10.Hafta

İçindekiler

- 10.Hafta
  - İçindekiler
  - Constexpr Fonksiyonlar
    - Consteval fonksiyonlar
    - Constinit Fonksiyonlar
  - Attributelar
    - [[nodiscard]] attribute
  - Concpetler
    - Conceptlerin Tanımlanması
      - Requires Clause
        - Conceptler ile birlikte kullanımı
      - Requires Expression
      - Subsumption

Constexpr Fonksiyonlar

Consteval fonksiyonlar

- Bu fonksiyona yapılan çağırıların hepsi compile-time içerisinde değerlendirilmek zorunda, compile-time context'i içerisinde kullanılmaması sentaks hatası oluşturuyor.

Constinit Fonksiyonlar

- Bu fonksiyonun compile-time init şekilde olması daha iyi olabilirdi çünkü buradaki context compile-time init'e karşılık geliyor. Statik veri değişkenlerin initialization sonucu static init fail olabilir.
- `constinit` keyword'ü ile derleyiciye değişkenin constant initialize edilmesini garanti edebiliyor.

```
constinit int x = 0;
//constinii = constexpr - conts;
```

- Bu ifadelere sabit bir ifade ile değer vermemiz zorunlu.
- Ama burada değişkenin değerini değiştirebildiğimiz için `constinit = constexpr - const`

```
constexpr int foo(int x)
{
    return x*6;
}

constexpr int x = foo(5);
constinit int y = foo(6);
```

```
int main()
{
    y++;
}
```

- Y değişkenini başka bir sınıf içerisinde kullanıyor olsa bile `constinit` anahtar sözcüğü ile bu değişkenin compile-time'da initialize ediyoruz.

Hangi değişkenleri `constinit` ile tanımlamalayabiliriz?

- Global değişkenleri
- static yerel değişkenleri
- sınıfların static veri elemanlarını

```
constinit int x = 10; // OK
int fo()
{
    static constinit int z = 30; // OK
}

class Myclass
{
public:
    static constinit int x = 10; // OK
};
```

```
constexpr std::array<int, 4> get_array()
{
    return {1, 2, 3, 4};
}

constinit auto g_ar4  = gee_array(); // OK

int main()
{
    for(auto i : g_ar4)
    {
        std::cout << i << "\n";
    }

    g_ar4[1]++;
    g_ar4[2] += 100;
    for(auto i : g_ar4)
    {
        std::cout << i << "\n";
    }
}
```

- Templateler içinde kullanabiliriz.

```
template <std::size_t N>
constinit std::array<int, N> get_array()
{
    std::array<int, N> ar;
    for(std::size_t i = 0; i < N; i++)
    {
        ar[i] = i;
    }
    return ar;
}

constinit auto g_ar4 = get_array<4>();

int main()
{
    for(auto i : g_ar4)
    {
        std::cout << i << "\n";
    }
    for_each(begin(g_ar4), end(g_ar4), [](int& x){ ++x; });
    for(auto i : g_ar4)
    {
        std::cout << i << "\n";
    }
}
```

- `constinit` ile `const` kullanabiliriz fakat bunun yerine constexpr kullanabiliriz.

Attributelar

- Derleyiciyi yönlendirmek için ve koda okuyana bilgi vermek için kullanılabilir. Derleyiciler daha önceden extension mekanizması ile bunu sağlıyabiliyordu.
- Dile eklenmesi ile derleyiciden bağımsız olarak bir standart yapı eklendi.
- Programcıyı korumaya yönelik.
- Bazen kodun yanlış yazılmasını engelemek için kullanıldığı gibi bazen de spesifik durumlarda bu yapının bilinçli olarak kullanıldığını belirtmek için kullanılabiliyor.
- Derleyiciler veya programcılar kendi attributelarını oluşturulabiliyor.

Sentaksı `[[attribute]]`

[[nodiscard]] attribute

Bir öğenin discard edilmesinin logic bir hata olduğunu belirtmek için kullanılıyor. Derleyiciyi de uyarı mesajı vermeye teşvik ediyor.

