11.Hafta

İçindekiler

> 19_02_09_2023

Conseptler Konusu Devam

Sadece kısıtlama amaçlı değil farklı tür argümanları için farklı conceptler'in kullanımını sağlıyabiliyoruz. Böylece daha fazla concept constraint edinmiş fonksiyon seçiliyor.

```
template <typename T>
concept has_foo = requires(T t) {
     t.foo();
};

template <typename T>
concept has_foobar = has_foo<T> && requires(T t) {
     //t.foo(); bu bir önceki concept'ten geldiği için tekrar yazmamıza
gerek yok
     t.bar();
};

void func(has_foo auto x) { std::cout << "has_foo\n"; } //abbreviated
function template
void func(has_foobar auto x) { std::cout << "has_foobar\n"; } //abbreviated
function template

struct A {
     void foo() const { std::cout << "A::foo\n"; }
};

struct B {
     void foo() const { std::cout << "B::foo\n"; }
     void bar() const { std::cout << "B::bar\n"; }
```

```
};

int main()
{
     A a;
     B b;
     func(a); // has_foo
     func(b); // has_foobar
}
```

Derste anlatılan conceptler

Standart kütüphaneye birden fazla kütüphane eklendi. Örneğin

std::convertible_to concept'i, 2 parametreye sahip 1.from ve 2.to parametreleri.

```
void foo(const std::convertible_to<std::string> auto &x) // x string'e
dönüşebilir mi?
{
     std::cout << x << "\n";
}

void bar(const atuo &x)  requires std::convertible_to<decltype(x),
std::string>
{
     std::string s = x;
}

template <std::convertible_to<std::string> T >
void baz(const T& x)
{
     std::cout << x << "\n";
}

template <typename T>
    requires std::convertible_to<T, std::string>
void func(const T& x)
{
     std::cout << x << "\n";
}
```

```
template <std::convertible_to<bool> T>
void foo(T x)
{

}

int main()
{
```

```
    int x {243};
    foo(x); // geçerli
    foo(&x);// geçerli
    foo(nullptr);// geçerli
    foo("deneme");// T'türünün çıkarımı const char * türünden çıkarılıcak
    //foo(std::string("deneme")); std::string türünden dmn
    //foo(std::make_unique<int>(12)); //Fonskiyon çağrılmaz çünkü tür
dönüştürme fonksiyonu explicit.
}
```

- **std::copyable**,
- **std::indirect_unary_predicate**,

```
template <typename F, typename Iter>
void bar(F fn, Iter iter)
requires std::indirect_unary_predicate<F, Iter>
{
    if(fn(*iter))
    {
        std::cout << *iter << "\n";
    }
}

int main()

{
    const auto pred = [](int x) { return x % 2 == 0; };
    std::vector<int> ivec {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    boolalpha(std::cout);
    for(auto iter = ivec.begin(); iter != ivec.end(); ++iter)
    {
        bar(pred, iter);
    }
}
```

```
void func(std::invocable<int> auto  fn, int x) //buraya gönderilecek bilir
callable'in int argüman ile çağırılması gerekiyor.
{
    fn();
}
```

Conceptler ile yeni algoritmalar için kendi constraintlerini oluşturuyorlar ve bu sayıda birden fazla avantaj elde edilmiş oluyor.

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

```
    vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    ranges::sort(v); //artık bu şekilde çağırıyoruz.
}
```

C++20 Gelen Yeni Özellikler

Bitsel İşlemler ve Bitler

- Negatif tam sayı gösterimi C++20'den önce farklı şekilde yapılabilirdi artık 20'den sonra 2'yü tümleyen yapılması garanti altına alındı.
- İşaretli bir int türden değişkeninin operandın sola doğru kaydırma operandı olduğunda, UB oluşturuyordu ve artık C++20 ile bu ifade geçerli.

```
int main()
{
    using namespace std;
    int x = -1;
    cout << x << "\n"; //
    x <<=1;
    cout << x << "\n"; //
    std::cout <<std::format("{0:032b}  {0}",x, x);
}
```

Sağdan yapılan besleme ile artık bu ifade 0 olmak zorunda.

- Soldan yapılan feed eskiden implemantasyona bağlı idi. Artık bu sign extension şeklinde olacağı garanti altında.
- İşaretli tam-sayılar ve işaretsiz tam sayıların karşılaştırılması daha önceden uyarı mesajı veriyordu. Size fonksiyonu yanına `ssize()` fonksiyonu eklendi ve bu fonksiyonun dönüş değeri artık işaretli bir sayı türünden bir değer döndürüyor.

```
int main()
{
    std::vector<int> ivec = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    for (int i = 0; i < size(ivec); ++i)  //ADL - Argument Dependend Lookup
    {
        cout << ivec[i] << "\n";
    }
}
```

