3.Hafta

İçindekiler

Perfect Forwarding

`std::forward<>` kullanımını incelemiştik.

Lambda ifadelerinde türün elimizde olmadığından bahsetmiştik.

```
int main()
{
    auto fn = [](auto&& t){
        std::forward<decltype(t)>(t);
    };
    // C++20 ile
    auto fn1 = []<typename T>(T && t){
        std::forward<T>(t);
    };
}
```

Kısaltımlış template sentaksı da eklendi

```
template <typename T>
void func(T x);
```

```
//C++20 ile aşağıdaki gibi yazılabilir.
void foo(auto &&x);
```

- Universasl reference'ı sadece perfect forwarding için kullanmak zorudnda değiliz ve terimde de bir uyuşmazlık oluşuyor.

```
void navigate(std::string iterator beg, std::string::iterator end)
{
    std::cout<< "non const semantics on the passed range \n";
}
void navigate(std::string::const_iterator beg, std::string::const_iterator
end)
{
    std::cout<< "const semantics on the passed range \n";
}
template <typename T>
void process_contianer(T &&t)
{
    navigate(std::begin(t), std::end(t));
}
int main()
{
    std::string name{"mutable test"}
    std::string const cname{"imutable test"};
    process_contianer(name); // non const semantics on the passed range
    process_contianer(cname); // non const semantics on the passed range
    process_contianer(std::string{"temporary"}); // non const semantics on
the passed range
    process_contianer(std::move(name)); // non const semantics on the
passed range
    process_contianer(std::move(cname)); // non const semantics on the
passed range
}
```

navigate fonksiyonun 2tane overloadi var birinin parametreleri const diğerininki const olmayan iterator. `process_container` için de const'uluk korunuyor ve çağrı buna göre yapılıyor.

Kendimiz de constluğa bağlı generic bir fonksiyonda oluşturabiliriz.

```
#include <type_traits>
#include <iostream>
#include <string>
using namespace std;

template <typename T>
void func(T&& )
{
    if constexpr(std::is_const_v<std::remove_reference_t<T>>)
    {
```

```
        std::cout<< "const ";
    }
    else
    {
        std::cout<< "non const ";
    }

    if constexpr(std::is_lvalue_reference_v<T>)
    {
        std::cout<< "lvalue\n";
    }
    else
    {
        std::cout<< "rvalue\n";
    }
}

class Myclass {};

int main()
{
    std::string name{"non const test"}
    func(name); // non const
    std::string cname{"const test"}
    func(cname); // const

    func(Myclass{}); // non const
    Myclass c;
    func(c); // non const
    func(std::move(c)); // non const
}
```

> Buradaki if constexpre run time'a değil compile time ile alakalı bir fonksiyon.

Universal reference parametre birden fazla yerde kullanılabiliyor ve burada value-category dependent code yazabiliyoruz. type_Traits kütüphanesi ile T türünün hakkındaki bilgileri sorgulayabiliyoruz

```
template <typename T>
void func(T,T);

int main()
{
    func(1,2); // burada sentaks hatası yok
    func(1,2.); // Sentaks hatası
    func("ali","ayse"); //sentaks hatası değil
}
```

Derleyici açısından ilk argümana bakılırsa T int, ikinci argümana bakılırsa T double. Burada sentaks hatası oluşuyor.

```
template <typename T>
void func(T&,T&);

int main()
{
    func("ali","ayse"); //SENTAKS HATASI
}
```

Burada aynı sentaks hatası oluşuyor çünkü çıkarım `const char [4]`, `const char[5]` olarak yapılıyor.

```
template <typename T>
void insert(std::vector<T> &vec, T &&val)
{
    vec.push_back(std::forward<T>(val));
}

int main()
{
    std::vector<std::string> vec;
    std:string name;
    insert(vec, name); // SENTAKS HATASI
}
```

Bu sentaks hatasının nedeni 1.parametrenin çıkarımı T'nin string olması üzerine, 2.parametre içinse T'nin çıkarımı T& olarak ve string & olarak yapılıyor. Bu yüzden hata oluşuyor.

- remove_reference kullanarak bu sorunu çözebiliriz.

```
template <typename T>
void insert(std::vector<std::remove_referednce_t<T>> &vec, T &&val)
{
    vec.push_back(std::forward<T>(val));
}

int main()
{
    std::vector<std::string> vec;
    std:string name;
    insert(vec, name);
}
```

- 2 tane template parametre kullanarak

```
template <typename ElemType, typename T>
void insert(std::vector<ElemType> &vec, T &&val)
{
```

```
    vec.push_back(std::forward<T>(val));
}

int main()
{
    std::vector<std::string> vec;
    std:string name;
    insert(vec, name);
}
```

- Ya da container türünü template parametresi olarak alabiliriz.

```
template <typename Container, typename T>
void insert(Container &vec, T &&val)
{
    vec.push_back(std::forward<T>(val));
}
int main()
{
    std::vector<std::string> vec;
    std:string name;
    insert(vec, name);
}
```

auto &&

- auto && ile universal reference'ı kullanabiliriz.

```
class Myclass {};

int main()
{
    Myclass m;
    const Myclass cm;
    auto &&r1 = Myclass{}; // Myclass =
    auto &&r2 = m; // auto = Myclass & reference collapsing ile Myclass &
oluyor
    auto &&r3 = cm;
    auto &&r4 = std::move(m);
    auto &&r5 = std::move(cm);
}
```

> fonksiyonun parametresi T'türü için yapılan çıkarım nasıl yapılıyorsa, auto için ayapılan çıkarım da aynı şekilde yapılıyor. Buradaki çıkarım `auto` için yapılıyor. Template çıkarımı da T'türü için çıkarım yapıyor.