- Fonksiyonun geri dönüş değeri tamamlayıcı bir değer değilse
- Çağıran kodun discard edebileceği bir hata kodu değilse ve bu bir risk içeriyorsa.
- C++17 ile geldi.

- C++20 ile
- - string literal opsiyonel `[[nodiscard("reason")]] function()` duruma göre reason'ı uyarı mesajı olarak yazdırıyor.

```
[[nodiscard]] int foo(int);
int main()
{
    foo(10); // warning
}
```

- Eğer geri dönüş değerini bilerek ve isteyerek kullanmıyorsak bunu `void` türüne cast edebiliriz.
- Türleri de niteleyebiliyoruz
- - Eğer geri dönüş değeri referans veya pointer ise burada geri dönüş değerinin discard edilmesi bir warning oluşturmayabilir.

```
class  [[nodiscard]] Myclass{};
Myclass foo();
Myclass bar();
int main()
{
    auto m = foo();
    bar(); // compiler will result a warning
}
```

- Sınıfların constructorları da `[[nodiscard]]` ile nitelenebilir.

```
class Myclass
{
public:
    [[nodiscard]] Myclass(int x);
};
int main()
{
    Myclass(10); // warning
    static_cast<Myclass>(12); // warning
}
```

- Standart kütüphanede de bazı fonksiyonlar C++20 ile bazı fonksiyonlar `nodiscard` ile bildirilmiş olabilir.
- - Örneğin Vector sınıfının empty fonksiyonu.

```
int main()
{
```

```
    std::vector vec{2,5,7,9};
    vec.size();
}
```

Concpetler

C++20 ile yapılan majör bir eklenti. Olabilecek tek eleştiri bu özelliğin çok geç gelmiş olması.

Template kodları constraint etmek için kullanılıyor. Template kodların derleyiciye kod yazdıran kodlar.

- Function template: ürünü bir fonksiyon ve buna bu fonksiyon türünün specialization'ı oluyor.
- Class template: ürünü bir sınıf ve bu sınıfa sınıf template'inin bir specialization'ı oluyor.
- Variable template: ürünü bir variable
- Alias template: Ürünü bir tür.

Burada şablonlar oluşturulurken kısıtlamalar olabilir. Örneğin bir şablon sadece tam sayılar için uygun olabilir.

- Şablonda bu kısıtlamayı anlatan/bildiren bir veri yok.
- Template'e eskiden yanlış bir değişken verildiğinde çok büyük bir hata mesajı veriyor.
- Compile time içerisinde derleyicinin üzerindeki yükü azaltmak.
- Bir veya birden fazla constraint sağlanmadığında hangi constraint'in sağlanmadığı hakkında bir hata mesajı verilsin.
- Ve buradaki constraint'lerin bir kısmı standart hale getirilsin, bunlara **concept** diyelim.
- Kendi kısıtlamalarımızı oluşturabilmemiz yanında standart'tan gelen kısıtlmaları da kullanabilelim.
- Kod seçiminde de kullanılabiliyor.

Comceptlerin dile eklemesi ile birlikte ranges kütüphanesi iş görüyor.

```
template <std::integral T>
void func(T x)
{
    std::cout <<"tam sayi " <<  x << "\n";
}
template<std::floating_point T>
void func(T x)
{
    std::cout << x << "\n";
}

void foo(std::integral auto x)
{
    std::cout << "tam sayi " << x << "\n";
}

int main()
{
    func(23L);
    func(3.4F);
}
```

- Concept'in kendisi hem başlık dosyasının ismi hem de concept'in ismi olabilir. Templateler ile de arttı. Concept template kategorisi de dil eklendi ve bu bize constraint'lerin isimlendirilmesi konusunda da fayda sağlıyor.

