2023.10.30

Reverse Iter

```
template<typename InIter>
void PrintRange(InIter beg, InIter end)
{
    while (beg != end)
    {
        std::cout << *beg++ << ' ';
    }
}
int main()
{
    using namespace std;

    vector<int> ivec;
    rfill((ivec, 20, Irand{0, 100};

    auto riter_beg = ivec.rbegin();
    auto riter_end = ivec.rend();

    PrintRange(riter_beg, riter_end); // sondan başa doğru yazar

    PrintRange(riter_end.base(), riter_beg.base()); // baştan sona doğru yazar

}
```

Back Insert Iterator

```
template <typename InIter, typename OutIter>
OutIter Copy(InIter beg, InIter end, OutIter destbeg)
{
    while (beg != end)
    {
        *destbeg++ = *beg++;
    }
    return destbeg;
}
/*
    Yukarıdaki Copy algoritmasına dokunmadan ona çağrı yapıp
    yapacağımız çağrı dest_vec source_vec'teki öğeleri sondan ekleyecek
*/
template<typename Container>
class BackInsertIterator
{
    public:
        BackInsertIterator(Container &c) : rc(c) {}
        BackInsertIterator& operator++() {return *this;}
        BackInsertIterator& operator++(int) {return *this;}
        BackInsertIterator& operator*() {return *this;}
        BackInsertIterator& operator=(const typename Container::value_type& val)
        {
            rc.push_back(val);
            return *this;
        }
}
```

```
int main()
{
    using namespace std;
    vector<int> source_vec {2, 5, 1, 3, 4, 6, 9, 7};
    vector<int> dest_vec; // empty vector

    BackInsertIterator<vector<int>> iter(dest_vec);
    BackInsertIterator iter(dest_vec);

    Copy(source_vec.begin(), source_vec.end(), BackInsertIterator(dest_vec));

    print(dest_vec); // 2, 5, 1, 3, 4, 6, 9, 7
}
```

Yukarıdaki Kodun STL ile yazılmış hali:

```
#include<iterator>
#include<algortihm>

int main()
{
    using namespace std;
    vector<int> source_vec {2, 5, 1, 3, 4, 6, 9, 7};
    vector<int> dest_vec;

    copy(source_vec.begin(), source_vec.end(), back_inserter(dest_vec));
    print(dest_vec);

}
```

Back Inserter

```
#include<iterator>
#include<algortihm>

int main()
{
    using namespace std;

    vector<string> svec;
    rfill(svec, 10'000, [] {return rname() + ' ' + rfname();});

    vector<string> dvec;
    size_t len = 13;

    // uzunluğu 13 olanlar kopyalanır
    copy_if(svec.begin(), svec.end(), back_inserter(dvec),
    [len](const string& s) {return s.length() == len;});
}
```

Front Inserter

```
#include<iterator>
#include<algortihm>

int main()
{
    using namespace std;
    vector<int> source_vec {2, 5, 1, 3, 4, 6, 9, 7};
    list<int>  ilist;

    copy(source_vec.begin(), source_vec.end(), front_inserter(ilist));
    print(dest_vec);

}
```

STL'de iteratorleri manipüle eden algoritmalar:

- -advance (bir iter'i npos arttırmak için kullanılır) advance(ref)
- -distance (iki iterator arasındaki farkı buluyor) distance(iter1, iter2)
- -next (iter, 5) 5 ilerisini alıyor
- -prev (iter, 5) 5 gerisini veriyor
- -iter_swap (iter_x, iter_y) iter'in x konumuyla y konumu yer değiştirir.

```
using iter = std::vector<int>::iterator

int main()
{
    iter::value_type type; // iter türü yani int
    iter::difference_type diff_type; // iki iter farkından oluşan tür ptrdiff_t
    iter::pointer pointer_type;  // int*
    iter::reference ref_type; // int&
    iter::iterator_category; // random_access_iterator_tag
}
```

Iterator Category'sine Göre Fonksiyon Yazma

```
// iterator category'sine göre func_impl etme
// tag dispatch
template <typename Iter>
void func_impl(Iter beg, Iter end, std::random_access_iterator_tag) {}

template <typename Iter>
void func_impl(Iter beg, Iter end, std::bidirectional_iterator_tag) {}

template <typename Iter>
void func_impl(Iter beg, Iter end, std::forward_iterator_tag) {}

template<typename Iter>
void func(Iter beg, Iter end)
{
    func_impl(beg, end, typename Iter::iterator_category{});
}
```

```
// static if

template<typename Iter>
void func(Iter beg, Iter end)
{
    using cat = typename std::iterator_traits<Iter>::iterator_category;

    if constexpr (std::is_same_v<cat, std::bidirectional_iterator_tag>)
    {
        std::cout << " bidirectional_iterator_tag" << "\n";
    }
    else if constexpr (std::is_same_v<cat, std::random_access_iterator_tag>)
    {
        std::cout << "random_access_iterator_tag" << "\n"
    }
}
```

