necatiergin / Cplusplus_Ders_Notlari

Sign up

Code Pull requests 0 Projects 0 Actions Security Pulse

Branch: master

Find file Copy path

Cplusplus_Ders_Notlari / unique_ptr.md

Fetching contributors...

829 lines (647 sloc) | 32.5 KB

Raw Blame History

# unique_ptr sınıfı

`C++11` standartları ile birlikte standart kütüphaneye dahil edilmiş olan `unique_ptr` bir akıllı gösterici `(smart pointer)` sınıfıdır. Bu akıllı gösterici sınıfı genel olarak "tek sahiplik" `(exclusive ownership)` stratejisini gerçekleştirir. Bir `unique_ptr` nesnesi bir dinamik sınıf nesnesini gösteren tek bir gösterici olarak kullanılır. `unique_ptr` nesnesi, kendi hayatı sona erince sahibi olduğu dinamik sınıf nesnesinin de hayatını sonlandırarak onun tutmakta olduğu kaynakların serbest bırakılmasını sağlar. Bu sınıfın temel varlık nedenlerinden biri, bir hata nesnesi `(exception)` gönderildiğinde söz konusu olabilecek kaynak sızıntısının `(resource leak)` engellenmesidir.

`unique_ptr` sınıfı `C++98` standartlarında var olan ancak dildeki araçların yetersizliğinden kaynaklanan kötü tasarımı nedeniyle eleştirilen `auto_ptr` sınıfının yerine getirilmiştir. `C++11` standartları ile `auto_ptr` sınıfı kullanımdan düşürülmüş `(deprecated)` onun yerine hem daha yalın ve daha net bir arayüze sahip olan hem de daha düşük kodlama hatası riski içeren `unique_ptr` sınıfı standart kütüphaneye eklenmiştir. `auto_ptr` sınıfının tasarlandığı dönemde `C++` dili taşıma semantiği, değişken sayıda tür parametresine sahip şablonlar `(variadic templates)` gibi araçlara sahip değildi. `C++11` standartlarıyla dile kazandırılan bu araçlar `unique_ptr` sınıfının güvenli bir biçimde tasarlanmasına olanak sağlamıştır.

### unique_ptr sınıfının kullanımı

Bazı işlevler işlerini şu şekilde görür:

- Önce işlerini gerçekleştirebilmek için sınıf nesneleri yoluyla bazı kaynaklar edinirler.
- Sonra yüklendikleri işleri gerçekleştirirler.
- İşlerini tamamladıktan sonra edindikleri kaynakları geri verirler.

İşlev içinde edinilen kaynaklar, işlev içinde tanımlanan yerel sınıf nesnelerine bağlanmışlarsa, işlevin kodundan çıkıldığında yerel sınıf nesnelerinin sonlandırıcı işlevinin çağrılmasıyla tutulan kaynaklar geri verilmiş olur. Ancak kaynaklar yerel bir sınıf nesnesine bağlanmadan dinamik olarak dışsal biçimde edinildiğinde dinamik sınıf nesnesini yöneten gösterici değişkenler tarafından kontrol edilirler. Bu durumda dinamik nesnenin hayatı `delete` ifadesi ile sonlandırılır. Aşağıdaki örneği inceleyin:

```
void func()
{
    class ResourceUser *pd = new ResoruceUse; // dinamik bir nesne oluşturuluyor
    // kaynaklar kullanılarak bazı işlemler gerçekleştiriliyor.
    delete pd; // Dinamik nesnenin ömrü sonlandırılarak kaynaklar geri veriliyor.
}
```

Böyle bir işlev sorunlara yol açabilir. Sorunlardan biri dinamik nesnenin hayatının sonlandırılmasının `(delete edilmesinin)` unutulmasıdır. Öneğin `delete` işleminden önce bir `return` deyimi yürütülürse dinamik nesnenin hayatı sonlandırılmayacaktır.

Başlangıçta kolayca görülüyor olmasa da bir başka sorun da bir fonksiyondan hata nesnesinin `(exception)` gönderilmesidir. Bir hata nesnesi gönderildiğinde programın akışı işlevden çıkacak böylece `delete` deyimi yürütülmeyecektir. Bu durum bir bellek sızıntısına `(memory leak)` neden olabileceği gibi daha genel olarak bir kaynak sızıntısına `(resource leak)` yol açabilir.

