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

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

Niektórzy twierdzą, że wyrażenia takie jak `SFINAE`[1], `constexpr if`[2], `static_assert`[3] i mądre techniki metaprogramowania w zupełności wystarczą do przeciążania. To oczywiście poprawne myślenie, lecz jest to obniżanie poziomu abstrakcji, co skutkuje tym, że programuje się w sposób żeby było zrobione a nie jak powinno być. Wynikiem jest więcej pracy dla programisty, zwiększona ilość błędów i mniej szans optymalizacyjnych. *C++* nie jest przeznaczony do metaprogramowania szablonów. Koncepty pomagają nam podnieść poziom programowania i ułatwić kod, bez dodawania kosztów czasu wykonania.

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

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

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

[3]Wykonuje sprawdzanie porównania w czasie kompilacji

```
}
```

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

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

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

```
template<typename R>
concept bool Range() {
  return requires (R range) {
    typename value_type_t<R>;
    typename iterator_t<R>;
    { begin(range) } -> iterator_t<R>;
    { end(range) } -> iterator_t<R>;
    requires Input_iterator<iterator_t<R>>();
    requires Same_as<value_type_t<R>,
      value_type_t<iterator_t<R>>>();
  };
}

template<typename S>
concept bool Sequence() {
  return Range<R>() && requires (S seq) {
    { seq.front() } -> const value_type<S>&;
    { seq.back() } -> const value_type<S>&;
  };
}
```

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

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

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

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