

- Tekrar:  $\rightarrow$  std::invoke : - kendisine argument olarak gonderilen fonksiyon, ilgili argumentlerle cagirir.  
 - generic function call iken yar  
 - member func pointer kullanmadan diger kompatibiligi ortadan kaldirir.

```

1 class MyClass {
2 public:
3     void func()
4     {
5         std::cout << "MyClass::func()\n";
6     }
7 };
8
9 int main()
10 {
11     MyClass m;
12     MyClass* p{ &m };
13
14     auto mfp = &MyClass::func;
15
16     (m.*mfp)();
17     (p->*mfp)();
18
19     std::invoke(mfp, m);
20     std::invoke(mfp, p);
21
22 }
23
24
25
26 }
```

Yukarıda:  $\rightarrow$  std::invoke(mfp, m);  
 $\rightarrow$  std::invoke(mfp, p);

- + NOT\_fn: - Birden bir unary predicate olur, karsılıklı yine bir unary predicate verr.  
 - Fakat gonderdigi diger, belli predictonun minin logic NOT nari

```

1 #include <iostream>
2 #include <functional>
3 #include "nutility.h"
4
5
6
7 int main()
8 {
9     using namespace std;
10
11     auto f = not_fn(isprime);
12     cout << boolalpha;
13
14     cout << f(19) << "\n";  $\rightarrow$  false
15     cout << f(20) << "\n";  $\rightarrow$  true
16
17 }
```

$\rightarrow$  isprime : cogur, onun sonucun logical NOT halini dondurur.

$\rightarrow$  Kullanim olan, algoritmlarla faydalı.

```

int main()
{
    using namespace std;

    vector<int> ivec;
    vector<int> destvec;

    rfill(ivec, 200, Irand(0, 100'000));

    copy_if(ivec.begin(), ivec.end(), back_inserter(destvec), not_fn(isprime));
    print(destvec);
}
```

prime de false dondu ve kopyalandi.

$\rightarrow$  !isprime yapilmaz, cunki operatörler farklı oluyor

### -Ornek 2:

```

int main()
{
    using namespace std;

    vector<string> svec;
    vector<string> destvec;

    rfill(svec, 100, rname);
    auto f = [] (const std::string& s) {return s.find('a') != string::npos; };

    copy_if(svec.begin(), svec.end(), back_inserter(destvec), not_fn(f));
    print(destvec);
}
```

$\rightarrow$  lambda expression for finding string contains 'a'

Stringde 'a' yoksa  
 stringler kopuler.

- \*STD Function: - Belirtti bir parametrik yapıdaki fonksiyonu sunmalar ve `std::function` nesnesinin fonksiyon çağrı operatörünü çağırınca ()() sunmasıdır. callableın fonksiyon çağırma direktörü.
- `std::function` bir fonksiyon adıptörü değil X
  - Vardır bir sınıf ✓
  - Genellikle ifadesi bir callback yapıyı oluşturur.
- ↳ Callable'lar C diliinde function pointer  
↳ C++ de ise function adımları da kullanılır.

```
main.cpp  notlar.txt  (Global Scope)
struct Functor {
    int operator()(int x) const
    {
        return x - 12;
    }
};

class MyClass {
public:
    static int foo(int a)
    {
        return -a;
    }
};

int main()
{
    using namespace std;

    std::function<int(int)> fn = cube; class template
    cout << fn(12) << "\n"; fn'ın obje olarak bir
                           fonksiyon çağırır

    fn = [](int a) {return a * a + 1;}; lambda ifadesi
                                       tek olarak bind edilebilir.
    cout << fn(12) << "\n";

    fn = Functor(); Functor class
    cout << fn(12) << "\n";

    fn = MyClass::foo; static member function
    cout << fn(12) << "\n";
}
```

- Her callable ile çalışabilir  
- Tekrar tekrar genel bir callable'a  
bağlanabilir.

→ Bir class template adı da var → Cpp 17 de genel CTAD ile kullanılabilir

```
notlar.txt  (Global Scope)
#include <vector>
#include <algorithm>
#include <iterator>
#include <string>

int foo(int a, int b)
{
    return a + b + 10;
}

int main()
{
    using namespace std;

    function<int(int, int)> fn = foo; class template argument deduction
                                         (CTAD) ile int olığını buldu
    cout << fn(12, 45) << "\n"
}
```

→ `std::function` default initialize edebilir. Herhangi bir fonksiyona sahip olmak zorunda değil!