Gönderilebilecek tüm hata nesnelerinin yine işlev tarafından yakalanması oluşabilecek kaynak sızıntısı engelleyebilir:

```
void f()
{
 class ResourceUser *ptr = new class ResourceUser; // bir nesne oluşturuluyor
 try {
  // bazı işlemler yapılıyor
 }
 catch (...) { // hata nesneleri yakalanıyor
  delete ptr; // kaynaklar geri veriliyor
  throw; // hata nesnesi yeniden gönderiliyor
 }

 delete ptr; // işlevden normal olarak çıkılırsa dinamik nesnenin hayatı sonlandırılıyor.
}
```

Bir hata nesnesinin gönderilmesi durumunda da kaynakların güvenli bir şekilde geri verilmesi sağlanmak istenirse hem daha fazla kod yazılması gerekir hem de yazılan kod çok daha karmaşık hale gelir. Dinamik olarak yaratılan nesnelerin sayısı birden fazla ise çok daha karışık bir durumun oluşacağı açıktır. Hata oluşumuna açık olan ve gereksiz bir karmaşıklığa neden olan bu kötü kodlama stilinden kaçınılmalıdır.

Bu amaçla tasarlanabilecek bir akıllı gösterici `(smart pointer)` sınıfı sorunu çözebilir. Bir akıllı gösterici nesnesinin kendi sonlandırıcı işlevinin çağrılmasıyla, akıllı göstericinin yönettiği dinamik nesnenin de hayatı sonlandırılabilir. Yerel bir akıllı gösterici nesnesi söz konusu olacağından artık işlevden ister normal yollarla ister bir hata nesnesi gönderilmesi yoluyla `(exception)` çıkılsın akıllı gösterici nesnesine bağlanmış olan dinamik nesnenin hayatı sonlanacak, böylece kaynak sızıntısı oluşmayacaktır. İşte `unique_ptr` sınıfı bu amaçla tanımlanmış bir akıllı gösterici sınıfıdır.

`unique_ptr` sınıfı türünden bir nesne, gösterdiği dinamik ömürlü nesnenin tek sahibi durumundadır. `unique_ptr` nesnesinin hayatı sonlandığında yani bir `unique_ptr` nesnesinin sonlandırıcı işlevi çağrıldığında onun sahip olduğu dinamik nesnenin de hayatı sonlandırılır, yani o dinamik nesne `delete` edilir.

Bir `unique_ptr` nesnesinin gösterdiği dinamik ömürlü nesneyi gösteren başka bir gösterici yoktur" ve bu semantik yapı kullanıcı kodlar tarafından da sürdürülmeli ve korunmalıdır.

Daha önceki örneğe geri dönüyouz:

```
#include <memory>  // unique_ptr için başlık dosyası

void f()
{
    // bir unique_ptr nesnesi oluşturuluyor ve bu nesneye ilk değer veriliyor.
    std::unique_ptr<ResourceUser>(new resourceUser);
    // işlemler yapılıyor
}
```

Hepsi bu kadar. Artık hata yakalamaya ilişkin deyimlere gerek kalmadığı gibi `delete` işleci de kullanılmıyor.

**bir unique_ptr nesnesinin kullanımı**

`unique_ptr` sınıf şablonu bir göstericinin özelliklerini destekleyen bir arayüze sahiptir: İçerik `(dereferencing)` işlecinin kullanılmasıyla `unique_ptr` nesnesinin gösterdiği dinamik nesneye erişilebilir. `unique_ptr` nesnesinin gösterdiği dinamik nesnenin öğelerine ok işleciyle erişmek de mümkündür. Aşağıdaki kodu inceleyin:

```
#include <iostream>
#include <string>
#include <memory>

int main()
{
 // oluşturulan unique_ptr nesnesine dinamik bir string nesnesi ile ilkdeğer veriliyor:

 std::unique_ptr<std::string> uptr(new std::string("Maya"));
 (*uptr)[0] = 'K'; // Yazının ilk karakteri değiştiriliyor
 uptr->append("can"); // Yazının sonuna karakterler ekleniyor.
 std::cout << *uptr << std::endl; // yazı yazdırılıyor.

 return 0;
}
```