Advance

```
// advance
int main()
{
    using namespace std;

    vector<int> ivec{ 2, 4, 6, 7, 9, 3, 1};
    list<int> ilist{ 2, 4, 6, 7, 9, 3, 1};

    auto vec_iter = ivec.begin();
    auto list_iter = ilist.begin();

    // 3 adım ilerler
    // ikisi farklı fonkisyonlar overloading var complite time'da kod seçimi
yapılır
    // type dispatch örneği
    advance(vec_iter, 3);
    advance(list_iter, 3);

}
```

```cpp
template<typename Iter>
void Advance(Iter& it, int n, std::random_access_iterator_tag)
{
    it +=n; // vector
}
template<typename Iter>
void Advance(Iter& it, int n, std::bidirectional_iterator_tag)
{
    while(n--)
        ++it; // list
}

template<typename Iter>
void Advance(Iter& it, int n)
{
    Advance(it, n typename std::iterator_traits<Iter>::iterator_category{});
}
```

```cpp
int main()
{
    using namespace std;

    vector<int> ivec{ 2, 4, 6, 7, 9, 3, 1};
    auto iter = ivec.end();
    advance(iter, -3); // *iter 9
}
```

Distance

```cpp
int main()
{
    using namespace std;

    list mylist { 2, 5, 8, 9, 3, 1, 8};

    auto iter1 = mylist.begin();
    auto iter2 = mylist.end();

    advance(iter1, 2); //*iter = 8
    advance(iter2, -1); // *iter 1

    auto n = distance(iter1, iter2); // n = 4
}
```

Next and Prev (CPP 11)

```cpp
// next and prev --cpp11

int main()
{
    using namespace std;

    vector<string> svec{ "ali", "can", "ece", "naz", "gul", "eda", "tan"};

    auto iter = next(svec.begin(), 3); // *iter = naz
    iter = next(svec.begin()); // *iter = can
    list<string> slist{ "ali", "can", "ece", "naz", "gul", "eda", "tan"};

    std::cout << *slist.end() << "\n"; // tanımsız davranış
    std::cout << *prev(slist.end()) << "\n"; // tan
}
```

advance: ref parametreli iter'in kendisini artırıyor (call_by_reference)

next: iter değişmez ancak assign etmem gerekiyor (call_by_value)

Swap

```cpp
template <typename Iter1, typename Iter2>
void IterSwap(Iter1 it1, Iter2 it2)
{
    std::swap(*it1, *it2);
}

////////

int main()
{
    using namespace std;
    vector<string> svec{ "ali", "can", "tan", "ata"};
    list<string> slist{ "gul", "eda", "naz", "ela"};

    iter_swap(next(svec.begin()), prev(slist.end(), 2);
}
```

Find If

```
// find_if
template<typename InIter, typename Pred>
InIter FindIf(InIter beg, InIter end, Pred f)
{
    while(beg != end)
    {
        if(f(*beg))
            return beg;
        ++beg;
    }
    return end;
}
int main()
{
    using namespace std;

    list<string> slist;
    rfill(slist, 20, name);
    print(slist);

    {
        char c = '0';

        auto iter = find_if(slist.begin(), slist.end(), [c](const string &s)
        {
            return s.contains(c);
        });

        if (iter != slist.end())
        {
            std::cout << "bulundu... idx = " << distance(slist.begin(), iter) <<
'\n';
        }
        else
        {
            std::cout << "bulunamadı" << "\n";
        }
    }

    // ya da
        char c = '0';
        if (auto iter = find_if(slist.begin(), slist.end(), [c](const string &s)
        {
            return s.contains(c);
        }); iter != slist.end())
        {
            std::cout << "bulundu... idx = " << distance(slist.begin(), iter) <<
'\n';
        }
        else
        {
            std::cout << "bulunamadı" << "\n";
        }

}
```

Transform

```
template<typename InIter, typename OutIter, typename F>
OutIter Transform(InIter beg, InIter end, OutIter destbeg, F f)
{
    while (beg !=end)
    {
        *destbeg++ = f(*beg++);
    }
    return destbeg;
}

////////
auto get_len(const std::string& s)
{
    return s.size();
}
int main()
{
    using namespace std;
    vector<string> svec;

    rfill(svec, 100, rname);
    list<size_t> lenlist;

    transform(svec.begin(), svec.end(),
        back_inserter(lenlist), get_len);

    print(lenlist); // print svec'tekilerin size'ni yazar
}
```

```
std::string revstr(std::string s)
{
    /*
    std::string temp(s);
    std::reverse(temp.begin(), temp.end());

    return temp;

    Eğer parametre değişikenin zaten kopyasını çıkarıyorsak (yukarıdaki gibi)
    const string:: &s yerine std::string s kullanıp aşağdaki gibi yazabiliriz.
    */

    std::reverse(s.begin(), s.end());
    return s;
}

int main()
{
    using namespace std;
    vector<string> svec;

    rfill(svec, 100, rname);
    list<size_t> lenlist;

    transform(svec.begin(), svec.end(),
        svec.begin(), revstr);
}
```

```cpp
// transform 2. overload'u

template<typename InIter1, typename InIter2, typename OutIter, typename F>
OutIter Transform(InIter1 beg, InIter1 end, InIter2 beg2, OutIter destbeg, F f)
{
    while (beg != end)
    {
        *desbeg++ = f(*beg++, **beg2++);
    }

    return destbeg;
}
```

```cpp
int main()
{
    using namespace std;
    vector<int> v{ 1, 4, 7, 2, 9, 9, 3 };
    deque<int> d{ 2, 5, 8, 1, 9, 3, 5};

    list<int> ilist;

    transform(v.begin(), v.end(), d.begin(), back_inserter(ilist)
        [](int x, int y) {return x * x + y * y});
    )

}
```