*Unique ptr ile Dinamik Bir Dizinin Hayatını Kontrol Etmek:

* Array new ile oluşan nesne, array delete ile silinmeli → undefined behavior yoksa

- Cpp 14 öncesinde unique_ptr<T[]> partial specialization yoktu. O yüzden array delete olan custom lambdalar kullanıldı.

Örnek:

```
class Nec {
public:
    Nec()
    {
        std::cout << "ctor\n";
    }

    ~Nec()
    {
        std::cout << "destructor\n";
    }
};

int main()
{

    std::cout << "main basladi\n";
    {
        std::unique_ptr<Nec[]> uptr{ new Nec[5]{} };

    }

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

→ [] partial specialization

→ Fakat örnekte *uptr olmuyor, uptr[] interface'i eklendi

```
int main()
{

    std::cout << "main basladi\n";
    {
        std::unique_ptr<Nec[]> uptr{ new Nec[5]{} };

        for (int i = 0; i < 5; ++i) {
            uptr[i]
        }


    }
```

```
int main()
{

    auto uptr = make_unique<Nec[]>(5);
```

→ Cpp 20'de new ile initialize zorunluluğu kalktı.

* unique ptr exception safety sağlar

```
void foo()
{

    throw std::runtime_error{ "hata hata\n" };

}


void func()
{
    auto pd = new Nec;
    ////
    foo();

    delete pd;
}


int main()
{
    try {

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

→ unique ptr olmadığı için exception yakalandıktan sonra → Destructor çağırılmayacak XX → resource leak!

→ bool nla buraya gitmeyecek

```cpp
void func()
{
    auto upd = make_unique<Nec>();
    ////
    foo();

}

int main()
{
    try {

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

→ exception throw edildiğinde, stack unwinding sağlanıp destructor çağrılır, sonra exception throw edilir.

Unique Ptr Öğelerini Containerda Saklamak:

```cpp
using namespace std;


int main()
{
    vector<Date*> dvec;

    for (int i = 1; i <= 12; ++i) {
        dvec.push_back(new Date{ i, i, 2022 });
    }

    for (auto p : dvec) {
        std::cout << *p << "\n";
    }

    for (auto p : dvec) {
        delete p;
    }
```

(Date* → raw pointer)

→ raw pointer ile saklandığında, tek tek delete etmemiz gerekir yoksa resource leak

* Daha tehlikeli bir durum, biz eğer vektörden eleman silseydik, pointer silinecekti ama nesne yaşamaya devam ediyor olacaktı.

→ Unique ptr kopyalamaya kapalı olduğu için, move semantiği ile vektöre eklenir

```cpp
using namespace std;

using DatePtr = std::unique_ptr<Date>;

using namespace std;

int main()
{
    vector<DatePtr> vec;

    vec.push_back(DatePtr{new Date{ 1, 5, 1998 }});
    vec.push_back(make_unique<Date>(4, 8, 2011));
    vec.emplace_back(new Date{ 2, 10, 1991});

    for (int i = 1; i <= 5; ++i) {
        vec.push_back(make_unique<Date>(i, i, 2011 + i));
    }

    std::cout << "vec.size() = " << vec.size() << "\n";

    _getch();



}
```

(DatePtr{new Date{ 1, 5, 1998 }} → geçici nesne pr value)

→ Ekleme alternatifleri

→ emplace back direct container'a argümanı gönderir.

* Böylelikle containerda pop-back yaparak, nesnenin destroy edilme garantisi oluyor.

```cpp
int main()
{
    vector<DatePtr> vec;
    vector<DatePtr> destvec(8);

    vec.push_back(DatePtr{new Date{ 1, 5, 1998 }});
    vec.push_back(make_unique<Date>(4, 8, 2011));
    vec.emplace_back(new Date{ 2, 10, 1991});

    for (int i = 1; i <= 5; ++i) {
        vec.push_back(make_unique<Date>(i, i, 2011 + i));
    }

    move(vec.begin(), vec.end(), destvec.begin());

    std::cout << "vec.size() = " << vec.size() << "\n";
    std::cout << "destvec.size() = " << destvec.size() << "\n";

    _getch();
```

→ copy yapılamaz! kopyalamaya kapalı. Bu yüzden copy'nin move yapanı "move" algoritm

→ kaynak range'deki öğeler başka range'e moveland

*** make_move_iterator:** - Bu bir iterator adaptörü

```cpp
using namespace std;

using DatePtr = std::unique_ptr<Date>;

using namespace std;

int main()
{
    vector<DatePtr> vec;

    vec.push_back(DatePtr{new Date{ 1, 5, 1998 }});
    vec.push_back(make_unique<Date>(4, 8, 2011));
    vec.emplace_back(new Date{ 2, 10, 1991});

    cout << (vec[0] ? "dolu" : "bos") << "\n";   // → dolu
    auto x = *make_move_iterator(vec.begin());

    cout << (vec[0] ? "dolu" : "bos") << "\n";   // → boş
    _getch();
```

→ * (dereference) ile yazılınca, unique_ptr
→ dez yazılsaydı iterator olacaktı

*** Unique_Ptr member'i olan Sınıflar:**

```cpp
class Myclass {

private:
    //members
    std::unique_ptr<std::string> mptr;

