14.Hafta

24_23_09_2023

25_24_09_2023

C++ Idioms

ADL Fallback

- Burada öncelikle

```
#include <iostream>

void func()
{
    using std::cout; //using declaration
    //artık bu coma separated list ile kullanılabiliyor
    //using std::cout, std::endl;

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

- template fonksiyon söz konusu olduğunda burada nitelenmemiş bir isim kullanıldığında önce blok scope ve daha sonra namespace scope aranır. Biz buna bir argüman gönderirsek ADL devreye girer ve bu durumda nitelenmemiş ismi ADL gereği o scope ta da aranıyor

ADL:

```
namespace A
{
    struct MyStruct
    {
    };
    void swap(Foo &, Foo &);
```

```
}

namespace B
{
    class Myclass{};
    void foo(std::vector<Myclass> &); // vector'ün myclass açılımı
türünden
}


int main()
{
    A::MyStruct ms;
    swap(ms, ms); // ADL devreye girer
    std::vector<B::Myclass> vec;
    foo(vec); // ADL devreye girer
}
```

- ADL normal isim arama ile birlikte çalışır ve bu durumda ambiguity oluşabilir.

```
#include <iostream>
#include <vector>

namespace A
{
    struct MyStruct
    {
    };
    struct Bar
    {

    };
    void swap(MyStruct &, MyStruct &)
    {
        std::cout << "A::swap\n";
    }
}

template <typename T>
void func(T)
{
    T x, T y;
    using std::swap; // std'den swap fonksiyonunu kullanılabilir hale
geliyor
    swap(x, x); // ADL devreye girer, bu durumda
}

int main()
{
    A::MyStruct ms;
    A::Bar bar;
    func(bar); // std olan çağırılır.
```

```
    func(ms); // ADL devreye girer.
}
```

Yukarıdaki kod içereisinde using bildirimi kullanıldığı için eğer T'nin bulunduğu namespace içinde swap fonksiyonu varsa ADL devreye girer ve bu durumda bu fonksiyon çağrılır. Eğer bu fonksiyon bulunmazsa std namespace içindeki swap fonksiyonu çağrılır.

Hidden Friend Bildirimi

- Bu fonksiyonlar ADL'e tabi

```
class Myclass
{
    int x;
    public:
    friend void foo(Myclass &); // hidden friend declaration sınıfın
member'ı değil o namespace içinde ve görülür değil.
    //Ancak bu isim ADL ile aranır.
};

int main()
{
    Myclass m;
    foo(m);
}
```

- Avantajları, Bizim birden fazla sınıfımız olduğunu düşünelim ve

```
struct A{A operator+(const A &, const A &);};
struct B{B operator+(const B &, const B &);};
struct C{C operator+(const C &, const C &);};
struct D{D operator+(const D &, const D &);};
struct E{E operator+(const E &, const E &);};
struct F{F operator+(const F &, const F &);};

/* A operator+(const A &, const A &); */
/* B operator+(const B &, const B &); */
/* C operator+(const C &, const C &); */
/* D operator+(const D &, const D &); */
/* E operator+(const E &, const E &); */
/* F operator+(const F &, const F &); */
class Nec{   };
int main()
{
    Nec n1,n2;
    //auto n3 = n1+n2; // burada hangi operator+ aranacak fakat
bulunamayacak ve burada hem hata mesajı karışık olucak hem de compile tam
yok.
}
```

- Bu fonksiyonlar hidden friend yapıldığı zaman ADL'ile bulunabilir ve global name alanında bulunabilir değil.
- Bazı durumlarda örütülü implicit dönüşümleri engelliyor.

Scope Guard

Otomatik ömürlü nesenler scope'larının sonunda destroy ediliyor, eğer otomatik ömürlü nesneler destroy edildiğinde stack unbinding sürecinde de destructor çağırılıyor. Bu da biz kaynakları sınıf nesnelerine bağladığımızda bu kaynakları destructor'da geri verdiğimiz de kaynak sızıntısı olmamasını sağlıyor (Resource Acquisition Is Initialization - RAII).

```
class Myclass
{
};

void foo()
{
    Myclass x;
}
```

O zaman burada kaynakları bir sınıf nesnesine bağlarsak bir sorun olmaması sağlanıyor. Böylece hangi clean-up fonksiyonunu çağırılmasını istiyorsak o fonksiyonu veriyoruz.

