD_COMPILER
#define SORTABLE Sortable
#define ITERATOR Iterator
#elseif
#define Sortable auto
#define ITERATOR auto
#endif
```

By dostać szorstki odpowiednik przeciążania opartego na konceptach, **enable_if** może być użyty. To działa ale raporty które słyszy się są takie że jest to dość bolesne do utrzymania pisząc kod prawdziwego świata (np. the Range library[Nie15]). W szczególności, pamiętaj używać obu:

pozytywnych i negatywnych sprawdzań.

6. Przeciążanie konceptów

Programowanie generyczne polega na używaniu tej same nazwy dla operacji które mogą być użyte równoważnie dla różnych typów. A tym samym, przeciążanie jest istotne. Tam gdzie nie możemy przeciążyć, potrzebujemy użyć obejść (np. **traits**, **enable_if** lub funkcji pomocnych). Koncepty pozwalają nam wybierać pośród funkcji opartych na właściwościach danego argumentu. Rozważ uproszczoną wersję algorytmu standardowej biblioteki **advance.**

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

Potrzebujemy różnych wersji **advance** włączając

- jednej prostej dla iteratorów wstecznych (forward iterators), przechodzącej po sekwencji jeden krok za drugim
- jednej szybkiej dla iteratorów randomowego dostępu by wykorzystać zdolność do awansowania iteratora do samowolnej pozycji w sekwencji jedną operacją

Taka selekcja czasu kompilacji jest istotna dla wykonania kodu generycznego. Tradycyjnie, zaimplementowaliśmy to używanie funkcji pomocniczych i **tag dispatch** ale z konceptami rozwiązanie jest proste i oczywiste

void advance(Forward Iterator p, int n) { while(n--) ++p; }

void advance(Random_access_iterator p, int n) { p += n; }

```
void use(vector<string>& vs, list<string>& ls) {
        auto pvs = find(vs, ”foo”);
        advance(pvs, 2); // użycie szybkiego advance
        auto pls = find(ls, ”foo”);
        advance(pls,2); // użycie wolnego advance
}
```

Skąd kompilator wie jak wywołać odpowiedni **advance?** Nie powiedzieliśmy mu bezpośrednio. Nie ma zdefiniowanej hierarchii iteratorów i nie zdefiniowaliśmy żadnych **traits** do użycia dla **tag dispatch.**

Przeciążone rozwiązanie bazujące na konceptach jest zasadniczo proste:

- Jeśli funkcja spełnia wymagania tylko jednego konceptu, wywołuje ją
- Jeśli funkcja spełnia wymagania żadnego konceptu wywołanie jest błędem
- Jeśli funkcja spełnia wymagania dwóch konceptów, zobacz czy wymagania jednego konceptu jest podzbiorem wymagań drugiego
    - Jeśli tak, wywołaj funkcję z największą liczbą wymagań (najściślejszych wymagań)
    - Jesli nie, wywołanie jest błędem (dwuznaczność(ambigoues))

w przykładzie **use Random_access_iterator** ma więcej wymagań niż **Forward_iterator (”Random_access_iterator** jest ściślejszy niż **Forward_iterator”**) więc wybieramy szybki **advance** dla iteratora **vectora.** Dla iteratora **listy,** pasuje tylko **Forward_iterator,** więc używamy wolnego **advance.**

Random_access_iterator jest ściślejszy niż **Forward_iterator** bo wymaga wszystkiego co robi

Forward_iterator dodatkowo operatorów takich jak [] i +.

Jest kilka technicznych szczegółów związanych z dokładnym porównaniem konceptów ścisłości ale nie potrzebujemy tego porównywać żeby używać przeciążania konceptów. Co jest ważne to że nie musimy wyraźnie określać ”hierarchii dziedziczenia” pośród konceptami, definiować klas **traits** lub dodawać **tag dispatch** funkcji pomocniczych. Kompilator przetwarza prawdziwą hierarchię dla nas. To jest bardziej prostsze, bardziej elastyczne i mniej podatne na błędy.

Przeciążanie oparte na konceptach eliminuje znaczącą ilość **boiler-plate** z programowania generycznego i kodu meta programowania (większość użyć **enable_if).** Ogólna zasada tutaj to że nie powinniśmy wymuszać na programiście żeby robił to, co kompilator umie robić lepiej. Przeciążanie oparte na konceptach zapewnia, że kod będzie podążał za ogólnymi i szeroko używanymi zasadami rozwijania (resolution), zamiast różniącymi się i potencjalnie subtelnymi szczegółami implementacji (takich jak pamiętanie by używać obu: pozytywnych i negatywnych form **enable_if** przy wyrażaniu przeciążania opartego o właściwość typu).

