Wprowadzenie.

Bibliteka STL jest przykladem programowania uogólnionego. Programowanie obiektowe koncentruje się na zagadnieniu danych, podczas gdy w programowaniu uogólnionym główny nacisk kladzie się na algorytmy.
Celem programowania uogólnionego jest utworzenie kodu niezależnego od typów danych. Aby sprawnie poruszać się po tych danych można wykorzystać do tego celu iteratory. Podobnie jak szablony powodują, że algorytmy są niezależne od typu przechowywanych danych w kontenerze danych, tak iteratory sprawiają, że alorytmy są niezależne od typu używanego kontenera, dlatego też są one kluczowym komponentem uogólnionego podejścia w bibliotece STL.

Mówiąc w skrócie, podejście stosowane w bibliotece STL rozpoczyna się od algorytmu przetwarzającego dane kontenera. Algorytm ten tworzy się w sposób możliwie ogólny, uniezależniając go od typu danych oraz rodzaju kontenera. Aby ogólny algorytm działał w konkretnych przypadkach, musimy zdefiniować iteratory, które spełnią wymagania algorytmu i określą wymagania dla projektu kontenera. Oznacza to, że podstawowe właściwości iteratorów i kontenerów wynikają z wymagań stawianych ogólnej postaci algorytmu.

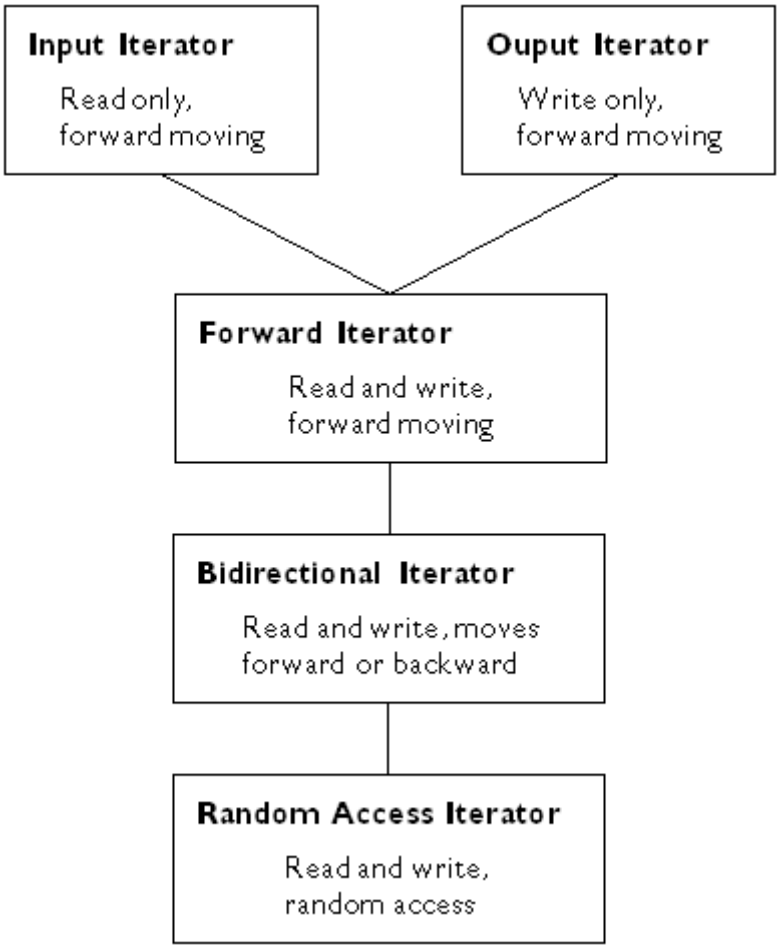
Kategorie iteratorów:
Począwszy od C++17 można wyróżnić 6 kategorii iteratorów. Są to:

- Input Iterator,
- Output Iterator,
- Forward Iterator,
- Bidirectional Iterator,
- Random Access Iterator,
- Contiguous Iterator.

W poniższej tabeli są przedstawione podstawowe właściwości iteratorów.