`unique_ptr` sınıfının kurucu işlevi `explicit` olduğundan bu türden bir nesne kopyalayan ilk değer verme `(copy initialization)` sözdizimiyle başlatılamaz.

```
std::unique_ptr<int> uptr = new int; // Geçersiz
std::unique_ptr<int> uptr(new int); // Geçerli
```

Bir `unique_ptr` nesnesi dinamik bir nesneye sahip olmadan da varlığını sürdürebilir. Varsayılan kurucu işlev ile hayata getirilen `unique_ptr` nesnesi hiçbir dinamik nesnenin sahibi değildir.

```
std::unique_ptr<std::string> up;
```

Bir `unique_ptr` nesnesine `nullptr` değeri doğrudan atanabileceği gibi bu amaçla sınıfın `reset` isimli üye işlevi de çağrılabilir:

```
uptr = nullptr;
uptr.reset();
```

Bu durumda eğer `unique_ptr` nesnesi bir dinamik nesneye sahipse sahip olduğu dinamik nesneyi `delete` eder.

Sınıfın `release` isimli üye işlevi bir `unique_ptr` nesnesinin sahip olduğu dinamik nesnenin sahipliğini bırakır. Bu işlev doğrudan `unique_ptr` nesnenin kontrol ettiği dinamik nesnenin adresini döndürmektedir. Bu işlevin çağrılmasıyla artık dinamik nesnenin sorumluğunu bu işlevi çağıran kod üstlenir:

```
std::unique_ptr<std::string> up(new std::string("Kaan Aslan"));
std::string* sp = uptr.release(); // uptr sahipliği bırakıyor
```

Sınıfın `bool` türüne dönüşüm yapan üye işleviyle bir `unique_ptr` nesnesinin dinamik bir nesneyi kontrol edip etmediği sınanabilir:

```
if (uptr) { // uptr dinamik bir nesneye sahip ise
 std::cout << *uptr << std::endl;
}
```

Bir `unique_ptr` nesnesinin dinamik bir nesnenin sahibi olup olmadığı `unique_ptr` nesnesninin `nullptr` değerine eşitliği ile de sınanabilir:

```
if (uptr != nullptr) // uptr dinamik bir nesneye sahip ise
```

`unique_ptr` nesnesininin veri elemanı olarak tuttuğu ham göstericinin değerinin `nullptr` değerine eşitliğiyle de aynı sınama gerçekleştirilebilir:

```
if (uptr.get() != nullptr) // uptr bir nesneye sahip ise
```

**unique_ptr ile sahipliğin devredilmesi**

`unique_ptr` sınıfı tek sahiplik semantiğini uygular. Sınıfın kopyalayan kurucu işlevi `(copy constructor)` ve kopyalayan atama işlevi `(copy assignment function)` `delete` edilerek sınıf kopyalamaya karşı kapatılmıştır. Ancak birden fazla `unique_ptr` nesnesinin aynı dinamik nesnesini adresiyle başlatılmaması programcının sorumluluğundadır:

```
#include <string>
#include <memory>
#include <iostream>

int main()
{
 std::string* sp = new std::string("hello");
 std::unique_ptr<std::string> up1(sp);
 std::unique_ptr<std::string> up2(sp); // Yanlış: up1 ve up2 aynı dinamik nesneye sahip
 //
}
```

Yukarıdaki gibi bir kod çalışma zamanı hatasına neden olur. Kodlayıcıların böyle hatalardan kaçınması gerekir. Peki, `unique_ptr` sınıfının kopyalayan kurucu işlevi ve atama işlecinin kodu nasıl olmalı? Bir unique_ptr nesnesini kopyalama yoluyla hayata başlatamamayız ve bir `unique_ptr` nesnesine kopyalama yoluyla atama yapamayız. `unique_ptr` sınıfında yalnızca taşıma semantiği kullanılmaktadır. `unique_ptr` nesneleri kopyalanamaz ama taşınabilir. Taşıyan kurucu işlev ve taşıyan atama işlevi sahipliğin başka bir göstericiye devredilmesini sağlar:

Kopyalayan kurucu işlevin kullanıldığını düşünelim:

```
#include <memory>

class Myclass {
 //
};

std::unique_ptr<Myclass> up1(new Myclass);
std::unique_ptr<Myclass> up2(up1); // Geçersiz
std::unique_ptr<Myclass> up3(std::move(up1)); // Geçerli
```

