* Std::move:

- move fonksiyonu. taşımaya hazırlar! Taşımanın kendisini yapmaz X

 # Look for Howard Hinnant

→ Rvalue Reference vs Forwarding Reference:

→ Forwarding referans olması için bir type deduction olmalı! Çünkü fırst de T&&

-
```
template <typename T>
void func(T&& r);
```
→ Forwarding Reference ✓

-
```
template <typename T>
void func(const T&& r);
```
→ Forwarding Reference Değil X

-
```
template <typename T>
class Myclass {
public:
    void foo(T&&);
};
```
→ Forwarding Reference Değil X

Çünkü, T'nin türü çıkarıldığında foo o türden && olacak.

-
```
template <typename T>
class Myclass {
public:
    template <typename U>
    void foo(U&&);
};
```
→ Forwarding Reference ✓

Çünkü bu bir member template

-
```
template <typename T>
void func(std::vector<T> &&)
```
→ Forwarding Reference Değil XX

-
```
template <typename T>
class Nec {
public:
    struct Erg{

    }
};

template <typename T>
void func(Nec<T>::Erg &&)
```
→ Forwarding Reference Değil XX

-
```
int main()
{
    auto&& x = 10;
}
```
→ Forwarding Reference ✓

-
```
int main()
{
    for (auto&& x : con) {

    }
}
```
↓ range based for loop!

→ Forwarding Reference ✓

* Reference Collapsing:

→ referansa referans olmaz! Bunun olduğu yerlerde ya L value ya da R value ref oluşmalı (pointer to pointer olur ✓)

→ reference collapsing → template parametrelerinde
→ using ile eş isim bildirimlerinde

```cpp
class Myclass{};

using LREF = Myclass&;
using RREF = Myclass&&;

int main()
{
    LREF &r
}
```

```cpp
//template parametre forwarding &
//auto &&
//decltype
//using
```

→ Collapse Cheat:

T& L	& L	T& L
T& L	&& R	T& L
T&& R	& L	T& L
T&& R	&& R	T&& R

```cpp
void func(std::string&& s);
```

↳ move garantisi yok!

* Her zaman move, copy'e kıyasla avantajlı olmayabilir!

```cpp
class Nec {

private:
    std::array<int, 500> mx;
};
```

→ heapte tutulan bir kaynak yok. std::array, C dizilerinin sarmalanmış hali. move ctor'un olması, kopyalamaya kıyasla avantajlı olduğu anlamına gelmiyor. Çünkü hem copy hem move aynı işi yapacak!

```cpp
class Nec {

private:
    std::vector<int> mx;
};
```

→ Fakat vector sınıfı 3 adet pointer tutuyor. 10.000 adet öğeyi kopyalamaktansa, bu pointerları çalmak (move) daha avantajlı!

*** Not:**

- Pr value expressionlar, en başta değere sahip değil. Cpp 17'de aslında bir formül/ formül gibi işlev. Kendi değeri yok. Hayata gelecek nesneye → (result object'e) dönüşünce değer olacak. ⇒ Bu olay temporary materialization! (örn: geçici nesne)

```cpp
auto s = string{"necati ergin"}
```

⇒ Burada ne copy ne move var!!

⇒ Cpp 17'de materialized olunca değer alırmı ??

* moved from state:

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

    string str(100'000, 'A');
    vector<string> svec;
    ///

    svec.push_back(move(str));

    str
}
```

→ Şu an str'nin değerini bilemeyiz X
Değer ile ilişkilendiremeyiz.
→ Ancak nesnenin kendisi hala valid!

* move'un implementasyonu: → prefix function adında bir sınıf
amacı compile time da bir işlem / hesaplama

```
template <typename T>
std::remove_reference_t<T>&& Move(T&& t)
{
    return static_cast<std::remove_reference_t<T>&&>(t);
}
```

```
template <typename T>
struct RemoveReference {
    using type = T;
};

template <typename T>
struct RemoveReference<T&> {
    using type = T;
};

template <typename T>
struct RemoveReference<T&&> {
    using type = T;
};

template <typename T>
using RemoveReference_t = RemoveReference<T>::type;
```

* Mandatory Copy Elision öncesi tekrar:

→ Special member functions: - Derleyici tarafından yazılır / Derleyici tarafından yazılması talimatı verilebilir.
- En iyisi (ideali) hepsini derleyici yazsın: Rule of zero
- Fonksiyonun kodunu derleyici yazmasın: default etmek
- Special member functionlar şu 3 durumda olabilir.

```
not declared

user declared

implicitly declared
```

```cpp
struct Myclass {
    Myclass(int);
};
```

→ Default ctor not declared!