Kullanılmasının gerekli olduğu senaryolar:

- Forwarding reference olarak kullanabiliriz.

```
class Myclass {};
void foo(const Myclass &)
{
    std::cout<< "foo(const Myclass &)\n";
}
void foo(Myclass &&)
{
    std::cout<< "foo(Myclass &&)\n";
}
void foo(Myclass &)
{
    std::cout<< "foo(Myclass &)\n";
}
void foo(const Myclass &&)
{
    std::cout<< "foo(const Myclass &&)\n";
}

int main()
{
    Myclass m;
    const Myclass cm;

    auto &&r1 = Myclass{};
    foo(std::forward<decltype(r1)>(r1)); // foo(Myclass{}) arasında bir
fark yok
    auto &&r2 = m;
    foo(std::forward<decltype(r2)>(r2)); // foo(Myclass &)
    auto &&r3 = cm;
    foo(std::forward<decltype(r3)>(r3)); // foo(const Myclass &)
    auto &&r4 = std::move(m);
    foo(std::forward<decltype(r4)>(r4)); // foo(Myclass &&)
    auto &&r5 = std::move(cm);
    foo(std::forward<decltype(r5)>(r5)); // foo(const Myclass &&)
}
```

Doğrudan argüman olarak göndermek ile argüman olarak göndereceğimiz ifadeye universal referans bağladık ve bunu fonksiyona argüman olarak forward ederek kullandık.

Return value perfect passing

```
class Myclass {};
void foo(const Myclass &)
{
    std::cout<< "foo(const Myclass &)\n";
}
void foo(Myclass &&)
{
```

```
    std::cout<< "foo(Myclass &&)\n";
}
void foo(Myclass &)
{
    std::cout<< "foo(Myclass &)\n";
}
void foo(const Myclass &&)
{
    std::cout<< "foo(const Myclass &&)\n";
}

const Myclass & func_const_lref(const Myclass & str){ return str;}
Myclass & func_non_const_lref(Myclass & str){ return str;}
Myclass && func_ref(Myclass && str){ return std::move(str);}
Myclass func_value(Myclass str){ return str;}

int main()
{
    Myclass m;
    const Myclass cm;
    foo(func_rref(Myclass{})); // foo(Myclass &&) yukarıdaki
fonksiyonlardan parametresi  Myclass && olan'ın çağırılması lazım.
    foo(func_non_const_lref(m)); // foo(const Myclass &)
    foo(func_const_lref(cm)); // foo(const Myclass &)
    foo(func_value(m)); // foo(Myclass &&)
}
```

Func_rref fonksiyonuna geçici nesne yolladık ve geçilen argümanı move etti. Geri dönüş değerini perfect forward aetmek için özel bir şey yapmamıza gerek yok.

Fakat biz bu geri dönüş değerini bir fonksiyonda tutmak ve daha sonra bu değişkeni kullanarak fonksiyon çağırısını yapmak istiyorsak bu durumda `auto &&` kullanmak zorudnayız.

```
template <typename T>
void func(T &&t)
{
    foo(bar(std::forward<T>(t)));
}

int main()
{
    Myclass m;
    const Myclass cm;
    //Type  ret = bar(std::forward<T>(t));
    //foo(ret); //buradaki ret'in türü ne olucak. Bu l'artık l value oluyor
    auto && ret = bar(std::forward<T>(t));
    foo(std::forward<decltype(ret)>(ret));
}
```

```
using namespace std;

int main()
{
    vector<int> ivec(10);

    for(auto val : ivec)
    {
        cout<< val;
    }
    for(auto i : ivec)
    {
        i = 10;
    }
    cout <<"\n";
    for(auto val : ivec)
    {
        cout<< val;
    }
}
```

Fakat eğer vector'ün bool specializasyonu varsa bu durumda partial specializasyonu var.

```
int main()
{
    vector<bool> ivec(10);


    auto x = ivec[2]; //burada proxy object döndürüyor.
    //x bir vector'un bool açılımının referansı türü nested type
    auto x = ivec.operator[](2);
    ivec[3] = true; // yazıldığında
    ivec.operator[](3).operator=(true); // yazılmış oluyor.

    for(auto val : ivec){ cout<< val; }
    /*derleyicinin ürettiği psuedo kod
    auto &&rng = ivec;
    auto pos = rng.begin();
    auto end = rng.end();
    for(;pos != end; ++pos)
    {
        auto temp = *pos; eğer  val referans türü olsaydı auto &temp
olurdu, sağ taraf referans türü olsaydı auto &&temp olurdu.
    }
    */

    for(auto i : ivec){ i = true;  }
    cout <<"\n";
    for(auto val : ivec){ cout<< val; }

    auto iter = ivec.begin();
```

```
    iter.operator*(0) = true; //bu bize referans döndüremeyeceğine göre,
proxy nesnesi dönüyor
    //reference nested type'ından nesneye yapılmış oluyor.
}
```

> Vector'un bool açılımının 2 sorunu var 1.container değil, 2.bool tutmuyor. 😄

```
template<>
class Vector
{
    class reference
    {
        operator=(bool);
        operator bool()const;
    }
    reference operator[](size_t idx);
};
```

Eğer `auto val` yerine `auto &&val` olsaydı bu durumun bir etkisi olmayacaktı.

Aşağıdaki kodda amaç öyle bir fill fonksiyonu oluşturmakki 1.argümandaki container'a 2.parametredeki val değerini yerleştirmek.

- `auto &&` kullanarak sentaks hatasının önüne geçilebiliyor.

```
template <typename C, typename T>
void Fill(C & con, const T& val)
{
    for(auto & elem : con)
    //Sentaks hatasını engellemek için universal refeerans kullanmak
gerekiyor.
    {
        elem = val;
    }
}

template <typename C, typename T>
void Fill_2(C & con, const T& val)
{
    auto &&rng = con;
    auto pos = rng.begin();
    auto end = rng.end();
    for(;pos != end; ++pos)
    {
        auto &elem = *pos; //pos.operator*() bu fonksiyonun geri dönüş
değeri referans ve r-value expression
        //sentaks hatasının nedeni l-value referansa  r-value expression
atanması
        *pos = val;
```

```
    }
}

inr main()
{
    vector<int> ivec(10);
    for(auto val : ivec)
    {
        cout<< val;
    }
    Fill(ivec, 10);
    cout <<"\n";
    for(auto val : ivec)
    {
        cout<< val;
    }

    vector<string> svec{"ali", "veli", "selami"};
    for(auto val : svec)
    {
        cout<< val;
    }
    Fill(svec, "necati");
    cout <<"\n";
    for(auto val : svec)
    {
        cout<< val;
    }
    vector<bool> bvec{false, false, false};
    for(auto val : bvec)
    {
        cout<< val;
    }
    //Fill(bvec, true); //SENTAKS HATASI
    cout <<"\n";
}
```

Perfect Returning

Amacımız foo'nun geri döndürdüğü gibi olduğu gibi func fonksiyonun aynı geri dönüş değerini döndürmek istiyoruz.

```
template <typename T>
??? func(T &&t)
{
    foo(std::forward<T>(t));
}
```

Geri dönüş değeri yerine `decltype(auto)` yazmamız gerekiyor. Peki bu ne demek?

decltype(auto)

decltype için 2 farklı decltype var.