İlk deyimden sonra, `up1` `new` işleciyle hayata getirilmiş nesnenin sahibi olur. Kopyalayan kurucu işlev gerektiren ikinci deyim sentaks hatasıdır. İkinci bir `unique_ptr` nesnesinin aynı dinamik `Myclass` nesnesinin sahipliğini almasına izin verilmez. Aynı zamanda tek bir sahibe izin verilmektedir. Ancak üçüncü deyimle sahiplik `up1` nesnesinden `up3` nesnesine devredilir. Artık `up1` nesnesi sahipliği bırakmıştır. new işleciyle hayata getirilmiş `Myclass` nesnesi `up3`'ün hayatının bitmesiyle delete edilir. Atama işleci de benzer şekilde davranır:

```
#include <memory>

class Myclass {
 //
};

int main()
{
 std::unique_ptr<Myclass> up1(new Myclass);
 std::unique_ptr<Myclass> up2;
 //up2 = up1; // geçersiz
 up2 = std::move(up1); //  sahiplik up1 nesnsinden up2 nesnesine devredilir.
}
```

Burada atama operatör işlevi sahipliği `up1` nesnesinden `up2` nesnesine devreder. Sonuç olarak, daha önce sahibi `up1` olan dinamik nesnenin artık yeni sahibi `up2`'dir. `C++11` öncesinde kullanılan `auto_ptr` sınıfında bu işlem doğrudan kopyalama semantiği ile yapılıyor bu da bir çok soruna neden oluyordu. Eger `up2` nesnesi atamadan önce bir dinamik nesnenin sahibi olsa idi bu dinamik nesne atamadan önce delete edilecekti:

```
#include <memory>

class Myclass {
 //
};

int main()
{
 // bir unique_ptr nesnesine dinamik bir nesneyle ilk değer veriliyor.
 std::unique_ptr<Myclass> up1(new Myclass);
 // bir başka unique_ptr nesnesine dinamik bir nesneyle ilk değer veriliyor
 std::unique_ptr<Myclass> up2(new Myclass);
 up2 = std::move(up1); // taşıyan atama işlevi up2'nin daha önce sahip olduğu nesne sonlandırılır.
 // Sahiplik up1'den up2'ye devredilir
}
```

Yeni bir sahiplik edinmeden sahip olduğu nesneyi bırakan bir `unique_ptr` nesnesi hiçbir nesneyi göstermez. Bir `unique_ptr` nesnesine başka bir `unique_ptr` nesnesinin değeri taşınarak atanmalıdır. `unique_ptr` nesnelerine adresler doğrudan atanamaz.

```
#include <memory>

class Myclass {
 //
};

int main()
{
 std::unique_ptr<Myclass> ptr;
 ptr = new Myclass; // Geçersiz
 ptr = std::unique_ptr<Myclass>(new Myclass); // Geçerli. Eski nesne sonlandırılır yenisi sahiplen

}
```

Bir `unique_ptr` nesnesine `nullptr` değerinin atanması nesnenin `reset` işlevinin çağrılmasına eşdeğerdir.

**nesne kaynağı ve boşaltım havuzu**

Sahipliğin devredilebilmesi `unique_ptr` nesnelerine özel bir kullanım alanı sunar: İşlevler dinamik nesnelerin sahipliğini `unique_ptr` nesneleri ile başka işlevlere aktarabilirler.

Bu iki ayrı yolla olabilir:

1. Bir işlev bir veri boşaltım havuzu `(sink)` olarak kullanılabilir.

Bu durumda, çağrılan işlevin parametre değişkeni kendisine sağ taraf değeri olarak gönderilen `unique_ptr` nesnesinin kaynağını devralır. Böylece, işlev sahipliğini devraldığı nesnenin sahipliğini yeniden bir başka koda devretmez ise işlevin kodunun çalışması sonlandığında `unique_ptr` nesnesinin sahiplendiği dinamik nesne silinir:

```
#include <memory>

class Myclass {
 //
};

void sink(std::unique_ptr<Myclass> up) // sink işlevi sahipliği devralır
{
 ///
}

int main()
{
 std::unique_ptr<Myclass>up(new Myclass);
 sink(std::move(up)); // up nesnesi sahipliği bırakır
 //
}
```