Jedne oczywiste pytanie: Jak rozróżniać typy które są składniowo identyczne ale różnią się semantyką? Standardowy przykład tego to **Input_iterator** i **Forward_iterator,** które różnią się tylko powtarzanym obchodem, który jest dozwolony tylko dla **Forward_iterator.** Najprostsza odpowiedź brzmi ”nie rób tego, dodaj operację do jednego z typów by uczynić je rozróżnialnymi”. Bardziej standardową i skomplikowaną odpowiedzią jest ”użyj klasy **traits**”. Ten ostatni to, co zrobimy gdy nie będziemy mogli zmodyfikować żadnego typu. W przypadkach **Input_iterator** i **Forward_iterator** moglibyśmy wprawdzie rozróżnić bo **Input_iterator** jest tylko moveable ale nie jest copyable (użycie **traitu** is_copy_constructible<T>) ale to subtelne.

7. Notacje 'short-form' (zapis skrótowy)

Jednym z moich zamiarów dla C++ jest robienie prostych rzeczy prostymi (make simple things simple). Mamy zatem zapis skrótowy, aby uniknąć denerwujących powtórzeń i baerdziej zwięźle określić nasze wymagania. Rozważ:

```
template<typename Seq> requires Sortable<Seq>
void sort(Seq &s);
```

możemy to skrócić do:

```
template<Sortable Seq>
void sort(Seq &s);
```

Jednakże to wciąż nie zbliża nas do ideału równoważności do kodu ”zwykłego niż ogólny”, co zostało sformułowane:

```
void sort(Sortable& s);
```

Aby tam dotrzeć, mamy kolejną "przepisaną regułę". Krótka forma i formy skrótowe są po prostu równoważne długiej, bardzo wyraźnej postaci powyżej. Używamy najkrótszej formy dla najprostszych przypadków, a pozostałe dwie formy - często w kombinacji - gdy musimy wyrazić bardziej skomplikowane wymagania, zwłaszcza w odniesieniu do wymagań dotyczących więcej niż jednego argumentu szablonu. Na przykład:

```
template<Sequence S, typename T>
requires Equality_comparable <Value_type<S>,T>
```

Iterator_of<S> find (S& seq, const T& value);

Po co się męczyć? Długa forma jest niewystarczająco wyrazistą dla większości kodów i od czasu wprowadzenia szablonów skomplikowana składnia szablonu była źródłem stałych skarg od użytkowników. Potrzebujemy jednak długiej formy, wyrażającej złożone wymagania, w skrócie: Najkrótsza forma jest celowo nie doskonale ogólna.

Ten wzór jest zgodny z "zasadą cebulową"(”onion principle”). Domyślne jest krótkie i proste. Kiedykolwiek musisz coś zrobić co nie może być wyrażone tak prosto, zerwij jedną warstwę z cebuli. Każda warstwa daje ci więcej elastyczności i sprawia że płaczesz więcej (z powodu dodatkowej pracy i dodatkowych możliwości popełnienia błędów. Innym przykładem tej zasady jest obecność zarówno starych (doskonale ogólnych) dla pętli i (prostszych i mniej podatnych na błędy) dla pętli for-loop.

Te 3 formy pasują do sposobu w jaki mówimy o funkcjach:

//argument szablonu musi być typu i ten typ musi być sekwencją i s musi być referencją do tego //typu

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

//argument szablonu musi być sekwencją i s musi być referencją do tej sekwencji
template<Sequence Seq>
void algo(Seq& s);

//s musi być referencją do sekwencji
void algo(Sequence& s);

Używamy skróconej formy dopóki nie mamy powodu by tego nie robić.

7.1 argumenty auto

Krótka forma również oferuje niewymuszony (unconstrained) wariant:

void f(auto x); //bierze argument jakiegokolwiek typu

Oznacza to, że auto jest najmniej ograniczonym konceptem. Najpierw zaproponowałem argumenty **auto** i typy zwrotne(return types) w 2001 r. [Jar02] i C ++ 14 obsługuje je dla wyrażeń lambda.

Możemy również zdefiniować coś takiego, używając trywialnego ograniczenia

concept bool Any = true; // każdy typ jest typu Any
void g(Any x); // bierze argument typu any

Te dwa sposoby określania nieograniczonych argumentów różnią się w jednym małym przydatnym sposobie

void ff(auto x, auto y); // x i y mogą być różnego typu
void gg(Any x, Any y); //x i y muszą być takiego samego typu

Oznacza to, że gg reprezentuje styl pary iteracyjnej STL i wiele innych "zestawów argumentów tego samego typu", podczas gdy ff reprezentuje zupełnie niepowiązane argumenty szablonu. Oba style są użyteczne i powszechne. Na przykład:

```
void user(vector& vs, listld) {
        ff(&vs,&ld); // przechowuje listę kontenerów (dowolnych typów)
        gg(vs.begin(),vs.end()); // OK: dwa iteratory tego samego typu
        gg(vs.begin(),ld.end()); // błąd: dwa iteratory różnego typu
}
```

7.2 Czytelność

Spodziewałem się, że ludzie używają konceptów, aby pochwalić ekspresję konceptów i poprawione komunikaty o błędach. Aspekty te zostały wspomniane (przez studentów i profesjonalnych programistów), ale po raz kolejny ludzie podkreślali znacznie lepszą czytelność kodu za pomocą konceptów. Powinienem był był spodziewać się, że ponieważ podstawowe koncepty umożliwiają lepsze specyfikacje interfejsów i dobre interfejsy upraszczają zrozumienie.