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

    function<int(int)> f;
    f = nullptr; mutable böyle "=nullptr" olsun
                  deniyor
}
```

→ Eğer = nullptr veya default int handle function çağırırsa → Bad Function Call exceptionını throw eder

```
CA_2022
1 #include <iostream>
2 #include <functional>
3 #include "nutility.h"
4 #include <vector>
5 #include <algorithm>
6 #include <iterator>
7 #include <string>
8
9 int foo(int x) { return x + 1; }
10
11 int main()
12 {
13     using namespace std;
14
15     function<int(int)> f;
16
17     try {
18         auto val = f(20);
19     } catch (const std::bad_function_call& ex) {
20     //catch (const std::exception& ex) {
21         std::cout << "exception caught: " << ex.what() << '\n';
22     }
23
24
25 }
26 }
```

I

→ Generalized callback örnek:

```
main.cpp
1 #include <vector>
2 #include <algorithm>
3 #include <iterator>
4 #include <string>
5
6 using fn_type = std::function<int(int)>;
7
8 void func(fn_type fn)
9 {
10     //code
11     auto x = fn(24);
12     std::cout << "x = " << x << "\n";
13     //..
14 }
15
16 int bar(int x) { return x * 5; }
17
18 int main()
19 {
20     func(bar);
21     func([](int a) {return a * a; });
22 }
```

I

→ Bir dizesi overajı ise Container'lar ne kullanılsın.

```
{ std::cout << "int foo(int)\n";
    return a * a;
}

struct Multiply {
    Multiply(int val) : mx{val}{}
    int operator()(int x)const
    {
        return x * mx;
    }
    int mx{};
};

int sum_square(int a, int b)
{
    return a * a + b * b;
}

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

    vector<fn_type> myvec;

    myvec.emplace_back(square);
    myvec.emplace_back(Multiply{10});
    myvec.emplace_back([](int a) {return 2 * a * a; });
    myvec.emplace_back(bind(sum_square, 10, _1));

    for (auto& fn : myvec) {
        std::cout << fn(40) << "\n";
    }
}
```

Vektörün tüm callback üyesinin 40 dğeri ile traverse etti.

arka callback birleştir.

## \* Dinamik Ömrülü Nesneler :

- Bir nesnenin ömrünün dinamik olmasıyla, dinamik storage kullanımda Ama Kasınlıkla AYNI ŞEVI DEĞİL XXX  
↳ malloc != new

- Statik ömrülü nesneler, ↳ string türkçeler  
    ↳ global nesneler  
    ↳ statik anahtar sırasıyla oluşturulan  
        yerel değişkenler.

- Otomatik ömrülü nesneler program akışı onların tanımladığı scopelerin arasında, ömrüne göre dirler.

- Dinamik ömrülü nesnenin lifespanı zamanın bize bağlı.

- new ifadesi, dinamik ömrülü nesnenin hayatı, basitanzı, Construct etti. Delete ile hayatı sonlandırır.

```
new string ↳ yedin new
new Fighter ↳ yedin new

new Fighter[n] ↳ array new

new (buffer)Fighter ↳ new vergiye giden
new (std::nothrow)Fighter

delete ptr;

delete[]ptr;
```

## ↳ (Yedlin) New İfadesi:

```
void* operator new(size_t); ↳ Operator new, malloc terci bir allocator function
void* malloc(size_t); ↳ I

new Fighter ↳ "now" + "Class" kullanıldığında (New İfadesi)
operator new(sizeof(Fighter)) ↳ Fighter, orgjnen class olup, Operator new'le
                                goeser.
```

- New expression != operator new
- malloc fisi ↳ nullptr
- operator new fisi ↳ bad alloc exp

- new İfadesi bir pointer oluşturur. Aynısı const olmasında sakınca yok

```
int main()
{
    auto p = new const Date(1, 4, 1987);
}
```

↳ Nesnenin tüm elementler read only!  
↳ const bir obje.

- () ve ya {} ile value init edilir. Bučuk onlar olmazsa, default init / default ctor çağırılır.

### \* Delete Operator:

- **memory leak:** Allocate edilen var olanının giri verilmemesi
- **resource leak:** Bulutken kaynakın giri verilmemesi

\* Eger delete kullanırsak, class destructor'ı çağırılmaz + sizeof(class) looks olsun giri verilmemis olur.

```
delete pm;

void free(void *p);
void operator delete(void *p);
```

### \* Operator new vs New operator (New expression)

The difference between the two is that operator new just allocates raw memory, nothing else.  
The new operator starts by using operator new to allocate memory, but then it invokes the constructor for the right type of object, so the result is a real live object created in that memory.  
If that object contains any other objects (either embedded or as base classes) those constructors are invoked as well.