1. Operandı olan ifadenin bir isim formunda olması, declaration türünü elde ediyor
2. Eğer bir expression olursa, bu durumda ifadenin değer kategorisi oluyor:

- PRvalue ==> T
- Lvalue ==> T&
- Xvalue ==> T&&

decltype(auto)' da ise, tıpkı auto da olduğu bir değişkene ilk değer verdiğimizde, `auto`nun kurallarına göre değil de `decltype`a göre belirleniyor.

```
int & foo();
int && bar();
decltype(auto) f1()
{
    return expr; // int & f1() return type
}
int main()
{
    int x = 4;
    decltype(auto) y = x; // int y = x;
    decltype(auto) z = 4; // int z = 4;
    decltype(auto) t = foo(); // int & t = foo();
    decltype(auto) u = bar(); // int && u = bar();
    decltype(auto) y = (x); // x ile yukarıdaki farklı anlamlara geliyor

    int m{}, * ptr{&m};
    decltype(auto) r = m;
    //decltpye(auto) ilave bir deklaratör alamıyor.
}
```

```
struct Myclass
{

};

decltype(auto) foo(Myclass m)
{
    //return m; // Myclass
    //return (m); // Myclass & olurdu ve otomatik ömürlü bir nesneye
referans döndürmüş olur ve hata olurdu.
}
```

```
decltype(auto) fn_A(int i){  return i; }
//decltype(auto) fn_B(int i){ return (i); }
//Sentaks hatası fonksiyonun geri dönüş değeri int & otomatik ömürlü
nesneyi döndürüyor
decltype(auto) fn_C(int i){  return (i+1); }
decltype(auto) fn_D(int i){  return i++; }
// Yok çünkü son ek ++ operatörü oluşturduğu ifade PR-value expression ve
geri dönüş değeri int
decltype(auto) fn_E(int i){  return ++i;}
// Ön ek ++ operatörü oluşturduğu ifade L-value expression ve geri dönüş
değeri int & otamtik ömürlü nesneye referans dönüyor.
decltype(auto) fn_F(int i){  return (i >= 0 ? i : 0); }
// Bir problem yok, çünkü ternary operatorünün operandları PR-value
expression ve geri dönüş değeri int
d//ecltype(auto) fn_G(int i, int j) {  return i >= j ? i : j; }
// ternary operatör'Ün return ifadesinin kategörüsü gene int & döndürüyor
struct S {  int i = 0; };
decltype(auto) fn_H(){   return (S{});}
// İfade PR-value expression ve geri dönüş değeri S
//decltype(auto)fn_I(){  return (S{}.i);}
// R-value nesnelerin non-static veri elemanlarına erişim ifadesi X-value
expression ve dönüş değeri int && oluyor ve geçici nesneye referans
dönüyor.
```

O zaman bizim bunu perfect return etmemiz için

```
template<typenamem T>
decltype(auto) foo(T &&val)
{
    return bar(std::forward<T>(val));
}

// lambda fonksiyonları için trailing return type kullanmamız gerekiyor.
int main()
{
    auto fn = [](auto &&r) -> decltype(auto)
    {
        return bar(std::forward<decltype(r)>(r));
    }
}
```

- Fonksiyonun geri dönüş değerini `decltype(auto)` ile bildirilen bir değişkende tuttuk. Böylece geri dönüş değeri hergangi bir valueType olabilir. Eğer burada geri dönüş değeri:
- - R-value referans türü ise, `if constexpr` ile bu saptanıyor.
- - ret value olabilir.
- - L-value referans olabilir. Bu iki durumda ya value return ya da l-value referans return ediyoruz.

```
template <typename Func, typename... Args>
decltype(auto) call(Func f, Args&&... args)
{
    decltype(auto) ret {f(std::forward<Args>(args)...)};
    if constexpr(std::is_rvalue_refeerence_v<decltype(ret)>)
        return std::move(ret);
    else
        return ret;
}
```

Bunu bir lambda fonksiyonu ile de yapabiliriz.

```
int main()
{
    auto f = [](auto &&...args) -> decltype(auto)
    {
        return func(std::forward<decltype(args)>(args)...);
    };
    auto f1 = [](auto &&...args) -> decltype(auto)
    {
        decltype(auto) ret = func(std::forward<decltype(args)>(args)...);
        if constexpr(std::is_rvalue_reference_v<decltype(ret)>)
            return std::move(ret);
        else
            return ret;
    }
}
```

- C++23 öncesinde ki sorun, hayatı bitmiş bir nesneye tekrar kullanmaya çalışıyoruz ve dangling reference oluyor.

```
std::vector<std::string> create_svec();

int main()
{
    for(std::string s : create_svec()){}
    // Aşağıdaki 3 döngüde tanımsız davranış oluşturuyor
    for(char c: create_svec().at(0)){} //
    for(char c: create_svec()[0]){} //
    for(char c: create_svec().front()){} //
}
//

int main()
{
    const auto & r1 = create_svec(); //life extension var.
    auto &r = creae_svec(); //SENTAKS HATASI sağ taraf değeri sol
refereansa bağlanamaz life extension yok
```

```
    vector<std::string> &&r2 = create_svec();

     const auto & r3 = create_svec().at(0); //UNDEFINED BEHAVIOR. Burada
life extension yok,
}
```

Yukarıdaki kodların sorununu anlamak için aşağıdkai kodu inceleyelim.

```
#include <iostream>
#include <vector>
#include <string>
class Myclass
{
public:
    ~Myclass(){ std::cout<< "object Destructed....\n";}
    std::vector<int> getVec()const{
        return ivec;
    }
private:
    std::vector<int> ivec{1,2,3,4};
};
Myclass foo()
{
    return Myclass{};
}
int main()
{
    {
        const auto & r = foo();
        cout<< "main devam ediyor..1\n"
    }//obje burada destruyor olmalı
    {
        const auto & r = foo().getVec(); //life-extension yok, obje burada
destruct ediliyor
        //r'yi kullansaydık tanımsız bir davranış oluşturmuş oluyor.
        cout<< "main devam ediyor..2\n"
    }
    cout<< "main devam ediyor..3\n"
}
```

C++23 ile bu implementasyon değişti. Umulmadık ve beklenmedik durumlar oluşuyor. Bu bizi neden ilgilendiriyor.

