Tekrar: · Conceptler, isimlendirilmiş constraintler ile ilgili;

· Templateların uygun olmayan specialization'a syntax hatası verdirtme, uygun olanı seçtirme.

```cpp
template <typename T, typename U>
requires std::convertible_to<T, U>
void foo(T, U);

int main()
{
    int x{};

    //foo(x, &x);
    foo(x, 3.4);
}
```

↓ conceptin kendisi,

```cpp
template <typename T, typename U>
requires std::is_convertible_v<T, U>
void foo(T, U);

int main()
{
    int x{};

    //foo(x, &x);
    foo(x, 3.4);
}
```

type traits kullanımı!

```cpp
template <typename T, typename U>
concept cnec = requires(T x, U p) {
    *p > x;
};
```

→ dereference edince karşılaştırılabilsin

```cpp
concept has_foo = requires(T x) {
    x.foo();
};

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


template <typename T, typename U>
requires has_foo<T> || has_bar<U>
class Myclass;


struct A {
    int foo();
};


struct B {
    //int bar();
};

int main()
{
    constexpr auto b = cnec<A, B>;
}
```

→ Compound Requirement:

```cpp
concept cnec = requires (T x) {

    {x.foo()}noexcept; //compound requirement
};

struct A {};

struct B {
    void foo();
};

struct C {
    void foo()noexcept;
};


int main()
{
    boolalpha(std::cout);
    std::cout << cnec<A> << '\n'; // → false
    std::cout << cnec<B> << '\n'; // → false
    std::cout << cnec<C> << '\n'; // → true

}
```

küme parantezi içinde

→ Nested Requires:

Requires clause, requires expr olarak kullanılabilir.

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

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

template <typename T>
concept PointerOrReference = Pointer<T> || Reference<T>;
```

↓ Nested olarak yazarsak:

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

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

template <typename T>
concept Nec = requires {

    requires Pointer<T> || Reference<T>;
}
```

* Mülakat Sorusu: Sadece kendi hasher'ı olan bir map/set template'i

```cpp
template <typename T>
concept shash = requires {
    std::hash<T>{};
};


void foo(shash auto)
{
}


struct Neco {};

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

int main()
{
    foo(Neco{});
}
```

→ Standart kütüphanenin sundüğü conceptler burada!

→ Josuttis Cpp 20 ve kendin kod yaz ← Marc Gregoire, kitabı: Professional C++

```cpp
int main()
{
    //geri dönüş değeri türü
    //trailing return type ile
    // constraint edi,lebilir mi

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


}
```

* Conceptler Genel Olarak Nerede Kullanılır:
Hangi Bağlamlarda Kullanılır:

1. Template Constraint edilirken boolean ifade gerekten her yerde

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

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

};

template <typename T>
requires HasFooBar<T>
T f1(T x);

template <typename T>
T f2(T x) requires HasFooBar<T>;
```

2. Bir başka concept tanımlayabiliriz:

```
template <typename T>
concept cn1 = std::integral<T> || HasFooBar<T>;
```

3. Constraint Template Parameter olarak

```
template <typename T>
requires HasFooBar<T>
class Myclass {

};

template <HasFooBar T>
class HerClass {

};
```

4. Abbreviated Template Syntax ile:

```
template <HasFooBar T>
void g(T x);


void func2(HasFooBar auto x);
```

→ Abbreviated template syntax ile, fonksiyonun return değeri de concept olabilir.

5. Lambdalarda

```
//geri dönüş değeri türü
//trailing return type ile
// constraint edi,lebilir mi

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

6. Conceptin kendisi, requires expr'da compound requirement olarak kullanılabilir.

```
template <typename T>
concept nec = requires  {

    requires std::integral<T>;

};
```

*** Cpp 20 öncesi, sadece int parametreyle çağırılabilen fonksiyon tanıtma:**

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

void func(int);

int main()
{

}
```

→ Exact match ise birinci func, başka bir parametre ise, deleted specialization olduğu için çağrılmaz!

(overload)

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

int main()
{
    func(1);
    func(1.);
    func(1.f);
    func(1u);
    func('1');
}
```

→ SFINAE

→ must include type traits

* Jonathan Boccara takip et → Fluentcpp.com!

*** Subsumption:**

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

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

void func(nec auto)
{
    std::cout << "nec auto\n";
}

void func(erg auto)
{
    std::cout << "erg auto\n";
}

struct A {
    void foo();
};

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

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

- Constraintler ayrıştırılır, daha kısıtlayıcı olan seçilir!

```cpp
template <typename T>
    requires requires(T x)
{
    x.foo();
}
void func(T x);

template <typename T>
requires requires(T x)
{
    x.foo();
    x.bar();
}
void func(T x);
```

Fakat, concept olmazsa, subsumption da olmaz