4.Hafta

İçindekiler

Literal Operator Functions

6.Derste anlatılan kısım Literal Operator Functions

- Uncooked ve cooked ayrıştımrak için

```
#include "Date.h"

Date operator""_dt(const char* p);

Date operator""_dts(const char* p, std::size_t n)
{
    std::puts(p);
    cout << "n = " << n << "\n";
    return {};
}

int main()
{
    //23051987_dt fonksiyonunu bir yazı olarak alsın
    Date dx = 23'05'1987_dt;  //Date dx = 23051987_dt;
    cout<<  "23-05-1987"_dts;
}
```

Çoğunlukla `size_t` değerini kullanmıyoruz.

- Parametre `char` olabilir.

```
int operator""_i(char c)
{
    return static_cast<int>(c);
}

void operator""_pr(const char *p)
{
    std::cout << p << std::endl;
}

int main()
{
    'A'_i;
    34.56_pr;
    123498_pr;
}
```

```
constexpr std::string operator""_ds(const char *p, std::size_t)
{
    return std::string{p} +" " + p;
}

int main()
{
    using namespace std;
    auto str = "murat yilmaz"_ds;
    cout <<"|"<< str << "|\n";
}
```

Geri dönüş değeri için bir kısıtlama yok

```
std::vectır<char> operator""_vec(const char *p, std::size_t n)
{
    std::vector<char> ret(n);
    int i{};
    while(*p)
        ret[i++] = (*p++);
    return ret;
}

int main()
{
    auto vec = "ahahahahahah"_vec;
    std::cout << vec.size() << "\n";
    for (auto c :vec)
    {
        std::cout << c << " ";
```

```
    }
    std::cout << "\n";
}
```

Strong Types

Bir çok programda gerçek sayı ve tam sayı türlerini daraltılmış bir konseptte kullanıyoruz. Bu kullanımların bazı sakıncaları var.

- Tür dönüşümleri kullanılan primitive türlere göre yapılacak ve kullanım amacımızı anlaması zorlaşıcak örneğin:

```
class Meter
{
public:
    struct PreventUsage{};
    //explicit Meter(double dval) : value{dval} {} // copy init sentaks
hatası oluyor.
    Meter(PeventUsage, double dval) : value{dval} {}
private:
    double value;
};

double operator""_m(long double dval)
{
    return Meter{Meter::PreventUsage{},static_cast<double> dval};
}

int main()
{
    Meter m1;
    double dval = 56.08;

}
```

Raw String Literal

- Bir string literali oluşturulduğuna çift tırnak karakteri escape edilmek zorundaydı

```
namespace STD
{
    inline namespace Literal{
        inline namespace StrinLiterals
        {
            std::string operator""_s(const char *p, std::size_t);
        }
    }
}
```

```
int main()
{
    using namespace std;
    const char *p = "\"\"\"\\C++\\C++\\ali";
    using namespace std::literals;

    auto name = "mustafa"s;
}
```

Normal string yazım notasyonunda bazı karakterleri *escape* etmemiz gerekiyor. Örneğin regex kullanırken 4 tane `\` karakteri kodun okunmasını/yazılmasını zorlaştırıyor ve dile `raw string literal` özelliği eklenmiş.

- Bu ayrı bir tür değil sadece yazma kolaylaştırılması için oluşturulmuş.

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

    cout << sizeof(R"(ali)") << "\n";
    cout << sizeof("ali") << "\n";
    cout << boolalpha<< is_same_v<const char[4], decltype(R"(ali)")>;
    auto p = R"("ahmet \mehmet \hasan
    uygar
    pinar
    omer

    )";
    puts(p);
}
```

- Yazının içerisinde `("murat")`yazıldığında sentaks hatası oluyor. Bunun için dile `<delimitir>(string)<delimiter>` şeklinde nasıl kullanıp kullanılamatyacağını programcı belirleyebiliyor.

I/O Manip

queted Fonksiyonu

- `quoted` fonksiyonun geri dönüş türü tamamen derleyiciye bağlı.

```
#include <iomanip>
#include <iostream>
int main()
{
    using namespace std;
    auto str = quoted("ali");
    constexpr auto b = is_same_v<const char[5], decltype(quoted("ali"))>;
//false
```

```
    cout << typeid(string).name() << "\n";
}
```

- Bu fonksiyon 2 tane daha parametre alıyor ve bu parametreler default edilmiş. Bu nasıl kullanılıyor.