Reference Qualifier

Bir üye fonksiyonun hangi değer kategorisindeki nesneler ile çalışabileceğini gösteriyor. Örneğin l-value referansı ile nitelendirilimş bir üye fonksiyon.

Aşağıdaki gibi bir fonksiyon yazılıdığına bir kopyalama oluyor. Fakat bunun geri dönüş değerini `const referans` yapılarak bir erişim verilebilir.

```
class myclass
{
public:
    std::string get_str() const{return m_str;}
private:
    std::string m_str;
};

Myclass create_myclass()
{
    return Myclass{};
}
int main()
{
    for(auto c : create_myclass().get_str()){}
}
```

Aşağıdaki kullanımlar legal fakat bunların semantik bir anlamı yok.

```
class Nec{};
Nec foo();
int main(){
    foo() = foo()
    Nec{} = Nec();
}
```

```
class Myclass
{
public:
    void foo()&; //l-value reference qualifier
    void func()&&; //l-value reference qualifier
    void bar() const &; //şeklinde de olaiblir.
}
int main()
{
    Myclass m;
    m.foo(); // foo(&m)
    Myclass{}.foo(); //SENTAKS HATASI
    std::move(m).foo(); //SENTAKS HATASI
    m.func(); //SENTAKS HATASI
    m.bar(); //SENTAKS HATASI

    cm.bar(); //
}
```

- Overload da edilebiliyor.

```
#include <iostream>
#include <string>
class Myclass
{
public:
    void foo()&{ std::cout<< "l-value reference qualifier\n"; } //l-value
reference qualifier
    void foo()&&{ std::cout<< "r-value reference qualifier\n"; } //l-value
reference qualifier
    void foo() const &{ std::cout<< "const l-value reference qualifier\n";
} //l-value reference qualifier
    void foo() const &&{ std::cout<< "const r-value reference qualifier\n";
} //l-value reference qualifier
};

int main()
{
    Myclass m;
    const Myclass cm;
    m.foo(); // foo(&m)
    cm.foo(); // foo(&cm)
    Myclass{}.foo(); // foo(&&Myclass{})
    move(m).foo(); // foo(&&m)
    move(cm).foo(); // foo(&&cm)
}
```

- Bazı senaryolarda bu çok büyük bir hata olabilir. Örneğin

```
void foo(bool){    std::cout<< "foo(bool)\n";}
void foo(std::string)  {    std::cout<< "foo(std::string)\n";}
std::string getStr(){ return "necati";}

int main()
{
    //if(getstr() = "necati")  //SENTAKS HATASI çünkü boola dönüştürülemez.
    foo(getstr() == "necati");
    foo(getstr() = "necati"); // Sentaks hatası yok ve parametresi string
olan fonksiyon çağırılıyor.
    // ilk olarak atama operatörü çağırılıyor ve bu atama operatör
fonksiyonu reference qualifier değil ve teknik bir engel yok.
    // bu kod foo(getstr().operator=("necati")) şeklinde çalışıyor.
Operator atama fonksiyonun geri dönüş değeri *this. ve string overload'da
çalışır.
}
```

Move semantiği ile kullanımına bir örnek

```
class Person
{
```

```
public:
    Person(const std::string &name) : m_name{name}{}
    std::string & get_name() &&{  std::cout<< "r-value reference
qualifier\n"; return std::move(m_name); }
    std::string & get_name() const &{  std::cout<< "l-value reference
qualifier\n"; return m_name; }
    std::string & get_name() & {  std::cout<< "const l-value reference
qualifier\n"; return m_name; }
private:
    std::string m_name;
};

template <typename T>
void foo(T &&x)
{
    auto name = std::forward<T>(x).get_name();
    std::cout<< name << "\n";
}

int main()
{
    Person p{"necati"};
    const Person cp{"necati"};
    foo(p); // l-value reference qualifier
    foo(cp); // l-value reference qualifier
    foo(Person{"necati"}); // r-value reference qualifier
    foo(std::move(p)); // r-value reference qualifier
    foo(std::move(cp)); // l-value reference qualifier
}
```

Universal Referans Olmayan Fakat Öyle Gözüken Senaryolar

Örneğin aşağıdaki kod universal referansı değil, const sağ taraf referans.

```
template <typename T>
void func(const T&&){}
```

Nested type'lar, burada x'in türü sağ taraf referansı.

```
template <typename T>
void func(Con & x, typename Con::value_type &&){}
```

```
template <typename T>
void func(std::vector<T> &&){}
```

generic bir sınıfta kullandığımız zaman universal referans **olmayan** bir fonksiyon yazmış oluyoruz.

```
template<typename T>
class Stack
{
public:
    void push(const T &&)
    {
        std::cout<< "push(const T &&)\n";
        m_con.push_back(val);
    }
    void push(T &&)
    {
        std::cout<< "push(T &&)\n";
        m_con.push_back(std::move(val));
    }

private:
    std::vector<T> m_con;
};

int main()
{
    Stack<std::string> istack;
    std::string str{"lvalue"};
    istack.push(str);
    istack.push(std::move(str));
}
```

Fonksiyonun parametresi ne olmalı?

- Elimizde bir sınıf türünden parametre alan ve salt okuma yapan bir fonksiyon olsun:
- - Eğer bu string'in tersten okunması gerekiyorsa

```
void func(const std::string & str)
{
    auto stem = str;
    reverse(str.begin(), str.end());
}
```

> ADL: Argument Dependent Lookup: Eğer fonksiyonua gönderilen agümanlardan biri bir namespace içerisinde tanımlanmış türe ilişkin ise, o zaman fonksiyon ismi o namespace içerisinde de aranır.

Move Only Types

Sınıflar problem domainindeki varlıkları temsil etme amacı olarak kullanılıyor ve bu varlıklardan bazıları kopyalamaya uygun ya da birden fazla probleme yol açılıyor. Sınıflar bunu engellemek için kopyalamaya karşı kapatılıyor. Fakat bu varlık eğer bir kaynağı tutuyorsa o kaynağı tekrar kullanmak için move edilebiliyor.