Co ciekawe, takie komentarze pochodzą od osób, które wyrażały różne preferencje w notacjach (a czasami nie lubiły innych notacji): niektóre preferują klauzulę **requires,** niektórzy preferują koncepty zamiast typename, a niektórzy wolą używać najkrótszej formy, gdy tylko mogą. Różni ludzie po prostu uważają jedną lub więcej notacji odpowiadające ich potrzebom. Powinienem był oczekiwać, że różne osoby mają różne potrzeby.

W obu przypadkach nie doceniłem wagi czytelności. Słyszę trzy aspekty:

- Deklaracje są bardziej precyzyjne i informacyjne. Deklaracje używające koncepty są po prostu łatwiejsze do odczytania i bardziej wiarygodne niż deklaracje używające nazw opisowych dla typów generycznych plus komentarze. Ponadto deklaracje używające pojęć są krótsze niż rozwiązania (przynajmniej w przypadku gdy komentarze są używane do dokumentowania ograniczeń).
- Zamiana **auto** konceptem w punkcie wywołania eliminuje niepewność co do charakteru wyniku. Widzę to głównie jako odpowiedź na nadmierne użycie **auto**, ale bez konceptów istnieje niewiele alternatyw dla szerokiego używania **auto** w generycznym kodzie, który nie jest całkowicie funkcjonalny. Na przykład **if (auto x = foobar (z))** jest znacznie mniej czytelny niż jeśli **InputChannel x = foobar (z)**
- Koncepty eliminują nieczytelne obejścia i skomplikowane 'boiler plates'. Tak, nie powinniśmy brać pod uwagę definicji szablonów i makr, ale często robimy to (aby zrozumieć, debugować i poprawiać) i często obejścia krwawią do interfejsów (np. W formie enable_if).

Oczywiście, te obserwacje są estetycznymi osądami, a nie faktycznymi eksperymentami, ale myślę, że są one znaczące. Oczywiście, te obserwacje są estetycznymi osądami, a nie faktycznymi eksperymentami, ale myślę, że są one znaczące. Zaskoczyły mnie estetyczny wyrok dla programistów i ilość pozytywnych komentarzy na temat czytelności.

Uważam, że kwestia czytelności przyczynia się do udoskonalenia projektowania i konserwacji przypisywanych pojęciom.

8. Pytania dotyczące projektowania języka

Dodałem tę sekcję, ponieważ ludzie często pytają o projektowanie języków i czasami sugerują alternatywy lub radykalne modyfikacje obecnego projektu konceptów. Ten artykuł nie jest

dokumentem do projektowania języków, więc omówienie jest krótkie.

- zaspokoić naturalną ciekawość decyzji o projektowaniu języków
- uspokoić potencjalnych użytkowników, że oczywiste rozwiązania alternatywne zostały uznane
- pokazać, jak projekt spełnia ogólne zasady projektowania dla C ++

8.1 Czy naprawdę potrzebujemy konceptów?

Już od bardzo wczesnego projektowania i korzystania z szablonów zdawaliśmy sobie sprawę, że niektóre formy sprawdzania interfejsu mogą być wyrażane przy użyciu szablonów bez wsparcia języka dodanego [str94]. Przykładem jest sprawdzenie konceptu Boost.

Dzisiaj, z funkcjami **static_assert**, **constexpr, constexpr if** [Vut16], nadmiarem traitów typów biblioteki standardowej (**standard-library type traits)** i zaawansowanymi technologiami metaprogramowania szablonów, widzę wiele wariantów tego argumentu. Zwykle, co uzyskuje się, prowadzi do sprawdzania czasu inicjowania, co jest mniej niż idealne. Bardziej zasadniczo obniża poziom programowania, czyniąc podstawą programowania generycznego zależą od różnych bibliotek METAPROGRAMOWANIE ad hoc. Twierdzenie, że opieranie się na prymitywach niskopoziomowych jest wystarczające, jest jak (poprawnie) wskazywanie, że nie potrzebujemy absolutnie instrukcji **for**, instrukcji **while** i instrukcji range-**for**, gdy mamy **if** i **goto**. Podobnie, podane wskaźniki do funkcji, funkcje wirtualne i wyrażenia lambda nie są absolutnie konieczne. Jednakże, C ++ nie ma być jedynie kodem składowym dla (metaprogramowania) szablonu:

- "Koncepty" nie są jakąś zaawansowaną cechą dla ekspertów przykręconych na początku czasu kompilacji typowania kaczkowego (**compile-time duck typing)**
- "Koncepty" to nie tylko cukier syntaktyczny dla cech typów i **enable_if.**
- "Koncepty" są podstawową cechą, która w idealnym świecie byłaby obecna w pierwszej wersji szablonów i stanowi podstawę do wszystkich zastosowań.