Iterator Capability	Input	Output	Forward	Bidirectional	Random Access
Dereferencing read	Yes	No	Yes	Yes	Yes
Dereferencing write	No	Yes	Yes	Yes	Yes
Fixed and repeatable order	No	No	Yes	Yes	Yes
`++i i++`	Yes	Yes	Yes	Yes	Yes
`--i i--`	No	No	No	Yes	Yes
`i[n]`	No	No	No	No	Yes
`i + n`	No	No	No	No	Yes
`i - n`	No	No	No	No	Yes
`i += n`	No	No	No	No	Yes
`i -=n`	No	No	No	No	Yes

Nietrudno zauważyć, że iteratory tworzą pewną hierarchię. Iterator postępujący(ForwardIterator) ma wszystkie właściwości iteratora wejsciowego oraz iteratora wyjsciowego wraz z pewnymi dodatkowymi cechami. Iterator dwukierunkowy(BidirectionalIterator) ma wszystkie właściwości iteratora postępującego oraz własne cechy. Iterator dostępu swobodnego łączy w sobie właściwości pozostałych iteratorów oraz ma własne dodatkowe cechy. Najbardziej wszechstronnym iteratorem jest ContiguousIterator dodany w C++17, który ma wszystkie cechy Iteratora dostępu swobodnego a ponadto wszystkie jego logiczne sąsiadujące elementy także sąsiadują fizycznie w pamięci.



Algorytm napisany dla konkretnego rodzaju iteratora może używać tego właśnie iteratora ***lub dowolnego innego, mającego wymagane właściwości.*** Dlatego na przykład kontener z iteratorem dostępu swobodnego może używać algorytmów napisanych dla interatora wejściowego.

Po co istnieją różne rodzaje iteratorów?
Umożliwiają one pisanie algorytmów, które używają iteratorów z najmniejszymi wymaganiami, dzieki czemu można ich używać dla większej liczby kontenerów.

Przykład na bazie algorytmu std::find

I tak np. algorytm std::find (https://en.cppreference.com/w/cpp/algorithm/find)

template< class InputIt, class T >
InputIt find(InputIt first, InputIt last, const T& value);

zawiera w swoich wymaganiach iż minimalnie powinien być to:

Type requirements
- `InputIt` must meet the requirements of *InputIterator*.

Poniżej przykładowy kod dla algorytmu std::find wykorzystujacego kontener std::vector.
Kontener std::vector posiada iterator.:

`iterator` *RandomAccessIterator*

W tym przypadku zastosowano iterator bardziej rozbudowany niż wymagał tego algorytm std::find.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>

int main()
{
    int n1 = 3;
    int n2 = 5;

    std::vector<int> v{0, 1, 2, 3, 4};

    auto result1 = std::find(std::begin(v), std::end(v), n1);
    auto result2 = std::find(std::begin(v), std::end(v), n2);

    if (result1 != std::end(v)) {
        std::cout << "v contains: " << n1 << '\n';
    } else {
        std::cout << "v does not contain: " << n1 << '\n';
    }

    if (result2 != std::end(v)) {
        std::cout << "v contains: " << n2 << '\n';
    } else {
        std::cout << "v does not contain: " << n2 << '\n';
    }
}
```

Przykład na na bazie algorytmu std::sort

A np. algorytm std::sort (https://en.cppreference.com/w/cpp/algorithm/sort)

template< class RandomIt >
void sort(RandomIt first, RandomIt last);

zawiera w swoich wymaganiach:

Type requirements
- `RandomIt` must meet the requirements of *ValueSwappable* and *RandomAccessIterator*.

Przykladowy kod dla algorytmu std::sort wykorzystujacego kontener std::array.
Kontener std::array posiada iterator:

`iterator`	*RandomAccessIterator* and *ConstexprIterator* (since C++20)that is a *LiteralType* (since C++17)

W tym przypadku zastosowano iterator zgodny z minimalnymi wymaganiami dla algorytmu std::sort.

```
#include <algorithm>
#include <functional>
#include <array>
#include <iostream>

int main()
{
    std::array<int, 10> s = {5, 7, 4, 2, 8, 6, 1, 9, 0, 3};

    // sort using the default operator<
    std::sort(s.begin(), s.end());
    for (auto a : s) {
        std::cout << a << " ";
    }
    std::cout << '\n';

    // sort using a standard library compare function object
    std::sort(s.begin(), s.end(), std::greater<int>());
    for (auto a : s) {
        std::cout << a << " ";
    }
    std::cout << '\n';

    // sort using a custom function object
    struct {
        bool operator()(int a, int b) const
        {
            return a < b;
        }
    } customLess;
    std::sort(s.begin(), s.end(), customLess);
    for (auto a : s) {
        std::cout << a << " ";
    }
    std::cout << '\n';

    // sort using a lambda expression
    std::sort(s.begin(), s.end(), [](int a, int b) {
        return a > b;
    });
    for (auto a : s) {
        std::cout << a << " ";
    }
    std::cout << '\n';
}
```