2023 12 22

std::bind

```
// std::bind
 #include <functional>
 int foo(int x, int y, int z)
 {
     std::cout << "x = " << x  << "y = " << y  << "z = " << z;
 }

 int main()
 {
     using namespace std;
     using namespace std::placeholders;
     auto f = bind(foo, 20, _2, _1);
     f(333,999); // foo(20, 999, 333); çağırır aslında
 }
```

```
struct Functor
{
    void operator()(int a, int b)
    {
        std::cout << "a = " << a << " b = " << b << "\n";
    }
};

int main()
{
     using namespace std;

     bind(Functor{}, placeholders::_1, 6512)(90); // function object
     bind([](int a, int b, int c)  // lambda fonksiyon
     {
         return a + b + c;
     }, 99, placeholders::_1, placeholders_1);

     auto val = fn(35); // val = 169
}
```

Class member fonksiyonlarda std::bind

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

        void bar(int a)const
        {
            std::cout << "Nec::bar(int a) a = " << a << "\n";
        }
 };

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

     Nec mynec;

     auto f1 = std::bind(&Nec::foo, _1);
     f1(mynec);

     auto f2 = std::bind(&Nec::bar, _1, 89);
     f2(mynec);
 }
```

```
void func(int& x, int& y, int& z)
 {
     x += 10;
     y += 10;
     z += 10;
 }

 int main()
 {
     int a = 72;
     int b = 82;
     int c = 92;

     auto f1 = std::bind(func, a, b, c);
     f1();
     std::cout << a << " " << b << " " << c << "\n"; // 72 82 92

     auto f2 = std::bind(func, ref(a), ref(b), ref(c));
     f2();
     std::cout << a << " " << b << " " << c << "\n"; // 82 92 102
 }
```

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

     vector<int> ivec(100);
     generate(ivec.begin(), ivec.end(), [] {return rand() % 1000;});

     int val = 900;

     cout << count_if(ivec.begin(), ivec.end(), [](auto i){return i > 900;}) <<
"\n";

     cout << count_if(ivec.begin(), ivec.end(), bind(greater{}, _1, val)) << "\n";
 }
```

```
// bind parametrelerin kopyasını çıkarır
 void increment(int& x)
 {
     ++x;
 }

 int main()
 {
     using namespace std;

     int ival{35};

     auto fn1 = bind(increment, ival);
     auto fn2 = bind(increment, ref(ival));

     fn1();
     cout << "ival = " << ival << "\n"; // ival = 35
     fn2();
     cout << "ival = " << ival << "\n"; // ival = 36
 }
```

std::function

function<>

template param. olarak bir callable'in cagrilmaya aday fonksiyon türünü kullanmamız gerekiyor.

```cpp
#include <functional>
 int foo(int x)
 {
     std::cout << "foo(int) cagrildi\n";

     return x * 19;
 }

int bar(int x)
 {
     std::cout << "bar(int) cagrildi\n";

     return x * x;
 }
 double baz(double);
 int main()
 {
     /*
        foo'un türü: int(int)
        &foo'un türü int(*)(int)
     */

     std::function<int(int)> f(foo);

     cout << "ret = " << f(90) << "\n"; // foo(90);   ret = 109

     f = bar;
     cout << "ret = " << f(90) << "\n"; // bar(90);   ret = 90 * 90

     std::function fctad = foo; // CTAD kullanabiliriz
     fctad = baz; // tür uyumusuzluğu olur.
 }
```

```cpp
class Nec
 {
     public:
        Nec(int x) : mx{x} {};
        void print_sum(int a) const
        {
            std::cout << mx << " + " << a << " = " << mx + a << "\n";
        }

        static void print(int a)
        {
            std::cout << mx << " + " << a << " = " << mx + a << "\n";
        }
    private:
        int mx;
 };

 void print_int(int x)
 {
     std::cout << "[" << x << "]\n";
 }

 int main()
 {
     using namespace std;

     function<void(int)> f(print_int); // function
     f(23);

     f = [](int x) // lambda
     {
         cout << " x = " << x << "\n";
     };

     f(45);

     f = Nec::print // class static function

     function fc = &Nec::print_sum; // syntax hatası
     function<void(const Nec&, int> fc = &Nec::print_sum; // legal

 }
```

std::function --exception

```cpp
// std::function --exception
 int main()
 {
     using namespace std;
     function<int(int)> f;
     try {
         auto val = f(45); // bad_function_call throw eder
     }catch (const std::bad_function_call& ex){
        std::cout << "exception caught: " << ex.what() << "\n";
     }
 }
```

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

     function<int(int)> f;

     if(f) //f.operator bool()
         std::cout << "bos degil\n";
     else
         std::cout << "bos\n";
 }
```