Concept öncesinde

- Sadece tam sayı türleri ile kısıtlamak istiyorsak

```
template <typename T, std::enable_if_t<std::is_integral_v<T>>* =nullptr >
void foo(T);

template <typename T>
std::enable_if_t<std::is_integral_v<T>öT> bar(T);

template <typename T>
void baz(T, std::enable_if_t<std::is_integral_v<T>>*p = nullptr);

int main()
{
    foo(10);
    bar(20);
    baz(30);
    foo(3.4); // Sentaks hatası oluşacaktı
}
```

Bir template'in kısıtlaması var ve bu kısıtlamaya uyulmasın

```
template <typename T>
void print(const T& x)
{
    std::cout << x << "\n";
}
int main()
{
    print(10);
    print(3.4);
    print("ali"s);

    std::vector vec{1,2,3,4,5};
    //print(vec); vector için ostream'e insert edecek bir fonksiyon yok.
}
```

Artık bir template'i birden fazla araç kullanarak constrait edebiliyoruz.

```
template <typename T>
void foo(const T& x)
{
    int t = x;
    std::cout << x << "\n";
}

int main()
{
   //foo(std::string{"ali"}); sentaks hatası mesajı çok uzun ve karışık
oluyor.
}
```

auto keyword'ü ile template olarak yazabiliyoruz.

```
temaplta <typename T>
class Myclass
{
public:
    void funct(auto x);
};
```

Conceptlerin Tanımlanması

Requires Clause

- 2 farklı şekilde tanımlanabilir.
- - prefix
- - trailing

```
requires sizeof(T) > 2
```

```
template<typename T>
requires (sizeof(T) > 2)  void foo(T x)
 void foo(T x) requires (sizeof(T) > 2) //parametre değişkenin ismini
buurada kullanabiliyoruz.
//her iki yerde de kullanabiliyoruz.
{
}
int main()
{
    //foo('A'); //sentaks hatası.
}
```

- Hem başlangıcında hem de sonunda kullanabiliriz.

```
template <typename T>
requires std::is_integral_v<T> void foo(T x) requires std::is_signed_v<T>
//requires std::is_integral_v<T> && std::is_signed_v<T> void foo(T x)
şeklinde de yazbilirdiik.
{
    std::cout << x << "\n";
}
{
    std::cout << x << "\n";
}
int main()
{
    foo(10);
    foo(3.4); //sentaks hatası
}
```

- Logic değil operatörünü kullanırsak requires clause'u parantez içerisine almalıyız.

Conceptler ile birlikte kullanımı

- Yukarıdaki constraintleri isimlendiriyoruz ve tekrar tekrar yazma zahmetinden kurtulabiliyoruz.
- Doğrudan requires clause'da kullanabiliyoruz. Hem prefix hem de trailing olarak kullanabiliyoruz.

```
template <typename T>
concept Integral = std::is_integral_v<T>;

template <typename T>
concept SignedIntegral = Integral<T> && std::is_signed_v<T>;

//constrained template parameter.
template <Integral T, Integral U>
class Myclass{};
```

```
template <typename T>
concept as_int = std::integral<T> || std::is_covertible_v<T, int>;

template <as_int T>
class Myclass{};

struct Cans{};

int main()
{
    Myclass<int> m1;
    //Myclass<Cans> m2; Sentaks hatası
}
```

- auto'nun kullanıldığı yerde bunu kullanabiliyoruz.

```
template <typename T>
concept as_int = std::integral<T> || std::is_covertible_v<T, int>;

void func(as_int auto x)
{
    std::cout << x << "\n";
}
```

- Sadece fonksiyon şablonlarında değil member fonksiyon template'lerde de kullanabiliyoruz.

```
class Myclass
{
public:
    void foo(std::integral auto);
};
```

- auto'nun kullanıldığı çoğu yerde de kullanabiliyoruz.

```
std::integral auto bar(int x)
{
    return x*x;
// return x*3.4; sentaks hatası bu ifadenin türü integral türünü
karşılamıyor.
}
int foo()
int main()
{
    std::integral auto x = foo(); //çıkarım yapılan türün sadece integral
olması durumunda bu kod  geçerli oluyor.

    vector<int> ivec(100);
    for(std::integral auto x : ivec)
        std::cout << x << "\n";
}
```

