*ADL Fallback Idiom:

*ADL:

```cpp
template <typename T>    <T> Provide sample template arguments for IntelliSense
void func(T)
{
    T x, T y;

    swap(x, y);
}
```

→ swap nitelendirilmediği için, önce önceki blok, sonra namespace içinde arar!

→ swap'a argüman olarak bloktan bir değer geçildiği için artık func içindeki swap olarak nitelendirilir!

```cpp
namespace nec {
    class Foo {

    };

    void swap(Foo&, Foo&)
    {

    }
}


int main()
{
    nec::Foo x, y;

    swap(x, y);
}
```

ADL ile nec::swap bulundu!

→ ADL Fallback:

```cpp
template <typename T>    <T> Provide sample template arguments for IntelliSense
void func(T x)
{
    T y;

    using std::swap;
    swap(x, y);


}

namespace nec {

    class Foo {

    };

    class Bar {

    };

    void swap(Foo&, Foo&)
    {
        std::println(_Fmt: "nec::swap(Foo&, Foo&)");
    }
}
```

→ Foo türünden nesne ile çağrılırsa nec türünün swap'ı, Bar türünden nesne çağrılırsa std::swap çağrılır!

* Hidden Friend Idiom:

```
class Myclass {

public:
    friend void swap(Myclass&, Myclass&);
};


int main()
{
    Myclass m1, m2;

    swap(&: m1, &: m2);
}
```

Swap fonksiyonu global namespace'te olmasına rağmen, Nitelendir. Nitelendir. Niteleyici X Çağrılması için ADL gerekir!

ADL ile swap bulundu! ADL olmadan, diğer bir şekilde swap çağrılamaz X

* Ortada Dolaşmaz önüne geçiyor!

```
namespace nec {
    struct A {

    };

    struct B {

        operator A();
    };

    void bar(A);
    void foo(A);

}

int main()
{
    nec::A a;
    nec::B b;

    bar(b);
}
```

A türüne dönüşüm fonksiyonu

Bu çağrı legal olarak ADL ile bulunabilir.

```
namespace nec {
    struct A {
        friend void bar(A);

    };

    struct B {

        operator A();
    };

    void foo(A);

}

int main()
{
    nec::B b;

    bar(b);
}
```

Bir ortak Hidden friend olduğu için, Tüm dönüşüme dönüşüp gitmez! ADL ile bulunur!

* Scope Guard Idiom:

* Otomatik ömürlü bir nesne, diyelim bir exception throw ettir. → yakalanıp catch edilebilirse destructor'ı çağırılır! (stack unwinding)

* Kaynak sızıntısı olma ihtimalini ortadan kaldırırız. Unique_ptr gibi. Ister scope_guard'ın sonuna gelinsin, ister exception throw edildiğinde kaynak geri verilir!

* Bunu sadece dinamik ömürlü nesneler için değil Cleanup gerektiren tüm işlemler için kullanabiliriz. → fopen sonrası fclose / lock sonrası unlock gibi

```cpp
#include <iostream>
#include <utility>

template <typename Func>
class scope_guard {

public:
    scope_guard(Func f) noexcept : m_f(f)
    { }

    ~scope_guard()
    {
        if (m_call)
            m_f();
    }
    scope_guard(scope_guard &&other) : m_f(std::move(other.m_f)), m_call(other.m_call) {}

    scope_guard(const scope_guard&) = delete;
    scope_guard& operator=(const scope_guard&) = delete;

private:
    Func m_f;
    bool m_call{ true };
};
```

→ burada istediğimiz clean up fonksiyonunu argüman olarak geçebiliriz!

→ geçerlise dismiss edebilmek için bu flag var! / Dismiss çağrıldığında bu member değişir.

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

int main()
{
    std::cout << "main basladi\n";

    if (1) {
        scope_guard g{ cleanup };

    }

    std::cout << "main devam ediyor\n";
}
```

```cpp
void foo()
{
    std::cout << "foobasladi\n";

    if (1) {
        scope_guard g{ cleanup };

        bar();

    }

    std::cout << "foo devam ediyor\n";
}

int main()
{
    try {
        foo();
    }
    catch (const std::exception& ex) {
        std::cout << "exception caught: " << ex.what() << '\n';
    }
}
```

⇒ Her iki durumda da cleanup fonksiyonumuz doğruca çalışır!

* Scope Guard Factory:

```cpp
template <typename Func>
scope_guard<Func> finally(const Func& f)noexcept
{
    return scope_guard<Func>(f);
}
```

* Return Type Resolver Idiom:

```
class String {

public:
    String(const char *p) : m_str{p} {}

    operator int()const
    {
        return std::stoi(m_str);
    }

    operator double()const
    {

        return std::stod(m_str);
    }

    operator long long()const
    {
        return std::stoll(m_str);
    }
private:
    std::string m_str;
};


int main()
{
    String s{ "3456.98" };

    int ival = s;
    double dval = s;
```

No issues found

* NON-virtual Interface (Idiom): By Nesb Sitter

polimorfik sınıflarda
Taban sınıfların destructorları
    ya public virtual
    ya da protected non-virtual olmalı

→ Virtual Dispatch Uygulanmayan Durumlar: 1. Constructor içinde yapılan virtual function çağrıları → yalnızca sınıf kendisi inşa edilmeden virtual func çağrılır

2. Destructor " " " " "

3. Qualified Name ile yapılan çağrılar

```
Base::vfunc
```

4. Object Slicing

```cpp
class Animal {
public:
    void speak()const
    {
        std::cout << get_sound() << std::endl;
    }
private:
    virtual std::string get_sound()const = 0;
};


class Cat : public Animal {
private:
    virtual std::string get_sound()const override
    {
        return "miyav miyav miyav";
    }

};

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

};
```

- Türeyen sınıf public interface'ini değiştirir.
- Türeyen sınıf invariantları control edebilir.

*Tuple Idioms: - Typedefler ya da enumlar ile

```cpp
{
    using namespace std;

    using id = int;
    using wage = double;
    using name = string;

    tuple<id, wage, name> x{ 345, 6.35, "murat" };

    get<name>(x)

}
```

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

    enum type { id, wage, name };
    tuple<int, double, string> tx{ 12, 5.6, "yasar" };

    get<id>(tx);
```

Templates:

function template
class template
variable template
alias template
concept

template parameter

type parameter
nttp
template parameter

parameter pack (variadic template)