```
int main()
{
    using namespace std;
    auto str = quoted("ali");
    cout << quoted(R"("ali" "can" "nur")")<< "\n";
    cout << str << "\n"; // akıma çift tırnaklı olarak veriyor

    ostringstream oss;
    oss << "\"Murat\"";
    cout << "|" << oss.str() << "|" << "\n";
    oss <<  quoted("\"Murat\"");
    cout << "|" << oss.str() << "|" << "\n";
}
```

- BU sadece ters bölü ve çift tırnak olması gerekmiyor. İlk parametrede hangi karakter için alınacağını belirliyoruz. İkinci parametrede hangi karakterin escape karakter olarak kullanılacağını belirliyoruz.

```
int main()
{
    using namespace std;
    ostringstream oss;
    oss << quoted("Murat",'*');
    cout << "|" << oss.str() << "|" << "\n";

    oss << quoted("*Murat* *hakan*", '*');
    cout << "|" << oss.str() << "|" << "\n";

    oss << quoted("*Murat* *hakan*", '*','+');
    cout << "|" << oss.str() << "|" << "\n";
}
```

- istringstream ile okuma işlemi yapılabilir.

```
int main()
{
    using namespace std;
    istringstream iss("\"Murat\" \"Hakan\"");
    string name;
    iss >> quoted(name);
    cout << name << "\n";
}
```

string_view Sınıf Şablonu

- basic_string_view sınıfının char specializationu.
- Bir yazınının gözlemleme pencerisini gösteriyor. Burada string aslında yazı anlamında. Yani ortada eğer bir bytelarda ardışık karakter olarak tutulan yazılar.

`std::sting`, `char str[120]`,`vector<char> cvec`, `std::array<char,10>` gibi yazının ardışık olarak byte stream tutması yeterli. Eğer yazı ilave bir bellek alanına kopyalama yapılıyorsa, bundan kaçabilmek için 2 pointer kullanılarak **yazıyı gözlemlemek** amaçlı kullanabiliriz.

```
class StringView
{
public:
    //string sınıfı içerisinde bulunan const tüm member functionlar burada
bulunursa ve bu fonksyionları const olarak implemente edersek.
private:
    const char *ps; //başlangıç adresi
    //std::size_t len; //uzunluk da tutulabilir.
    const char *pe;
};
```

Bu şekilde artık sadece yaznının gözlemcisi olarak kullanabiliyoruz. Eğer yazıyı sadece okuma amaçlı kullanıyorsak burada sadece bu sınıfı kullanabiliriz.

```
#include <string_view>
//void func(const std::string &s) //burada bir kopyalama olup olmaması
kullanıma bağlı.
void func(std::string_view s)
{
    std::cout << s << "\n";
}

int main()
{
    func("bugün yine hava cok sicak ve ne yazik ki orman yanginlari var");
}
```

- Burada kopyalanan string_view sınıf nesnesin ve bu sınıf sadece 2 pointerı var.

```
void func(std::string_view s)
{

}

int main()
```

```
{
    std::string str(100'000,'a');
    size_t idx{4000};
    size_t n{50'000};
    auto s = str.substr(idx,n); //burada yeni bir string oluşturuluyor.
    //50'000 karakterlik bir bellek alanında allocate ediliyor ve substring
kısmı kopyalanıyor
    //Burada substr içerisinde sadece bir arama işlemi yapmak istiyorsak

    string_view sv = str;
}
```

- Dikkatli olunması gereken yerler var çünkü, yazının kaynağının sahibi biz değiliz. Örneğin string_view sınıfının gözlemlendiği yerlerde `dangling pointer` oluşabilir. Generic programlarda kullanım alanına göre kritik hata senaryoları oluşuturabilir.
- String_view sınıfının *null terminated byte stream* olması gereken değil. Eğer null terminated bekleyen bir API'ye bunu gönderirsek run-time hatası oluşucak.
- Substr'si substr döndürüyor. Gözlemcisi olduğu yazıyı salt okuma erişimli olarak okuyorlar.
- remove-prefix, remove-suffix, ile gözlemlendiği aralığı değiştirmemize yarıyor.

Nasıl Construct Edilebilir?

1. Yolu default construct edebiliriz ve bu string_view sınıf nesnesinin boş olduğunu gösteriyor.
2. c-string constructor (null-terminated) bir yazıyı gözlemleyebiliriz.
3. Data constructor 1 pointer ve 1 tam sayı istiyor.