Bu sınıfı:

```
class MoveOnly
{
public:
    MoveOnly() = default;
    MoveOnly(const MoveOnly&) = delete;
    MoveOnly& operator=(const MoveOnly&) = delete;
    MoveOnly(MoveOnly&&) = default;
    MoveOnly& operator=(MoveOnly&&) = default;
};
```

Tanımlayabiliyoruz. Move memberlar için 2 seçeneğimiz var, bu memberları biz yazabiliriz ya da compiler bunu bizim için yazabilir.

```
class MoveOnly
{
public:
    MoveOnly() = default;
    MoveOnly(const MoveOnly&) = delete;
    MoveOnly& operator=(const MoveOnly&) = delete;
    MoveOnly(MoveOnly&&){ std::cout<< "MoveOnly(MoveOnly&&)\n"; }
    MoveOnly& operator=(MoveOnly&&){ std::cout<< "MoveOnly& operator=
(MoveOnly&&)\n"; }
};

void func(MoveOnly m)
{
    std::cout<< "func(MoveOnly m)\n";
}

int main()
{
    MoveOnly m;
    func(m); // SENTAKS HATASI
    func(std::move(m)); // MoveOnly(MoveOnly&&)
}
```

Inittializer List Hatırlatma:

- Container'lar söz konusu olduğunda **initializer listleri** var:
- - Derleyici burada arka planda bir array oluşturuluyor ve bu arrayin öğelerini initializer listteki öğelerle hayata başlıyor ve burada bir **kopyalama** oluyor.
- - Yani bunları move only type'lar ile kullanamıyoruz. Initializer listessi arka planda 2 tane pointer tutuyor (Biri başlangıcını öbürü de bittiği yerini tutuyor).

```
void func(std::initializer_list<MoveOnly> ilist)
{
    cout << "&ilist "<< &ilist<< "\n";
```

```
    cout << "dizi adresi "<< ilist.begin()<< "\n";
}
int main()
{
    std::initializer_list<int> mylist{2,7,6,9,1};
    cout << "&ilist "<< &mylist<< "\n";
    cout << "dizi adresi "<< mylist.begin()<< "\n";
    func(mylist);
}
```

- Size veya distance fonksiyonu çağırılarak boyutu elde edilebilir. For-based range loop ile de kullanılabilir.
- Constructor'larda kullanılıyor.

```
void func(int a, int b, int c)
{
    std::vector<int> ivec{a,b,c};
    ivec = {a,b,c};
}

int main()
{
    using namespace std;
    vector<int> ivec{1,2,3,4,5};
}
```

- Initializer list parametreli bir constructor'ın önceliği daha yüksek.

```
class Myclass
{
public:
    Myclass(int){ std::cout<< "Myclass(int)\n"; }
    Myclass(std::initializer_list<int>){ std::cout<<
"Myclass(std::initializer_list<int>)\n"; }
};
int main()
{
    Myclass myNec{10};
    Myclass myNec1(1);
}
```

Burada gennnellikle string sınıfı için soruluyor.

```
int main()
{
    using namespace std;
    cout << string(66,"X")<<"\n";
```

```
    cout << string{66,"X"}<<"\n";
    cout << vector<int>(19,10).size()<<"\n";
    cout << vector<int>{19,10}.size()<<"\n";
}
```

> STL'de genellikle insert işlemlerinde ayrı ayrı çağırmak yerine tek bir fonksiyon çağrısı ile yapabiliyorsak o çağrı ile yapabiliriz.

```
class MoveOnly
{
public:
    MoveOnly() = default;
    MoveOnly(const MoveOnly&) = delete;
    MoveOnly& operator=(const MoveOnly&) = delete;
    MoveOnly(MoveOnly&&){ std::cout<< "MoveOnly(MoveOnly&&)\n"; }
    MoveOnly& operator=(MoveOnly&&){ std::cout<< "MoveOnly& operator=
(MoveOnly&&)\n"; }
};

int main()
{
    using namespace std;
    //vecotr<MoveOnly> ivec{MoveOnly{}, MoveOnly{}, MoveOnly{}}; // SENTAKS
HATASI çünkü burada kopyalama semantiği işliyor.
    vector<MoveOnly> myvec(100);
    //for(auto x : myvec){} //SENTAKS HATASI çünkü burada gene kopyalama
yapılıyor.
}
```

`print_tr()` fonksiyonu ile amacımız sınıfın hangi özelliklere sahip olduğunu görmek.

```
class MoveOnly
{
public:
    MoveOnly() = default;
    MoveOnly(const MoveOnly&) = delete;
    MoveOnly& operator=(const MoveOnly&) = delete;
    MoveOnly(MoveOnly&&){ std::cout<< "MoveOnly(MoveOnly&&)\n"; }
    MoveOnly& operator=(MoveOnly&&){ std::cout<< "MoveOnly& operator=
(MoveOnly&&)\n"; }
};

template <typename T>
void print_tr()
{
    if constexpr(std::is_default_constructible_v<T>)
    {
        std::cout<< "default constructible\n";
```

```
    }
    else
    {
        std::cout<< "not default constructible\n";
    }
    if constexpr(std::is_destructible_v<T>)
    {
        std::cout<< "destructible\n";
    }
    else
    {
        std::cout<< "not destructible\n";
    }
    if constexpr(std::is_copy_constructible_v<T>)
    {
        std::cout<< "copy constructible\n";
    }
    else
    {
        std::cout<< "not copy constructible\n";
    }
    if constexpr(std::is_copy_assignable_v<T>)
    {
        std::cout<< "copy assignable\n";
    }
    else
    {
        std::cout<< "not copy assignable\n";
    }
    if constexpr(std::is_move_constructible_v<T>)
    {
        std::cout<< "move constructible\n";
    }
    else
    {
        std::cout<< "not move constructible\n";
    }
    if constexpr(std::is_move_assignable_v<T>)
    {
        std::cout<< "move assignable\n";
    }
    else
    {
        std::cout<< "not move assignable\n";
    }
}

class Myclass{}

int main()
{
    print_tr<Myclass>();
    print_tr<MoveOnly>();
}
```