Jestem pewien, że gdyby mieliśmy już koncepty w 1990 r., Szablony i biblioteki szablonów byłyby znacznie prostsze.

Zauważ, że w deklaracjach argumentów szablonów typename jest po prostu najmniej wymagającym konceptem: wymaga tylko argumentu szablonu jako typu (a nie wartości bez typu). Tak więc stare szablony prekonceptów integrują się płynnie z konceptami. Jeśli ograniczenia nie są potrzebne, po prostu nie używamy konceptów ani nie używamy konceptów zapewniających bardzo minimalne ograniczenia. Przykłady są bardzo ogólnym metaprogramowaniem szablonu manipulującym AST i argumentami szablonów, dla których wymogi są wyrażane wyłącznie jako relacje z innymi argumentami szablonu.

8.2 Sprawdzanie definicji

Koncepty obecnie nie uniemożliwiają szablonom korzystania z operacji, które nie zostały określone w wymaganiach. Rozważ:

```
template<Number N>
void algo(vector<N>& v) {
        for (auto& x : v) x%=2;
}
```

Nasz koncept Number nie wymaga %=, więc czy wywołanie algo odniesie sukces będzie zależeć nie tylko na tym czy będzie sprawdzony przez koncept ale na rzeczywistych właściwościach typu argumentu: czy typ argumentu ma% =? Jeśli nie, otrzymamy spóźniony (czasu inicjowania) błąd.

Niektórzy uważają to za poważny błąd. Ja nie: niesprawdzanie definicji szablonu względem konceptów szablonu było rozmyślnym wyborem projektowym. My (Gabriel Dos Reis, Andrew Sutton, i ja) wiemy, jak wdrożyć sprawdzenie definicji zgodnie z aktualnie określonymi konceptami. Przeprowadziliśmy analizę i eksperymenty (np. [Rei12]), ale zdecydowanie postanowiliśmy nie uwzględniać cechy we wstępnym w projekcie konceptu.

- Nie chcieliśmy opóźniać i komplikować pierwotnego projektu (co opóźniałoby otrzymanie istotnych informacji zwrotnych i opóźniło budowę biblioteki).
- Szacujemy, że coś około 90% korzyści konceptów ma wartość lepszej specyfikacji i sprawdzania punktu użytkowania
- Implementator szablonu może zrekompensować normalne techniki testowania.
- Jak zawsze, błędy typu są zawsze złapane, tylko niewygodnie późno.
- Sprawdzając definicje skomplikowaliśmy przejście od starszego, nieograniczonego kodu do szablonów opartych na konceptach.
- Sprawdzając definicje nie moglibyśmy w szablonie wstawić pomocniczych narzędzi do debugowania, kodu rejestrowania, kodu telemetrii, liczników wydajności i innych "scaffolding code" bez wpływu na jego interfejs.

Ostatnie dwa punkty mają kluczowe znaczenie:

- Typowy szablon wywołuje inne szablony w jego implementacji. Chyba że szablon używający konceptów może wywołać szablon z biblioteki, która nie wywołuje, biblioteka z konceptami nie może używać starszej biblioteki, zanim ta biblioteka zostanie zmodernizowana. To poważny problem, zwłaszcza gdy obie biblioteki są rozwijane, utrzymywane i używane przez więcej niż jedną organizację. Stopniowe przyjęcie konceptów ma zasadnicze znaczenie w wielu podstawach kodu
- **Scaffolding code** (zarówno szablonów jak i kodu który nie jest szablonem) jest bardzo popularny i zmienia się w trakcie istnienia biblioteki. Kod rusztowania (zarówno szablony, jak i kod inny niż szablon) jest bardzo popularny i zmienia się w trakcie istnienia biblioteki. Jeśli interfejs musi się zmienić aby się dostosować, mówić, rejestrować, mamy pierwszorzędny problem z utrzymaniem.