```
void* operator new(std::size_t n)
{
    std::cout << "my operator new function called!  n = " << n << "\n";
    void* vp = std::malloc(n);
    if (!vp) {
        throw std::bad_alloc{};
    }

    std::cout << "the address of the allocated block is: " << vp << "\n";
    return vp;
}

void operator delete(void *vp)
{
    std::cout << "my operator delete function called!  vp = " << vp << "\n";
    std::cout << vp << " the memory block at the address of " << vp << " freed\n";
    std::free(vp);
}
```

- Operator new / operator delete global direkt overload edilebilir → her new birim overload'u çağırır. / Sınıfın overload da mention
- new operator / delete operator overload edilemez X
- (new student) (delete student)

## \* Array New Operator / Array Delete Operator:

- Belirtilen adresteki dinamik olarak sınırlı dizimi siler.

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

    size_t n;
    cout << "kac nesne: ";
    cin >> n;

    auto p = new Myclass[n];
    delete[] p;
}
```

*Mənələrə gəldiklər sırasında  
destruct edilir.*

## \* Direct !!

- new ve delete uygunlu olmali yoxsa undefined behavior
- Yəni new[] ne sədəcə delete()

```
    ... row >> col,
    int** pd = new int* [row];

    for (size_t i{}; i < row; ++i) {
        pd[i] = new int[col];
    }

    Irand myrand{ 0, 9 };

    for (size_t i{}; i < row; ++i) {
        for (size_t k{}; k < col; ++k) {
            pd[i][k] = myrand();
        }
    }

    for (size_t i{}; i < row; ++i) {
        for (size_t k{}; k < col; ++k) {
            cout << pd[i][k];
        }
        cout << '\n';
    }

    for (size_t i{}; i < row; ++i) {
        delete[] pd[i];
    }

    delete[] pd;
```

*\*\*  
\*\**

## \* Placement New Operator:

```
int main()
{
    std::cout << "sizeof(Myclass) = " << sizeof(Myclass) << "\n";
    char buffer[sizeof(Myclass)];
    new(buffer)Myclass
}
```

*Placement new, bəzən arguman olur,  
sənət construct edilecək məməriyə alır.*

\* İsteklerde Sorular: Ne zaman, hangi durumda destructor explicit olarak çağırılır?

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

    std::cout << "sizeof(Myclass) = " << sizeof(Myclass) << "\n";
    char buffer[sizeof(Myclass)];
    cout << "buffer dizi adresi " << (void*)buffer << '\n';

    auto p = new(buffer)Myclass;

    p->~Myclass(); //dikkat!!
}
```

→ Cevap:

- Placement new ile bizzat türün etkileşimiz belirtilebileceği alanında, bir nesne oluşturulduğunda destructor, delete operatörü ile yığılmazı undefined behaviordir. (Buffer'in free etimde çalışmaz)
- Buffer < Class → türümüz olursa,

\* Placement new operator overloaded edilemez.

\* NO Throw New:

- Gerekenen operator fonksiyonu boşlukızırsa, exception throw etmeden yerine nullptr return eder.

```
int main()
{
    auto p = new(std::nothrow)Myclass;
    if (!p) {
        // Standard istemeden
        // bir empty class istenir
    }
}
```

→nothrow + throw, constexpr varlığı  
malloc gibi, bizzat böyle bir  
handle yapmanız gerekebilir.

\* Operatör new / Operatör Delete Fonksiyonlarının Bir Sınıf İçin Overload Edilmesi:

→ solve o class türünden nesne oluşturulduğunda, bizzat fonksiyonuz allocation gerçekleştirecek. .... (delete editir)

\* Sınıfların operatör new / fonksiyonu = Implicitly Static  
operatör delete

```
int main()
{
    auto p = ::new Myclass;
    ::delete p;
}
```

Scope resolution operatörü ile  
standart kütüphanesinin operatör new/delete

```
int main()
{
    auto p = new Myclass;
    delete p;
}
```

Yoksa, Myclass'in operatör new/delete fonksiyonu

+ Compiler New implementation:

```
new_handler get_new_handler();  
new_handler set_new_handler(new_handler);  
  
void* operator new(size_t n)  
{  
    for (;;) {  
        void* vp = std::malloc(n);  
        if (vp) {  
            return vp;  
        }  
        new_handler fp = get_new_handler();  
        if (!fp) {  
            throw std::bad_alloc{};  
        }  
        fp(); } } ↳ loopla bâzın betirildiyiniz func losur.
```

```
using new_handler = void (*)(void);  
//typedef void (*new_handler)(void);  
new_handler gp{};
```

\* Bu implementasyon esnekdir.  
\* std::get\_new\_handler kullanılır.