2. Bir işlev nesne kaynağı `(factory)` olarak davranabilir. `unique_ptr` geri döndürüldüğünde geri döndürülen sınıf nesnesinin sahipliği işlevi çağıran koda devredilir. Aşağıdaki örnek bu tekniği gösteriyor:

```
#include <memory>

class Myclass {
 //
};

std::unique_ptr<Myclass> source()
{
 std::unique_ptr<Myclass> ptr(new Myclass); // ptr dinamik nesnenin sahibi
 ///
 return ptr; // sahiplik çağıran işleve devrediliyor.
}

void g()
{
 std::unique_ptr<Myclass> p;

 for (int i = 0; i<10; ++i) {
  p = source(); // p geri döndürülen nesnenin sahipliğini alır
  //f işlevinin geri döndürdüğü bir önceki nesne silinir

 }
 // p'nin son sahip olduğu nesne silinir.
}
```

`source` işlevi her çağrıldığında `new` işleciyle dinamik bir `Myclass` nesnesi yaratılmış olur ve source işlevi bu nesneyi sahipliği ile birlikte kendisini çağıran koda gönderir. İşlevin geri dönüş değerinin `p` isimli `unique_ptr` nesnesine atanması dinamik nesnenin mülkiyetini bu nesneye devreder. Döngünün ikinci ve daha sonraki turlarında `p` nesnesine yapılan her atama `p`'nin daha önce sahiplendiği dinamik nesneyi siler.

`g` işlevinin çıkışında `p` nesneninin ömrü sona erdiğinden `p` için çağrılan sonlandırıcı işlevin çağrılması `p`'nin sahipliğini üstlendiği son dinamik `Myclass` nesnesinin de `delete` edilmesini sağlar. Bir kaynak sızıntısı mümkün değildir. İşlev içinden bir hata nesnesi gönderilse dahi, bir `unique_ptr` nesnesinin sahibi olduğu dinamik nesne silinecektir.

**unique_ptr nesnelerinin veri öğesi olarak kullanılması**

`unique_ptr` nesnelerinin sınıfların veri öğeleri yapılmasıyla kaynak sızıntıları engellenebilir. Ham göstericiler yerine akıllı göstericilerin kullanılması durumunda sonlandırıcı işleve gerek kalmaz. Nesnenin ömrünün bitmesiyle, veri elemanı olan akıllı gösterici nesnelerinin de hayatı sonlanacak bu da dinamik nesnelerin `delete` edilmesini sağlayacaktır. Ayrıca `unique_ptr` nesnelerinin kullanılmasıyla bir sınıf nesnesinin hayat başlama sürecinde bir hata nesnesini gönderilmesi durumunda kaynak sızıntısı engellenmiş olur. Bir sınıf nesnesi için sonlandırıcı işlevin çağrılabilmesi için söz konusu nesnenin kurucu işlevinin kodu tamamen çalışmış olmalıdır. Kurucu işlev içinden bir hata nesnesi gönderilirse yalnızca kurulumu tamamlanmış veri öğeleri olan sınıf nesneleri için sonlandırıcı işlev çağrılacaktır. Eğer sınıfın birden fazla ham gösterici veri öğesi var ise, birinci `new` işlemi başarılı olduktan sonra ikincisi başarısız olursa kaynak sızıntısı oluşur.

Örneğin:

```
class A {
public:
 A(int);
};

class B {
private:
 A* ptr1; // gösterici veri öğeleri
 A* ptr2;
public:
 // göstericilere ilk değer veren kurucu işlev
 // - ikinci new hata nesnesi gönderirse kaynak sızıntısı oluşur.

 B(int val1, int val2) : ptr1(new A(val1)), ptr2(new A(val2)) {}
 // kopyalayan kurucu işlev
 // ikinci new hata gönderirse kaynak sızıntısı olur
 B(const B& x) : ptr1(new A(*x.ptr1)), ptr2(new A(*x.ptr2)) {}

 // atama işlevi
 const B& operator= (const B& x)
 {
  *ptr1 = *x.ptr1;
  *ptr2 = *x.ptr2;
  return *this;
 }

 ~B()
 {
  delete ptr1;
  delete ptr2;
 }
};
```