- Sabit ifadesi ile de kullanabiliyoruz. Çok daha karmaşık bir concept ile de kullanabilioyruz.

```
template <typename T>
void func(T x)
{
    if constexpr(std::integral<T>)
    {
        std::cout << x << "\n";
    }
```

```
    else if constexpr(std::floating_point<T>)
    {
        std::cout << "floating pıint \n";
    }
    else
    {
        std::cout << "other type\n";
    }
}
int main()
{
    func(10);
    func(3.4);
    func("ali"s);
}
```

```
template <typename T>
concept additive = requires(T xö T y){
    x+y; // T'türünden 2 nesnse toplandığında geçerli olucak
    x-y; // T'türünden 2 nesnse çıkartıldığında geçerli olucak
};

template <typename T>
void func(T x)
{
    if constexpr(additive<T>)
    {
        std::cout << x << "\n";
    }
    else
    {
        std::cout << "not additive\n";
    }
}

int main()
{   int x = 10;
    int *p = &x;
    func(x);
    funct(p);
}
```

- constexpr fonksiyonları da kullanabiliyoruz.

```
template <std::size_t N>
requires (std::has_single_bits(N)) && (N>32)
class Myclass{};

int main()
```

```
{
    //Myclass<5> m1;
    //Myclass<45> m1;
    Myclass<64> m1;
};
```

. Isprime örneği

```
constexpr bool isprime(int x)
{
    if(x<2)
        return false;
    if (x %2 == 0) return x == 2;
    if (x %3 == 0) return x == 3;
    if (x %5 == 0) return x == 5;

    for(int i{7}; i *i <= x ; i+2)
        if(x%i == 0)
            return false;
    return true;
}

template <int N>
requires isprime(N)
class Myclass {};

template <int N>
concept Prime = isprime(N);

template <int N>
requires Prime<N>
void func();

int main()
{
    Myclass <18> x;//sentaks hatası
    func<19>();
//func<6>(); sentaks hatası
}
```

Requires Expression

- Boolean bir değer üretiyor.
- Blok içerisinde birden fazla constraint tanımlayabilmek için değişken yapabiliyoruz.
- Bu ifadeler yürütülmüyorlar.
- Birden fazla kategöride bir şeyler belirtebiliyoruz.
- - simple requirements: ifadenin geçerli olup olmadığını test ediyoruz.

```
template <typename T>
concept Den = requires(T x) {
    x++; //örneğin int türü ++ operatörünü destekliyor ve geçerli
};

template <typename T>
concept Den1 = requires(T x) {
    x == nullptr;
};

template <typename T>
concept Den2 = requires(T x) {
    std::is_integral_v<T>;
};

struct A{};
int main()
{
    static_assert(Den<int>);
    static_assert(Den<int*>);
    static_assert(Den<std::vector<int>::iterator>);
    //static_assert(Den<A>);
    constexpr auto b = Den1<std::unique_ptr<int>>; //true uniuqe_ptr
nullptr ile fonksiyon ile karşılaştırılabiliyor
    constexpr auto c = Den2<double> ; //true
}
```

- type Requirements:
- - Derleyici bir türün var ve geçerli olduğunu sınamak zorunda

```
template <typename T>
concept Den = requires(T x) {
    typename T::value_type;
};



int main()
{
    constexpr auto b = Den<int>;
    constexpr auto b1 = Den<std::vector<int>>;
}
```

18_27_08_2023

- type parametresinin veya non-type parametresinin requires clause içerisinde geçmesi zorunlu değil.

```
template <typename T>
    requires true
class Myclass{};

int main()
{
    Myclass<int> m1;
}
```

- standart kütüphanede de conceptler var, bunları kullanmak için `concepts` başlık dosyasını eklememiz gerekiyor.

```
#include <concepts>
template <typename T>
requires std::is_pointer_v<T> || std::same_as<T, std::nullptr_t>
void foo(T);

int main()
{
    int ival {10};
    foo(nullptr);
    foo(&ival);
}
```