STL'de Move Semantics ve Perfect Forwarding

- iostream sınıfı
- ofs move-only bir sınıf olmasına rağmen nesneyi döndürdü.

```
#include <iostream>
#include <string>
#include <vector>
#include <fstream>
#include <initializer_list>

std::ofstream create_file(const std::string &name)
{
    std::ofstream ofs{name};
    if(!ofs)
    {
        throw std::runtime_error{ name + "File cannot be opened"};
    }
    return ofs;
}

void write(std::ofstream ofs){}

void func(std::ofstream & ofs);

int main()
{
    auto ofs = create_file("test.txt");
    //write(ofs); SENTAKS HATASI çünkü kopyalamaya karşı kapalı.
    //write(std::move(ofs));
    func(ofs); // bu aktarımda dikkatli olmak gerekiyor
    //Burada dosyanın durumunu bilmemiz için func fonksiyonunun ne
yaptığını bilemiyoruz.
}
```

- string sınıfı

```
int main()
{
    string str{"dogruyum\ncaliskanim\nbuyuklerimi saymak\nkucuklerimi
sevmek\n"};
    istringstream iss{str};
    string name;
    iss >> name;//geline(iss,name) de yapabilirdik.
    cout <<"|" << name <<"|\n"
}
```

Modern cpp ile bu inserter ve extractor'un sağ taraf referansllı overloadları da ekledi

```
int main()
{
    using namespace std;
    string str{"dogruyum\ncaliskanim\nbuyuklerimi saymak\nkucuklerimi
sevmek\n"};
    int x = 345;
    string name {"cansin"};
    double dval = 4.2;
    auto s = (ostringstream oss()<< x << name << dval).str();
    cout << s << "\n";

    ofstream ofs{"test.txt"};

}
```

- Bir vector taşındığı zaman, moved-from state'deki vector'ün size'ı 0 oluyor. Bu standartlarda garantili bir durum.

Algoritmaları

remove & remove_if & unique

- Stl'deki bazı silme algoritmaları biz farkında olmasakta taşınmaya neden olabilir. Algoritmaların çoğunun parametreleri iterator olduğu için silme işlemleri yapamıyor ve bu algoritmalar o range'teki öğeeleri arrange ediyorlar.
- Öğelerin sayısı değişmiyor fakat bazı öğeler moved-from state'de olabilir.

```
template <typename ForIt, class T>
ForIt Remove(ForIt first, ForIt last, const T &val)
{
    first = std::find(first, last, val);
    if(first != last)
         for(auto i = first; ++i != last;)
             if(!(*i == val))
                *first++ = std::move(*i); //Burada neseneler moved-from
state kalıyor olabilir.
     return first;
}

template <typename ForIt, class UnaryPred>
ForIt remove_if(ForIt first, ForIt last, UnaryPred f)
{
    first = std::find_if(first, last, f);
    if(first != last)
         for(auto i = first; ++i != last;)
             if(!f(*i))

                *first++ = std::move(*i);
```

```
     return first;
}
```

> Remove bir range'teki öğeleri logic silme işlemine tabi tutuyor. Remove_if bir unaryPredicat'in true değer verdiği durumlarda logic silme yapıyor. Unique ise aynı değere sahip üyelerin sayısını 1'e indiriyor.

```
int main()
{
    using namespace std;
    vector<string> svec{"ali", "veli", "selami", "necati", "ali", "veli",
"selami", "necati"};
    auto iter = remove_if(svec.begin(),svec.end(), [](const string &s){
return s.find('i') != std::string::npos;});
    int cnt{};
    while(iter != svec.end())
    {
        cout<< ++cnt; <<" "<<*iter++ << "\n";

    }
}
```

Sınıfın veri elemanlarını initalize etmek için birden fazla yöntem kullanıyor olabiliriz. Josutis 3 tane ayrı sınıf oluşturuyor.

```
#include <iostream>
#include <string>
#include <vector>
#include <chrono>
class PersonClassic
{
public:
    PersonClassic(const std::string &name, const std::string &surname) :
m_name{name}, m_surname{surname}{}
private:
std::string m_name;
std::string m_surname;
};

class PersonInitMove
{
public:
    PersonInitMove(std::string name, std::string surname) :
m_name{std::move(name)}, m_surname{std::move(surname)}{}
private:
    std::string m_name;
    std::string m_surname;
};
class PersonInitOverload
```

```
{
public:
    PersonInitOverload(std::string name, std::string surname) :
m_name{std::move(name)}, m_surname{std::move(surname)}{}
    PersonInitOverload(const std::string &name, const std::string &surname)
: m_name{name}, m_surname{surname}{}
    PersonInitOverload(std::string&& name, const std::string &surname) :
m_name{std::move(name)}, m_surname{surname}{}
    PersonInitOverload(const std::string &name, std::string&& surname) :
m_name{name}, m_surname{std::move(surname)}{}
    PersonInitOverload(std::string&& name, std::string&& surname) :
m_name{std::move(name)}, m_surname{std::move(surname)}{}
    PersonInitOverload(const char *name, const char *surname) :
m_name{name}, m_surname{surname}{}
    PersonInitOverload(const char *name, const std::string &surname) :
m_name{name}, m_surname{surname}{}
    PersonInitOverload(const std::string &name, const char *surname) :
m_name{name}, m_surname{surname}{}
    PersonInitOverload(std::string&& name, const char *surname) :
m_name{std::move(name)}, m_surname{surname}{}
    PersonInitOverload(const char *name, std::string&& surname) :
m_name{name}, m_surname{std::move(surname)}{}
private:
    std::string m_name;
    std::string m_surname;
};

const char* pname =
"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxx";
const char* psurname =
"yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyy";

using Person = PersonClassic;
//using Person = PersonInitOverload;
//using Person = PersonInitMove;

std::chrono::nanoseconds measure(int num)
{
    std::chrono::nanoseconds total_duration{};
    for(int i = 0; i<num ; ++i)
    {
        std::string name(100,'N');
        std::string surname(100,'S');
        auto start = std::chrono::steady_clock::now();
        Person px{pname, psurname};
        Person p1{name, surname};
        Person p3{std::move(name), std::move(surname)};
        auto end = std::chrono::steady_clock::now();
        total_duration += end - start;
    }
    return total_duration;
}
```

```
constexpr int n = 100'000;

int main()
{
    measue(20);
    std::chrono::nanoseconds nanosecond_duration{measure(n)};
    std::chrono::duration<double,
std::milli>millisecond_duration{nanosecond_duration};
    std::cout<< "test resuls for "<< n << " runs\n" <<
millisecond_duration.count() << "ms\n";
    std::cout<< "test resuls for "<< n << " runs\n" <<
nanosecond_duration.count() << "ns\n";
}
```

onlinegdb

En yavaş çalışan parametresi const referans olan. En hızlı çalışan ise move olan.

