0.1 Specjalizacja algorytmów

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

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

Prosty koncept iteratora forward:

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

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

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

```
  return n;
}
```

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

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

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

Koncept `Bidirectional_iterator` jest zbudowany na podstawie `Forward_iterator`. Czyli *iterator bidirectional* jest *iteratorem forward*, który również może poruszać się do tyłu. Zestaw wymagań konceptu `Bidirectional_iterator` całkowicie zalicza się do zestawu konceptu `Forward_iterator`. W wyniku czego, za każdym razem gdy `Bidirectional_iterator<X>` jest prawdziwe (dla wszystkich `X`), `Forward_iterator<X>` musi tez być prawdziwy. W tym przypadku mówimy, że koncept `Bidirectional_iterator` *udoskonala*[1] koncept `Forward_iterator`.

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

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

[1](ang. refine)

```
  }
```

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

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

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

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

Te algorytmy można używać do zdefiniowania dużej liczby użytecznych operacji.

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

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

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

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

tacja tych funkcji jest doskonale zdefiniowana dla wskaźników.

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

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

2. Funkcje nieszablonowe są lepsze niż specjalizacje szablonów funkcji.

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

   Specyfikacja techniczna konceptów dodaje jeszcze jedną zasadę:

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

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

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

Koncept	Atomowe wymagania

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

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

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

Dla zwięzłości wyłączyłem ograniczenie `Regular<I>` pojawiające się w `Forward_iterator`, ponieważ on(i jego wymagania) są wspólne dla wszystkich konceptów iterujących. Porównując powyższe stwierdzimy, że `Bidirectional_iterator` ma ścisły nadzbiór wymagań `Forward_iterator`, a `Random_access_iterator` ma ścisły nadzbiór wymagań `Bidirectional_iterator`. Z tego względu `Random_access_iterator` jest najbardziej ograniczony i to przeciążenie zostało wybrane. Nowa reguła przeciążania nie gwarantuje, że rozwiązanie przeciążenia odniesie sukces. W szczególności, jeśli dwóch realnych kandydatów ma nakładające się lub logicznie równoważne ograniczenia, rozwiązanie będzie niejednoznaczna. Jest kilka powodów, dla których to miałoby się zdarzyć.