Bu tür bir kaynak sızınıtısını önlemek için `unique_ptr` sınıf nesneleri kullanılabilir:

```
#include<memory>

class A {
public:
 A(int);
};

class B {
private:
 std::unique_ptr<A>ptr1; // unique_ptr veri öğeleri
 std::unique_ptr<A>ptr2; // unique_ptr veri öğeleri
public:
 // kurucu işlevler unique_ptr veri öğelerine ilk değer verir
 // kaynak sızıntısı mümkün değildir
 B(int val1, int val2): ptr1(new A(val1)), ptr2(new A(val2)) {}

 // kopyalayan kurucu işlev
 // kaynak sızıntısı mümkün değildir
 B(const B &x) : ptr1(new A(*x.ptr1)), ptr2(new A(*x.ptr2)) {}
 // atama işlevi
 const B& operator= (const B&x)
 {
  *ptr1 = *x.ptr1;
  *ptr2 = *x.ptr2;
  return *this;
 }
 // sonlandırıcı işleve gereke kalmaz
 // Derleyici tarafından yazılan sonlandırıcı işlev
 //ptr1 ve ptr2 göstericilerinin nesnelerini delete eder
};
```

Artık sonlandırıcı işleve gerek kalmaz çünkü `unique_ptr` nesnelerinin sonlandırıcı işlevleri dinamik `A` nesnelerinin `delete` edilmesini sağlar. `B` sınıfı için kopyalayan kurucu işlevin ve kopyalayan atama işlevinin de yazılması gerekir. Çünkü öğe olarak kullanılan `unique_ptr` nesneleri derleyicini yazacağı kodla kopyalanamaz. Eğer bu işlevler tanımlanamaz ise `B` sınıfı türünden nesneler kopyalanamaz yalnızca taşınabilir.

**unique_ptr ve diziler**

Bir `unique_ptr` nesnesi aşağıdaki durumlarda sahip olduğu nesneyi `delete` eder:

- `unique_ptr` nesnesinin hayatı sona erdiğinde
- `unique_ptr` nesnesine yeni bir `unique_ptr` değeri atandığında
- `unique_ptr` nesnesine `nullptr` değeri atandığında
- `unique_ptr` nesnesi için sınıfın `reset` işlevi çağrıldığında

Bu durumlarda silme işlemi `delete` işleci ile yapılmaktadır. Ne yazık ki C dilinden gelen kurallar nedeniyle bir göstericinin tek bir nesneyi mi yoksa bir diziyi mi gösterdiği bilinemez. Ancak dinamik dizilerin silinmesi `delete` işleci ile değil `delete[]` işleci ile yapılmalıdır. Dinamik bir dizinin `delete` işleci ile sonlandırılması çalışma zamanı hatasıdır. Aşağıdaki kod geçerli olsa da çalışma zamanı hatasına neden olur:

```
std::unique_ptr<std::string>up(new std::string[10]); // çalışma zamanı hatası
```

`shared_ptr` sınıfı için diziler için silme işlemini gerçekleştirecek özel bir `deleter` türünün kullanılması zorunludur. İstersek `unique_ptr` sınıfı için de bu araçla bir `deleter` oluşturabiliriz. Ama buna gerek yoktur. `C++` standart kütüphanesi `unique_ptr` sınıfını dizi türleri için özelleştirmiştir `(specialization)`. Dizi türleri için yapılan özelleştirme sahiplik sona erdiğinde `delete` işleci yerine `delete[]` işlecini kullanır. Eğer bir `unique_ptr` nesnesi dinamik bir diziyi gösterecekse bildirim aşağıdaki gibi yapılmalıdır:

```
std::unique_ptr<std::string[]> up(new std::string[10]); // OK
```

Ancak bu özelleştirmede sunulan arayüz birincil şablondakinden farklıdır. `operator*` ve `operator->' işlevleri yerine 'operator[]` işlevi sunulmuştur.

```
std::unique_ptr<std::string[]> up(new std::string[10]); //
std::cout << *up << std::endl; //Geçersiz * işlemi diziler için tanımlı değil.
std::cout << up[0] << std::endl; // Geçerli
```

Köşeli parantez işlevine gönderilen indisin geçerli bir değerde olmasından programcı sorumludur. Geçersiz bir indis değeri çalışma zamanı hatasına neden olur. Bu özelleştirilmiş sınıf taban sınıf türünden bir akıllı göstericinin türemiş sınıf türünden bir diziyle başlatılmasına da izin vermez. Yani çalışma zamanı çokbiçimliliği dizilerde `unique_ptr` sınıfı yoluyla desteklenmemektedir.