```
int main()
{
    std::string_view sv{};
    cout << boolalpha << sv.empty() << "\n"; //true
    std::cout << sv.size() << "\n"; //0
    std::cout << sv.data() << "\n"; //nullptr

    char str[] = "necati ergin";
    std::string_view sv2{str};
    sd::string_view sv1{"ali ata bak"};
    std::cout << sv1 << "\n";
    std::cout << sv2 << "\n";
}
```

3.Konu hakkında:

- İkinci parametrede verdiğimiz uzunluk kadar gözlemleyebiliyoruz. Ve eğer data fonksiyonunu çağırırsak bu fonksiyon bize başlangıç adresini döndürücek bu durumda eğer sv2 değişkenini null-terminated byte stream isteyen bir yerde kullanırsak run-time hatası oluşucak.

```
int main()
{
    using namespace std;
    char str[] = "ali ata bak";
    string_view sv1 = str;
    string_view sv2(str,7);
    cout << sv1.length() << "\n";
    cout << sv2.length() << "\n";

    std::array<char,6> ar{'t','a','y','l','a','n'};
    string_view sv3{ar.data(),ar.size()};//burada tanımsız davranış yok ve
taylan yazısınıın gözlemcisi
    cout << sv3.length() << "\n";
    //cout << sv3.data() << "\n"; tanımsız davranış
    //puts(sv3.data()); tanımsız davranış
    //printf("%s\n",sv3.data()); //tanımsız davranış
}
```

String türünden bir constructor'ı yok:

string sınfının `std::basic_string_view<char, std::char_traits<char>>()` ile string_view sınıfına dönüştürülüyor.

```
int main()
{
    using namespace std;
    string s = "ali ata bak";
    string_view sv = s; //burada string sınıfının bir constructor'ı yok.
burada std::strin string_view sınıfına dönüştürülüyor
}
```

İki char * ile de hayata başlatabiliriz

```
int main()
{
    using namespace std;
    char name[] = "ali ata bak";
    string_view sv1{name +2, name +6};
    cout << sv1.length() << "\n";
    cout << sv1 << "\n";
}
```

C++20 ile template range parametreli constructor'da dile eklendi.

```
int main()
{
```

```
    using namespace std;
    string str{"deneme"};
    string_view sv{str.begin(), str.end()};
    cour << "|" << sv << "|\n";
    vector<char> c_vec{'a','l','i',' ','a','t','a',' ','b','a','k'};
    string_view sv2{c_vec.begin(), c_vec.end()};
    cout << "|" << sv2 << "|\n";
}
```

- Burada yazı değiştiğinde string_view sınıf nesnesi de değişmiş yazıyı gözlemliyor

```
int main()
{
    using namespace std;
    char name[] = "ali ata bak";
    string_view sv1{name};

    cout << sv1 << "\n";
    cout << sv1.front() << "\n";
    cout << sv1.back() << "\n";
    name[0] = 'x';
    name[2] = 'y';
    cout << sv1 << "\n";
}
```

- Contructor'ı explicit.
- `nullptr` parametreli contructor'ı delete edilmiş durumda

Dangling Pointer

```
std::string foo()
{

    return "deneme";
}

int main()
{
    using namespace std;
    const string &s = foo(); //life extension.
    string && s = foo(); //life extension oluyor.
    //auto p = foo().c_str(); Burada life-extension olmuyor. Dangling
pointer oluşuyor.
    cout << s << "\n";
}
```

- Eğer bu kodu oluşturursak burada tanımsız davranış oluşuyor. String sınıfı dönüş fonksiyonuunu string_view nesnesine atayamayız

```
std::string foo(){    return "deneme";}

int main()
{
    using namespace std;
/*  TANIMSIZ DAVRANIŞ
   string_view  sv = foo();
    cout << sv << "\n";
*/
}
```

- Eğer burada Person sınıfının hayatı biterse dangling pointer oluşma ihtimali çok yüksek.