- T türünden geçici bir nesnse oluşturarak kullanmak istersek o türün default constructor'ı olması lazım bu durumda sentaks hatası vermemesi için declval aşağıdaki gibi çağırıldığında default constructor'ı olmöasına gerek yok.
- optional nesnesi de dereferenc edilebiliyor ve bu nesne dereference edildiğinde çağırılan fonksiyonun geri dönüş değeri int& türünden bir nesne.

```
template <typename T>
 requires std::integral<std::remove_reference_t<decltype(*std::declval<T>
())>>
/*
İçeriden dışarıya doğru
- decltype ile T türünden bir nesne oluşturuyoruz.
- Referans türüyose referanslığından kurtarıp, eğer T'türünden bir nesne
referanslıktan sıyrılıdığında bir integral türüne dönüşebiliyorsa bu şablonu
kullanabiliriz.
*/
void foo(T);

int main()
{
    int x{};
    double d{};
    std::optional opt{10};
    //foo(&x);
```

```
    //foo(&d);
    //foo(&opt);

}
```

- Aşağıdaki kodda veya ifadesi ile oluşturulan ifadenin hepsinin geçerli bir ifade olması gerekiyor.l

```
template <typename T, typename U>
concept cden = requires(T x, U y){
    //*y > x; //2.tamplate parametresi türünden bir nesne dereferans
edildiğinde o 1. template parametresi türünden bir nesne ile
karşılaştırılabiliyor olması gerekiyor.
    x.foo() || y.bar(); //x.foo() veya y.bar() çağrılabiliyor olması
gerekiyor anlamına gelmiyor!!!!!!!!!!!!
};

struct A{void foo();}
struct B{}
int main()
{
    constexpr auto b = cden<A,B>;
}
```

- Eğer veya şeklinde yapmak istiyorsak

```
template <typename T, typename U>
concept cden = requires(T x){
    x.foo();
}
||
requires(T x){
    x.bar();
};
```

3 türde requirement ifadesi var:

- simple requirements
- type requirements
- compound requirements

```
template <typename T>
concept cden = requires(T x){
    x.foo();
};
//T türünün çağırılabilir bir foo fonksiyonu olması lazım.
```

- type requirement için
- T'nin print fonksiyonun çağrılabilir olması, first ve secont type'larının olması gerekiyor.

```
template <typename T>
concept cden = requires(T x){
    x.print();
    typename T::value_type::first_type;
    typename T::value_type::second_type;
};
```

```
template <typename T>
concept cden = requires(typename T::value_type x) {std::cout << x;};

using mytype = std::vector<int>;
int main()
{
    static_ssert(cden<mytype>);
}
```

- Compound requirements:
- Hem çağırılabillir foo fonksiyonu olucak hem de bu fonksiyon exception throw etmeyecek bir concept: Burada sentaks:

```
template <typename T>
concept cden = requires(T x){
    {x.foo()}; // bu küme parantezsiz yazmak ile aynı
    {x.bar()}noexcept;

};
```

- Aşağıdaki fonksiyonun geri dönüş değerinin int olmasını istiyorsak:
- - same_as 2 parametreli bir concept ve derleyici burada `same_as</*a*/,int>`

```
template <typename T>
concept cden = requires(T x){

    {x.foo()}noexcept -> std::same_as<int>; //fonksiyonun geri dönüş
değeriinin int olmasını istiyoruz.
}
```

```
template <typename T>
concept Pointer = std::is_pointer_v<T>;
```

```
template <typename T>
concept Reference = std::is_reference_v<T>;

template <typename T>
concept Den = requires{
    requires Pointer<T> || Reference<T>;
};
```

- `common_type<...>` variadic parametre alan bir tür ve argümanlar arasında bir tür çıkarımı yapacak.