Zauważ, że ograniczony szablon (szablon używający konceptów) może wywołać nieograniczony szablon. W takim przypadku błędy w implementacji nie są odnalezione do czasu inicjowania. Podobnie, nieograniczony (tradycyjny szablon) może wywołać ograniczony szablon. W takim przypadku błędy użytkowania nie są znalezione do czasu inicjowania. Te pierwsze sugerują, że koncepty można wprowadzać "na górę", podczas gdy drugie oznacza, że w wyniku tego robi się najwięcej korzyści.

Więc wiemy, jak zrobić "sprawdzenie definicji", ale nie zrobimy tego dopóki te dwa problemy nie zostaną rozwiązane. Istnieją oczywiste rozwiązania, takie jak wskaźnik w ciele / implementacji szablonu (nie w deklaracji / interfejsie), który będzie korzystać z obiektów nie zagwarantowanych przez koncepty, ale każdy taki mechanizm musiałby być poważnie analizowany i testowany. To nie może być warte wysiłku. Warto pamiętać o jednej z podstawowych zasad, które kierowały projektem C ++: "Ważne jest, aby umożliwić użyteczność, niż zapobiegać każdemu nadużyciu" [str94]. To jedna z zasad wyróżniających C ++ od wielu innych języków. Istnieją sposoby zapobiegania złemu programowaniu poza regułami językowymi, ale stopniowe 'scaffolding code' musi być w jakiś sposób być umożliwiony przez język.

8.3 Osobna kompilacja szablonów

Całkowita oddzielna kompilacja szablonu zakłada sprawdzenie definicji, a najbardziej oczywista implementacja wymaga wielu pośrednich wywołań funkcji, które mogłyby zabić wydajność. Oczywistą alternatywą jest poł-skompilowana forma szablonu jako część systemu modułowego.

8.4 Wielokrotne notacje (Multiple notations)

Koncepty dostarczają wielokrotne sposoby mówienia rzeczy. Pogodziłem się z tym. Uważam że to idealne ale są ludzie oddani pojęciu ”tylko jeden sposób”. To pojęcie prostoty nieuchronnie prowadzi do ograniczenia ekspresji (np. Tylko krótkiej formy używania konceptu) lub systematycznej wielomówności (np. Tylko długa forma używania konceptu).

Koncepty mogą być definiowane jako zmienne szablonu lub jako funkcje szablonu. Na przykład:

```
template<typename T>
concept bool Eq1 =
        requires (T a, T b) {
                { a == b } -> bool; // porównuje zmienne typu T z operatorem ==
                { a != b } -> bool; // porównuje zmienne typu T z operatorem !=
        };
```

i

```
template<typename T>
concept bool Eq2() {
        return requires (T a, T b) {
                { a == b } -> bool; // porównuje zmienne typu T z operatorem ==
                { a != b } -> bool; // porównuje zmienne typu T z operatorem !=
        };
}
```

Niestety składnia używania szablonu zmiennego i składnia wywołania funkcji szablonu różnią się. Nie ma to nic wspólnego z konceptami, ale prowadzi do takich ciekawostek:

```
template<typename T> requires Eq1<T> void f(T&);
template<typename T> requires Eq2<T>() void g(T&);
```

Pamiętanie czy dodać **()** czy nie jest uciążliwością. Jednak wywoływanie funkcji w C++ wymaga **( )** a pobranie wartości zmiennej nie. Powodem dopuszczenia zarówno zmiennych, jak i funkcji jest ogólność. To wynika ze sposobu w jaki wyrażenia są zdefiniowane w C++.

Zauważ, że C ++ wspiera przeciążanie funkcji, więc jeśli potrzebujesz dwóch konceptów o tej samej nazwie, musisz użyć formatu funkcjonalnego. Jak dotąd było to rzadkie.

Dlaczego wymagamy, aby programiści napisali koncept bool zamiast zwykłego konceptu (sugerującego bool)? Chcieliśmy aby koncept podążał za zwykłą definicją, zamiast wymyślać coś nowego, a definicje zaczynały się od typu. Istnieje wyjątek od tej reguły w gramatyce C ++: nie określamy typu dla konstruktorów, destruktorów i operatorów konwersji. To spowodowało trochę zamieszania i złożoności oraz kilka skarg, więc zdecydowaliśmy się nie stosować tego precedensu.

8.5 Opt in

Projekt konceptu jest zgodny z zasadą, że nie powinniśmy wymuszać programistów by mówili rzeczy, które kompilator już wie (i często wie lepiej niż programista). Prowadzi to do krótszego, czystszego kodu i mniej błędów.

- Nie decydujesz się na przeciążanie opartego na koncepcji, podobnie jak nie chcesz używać zwykłego przeciążania.Jest to spójne, a nie verbose, w duchu C ++. Niektórzy ludzie przyzwyczaili się do wyboru obiektów, dodając cechy. To ma swoje uroki, ale jest zasadniczo obejściem i fałszywą różnicą między programowaniem generycznym a "zwykłym programowaniem". Dodało pracę dla programisty aplikacji i możliwość popełnienia błędów.