```
class Person
{
public:
    Person(const std::string &name) : m_name{name} {}
    void print()const
    {
        std::cout << m_name << "\n";
    }
    std::string_view get_name()const
    {
        return m_name;
    }
private:
    std::string m_name;
};

Person create_person()
{
    return Person{"deneme person"};
}

int main()
{
    using namespace std;
    Person p{"ali"};
    p.print();
    //auto sv = create_person().print(); //Tanımsız Davranış
    //string_view sv = p.get_name(); //Tanımsız Davranış
    cout << sv << "\n";
}
```

- Örneğin string sınıfında bir re-allocation olması da bu pointerların dangling olmasına sebep olabilir.

```
int main()
{
```

```
    using namespace std;
    string str(10,'a');
    string_view sv {str};
    str.append(100,'b');
    //cout << sv << "\n"; DANGLING POINTER OLUŞUYOR.
}-
```

Bazen dangling pointer'ı saptamak çok kolay olmayabilir:

- Aşağıdaki kod içerisinde `operator+` fonksiyonun da herhangi bir problem yok. Fakat concat fonksiyonunu çağırırsak run-time hatası oluşuyor.
- Template parametresi std::string_view oluyor çıkarım yapılıyor `s1 + s2` geri dönüş değeri string fakat `concat` fonksiyonun geri dönüşü string_view ve burada life extension olmuyor.
- Eğer auto ile yapılsaydı bu hata oluşmazdı çünkü burada çıkarım string olarak yapılıyor.

```
template<tyepname T>
T concat(const T &s1, const T &s2)
//auto concat(const T &s1, const T &s2)
{
    return s1 + s2;
}

std::string operator+(std::string_view sv1, std::string_view sv2)
{
    return std::string{sv1} + std::string{sv2};
}

int main()
{
    std::string_view sv = "Merhaba";
    auto val = concat(sv,sv);
    std::cout << val << "\n";
}
```

> auto return type ve retailing auto return type bir birinden farklı

```
template<typename T>
auto foo(T x)
{
    return x+x;
}

template<typename T>
auto foo(T x) -> decltye(x+x) //yukarıdaki ile aynı
{
    return x+x;
}

template<typename T>
```

```
auto func(T x) -> decltype(x.foo())
{
    return x+x; //bu ifadeden artık bir çıkarım yapılmıyor.
}
```

- Aşağıdaki kullanımda bir hata

```
std::string_view foo(std::string s)
{
    return s;
}
```

string_view Sınıfının Üye Fonksiyonları

- Bir çok üye fonksiyonu constexpr

```
int main()
{
    constexpr std::string_view sv{"deneme"};
    constexpr auto len = sv.length();
    constexpr auto cs = sv.front();
    constexpr auto ce = sv.end();

    constexpr auto iter_begin = sv.begin();
    constexpr auto iter_end = sv.end();

    constexpr auto idx = sv.find('e');
    constexpr auto idx_2 = sv.find_first_of("nem");
}
```

- remove_prefix ve remove_suffix ile string_view nesnesinin gözlemlediği aralığı değiştirebiliyoruz. Data fonksiyonu ile verdiğimiz noktadan null-terminated kullanıldığı yere kadar yazıyı bastırıyor.

```
int main()
{
    using namespace std;
    string_view sv{"ali ata bak"};
    cout << "|"<< sv <<"|"<< "\n";
    cout << sv.size() << "\n";
    cout << sv.data() << "\n";
    sv.remove_prefix(4);
    cout << "|"<< sv <<"|"<< "\n";
    cout << sv.size() << "\n";
    cout << sv.data() << "\n";
    sv.remove_suffix(2);
    cout << "|"<< sv <<"|"<< "\n";
    cout << sv.size() << "\n";
```

```
    cout << sv.data() << "\n";
}
```

- Yazının başındaki boşlukları gözlemlemek istemiyoruz bunu:

```
int main()
{
    std::string str {" baaaaaaabaaaaaaaalallala"};
    std::string_view sv{str};
    sv.remove_prefix(std::min(sv.find_first_not_of(' '), sv.size()));
    std::cout << sv << "\n";
    std::cout << str << "\n";
}
```

şeklinde yapabiliriz.