```cpp
// std::function amacı
 void foo(int (*fp)(int))
 {
     /*
        Bu fonksiyona sadece function pointer gönderebilirim. Ama,
        Function object, state lambda gönderememem
     */
     auto val = fp(12);
 }
 // int(int) olan her türlü callable bar fonksiyonu kabul eder.
 void bar(std::function<int(int)>);

 int f(int);
 struct Nnec
 {
    static int foo(int);
 };

 struct Erg
 {
     bool operator()(int)const;
 };

 int func(int, int, int);

 int main()
 {
     auto fn = [](int x){return x;};
     // hepsi geçerli
     bar(f);
     bar(Nec::foo);
     bar(Erg{});
     bar(fn);

     auto fb = bind(func, _1, _1, _1);
     bar(fb);
 }
```

```cpp
// Kullanım Senaryosu 1:
class Myclass
{
    public:
        void foo()
        {
            auto val = mf(12);
        }
    private:
        std::function<int(int)> mf;
}
```

```cpp
// Kullanım Senaryosu 2:
int f1(int);
int f2(int);
int f3(int);
int f4(int);
struct Functor
{
    int operator()(int)const;
};

using fntype = std::function<int(int)>;
int main()
{
    using namespace std;
    vector<fntype> myvec{ f1, f2, f3, f4 };

    myvec.push_back(Functor{});

    for (auto& f : vec)
    {
        auto val = f(12);
    }
}
```

std::mem_fn

```
class Myclass
{
    public:
        void func()const
        {
            std::cout < "Myclass::func()\n";
        }
        void foo(int x)const
        {
            std::cout < "Myclass::foo(int x) x = \n" << x <<"\n";
        }
}

int main()
{
    using namespace std;
    auto f1 = mem_fn(&Myclass::func);

    Myclass mx;
    f1(mx); // "Myclass::func()"

    auto f2 = mym_fn(&Myclass::foo);
    f2(mx, 768); // "Myclass::foo(int x) x = 768"
    Myclass* ptr = &mx;
    f1(ptr);
}
```

```
//std::mem_fn Kullanım Senaryosu 1
int main()
{
    using namespace std;
    vector<string> svec;

    rfill(svec, 20, [] {return rname() + ' ' + rfname();});

    vector<size_t> lenvec(svec.size());

    transform(svec.begin(), svec.end(), lenvec.begin(), [](const string& s)
    {
        return s.size();
    });

    // ya da

    transform(svec.begin(), svec.end(), lenvec.begin(), mem_fn(&string::size));
}
```

```
//std::mem_fn Kullanım Senaryosu 2
class Nec
{
    public:
        Nec(int val) : mval(val) {}
        void print()const
        {
            std::cout << "[" << mval << "]\n";
        }
    private:
        int mval;
};

int main()
{
    using namespace std;
    vector<Nec> nvec;

    for(int i = 0; i < 20; ++i)
    {
        nvec.emplace_back(i);
    }

    for_each(nvec.begin(), nvec.end(), mem_fn(&Nec::print));
}
```

std::not_fn

```
int main()
{
    int x = 11;
    auto b = std::isprime(x); // true

    auto f = not_fn(isprime);
    b = f(x); // false
}
```

```
//std::not_fn Kullanım Senaryosu

int main()
{
    using namespace std;
    vector<int> ivec;
    rfill(ivec, 100, Irand{0 , 100'000});

    copy_if(ivec.begin(), ivec.end(), ostream_iterator<int>{cout, "\n"}, [](int x)
{ return !isprime(x)});
    copy_if(ivec.begin(), ivec.end(), ostream_iterator<int>{cout, "\n"},
not_fn(isprime));
}
```

std::array (C dizilerini sarmalayan bir wrapper)

```cpp
int main()
{
    std::array<int, 5> ar; // ar'ın elemanları garbage durumdalar
}
```

Neden std::array kullanalım?

- STL uyumlu
- Interface var
- .at exception throw ediyor
- array decay yok (dizinin ilk elemanın bir pointer adresine dönüşmesi)
- Geri dönüş değeri ya da parametresi std::array olan fonksiyonlar yazabiliriz.

```cpp
int main()
{
    int a[0]; // C-style array -sytantax hatası

    std::array<int, 0> ax; // boş array geçerli
    boolalpha(cout);
    cout << "ax.size() = " << ax.size() << "\n";
    std::cout << ax.empty() << "\n";
}
```

```cpp
std::array<int,3> foo(int a, int b, int c)
{
    // hepsi geçerli
    //return std::array<int,3> {a, b, c};
    //return std::array {a, b, c}; //
    return {a, b, c};
}
```

```cpp
template <typename T, std::size_t n>
std::ostream& operator << (std::ostream& os, const std::array<T, n>& ar)
{
    for (std::size_t i{}; ar.size();  ++i)
    {
        os << ar[i] << ", ";
    }

    return os << ar.back() << "]";
}

int main()
{
    using namespace std;

    array<int, 3> ax{2, 5, 7};
    cout << ax << "\n";
}
```

```cpp
int main()
{
    using namespace std;
    array a = {"naci", "cemal", "emre"}; // std::array<const char*>
    // literal operator function
    array b = {"naci"s, "cemal"s, "emre"s}; // std::array<string>
}
```