- Nie decyduj się albo jednoznacznie nie określaj hierarchię relacji między konceptami. Kompilator oblicza właściwe relacje między pojęciami i stosuje bardzo proste mechanizmy rozdzielczości(**resolution mechanisms**) do zastosowań. Wymaganie określenia wyraźnej hierarchii konceptów ograniczyłoby elastyczność, wymagałoby, aby koncept był częścią kilku hierarchii i / lub wymagała większej przewidywalności od projektantów bibliotek.
- Nie decyduj się na dopasowywanie względem typu konceptu: Jeśli typ ma właściwości składniowe wymagane przez koncept, to znaczy że pasuje**(it matches).** Oznacza to, że nie musisz zaśmieca swojego kodu "deklaracjami modelowania", takimi jak "**vector<T>** modeluje **Kontenet(Container)**" lub "**List<T>::iterator** jest **Bidirectional_iterator**". Wymagania, które skomplikowałoby korzystanie z biblioteki, (chyba że sprawdzone przez kompilator) mogłyby być źródłem błędów i / lub wymagałyby większego przewidywania od projektantów bibliotek.

W rzadkich przypadkach, gdy dwa pojęcia są identyczne pod względem składni, ale różnią się semantycznie (np. Input_iterator i Forward_Iterator są prawie niemożliwe do odróżnienia), musimy coś zrobić by je jednoznacznie zidentyfikować: albo robimy je jednoznacznymi przez dodanie operacji lub typu (member type) albo używamy klas cech (traits class) w trakcie wykonywania operacji na nich.

Jeśli chcesz zagwarantować, że typ spełnia koncept, użyj **static_assert.**

8.6 Nie można rozpoznać szablonów

Rozważ:

void sort(Sortable&);

Niektórzy doświadczeni programiści C ++ martwią się, że nie ma pojęcia składniowego, że jest to szablon. Inni, jak ja, uważają to za idealne: wreszcie generyczne funkcje mogą być traktowane tak samo jak inne funkcje! Zauważmy, że posiadaliśmy tę własność dla operatorów "zawsze". Uczyłem konceptów dziesiątki studentów. Studenci nie martwią się. Mówią "fajnie" lub (częściej) po prostu biorą to za pewnik. Nie gubią się ani nie piszą szczególnie złego kodu używając tej notacji. Duże problemy i zamieszanie są gdzie indziej - w obszarach, które zostały ustanowione w C + + od dziesięcioleci. Nowsze funkcje, takie jak koncepty, są prostsze w użyciu niż ich bardziej ustalone rozwiązania. Początkowo ja też się martwiłem o możliwość zamieszania i zastanawiałem się, czy potrzebujemy konwencji nazewnictwa lub pojęcia składniowego. Próbowałem **Sortable_c,** podobnego do sposobu w jaki niektóry kod (styl języka C) używa **foo_t** do rozróżnienia typu **foo** od zmiennej **foo** i **cSortable,** ale po chwili stało się to nużące i zawsze było brzydkie - rodzaj odwrotnej węgierskiej notacji. Nie ułatwiło to obsługi kodu.

Ponadto, wprowadziliśmy już równoważne "skasowanie" notacji rozróżniającej pomiędzy nazwami typu i nazwami szablonów, gdy (w końcu!) Postanowiliśmy zezwalać na wyciągnięcie argumentów szablonu z argumentów konstruktora [Spe16]. Teraz możemy powiedzieć

pair p {9.2, 4};

Bardziej niż
pair<double, int> p {9.2, 4};

Co ciekawe, nie usłyszałem sugestii, aby zmienić nazwę pary (i większość innych szablonów) na coś takiego jak pair_tmpl w celu zwiększenia czytelności. Nie słyszałem też sugestii, że para potrzebowała przedrostka, np. Szablonu, aby powiedzieć użytkownikowi, że jest szablonem.

Zauważ, że w C++ mówimy:

void f(Node*);

bardziej niż:

void f(struct Node*);

To rzadko powoduje zamieszanie i nie słyszałem o tym narzekań od najwcześniejszych dni C ++ (powiedzmy 1982), gdy niektórzy uważali, że dodana struktura jest pomocna, gdyż była znana z języka C. Podczas pracy z prawdziwym kodem ludzie wiedzą, czy nazwa odnosi się do zmiennej, typu, szablonu czy konceptu. Tak samo wiedzą kompilatory.