```
void foo(std::string)
{
    std::cout << "string\n";
}
void foo(std::string_view)
{
    std::cout << "string_view\n";
}
void foo(const char *)
{
    std::cout << "const char *\n";
}

int main()
{
    // foo("ali ata bak"); //Sentaks hatası ambugity oluşuyor.
    foo("ali ata bak"s);
    foo("ali ata bak"sv);
    foo("ali ata bak"); //const char * parametreli overloadu ekledikten
sonra artık o çağırılıyor.
}
```

- Programcıların bunu tercih ettiği bir yerden veri elemanlarını initialize etmek için kullanılıyor.

```
class Myclass
{
public:
    //Myclass(std::string & s) : ms{s} {}
    //Myclass(std::string_view sv) : ms{sv} {}
    Myclass(std::string s) : ms{std::move(s)} {}
```

```
private:
    std::string ms;
};
```


alt text

Eğer input parametreleri gönderirsek ve small string optimization yapılmıyorsa, parametrenin ne olduğuna göre değişiyor.

- C++20 ile starts_with ve ends_with fonksiyonları eklendi. Yazının başında ve sonunda belirli bir karakter olup olmasdığnı kontrol ediyor ve bu fonksiyonlar bool değer döndürüyor

```
int main()
{s
    using namespace std;
    string_view sv{"ali ata bak"};
    cout << boolalpha << sv.starts_with('a') << "\n";
    cout << boolalpha << sv.ends_with('k') << "\n";
}
```

std::optional

- C++17 ile dile eklendi. Vocabulary type türünden biri.

Bu sınıf türünden nesnenin ya bir değeri var ya da boş. Değeri olması kadar değeri olmaması da çok doğal. Normlde bu durumu genelde başka araçlar ile test ediyoruz. Bu araçların kullanıldığı her yerde kullanabiliriz.

- 2 fonksiyon ile değere sahip mi değil mi sorgulamasını yapabiliyoruz.
- - default construct edilmiş bir optional nesnesi boş durumda.

```
#include <optional>

template<typenameT >
class Optional{};
int main()
{
    using namespace std;
    optional<int> ox;
    optional<string> os;
    optional<vetor<int>> ov;
    optional<int> oy{12};
    if(ox.operator bool()){ cout << "ox has value\n";}
    else
    {
        cout << "ox has no value\n";
    }
    if(os){cout << "os has value\n";    }
    else
```

```
    {
        cout << "os has no value\n";
    }
    cout << boolalpha << ox.has_value()? "dolu" : "bos" << "\n";
    cout << boolalpha << oy.has_value()? "dolu" : "bos" << "\n";
}
```

- Boş olarak başlatmak için `nullopt_t` türünden bir değişken kullanılabiliriz. Bu `nullopt` değer ile hayata başlatmak istiyorsak bunu kullanabiliriz.

```
int main()
{
    using namespace std;
    optional<int> ox = nullopt;
}
```

optional Sınıfının Tutuğu Değerin Erişimi

İçerik operatör fonksiyonu ile erişebiliriz. Bu fonksiyon bir referans döndürüyor. Benzer şekilde `operator->` fonksiyonu da kullanılabilir.

- Bir pointer sınıf olmamasına rağmen, bu sınıf bir pointer sınıfıymış gibi overload edilmiş.

```
int main()
{
    using namespace std;
    optional<string> ox = "Deneme test";
    cout << *ox << "\n";
    *ox = "ali ata bak";
    cout << *ox << "\n";
    cout << ox.value() << "\n";
}
```

- CTAD ile optional sınıfları kullanılabiliyor.

```
int main()
{
    optional ox = "ali ata bak";
}
```

- Eğer içerik ve ok operatör fonksiyonu kullanıldığında optional nesnesi boş ise exception throw **etmiyor**.
- Ama eğer `value()` fonksiyonunu kullanarak boş bir değere erişmeye çalışırsak burada exception throw ediliyor.

- `std::bad_optional_access` türünden bir exception throw ediliyor.

```
int main()
{
    using namespace std;
    optional<string> ox;
    //cout << *ox << "\n"; //tanımsız davranış
    try {
        cout << ox.value() << "\n"; //exception throw ediyor.
    } catch (const std::bad_optional_access &ex)
    {
        cout << ex.what() << "\n";
    }
}
```