```
template <typename Func>
class scope_guard
{
public:
    scope_guard(Func f) noexcept : m_f{f}    {    }
    scope_guard(scope_guard &&other) noexcept :
m_f{std::move(other.m_f)},dismissed(other.dismissed) {  }
    scope_guard(const scope_guard &) = delete;
    scope_guard &operator=(const scope_guard &) = delete;
    ~scope_guard()
    {
        if (!dismissed)
        {
            m_f();
        }
    }
private:
    Func m_f;
    bool dismissed{true;};
};

void cleanup()
{
    std::cout << "cleanup\n";
}
```

```
int main()
{
    if(1){
        scope_guard sg{cleanup};
    }
}
```

Return Type Resolver

Tür dönüşümleri ile alakalı, neye ilk değer verildiğine bağlı olarak tür dönüşümü yapılmasını istiyoruz.

- Tür dönüştürme operatörlerinin bazı özellikleri var:
    - `operator <Dönüştürülücek Tür>` yazılıyor.
    - const/non-const olabilir.
    - Referans türüne dönüşüm olabilir.
    - Overload edilebilir.
    - Member-template fonksiyonda olabilir. `template<typenae T> operator T() const;` şeklinde.
    - auto keyword'ü kullanılabilir.
    - explicit keyword'ü kullanılabilir.

```
class String
{
public:
    String(const char *p) : m_str{p} {}
    operator int() const
    {
        return std::stoi(m_str);
    }
    operator long long() const
    {
        return std::stoll(m_str);
    }
    operator double() const
    {
        return std::stod(m_str);
    }
private:
    std::string m_str;
};

int main()
{
    int ival = String{"23445"};
    double dval = String{"23445.2123"};
}
```

Bunu

```
vecotr<int> = Range(19,56); //19'dan 56'ya kadar olan öğeleri bir
container'a ilk değer vermek için kullanabilmek istiyoruz.
list<int> = Range(20, 97);
```

Bunu gerçekleştirmek istiyoruz

```
class Range
{

public:
    Range(int from, int to): m_from{from}, m_to{to} {
        if(m_from > m_to)
            throw std::invalid_argument{"Invalid Range"};
    }

    template <typename C>
    operator C() const
    {
        C con;
        for (int i = m_from; i < m_to; ++i)
        {
            con.insert(con.end(), i);
        }
        return con;
    }
private:
    const int m_from;
    const int m_to;
}

int main()
{
    using namespace std;
    vector<int> ivec = RAnge(19, 56);
    std::cout<< "ivec.size()"<<ivec.size()<< "\n";
    for (auto val : ivec)
    {
        std::cout << val << " ";
    }
    set<int> iset = Range(5, 13);
    std::cout<< "iset.size()"<<iset.size()<< "\n";
    for (auto val : iset)
    {
        std::cout << val << " ";
    }
}
```

Non Virtual Interface Idiom

- Herb Sutter

Polimorfik sınıflarda taban sınıfların destructorları ya public virtual ya da protected non-virtual olmalı.

```
class Base
{
public:
    virtual void foo();
    ~Base() // virtual destructor
    {
        std::cout << "Base destructor\n";
    }
};

class Der : public Base
{
public:
    ~Der() // non-virtual destructor
    {
        std::cout << "Der destructor\n";
    }
};

int main()
{
  Base *ptr = new Der;
  delete ptr; // burada UB oluşur.
}
```

- delete operatörü çağırıldığında, önce destructor çağırılıyor, eğer sanal değilse base sınıfın destructoru çağırılıyor. Burada destructor'ı virtual yapsaydık

```
class Base
{
public:
    virtual void foo();
    virtual ~Base() // virtual destructor
    {
        std::cout << "Base destructor\n";
    }
};

class Der : public Base
{
public:
    ~Der() // non-virtual destructor
    {
        std::cout << "Der destructor\n";
    }
};
```

```
int main()
{
  Base *ptr = new Der;
  delete ptr; // burada UB oluşur.
}
```

- Virtual dispatch uygulanmayan durumlar
- - Constructor içerisinde yapılan çağırılan fonksiyonlar için bu uygulanmaz, taban sınıf içinde bunu çağırma
- - Aynı şey için destructor içinde de geçerli
- - Object slicing olduğunda `Base b = der` olduğunda `b.vfunc()` dediğimizde virtual fonksiyon çağrılmaz.

Bazı durumlarda türemiş sınıf nesnesini taban sınıf pointer'ı ile delete etmeyeceğiz fakat virtual kullanımı engelleyemiyoruz.

```
class Base
{
    virtual void foo();
protected:
    ~Base() // virtual destructor
    {
        std::cout << "Base destructor\n";
    }
};

class Der : public Base
{
public:
    ~Der() // non-virtual destructor
    {
        std::cout << "Der destructor\n";
    }
};

int main()
{
  Base *p = new Der;
  //delete ptr; // erişim hatası oluşacak.
  Der myDer; // Erişim kontrolüne takılmayacak
}
```

- NVI: taban sınıfın sanal fonksiyonlarını sınıfın private/protected koyun, taban sınıfın sanal olmayan fonksiyonları taban sınıfın sanal olan fonksiyonlarını çağırsın.
- private virtual fonksiyonlar'da override edilebilir.