Podejrzewam, że kolorowanie tekstu zapewni dalszą pomoc czytelnikom ludzkim w odróżnieniu typów, szablonów i koncepcji, ale zasadniczo nie jest to problem. Przypisuje obawy dotyczące notacji i ewentualnego "zamieszania między typami i szablonami" dobrze znanemu zjawisku ludzi wyobrażających sobie problemy z nowymi cechami języka i zastanawiającymi czy składnia pisana ciężką ręką lub konwencje notacji są potrzebne do odnoszenia się do wyobrażonych problemów. Czas płynie i nowatorskie funkcjonalności stają się znane, zwięzłe notacje staje się niezmiernie korzystne. Mój wstępny projekt szablonu nie miał prefiksu szablonu <typename T>. Został on wprowadzony przede wszystkim w celu zapewnienia "martwiących się". Obecnie powszechnie się nie podoba, a często wyśmiewa jako przykłady wielomówności i złego projektu.

8.7 Czy pojęcia powinny być rodzajem klas?

W oparciu o doświadczenia z innymi językami i eksperymentowanie z konceptami C ++ 0x niektórzy przekonali się, że koncepty powinny być definiowane podobnie do klas. To jest lista deklaracji. Oczywiście, projektanci konceptów nie zgadzają się. Używane przez nas schematy użytkowania są bardziej elastyczne, krótsze do pisania w rzeczywistych przypadkach i lepiej obsługują przeciążenie i pośrednie konwersje. Ponieważ jednak wzorce użycia są bardziej ogólne niż koncepty oparte na deklaracjach, możemy faktycznie zdefiniować koncepcty pod względem podpisów. Nie zalecamy tej techniki, ale jest jeden sposób, w jaki można zdefiniować koncept typu wymaganego do wyprowadzenia z **Base**, mieć member type **int f (double)**, i member **int m**:

```
template<class T>
concept bool Hack =
        requires(T t, Base* p, int(T::*pp)(double), int* ppp) {
                { &t==std:addressof(t) }; // miej pewność że operator & nie jest przeciążony w
brzydki sposób
                { p = &t }; // T jest pochodny z Base
                { pp = &T::f }; // T ma member int f(double)
                { ppp = &t.m}; // T ma member int m
        };
```

To nie dotyczy przeciążonych funkcji, ale wiele języków w oparciu o sygnatury nie radzi sobie z przeciążaniem, a przecież nie jest to technika zalecana do ogólnego użytku.

Nie jest rzadkością, aby zbliżyć się nowego języka lub nowej funkcję języka, starając się go używać w stylu znanym z innego języka. W rzeczywistości wydaje się to niemal nieuniknione: "Możesz pisać Fortran" w dowolnym języku ". Podobnie można napisać "Java", "C", "C #", "Haskel" itd. W języku C ++, ale zachęcam ludzi do spróbowania nie wpaść w tę pułapkę podczas wypróbowywania pojęć (np. opieranie ograniczeń na na wyraźnych hierarchiach I klasach bazowych).

Dawno temu Andrew Koenig zrobił interesujący eksperyment (który niestety nie sądzę, że spisał). Wziął kilka prostych programów C ++ i przerobił je na ML i kilka prostych programów ML i przrrobił je na C ++. Transkrypcje były jednolicie brzydkie, wielomówne i powolne. Następnie rozważał problemy, które zostały spisane przez te małe programy i rozwiązał je w natywnym stylu w "innym języku". Programy te były we wszystkich przypadkach rozsądnie miłe i szybkie. Staraj się używać pojęć, jak zostały zaprojektowane do użycia, przynajmniej na początku. Oczekuję, że są wystarczająco ogólne, aby znaleźć zastosowanie poza moją wyobraźnią, a to byłoby dobre. Jednak nie oceniaj ich na podstawie prostego przełożenia czegoś z jednego języka na inny.

8.8 Koncepty nie są klasami typu

Uwagi na temat wcześniejszych projektów tego dokumentu przekonały mnie, że potrzebna jest uwaga do rozwiązywania konfuzji i nieporozumień.

Jak wspomniano w §1 i [Str94], kiedy zaprojektowałem szablony w latach 1987-8, chciałbym ograniczyć argumenty szablonów. Jest to idea, która cofa się co najmniej o Clu [Lis77] i prawdopodobnie dalej. Wydawało się to wtedy oczywistym pomysłem. Powodem, dla którego się wycofałem, była paskudna kombinacja problemów związanych z implementacją i problemami notacji. W szczególności nie sądziłem, że system typu, który sugerował użycie wywołań funkcji pośrednich, był akceptowalny w kontekście programowania systemów. Dalej tak nie sądzę. Tak, podobnie jak szablony, koncepty nie polegają na wywołaniu funkcji pośrednich. Jakiekolwiek koszty czasu wykonania prowadziłyby do nieużytkowania w krytycznych rejonach. Koncepty nie są przeznaczone by być klasami abstrakcyjnymi: jeśli chcesz vtable, wiesz gdzie to znaleźć.