- yapıların *bitfield members* alanları olabiliyor

```
struct A
{
```

```
    int a : 3;
    int b : 4;
    int c : 5;
};
```

gibi. Burada kaç bit olacağını gösteren ifadeler de constexpr olmak zorunda. Bunlar sınıfın üyesi, C++20 öncesinde bunlar `default member initializer` olamazdı. C++20 ile birlikte yapıların bitfield memberlarına default member initializer atanabilir.

```
struct A
{
    int a : 3{2};
    int b : 4{2};
    int c : 5{2};
};
```

include<bit>

Sistem programlama tarafında kullanılabilen constexpr bağlamında kullanılabilen fonksiyonlar.

```
#include <bit>
```

- `rotatae_left` ve `rotate_right` fonksiyonları
- `bit_ceil` ve `bit_floor` fonksiyonları
- `has_single_bit` fonksiyonu,
- `count_leadign_zeros`, `count_trailing_zeros` ve `popcount` fonksiyonları
- `count_right_zeroes` ve `count_right_ones` fonksiyonları

Compile time'da constexpr gereken bağlamda kullanılmaya zorlanırsa geri dönüş değeri verilebiliyor.

- `std::bit_cast`: Şimdiye kadar bazı durumlarda UB oluşturan, farklı türden bir değerin aynı büyüklükten farklı bir türden kullanılmasını sağlayan fonksiyon.
- Modern C++ ile tamsayı ile önemli eklemeler yapıldı ve artık bir sabiti 2lik sayı sisteminde yazabiliyoruz. Digit separatorler ile birlikte kullanılabiliyor. `x = 0b1010'1010'1010'1010;`

```
int main()
{
    using namespace std;
    uint16_t x = 0b1010'1010'1010'1010;
    //[[fill]align][sign][#][0][width][,][.precision][type]
    cout << format("{0:016b} {0:8}\n",x);
    cout << format("{0:016b} {0:8}\n",rotl(x,2));//rotate left için bitsel
bir operatör yok 2.parametreye negatif değer
    //geçebiliyoruz ve böylece ters yönde dönmüş oluyoruz. rotr(x,2) olmuş
gibi düşünülebilir.
}
```

- bit_floor ve bit_ceiling 2 tabanına göre bu işlemleri gerçekleştiriyor.
- PowerOfTwo için artık bir constexpr fonksiyon var.

```
int main()
{
    using namespace std;
    has_single_bit(0b1000); //true
    has_single_bit(0b1001); //false
}
```

Bit Cast

Bazı durumlarda bir değişkenin türünü değiştirip aynı bitleri başka bir türdenmiş gibi ele almak istiyoruz. Fakat burada bazı girişimler UB oluştururken bazı değişiklikler istenilen şekilde olmayabilir.

```
struct Nec
{
    std::uint16_t x;
    std::uint16_t y;
};

int main()
{
    using namespace std;
    float e = 10;
    auto val = bit_cast<unsigned long long, float>(e);
    Nec n = {243u,8734u};
    auto uval = bit_cast<unsigned long, Nec>(n);
}
```

Dependant Type

Eğer elimizde ki tür template türünün bir üyesine bağlı ise

```
template <typename T>
void func(typename T::value_type)
{}
```

Derleyiciye bunun dependant type olduğunu `typename` keyword'ü ile anlatmak zorundaydık. C++20 ile birlikte eğer bunun dependant type olduğunda derleyici burada kullanmayı koşullu kılmıyor.

```
template <typename T>
struct PointerTrait
```

```
{
    using Pointer = void*;
};

template <typename T>
struct Den
{
    using Pointer = PointerTrait<T>::Pointer;

    T::Den bar(T::Den p)
    {
        return static_cast<T::Den>(p);
    }
};
```

- ismi cmp ile başlıyan fonksiyonlar mixed-typelar söz konusu olduğunda bu fonksiyonları kullanmamız gerekiyor.
- using bildirimleri onlarda constraint edilebiliyor.

```
template <typename T>
using fintpair = std::pair<int, T>;

int main()
{
    fintpair f = {2, 5.6};
    fintpair f1 = {2, 5.6};
}
```

inline olarak namespace bildirebiliyoruz

```
namespace A
{
    namespace B
    {
        inline namespace C
        {
            int b;
        }
    }
}
```

Source Location

- Source location başlık dosyasından gelen bir değişken var, sadece derleme zamanında çağırılabilen fonksiyonlar.

```
int main()
{
    auto loc = source_location::current(); //sadece
    constexpr auto pfname = sl.file_name();
    constexpr auto pline = sl.line();
    constexpr auto pfunc = sl.function_name();
}
```

- Bir çok yerde artık kullanılabiliyor, örneğin fonksiyonun geri dönüş değeri olarak kullanılabilir.

Using Enum bildirimi

- Bir enum türü:
- - Eski enum türlerinden tam sayı türlerine dönüşüm yapılabilirdi.
- - Enumaration constant isimler ayrı bir scope içerisinde değildi ve bu isim çakışması türünden problem oluşturabilirdi.
- - Bir class değil!
- - scoped enum olarak tanımlanabilir. Böylece kendi scope türü oluyor ve çözünürlük operatörleri ile kullanılabiliyor.

```
enum class Color {red, green, blue};
```