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

int main()
{
    using namespace std;
    common_type_t<int, long> x; //int
    common_type_t<int, Myclass> y; //Myclass
}
```

- Bunu eğer concept içerisinde kullanmak istersek:

```
template <typename T, typename U>
concept Den = requires(T x, U y){
    typename std::common_type_t<T,U>; //böyle bir tür oluşabilsin diye
talep ediyoruz
};

struct A{};

int main()
{
    static_assert(Den<int, long>);
    static_assert(Den<const char *, std::string>);
    static_assert(Den<int, A>);
}
```

- `common_type` ın concept versiyonu `common_with`.

```
template <typename T, typename U>
concept Den = requires(T x, U y){
    {x+y}->common_with<U>;
};
```

- Compile time'da `std::hash`in bizim türümüz için specialization'ı olmasını istiyoruz.

```
#include <functional>
tepmplate <typename T>
void foo(T y)
{
    std::hash<T> x;
    ////
}

struct Dene{};

template <>
struct std::hash<Dene>
{
    std::size_t operator()(const Dene&) const
    {
        return 1;
    }
};

int main()
{
    foo(34);
    //foo(Dene{}); sentaks hatası. Eğer Dene türü için hash
specialization'ı olmasını istiyorsak bunu yapmamız gerekiyor.
    foo(Dene{});
}
/// Bunun için bir concept oluşturabiliriz. Aşağıdaki concept yanlış çünkü
bu böyle bir tür oluşturulabilir mi demek.
template <typename T>
concept shash = requires(T x){
    //typename std::hash<T>; bunun yerine
        std::hash<T>{};
};

template <typename T>
requires shash<T>
void foo(T x) // ya da void foo(shash auto)
{}
```

- Bir conceptin specialization'ı bir boolean sabit oluyor.
- abbreviated sentaks ile de kullanılabiliyor:

```
template <typename T>
concept HasFooBar = requires (T x){
    {x.foo()}noexcept->std::convertible_to<bool>;
    {x.bar()}noexcept->std::same_as<bool>;
};
```

```
HasFooBar auto foo(HasFooBar auto ); //Fonksiyonun geri dönüşü ve
parametresi aynı concept'i sağlayan türler.
// Türleri aynı olmak zorunda değil.

template <typename T>
requires HasFooBar<T>
class Myclass{};

void func2(HasFooBar auto x );
```

- lambda ifadelerinde de kullanabiliyoruz.

```
int main()
{
    auto f = [](std::integral auto )-> std::integral auto {return 1.2;}

}
```

- compound ifadelerde türü belirlemek için kullanılabilir.

```
template <typename T>
concept Den = requires(T x){
    {++x}noexcept->std::same_as<int>;
};
```

- type veya non-type parametreler ile de kullanılabilir.,

```
template <typename T>
void func(T) = delete;

void func(int);
int main()
{
    func(10);
    func(3.4);
}
```

- SFINAE ile aynı işi yapıyor.

```
#include <type_traits>
template <typename T, typename std::enable_if_t<std::is_same_v<T,int>,
int>> * = nullptr>
void func(T);
```

- doğrudan concept ile de yapabiliriz.

```
template <typename T>
requires std::same_as<T, int>
void func(T);
```

Standart'da çok sayıda concept var.

```
template <typename T, typename U, typename W>
concept Denable = requires (T t, U u, W w){};

void foo(Denable<int,double,>)
```

- Genellikle type traitsler ile concept isimleri birbirlerine benziyor.

```
void func(std::invocable<int> auto f)
{
    f(10);
}
///
template <typename F>
requires std::invocable<F,int>
void func(F fn)
{
    fn(10);
}
///
template <std::invocable<int> F >
void func1(F fn)
{
    fn(10),
}
// Bu 3 fonksiyon da aynı anlama geliyor.