```
class Base
{
public:
    void foo()
    {
        foo_impl();
    }
    void bar()
    {
        bar_impl();
    }
private:
    virtual void foo_impl() = 0;
    virtual void bar_impl() = 0;
};
```

- bunun ne faydası var? Interface ile implementasyonu ayırma ilkesi. İmplementasyon sınıfın private bölümğnde kalıyor. Interface'de bir değişiklik yapılmadığı sürece interface değişmeyecek. Artık bu şekilde interface ve implementasyon ayrılmış.
- Asıl faydası, burada artık taban sınıf bir takım kontrol işlemleri yapabilir.

```
class Animal
{
public:
    virtual void speak()const = 0;

}
class Cat : public Animal
{
public:
    void speak()const override
    {
        std::cout << "Meow\n";
    }
}

class Dog : public Animal
{
public:
    void speak()const override
    {
        std::cout << "Hav Hav\n";
    }
}
```

- Yapıda değişiklik yapıp speak fonksyionu sanal fonksiyon yapmıyoruz.

```
class Animal
{
public:
    void speak()const {
        std::cout <<  get_sound() << std::endl;
    }
private:
    virtual std::string get_sound()const = 0;
}
class Cat : public Animal
{
public:
    std::string get_sound()const override
    {
        return "Meow";
    }
}

class Dog : public Animal
{
public:
    std::string get_sound()const override
    {
        return "Hav Hav";
    }
}
```

- kalıtım ile Fragile Base Problem'i var.

Aşağıdaki kodda, element olarak bir Set alıyoruz ve bunu kendi ihtiyacımıza göre adapte etmek istiyoruz.

- add fonksiyonu: underlining container'ın insert fonksiyonunu çağırıyor ve add_impl() modify yapabiliyor.
- add_range fonksiyonu: 2 pointer parametreli range alıyor ve aralıktaki öğeleri set ediyor. range parametreli bütün öğeleri set ediyor.
- custimizing işlemleri için add_impl ve add_range_impl fonksiyonları virtual olarak tanımlanıyor.

CountingSet ise bu fonksiyonları override ediyor ve add_impl/add_range_impl fonksiyonlarını override ediyor. count'ı arttırıyor.

```
class Set
{
    std::set<int> m_set;
public:
    void add(int val)
    {
        m_set.insert(val);
        add_impl(val);
    }
    void add_range(const int* begin, const int* end)
```

```
    {
        auto beg = begin;
        auto en = end;
        while(beg != en)
        {
            add(*beg++);
        }
        add_range_impl(begin, end);
    }
private:
    virtual void add_impl(int val) = 0;
    virtual void add_range_impl(const int* begin, const int* end) = 0;
};

class CountingSet : public Set
{
private:
    int m_count{};
    virtual void add_impl(int val) override
    {
        ++m_count;
    }
    virtual void add_range_impl(const int* begin, const int* end) override
    {
        m_count += std::distance(begin, end);
    }
};
```

Buradaki problem add fonksiyonu çağırıldığında add_impl override çağırılacak ve count birden fazla kez arttırılmış olucak.

Tuple ile alakalı Idiomlar

- Tür eşim bildirimi veya Enum bildirimleri kullanmak.

```
int main()
{
    using namespace std;
    tuple<int, double, string> t{12, 3.4, "deneme"};

//
    using id = int;
    using wage = double;
    using name = string;
    enum {id, wage, name}; //gibi
    tuple<id, wage, name> t2{12, 3.4, "test"};
    get<id>(t2) = 23;
//
    get<0>(t) = 23;
    //get<3>(t) // burada compile time error oluşur.
```

```
    get<int>(t) = 23;
}
```

- tie fonksiyonu: tuple'ı unpack etmek için kullanılır.

```
std::tuple<int, double, std::string> foo()
{
    return {12, 3.4, "deneme"};
}

class Date
{
public:
    friend bool operator<(const Date &lhs, const Date &rhs)
    {
        return std::tuple(lhs.year, lhs.month, lhs.day) <
std::tie(rhs.year, rhs.month, rhs.day);
    }
private:
    int year;
    int month;
    int day;
};

int main()
{
    using namespace std;
    int x{12};
    double y = 3.4;
    string z= "aaaaa";
    tie(x, y, z) = foo; //tuple'ı unpack ediyor. burada bu referans
açılımlarını döndürüyor
}
```

- x, y, z, birbirine atamak istiyoruz

```
int main()
{
    int x = 10, y = 20, z = 30;
    int temp = x;
    x = y;
    y = z;
    z = temp;
    tie(x,y,z) = tuple(y,z,x);
    cout << x << " " << y << " " << z << "\n";
}
```

templates

function template class template variable template alias template concept

- template parameter
- type parameter
- nttp
- parameter pack
- deduction
- CTAD
- name lookup
- dependent names

full/explicit specialization partial specialization

overloading partial ordering rules

friend declarations

meta functions ==> type_traits

sfinae tekniks tag dispatch

default template arguments

C++20 abbreviated template syntax eski kurallardaki genişletmeler.

constrained templates

fold expression unary fold binary fold unary left fold binary left fold unary right fold binary right fold

CRTP STL deki yardımcı öğeler, member templates