};

int main()
{
    Myclass m1;
    Myclass m2(std::move(m1));

    m1 = std::move(m2);
}
```

* Unique Ptr kopyalanamaz, bu yüzden sınıfın copy constructor ve copy assignment fonksiyonu oluştururken syntax hatası alınır. Bunun sonucunda da copy ctor compiler tarafından delete edilir.

* Sınıfın kendisi de move-only olur!

* move ctor / move assignment, implicitly declared

→ Unique Ptr member'ı olan sınıfı kopyalayabilmek için:

```cpp
class Myclass {

public:
    Myclass(const Myclass& other) : mptr{ other.mptr ? std::make_unique<std::string>(*other.mptr) : nullptr}
    {

    }
private:
    std::unique_ptr<std::string> mptr;

};

int main()
{
}
```

`?` : bunun olmasının nedeni, eğer other'ın unique-ptr'ı == nullptr ise bizimkini de null'a çekmeliyiz. Nullptr'ı kopyalayamayız XX

`*other.mptr` : dereference ederek other'ın değerini alırız

make_unique yerine, new std::string(*other.mptr)

*Unique_ptr ile incomplete typeların ilişkisi:

```cpp
class Myclass;

int main()
{
    std::unique_ptr<Myclass> uptr;
}
```

→ Syntax hatası:

Burada hatanın nedeni, delete kodunun olduğu yerde complete type olmaması. Classların destructorları, static inline olarak derleyici tarafından oluşturulur.

→ Çözüm: sınıfın destructor'ının önce sınıf içinde declaration'ını yapıp, sonra başka bir noktada (.cpp file) ayrı bir bildirimle onu =default; etmek

(Dakika 1.55'te anlatım başlıyor, geri dön izle)

```cpp
#include <memory>

class Nec;

class Myclass {
public:
    ~Myclass();
private:
    std::unique_ptr<Nec> mp;
};
```

3. Ama derleyici burada destructor'ı inline olarak default etseydi, o zaman incomplete type olduğu için syntax hatası.

```cpp
//complete type
class Nec {

};

Myclass::~Myclass() = default;

void func()
{
    Myclass m;
}
```

2. → Nec complete type olduğu için syntax hatası olmaz

1. → scope sonunda myclass destructor çağrılır.

* Pimpl Idiom (Technique)

* Pointer Implementation
  - → handle-body idiom
  - → opaque pointer
  - → cheshire cat

  hepsi aynı

* Amaç, sınıfın private interface'ini client'tan gizlemek.

```cpp
#pragma once

#include <string>
#include "date.h"
#include <vector>

class Student {
public:


private:
    std::string m_name;
    Date m_bdate;
    std::vector<int> m_grades;
    //...

}
```

Bu memberlar olduğu için ilgili kütüphaneler include edildi.

- Eğer bir şekilde private'ı gizleyebilirsek bu kütüphaneleri include etmemize gerek kalmaz.

→ Peki neden gizlemek isteriz:

Çünkü, sürekli private için gereken kütüphaneleri client da dolaylı yoldan include edecek

↳ Bu durumun dezavantajları: - Bilgi gizliliği yok

- Client code, çok fazla include edeceği için Compile Time oldukça uzar.

- Fakat en önemlisi, bağımlılık. Eğer içlerinden bir kütüphane değişirse, onu include eden tüm kütüphaneler yeniden derlenmeli, Eğer derlenmezse Abstract Binary Interface farklı olacak → Bu tanımsız davranışa sebebiyet verir.

```cpp
#pragma once

class Student {
public:
    Student();
    ~Student();
    void print()const;
    void set();
    int get_best_grade()const;

private:
    struct pimpl;
    pimpl *mp;

};
```

→ Header File

Bu veri elemanları yerine Incomplete type bir struct ve onun pointerını koyduk

- Bu pimple idiom'ın eski implementasyonu. Yine resource leak riski var, çünkü bu mp'yi Student destructor delete ediyor. ⇒ Unique pointerla implement etmeliyiz.

```cpp
#include "student.h"
#include "date.h"
#include <string>
#include <vector>

struct Student::pimpl {

    std::string m_name;
    Date m_bdate;
    std::vector<int> m_grades;
};

Student::Student() : mp{ new pimpl } {

}

Student::~Student()
{
    delete mp;
}
```

→ Şimdi bunları Cpp File'a ekledik

→ Student hayata gelince pimple olacak ve biz onun içindekileri kullanıyoruz.

→ Header File

```cpp
#pragma once

#include <memory>


class Student {
public:
    Student();
    ~Student();
    void print()const;
    void set();
    int get_best_grade()const;

private:
    struct pimpl;
    std::unique_ptr<pimpl> mp;


};
```

pimpl açılımı incomplete type olduğu için, destructorın yazdığı destructor inline olursa incomplete type'ı delete etmeye çalışır.

Burada bildirimini yaptığı cpp dosyasında default edicez !!

→ Cpp File

```cpp
struct Student::pimpl {

    std::string m_name;
    Date m_bdate;
    std::vector<int> m_grades;

    void print()const
    {
        std::cout << m_name << "\n";
        std::cout << m_bdate << "\n";
        for (const auto const int i : m_grades)
            std::cout << i << " ";
        std::cout << "\n";
    }

};

Student::~Student() = default;

Student::Student() : mp{ std::make_unique<pimpl>()} {
    //
}

void Student::print()const
{
    mp->print();
```