```cpp
struct Myclass {
    Myclass();
    Myclass(const Myclass&);
};
```

→ Move ctor not declared!

⇒ Yani eğer move çağrısı yapılırsa, fallback olup copy çağrılıyor olucak!

```cpp
struct Myclass {

    Myclass(const Myclass&) = delete;
```

→ Copy ctor user declared!

ama deleted. Çağrılması sentaks hatası!

```cpp
struct Myclass {

    Myclass(const Myclass&) = delete;
    Myclass& operator=(const Myclass&) = delete;
    Myclass(Myclass&&);
    Myclass& operator=(Myclass&&);
};

struct Myclass {

    Myclass(Myclass&&);
    Myclass& operator=(Myclass&&);
};
```

Bu iki bildirim aynı!

move memberlar declare edilirse, copy memberlar delete edilir.!

```cpp
// Myclass.h
class Myclass {
public:
    ///...

    ~Myclass();

};


///myclass.cpp
Myclass::~Myclass() = default;
```

→ Pimpl idiomunda kullanılıyor

⇒ Bu şekilde header'da user declared ama cpp kodunda default → Bu legal bir kullanım.

```
not declared


user declared
     defined
     default
     delete

implicitly declared
     - defaulted
     - deleted
```

implicitly declared → Derleyicinin duruma göre, dilin kuralları gereği biz bildirmesek bile sınıfın special member functionlarını bildirir. ve nasıl tanımlanırsa o şekilde tanımlar!

deleted → eğer yazması gerektiği gibi yazarken, bir syntax error ile karşılaşırsa DELETE EDER!!

```
7  class Myclass {
8  public:
9
10 private:
11     const int mx;
12 };
13
14 int main()
15 {
16     Myclass m;
17 }
```

→ Syntax hatası:

Çünkü default constructor, non-static data memberları default init eder. Const nesneler default init edilemez XX

Default ctor is deleted!!

(Benzer durum referanslar için de geçerli)

```
7  class Member {
8  public:
9  protected:
10     Member();
11 };
12
13 class Myclass {
14 public:
15
16 private:
17     Member mx;
18 };
19
20 int main()
21 {
22     Myclass m;
23 }
```

→ protected'a çağrı yaptık, Myclass default ctor is deleted!

```
class Base{
public:
    Base(int);
protected:

};

class Myclass : public Base{
public:

private:

};

int main()
{
    Myclass m;
}
```

Hazır default ctor yok!

Taban sınıfın default ctor'ına çağrı

* Howard Hinnat's Special Member Function Table:

	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
hiçbiri	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

* Hatırlatma:

```cpp
class Myclass {

private:
    int mx{};
    std::string mname{ "noname yet" };
};
```

→ Default member initialization Fakat bu, eğer bir default constructorda başka değer vermezsek bu değeri alınacak! Default ctor tüm veri elemanlarını default initialize edecek → garbage value
in determined value

```cpp
class Myclass {
public:
    Myclass() : mx(98) {}
    void print()const
    {
        std::cout << mx << " " << mname << "\n";
    }

private:
    int mx{ 34 };
    std::string mname{"necati ergin"};
};

int main()
{
    Myclass m;

    m.print();
}
```

98 ve necati ergin print edilecek

* PR value'nun Cpp 17'deki Yeni Tanımı:

→ mandatory copy elision için yol açtı!

- PR value artık bir değere sahip bir nesne değil. Bir nesneyi initialize eden bir formül gibi
  PR value ile copy ya da move etmeden nesne init ediyoruz!

```cpp
//copy elision

class Myclass {
public:
    Myclass(const Myclass&) = delete;
};

Myclass foo()
{
    return Myclass{};
}

void bar(Myclass);

int main()
{
    Myclass m = Myclass{};

    bar(Myclass{});
}
```

(Myclass{} — PR value; Myclass{} — PR value)

→ Cpp 17 öncesinde:

- Myclass{} PR value olmasına rağmen, bir fonksiyona argüman olarak geçmek için kullanılmasa bile copy constructor olmak zorunda idi!

→ Cpp 17 ve sonrasında:

- Bir materialization süreci başlıyor, ve obje ilk değer alıyor. PR valueler bir final object'e bağlanmalı / final object init etmeli

```
class Myclass{
public:
    Myclass();
    Myclass(int);

};

void func(Myclass);


int main()
{
    func(Myclass{ 12 });
    func(Myclass{ });
}
```

→ Sadece 1 ker constructor çağrılır!

Copy çağrılmaz!

```
Myclass func(int x)
{
    return Myclass{ x };
}


int main()
{
    auto m = Myclass{ Myclass{ func(12) } };
}
```

sadece 1 ker constructor çağrılır!

→ Bu durum "unmaterialized object passing" denir!