Emplace

- Push back eklenecek nesnenin nerede construct edildiğini biliyor. Sol referans olan overload'ın da copy-constructor çağırıyor, sağ taraf referansta olan ise move constructor'ı çağırıyor.
- Emplace back C++17 ile artık geri dönüş değeri var ve `constexpr T&` dönüyor.
- - emplace yapıldığında ne copy ne de move constructor çağırılıyor. Burada container'ın allocator'ının edindiği alanda nesneyi oluşturuyor. Move-only bir sınıf içerisinde bu çağırılabilir.
- - emplace back çağırıldığında argümanların gönderildiği constructor'da diğer sınıf nesneleri için move-constructor'ı çağırabiliyor.

```
template <typename T, typename Allocator = std::allocator<T>>
class Vector
{
public:
    void push_back(const T &val)
    {
        //new(place)T(t);
        if(m_size == m_capacity)
            reserve(m_capacity * 2);
        m_allocator.construct(m_elem + m_size, val);
        ++m_size;
    }

    void push_back(T &&val)
    {
        //new(place)T(std::move(t));
        if(m_size == m_capacity)
            reserve(m_capacity * 2);
        m_allocator.construct(m_elem + m_size, std::move(val));
        ++m_size;
    }
```

```
    template <typename... Args>
    constexpr T& emplace_back(Args&&... args)
    {
        //new(place)T(std::forward<Args>(args)...

        if(m_size == m_capacity)
            reserve(m_capacity * 2);
        m_allocator.construct(m_elem + m_size, std::forward<Args>
(args)...);
        ++m_size;
        return m_elem[m_size - 1];
    }
private:

};

//Emplace_back 2. faydası, birden fazla parametreli olan olabilir.

vector<Fighter> fvec;
fvec.emplace_back(exp1, exp2, exp3); //bu argümanlarda sınıfın
constructor'ına geçiriliyor
// ve  bunlar R-value expresionsa bunlar için de move constructor
çağırılıyor.
```

Copy_backward algoritması

Elimizde bir vektör var ve

```
#include "nutility.h"
int main()
{
    using namespace std;
    vector<int> ivec(10);
    iota(ivec.begin(),ivec.end(),0);
    print(ivec);
    //ilk 1.öğeyi 3.öğeye gelecek şekilde kopyalamak istiyorsak.
    copy_backward(ivec.begin(), ivec.begin() + 5, ivec.begin() + 7);
//yazma range'nin end iterator'ü
    print(ivec);
}
```

> `std::iota(ivec.begin(),ivec.end(),0)` ile 0'dan başlayarak 10'a kadar olan sayıları ivec'e atıyoruz. copy_backward ile reverse copy arasında fark var.

Copy_backward:

```
template<typename InIter, typename OutIter>
OutIter Copy(InIter first, InIter last, OutIter dest_first)
{
```

```
    while (first != last)
    {
        *dest_first++ = *first++;
    }
    return dest_first;
}

template<typename BidIt1, typename  BidIt2>
BidIt2 CopyBackward(BidIt1 first, BidIt1 last, BidIt2 dest_last)
{
    while(last != first)
        *--dest_last = *--last;
    return dest_last;
}
```

Move_backward algoritması

```
template<typename InIter, typename OutIter>
OutIter Move(InIter first, InIter last, OutIter dest_first)
{
    while (first != last)
    {
        *dest_first++ = std::move(*first++);
    }
    return dest_first;
}

template<typename BidIt1, typename  BidIt2>
BidIt2 MoveBackward(BidIt1 first, BidIt1 last, BidIt2 dest_last)
{
    while(last != first)
        *--dest_last = std::move(*--last);
    return dest_last;
}
```

Örneğin:

```
int main()
{
    using namespace std;
    vector<string> sv{"ali", "veli", "selami", "necati", "ali", "veli",
"selami", "necati"};
    vector<string> destvec(svec.size());
    copy(svec.begin(), svec.end(), destvec.begin());
    print(destvec);
    int count{};
    for(const auto &name:svec)
    {
        cout<< ++count << " " <<name << " ";
```

```
    }
    cout<< "\n";
    move(svec.begin(), svec.end(), destvec.begin());
    count = 0;
    for(const auto &name:svec)
    {
        cout<< ++count << " " <<name << " ";
    }
}
```

Move Iteratör Adaptörü

Move iteratör C++11 ile dile eklendi.

Adaptör: Adaptör tasarım paternini implemente eden bir sınıf. STL içinde:

- Fonksiyon adaptörleri: Bir callable alıyor ve bir callable geri veriyor.
- Container adaptörleri: `Stack`, `queue`, `priority_queue`. Bir container'ı data member olarak alıyorlar ve o container'ın interfaceini kullanarak kendi interface'ini oluşturuyorlar.
- Iterator adaptörleri: `reverse_iterator`, `move_iterator`, Bir tane iteratörü alıyor ve onun interface'ini kendi kullanım amacına göre uyarlıyor.

2 şekilde adaptör oluşturulabilir:

- Biri container adaptörü gibi veri elemanı yaparak

```
template <typename T>
class Myclass
{
public:
    void foo()
     {
        mc.func();
     }
private:
    C mc;
};
```

- Kalıtım yoluyla:

```
template <typename Iter>
class Myiterator : public Iter
{

};
```

`move_iterator<vector<int>::iterator> iter;` gibi kullanılabilir.

```cpp
#include <iterator>
template <typename Iter>
typename std::move_iterator<Iter> MakeMoveIterator(Iter it)
{
    return std::move_iterator<Iter>{it};
}

int main()
{
    using namespace std;
    vecor<string>svec{"ali", "veli", "selami", "necati"};
    //biz move iterator oluşturmak istiyoruz

    //1.si açıkça türü yazmak
    moev_iterator<vector<string>::iterator> iter{svec.begin()};

    //2.si CTAD kullanılarak. C++17 ve sonrasında geldi
    move_iterator iter2{svec.begin()};

    //3.sü ise bir fabrika fonksiyonu ile
    auto miter = make_move_iterator(svec.begin());
}
```

```cpp
int main()
{
    using namespace std;
    vecor<string>svec{"ali", "veli", "selami", "necati"};
    auto miter = make_move_iterator(svec.begin());
    auto name = *miter;
    cout<< svec[0].size() << "\n"; //ilk öğe taşınmış durumda ve 0 oluyor.
}
```

Move iteratör dikkatli olunmak zorunda çünkü eğer bu nesne 2.kez dereferense ederse move-from state'deki nesneyi kullanmış oluyoruz ve bu tanımsız davranış oluşturuyor.