int main()
```

Concept	Constraint
integral	Integral type
signed_integral	Signed integral type
unsigned_integral	Unsigned integral type
floating_point	Floating-point type
movable	Supports move initialization/assignment and swaps
copyable	Supports move and copy initialization/assignment and swaps
semiregular	Supports default initialization, copies, moves, and swaps
regular	Supports default initialization, copies, moves, swaps, and equality comparisons
same_as	Same types
convertible_to	Type convertible to another type
derived_from	Type derived from another type
constructible_from	Type constructible from others types
assignable_from	Type assignable from another type
swappable_with	Type swappable with another type
common_with	Two types have a common type
common_reference_with	Two types have a common reference type
equality_comparable	Type supports checks for equality
equality_comparable_with	Can check two types for equality
totally_ordered	Types support a strict weak ordering
totally_ordered_with	Can check two types for strict weak ordering
three_way_comparable	Can apply all comparison operators (including the operator <=>)
three_way_comparable_with	Can compare two types with all comparison operators (including <=>)
invocable	Type is a callable for specified arguments
regular_invocable	Type is a callable for specified arguments (no modifications)
predicate	Type is a predicate (callable that returns a Boolean value)
relation	A callable type defines a relationship between two types
equivalence_relation	A callable type defines an equality relationship between two types
strict_weak_order	A callable type defines an ordering relationship between two types
uniform_random_bit_generator	A callable type can be used as a random number generator

```
void func (std::regular auto);
struct Den{};

int main()
{
    func(12);
    func(std::string{"test"});
//    func(Den{}); //equalitiy comparable sağlanmıyor. bool operator==()
eklenirse bu sorun ortadan kalkar
}
```

- Jonathan bockara'nın tekniği:

```
void aggregateAndDisplay(std::map<int, std::string> const& source,
std::map<int, std::string> const & destination)
{
    auto aggregatedMap = destination;
    for(auto const & sourceEntry : source)
    {
        auto destinationPosition = aggregatedMap.find(SourceEntry.first,
SourceEntry.second );
        if(destinationPosition == aggregatedMap.end())
        {
            aggregatedMap.insert(std::make_pair(sourceEntry.first,
sourceEntry.second));
        }
```

```
        else
        {
            aggregatedMap[sourceEntry.first] = sourceEntry.second + " or "
+ destinationPosition->second;
        }
    }
    for(auto const & entry : aggregatedMap)
    {
        std::cout << "Available tranlations for " << entry.first <<
entry.second << "\n";
    }
}

//
auto foo(std::map<int, std::string> const& source, std::map<int,
std::string> const & destination)
    {
        auto aggregatedMap = destination;
        for(auto const & sourceEntry : source)
        {
            auto destinationPosition =
aggregatedMap.find(SourceEntry.first, SourceEntry.second );
            if(destinationPosition == aggregatedMap.end())
            {
                aggregatedMap.insert(std::make_pair(sourceEntry.first,
sourceEntry.second));
            }
            else
            {
                aggregatedMap[sourceEntry.first] = sourceEntry.second + "
or " + destinationPosition->second;
            }
        }
        return agggregatedMap;
}

void aggregateAndDisplay(std::map<int, std::string> const& source,
std::map<int, std::string> const & destination)
{
    auto aggregatedMap = foo(source, destination);
    for(auto const & entry : aggregatedMap) //aggregateMap üstteki
fonksiyonun geri dönüş değeri
    {
        std::cout << "Available tranlations for " << entry.first <<
entry.second << "\n";
    }
}
```

Subsumption

- Daha fazla kısıtlayıcı olan konspet daha gevşek olanı içine alıyor.

```
void foo(std::integral auto)
{
    std::cout << "integral\n";
}
void foo(std::unsigned_integral auto )
{
       std::cout << "unsigende integral\n";
}
int main()
{

}
```

```
#include <concepts>
#include <iostream>

template <typename T>
concept den = requires (T x)
{
    x.foo()
};

template <typename T>
concept arg = den<T> && requires(T x)
{
    x.bar();
}

void func(den auto)
{
    std::cout << "integral\n";
}
void func(arg auto)
{
    std::cout << "arg auto \n";
}

struct A {
    void foo();

};

struct B{
    void foo();
    void bar();
};

int main()
{
    func(A{});
```

```
    func(B{});
}
```