Ponadto niejawne konwersje, ogólne przeciążenie i specjalizacja nie pasują do konceptów zaprojektowanych koncepcyjnie jako tabel (zbiorów) funkcji (tak jak próbowano C ++ 0x). Koncepty nie są klasami w żadnym wspólnym sensie informatyki słowa "klasa".

Koncept w C++ jest predykatem czasu kompilacji opartym na zeru lub wielu argumentach typów argumentów szablonu lub argumentach wartości.

- Wymagania operacyjne dotyczące konceptów są określone w kategoriach wzorców użycia ("poprawne wyrażenia"), a nie sygnatur funkcji. §4.
- Koncepty są określone jako ogólne predykaty (wyrażenia Boole'a), w tym funkcje zwracające bool
- Typ nie musi być jednoznacznie określony, aby pasował do konceptu; przypasowanie jest dedukowane
- Koncepty mogą przyjmować argumenty wartości (a nie tylko argumenty typu).
- Koncepty mogą przyjmować wiele argumentów
- Funkcje konceptów mogą być przeciążone
- Algorytmy mogą być przeciążone na konceptach.
- Koncepty nie są zdefiniowane jako członkowie hierarchii. Relacje pomiędzy konceptami są dedukowane
- Koncepty mogą ograniczać argumenty szablonu

To różni się od klas typu (np. Klasy typu w języku Haskell). Wiemy, że klasy wielu parametrów zostały omówione i używane w dialektach Haskell przez długi czas, ale ponieważ nie są one oficjalnie częścią Haskell i kontrowersyjne w miejscach, wymienię je tutaj jako różnicę. Wieloargumentowe koncepty zawsze stanowiły integralną część pojęcia konceptu C ++. STL - zaprojektowany i wdrożony przed rokiem 1984 przez Alexa Stepanova - został zaprojektowany z myślą o konceptach [Kap81] i nie może być odpowiednio ograniczony bez konceptów wielo typowych.

Należy pamiętać, że dwa projekty mające na celu rozwiązanie podobnych problemów mogą różnić się znaczącymi sposobami z uzasadnionych powodów. Uważam, że ma to miejsce w przypadku klas i pojęć typu. Haskell klasy typu nie służyłoby dobrze w kontekście C ++ i I wątpię by koncepty w C++ byłoby idealnym rozwiązaniem dla Haskell. Odwrotnie, odnosząc się do dwóch różnych rozwiązań problemu o tej samej nazwie może powodować zamieszanie. Gdybym pomyślał, że pewna forma klas typu mogłaby rozwiązać problem elegancko i wydajnie ograniczania szblonów C++, zaproponowałbym ją.

Koncepty, jak zdefiniowane dla C ++, wracają do prac Alexa Stepanova nad programowaniem generycznym w późnych latach 1970 roku i udokumentowane pod nazwą "Algebraic structures". Zauważ, że to prawie dziesięć lat przed projektem Haskell. Alex użył pojęcia "koncept" na wykładach w późnych latach 1990 i udokumentował to. Obecne zastosowanie predykatów polegających na wzorcach użycia do opisywania operacji ma swoje początki w pracach Stroustrup i Dos Reis w latach dziewięćdziesiątych i na początku 2000 i udokumentowano. Powodem używania wzorców użytkowania jest obsłużenie niejawnych konwersji i przeciążenia. Chociaż wiedzieliśmy o Haskellu i ML, nie miały one istotnego wpływu na obecny projekt C ++.

WNIOSKI

Koncepty zawierają kompletne szablony C ++, zgodnie z pierwotnym założeniem. Nie widzę ich jako rozszerzenia, ale jako ukończenia.

Pojęcia są dość proste w obsłudze i definiowaniu. Są zaskakująco pomocne w poprawie jakości kodu generycznego, ale ich zasady - a nie tylko ich szczegóły dotyczące języka - techniczne - muszą być zrozumiane dla skutecznego wykorzystania. W tym celu koncepty są podobne do innych podstawowych struktur, takich jak funkcje, klasy i szablony. W porównaniu do szablonów nieograniczonych, nie ma kosztów czasu wykonania poniesionych przy użyciu konceptów.

Koncepty są starannie zaprojektowane, aby pasowały do C ++ i stosować się do zasad projektowania C ++:

- dostarczają dobre interfejsy
- szukają spójności semantycznej
- nie zmuszają użytkownika do robienia tego, co maszyna robi lepiej
- zachowują proste rzeczy prostymi
- zero kosztów dodatkowych (overhead)

Nie myl znajomości i prostoty. Nie myl wielomówności z "łatwym do zrozumienia". Spróbuj konceptów! Poprawią one znacznie twoje programowanie generyczne i sprawią, że bieżące obejścia (np. Klasy cech (traits class) i techniki niskiego poziomu (np. Przeciążenie oparte na technologii enable_if) sprawią wrażenie podatnych na błędy i żmudnych w programowania montażu (tedious assembly programming).