- Bu öğeleri container'ın üyelerini kullanarak yeni bir container'da kullanmış olmak istiyoruz.

```cpp
int main()
{
    using namespace std;
    vecor<string>svec{"ali", "veli", "selami", "necati"};
    vecor<string>svec1{svec.begin(), svec.end()}; //copy constructor
çağırıldı.

    for(int cnt = 0; const auto & name : svec1)
    {
        cout<< ++cnt << " " << name << "\n";
    }
```

```
    vector<string>mvsvec{move_iterator{svec.begin()},
move_iterator{svec.end()}}; //move constructor çağırıldı.
    for(int cnt = 0; const auto & name : svec1)
    {
        cout<< ++cnt << " " << name.length() << "\n";
    }
}
```

```
void do_something(std::string s)
{
    std::cout << "fonksiyona gelen sisim" << s << "\n";
}
int main()
{
    using namespace std;
    vector<string>svec;
    rfill(svec,10,rname); //nutility.h'dan geliyor ve vector'ü random
olarak dolduruyor.
    //fonksiyona içinde i 'harfi geçenleri fonksiyona göndermek istiyoruz.
    for_each(svec.begin(), make_move_iterator(svec.end()),
    //referans olursa sentaks hatası olur çünkü r-value dönüşmüş olur
    //[](string &str)
    //[](string &&str) bu-da olmaz çünkü hala l-val expression. Bunun için
do_something'te move yapmamız lazım.
    [](string str)
    {
        if(str.find('i') != string::npos)
            do_something((str));
    });

    for(int count = 0; const auto &name : svec)
    {
        cout<< ++count << " " << name << "\n";
    }
}
```

Literal Operator Functions

- Bir tamsayı sabitini yazarken digit separator kullanılabiliyor.

`87'13821312'2132313'` gibi. Kullanılabiliyor. Magic numberlar ile kullanıldığın da bu kodun okunabilirliğini arttırıyor. Hangi sayı türü ile yazıldığının sabitin hangi tabanda yazıldığı ile alakası yok `0x1a'cff`.

Userdefined Literal

Dilin core sentaksı içerisinde nasıl kullanılan sabitler olduğu gibi. Biz de kendi sabitlerimizi oluşturabiliyoruz.

```
int main()
{
    354u;
    "alican"s;
}
```

- STL'in sunduğu literal operatörler.
- Programcının kendisi de böyle fonksiyonları tanımlayabiliyor ve programcının oluşturduğu türler için de kullanılabiliyor.

`743ms` yazdığımızda burada bir fonksiyon çağırısı yapılıyor. Biz böyle bir ifadeyi derleyici bazı fonksiyonlar var ise bunu bir çağrı koduna değiştiriyor. Burada çağırılan free function `constexpr` bir fonksiyonda olabilir ve buradan elde edilen ifade bir compile time sabiti oluyor.

- Bu fonksiyonların görünür olması durumunda isimleri ile çağırabiliyoruz.

1. Bu fonksiyonlar keyfi olarak parametre türleri alamıyor. Parametre değişkenlerinin hangi türden olabileceğine ait sentaks kuralları var.
2. Fonksiyonun sadece parametrelerine ilişkin bir kural var geri dönüş değeri için programcıyı engelleyen bir durum yok.
3. Suffix olarak kullanılan isim fonksiyonun ismi ve programcının yazacağı custom type için `_<isim>` şeklinde olmalı.

Bunlar 2 ayrı kategoride oluyor:

- **Cooked**: Eğer bir tam sayı kullanarak ve ona son ek eklenecekse, derleyici bu ifadeyi doğrudan bir tam sayı olarak gönderiyor. `312321_m` => `operator""_m(unsigned long long)` ile türden unsigend long long paramtresine 312321 olarak gönderiliyor.

- **Uncooked**: Çağırılan fonksiyonun parametresi c-string ve bu yukardaki gibi bir fonksiyon ise `operator""_m(const char *p)` bu fonksiyona "312321" olarak gönderiliyor ve artık bu yazıyı karakter karakter olarak dolaşabiliriz.

- Tam sayı ve gerçek sayılar için cooked/uncooked yazma şansımız var.

- Bu fonksiyonları namespace içerisinde sun.

- **unsigned long long/char** olmalı

```
int operator""_sr(unsigned long long val)
{
    std::cout<<"operator_k ->  val = " << val << "\n";
    return static_cast<int>(std::sqrt(val));
}
int main()
{
    using namespace std;
    auto x = 843712_sr;
    cout<< x << "\n";
```

```
    auto = operator""_sr(843712); //şeklinde de çağırılabilir.
}
```

- **uncooked** olarak bu fonksiyon yazılsaydı:

```
int operator""_sr(const char *p)
{
    std::cout<<"operator_k ->  val = " << std::strlen(p) << "\n";
    while(*p)
    {
        std::cout << *p << " " << (int)*p<< "\n";
        ++p;
    }
    return 0;
}
int main()
{
    auto val = 823.3232_sr
}
```

Öyle bir son ek olsunki 2'lik sayı sisteminde bir sayıya denk gelsin

```
int operator""_b2(const char *p)
{
    int val{};
    while(*p)
    {
        if(*p == '0'|| *p == '1')
            val = val * 2 + (*p - '0');
        else
            throw std::invalid_argument{"invalid binary digit"};
        ++p;
    }
    return val;
}

int main()
{
    auto val = 101011101_b2;
}
```

- Gerçek sayı sabiti içinse `long double val` türden bir değişken olmak zorunda.