**default_delete sınıfı**

`unique_ptr` sınıf şablonunun (basitleştirilmiş) tanımı aşağıdaki gibidir:

```
namespace std {
 // birincil şablon
 template <typename T, typename D = default_delete<T>>
 class unique_ptr
 {
 public:
  T& operator*() const;
  T* operator->() const noexcept;
 };

// dizi türleri için kısmi özellştirme:
 template<typename T, typename D>
 class unique_ptr<T[], D>
 {
 public:
  T& operator[](size_t i) const;
 };
}
```

Yukarıdaki kodda `unique_ptr` sınıfının diziler için özelleştirilmesi görülüyor. Tanımdan da görüldüğü gibi özelleştirilmiş sınıfın arayüzünde `operator*` işlevi ve `operator->` işlevi yer almamakta fakat `operator[]` işlevi bulunmaktadır. `unique_ptr` sınıfının standart kütüphanede bulunan gerçekleştirimi `operator*` ve `operator->` işlevlerini geri dönüş değerleri türlerinin tam olarak elde edilebilmesi için bazı şablon hileleri kullandığından biraz daha karmaşıktır. Özelleştirilmiş sınıf için kullanılacak `std::default_delete<>` sınıfı silme işlemini `delete` yerine `delete[]` ile yapar:

```
namespace std {
 // birincil şablon
 template<typename T>
 class default_delete {
 public:
  void operator()(T* p) const; // delete p işlemini yapar
 };

 // dizi türleri için kısmi özelleştirme:
 template <typename T>
 class default_delete<T[]> {
 public:
  void operator()(T* p) const; // delete[] p işlemini yapar
 };
}
```

Varsayılan şablon tür argumanları otomatik olarak özelleştirmelere de uygulanmaktadır.

**make_unique işlev şablonu**

Hatırlayacağımız gibi `unique_ptr` sınıf şablonu dile `C++1` standartlarıyla eklenmişti. Ancak `C++11` standartlarında `make_unique` işlev şablonu yer almıyordu. Bu eksiklik `C++14` standartlarıyla karşılandı. Mükemmel gönderim `(perfect forwarding)` mekanizmasından faydalanan `make_unique` değişken sayıda parametreli `(variadic)` işlev şablonu, bir `unique_ptr` nesnesini sarmalayarak geri döndürüyor:

```
template <class T, class... Args>
unique_ptr<T> make_unique(Args&&... args);
```

Bu işlev şablonunun aşağıdaki gibi gerçeklendiğini düşünebiliriz:

```
template<typename T, typename... Args>
std::unique_ptr<T> make_unique(Args&&... args)
{
 return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}
```

İşleve kontrol edilecek dinamik nesnenin kurucu işlevinin kullanacağı argümanlar gönderiliyor. Aşağıdaki örneğe bakalım:

```
#include <memory>
#include <string>
#include <iostream>

int main()
{
        using namespace std;
 auto up = make_unique<string>(10, 'A');
 cout << *up << "\n";
 //...
}
```

Yukarıdaki kodda `make_unique` işlev şablonundan üretilecek bir işlevle `string` sınıfının `size_t` ve `char` parametreli kurucu işlevine `10` ve `'A'` değerleri gönderilerek önce dinamik ömürlü bir `string` nesnesi hayata getiriliyor. Daha sonra hayata gelen dinamik nesneyi kontrol eden bir `unique_ptr` nesnesi geri döndürülüyor.

**unique_ptr::get üye işlevi**

`unique_ptr` sınıfının `get` işlevinin geri dönüş değeri kontrol edilen dinamik nesnenin adresi. `unique_ptr`

nesnemizin yaşamını kontrol ettiği bir dinamik nesne yoksa işlev `nullptr` adresini döndürüyor:

```
#include <memory>
#include <string>
#include <iostream>

using namespace std;

int main()
{
 auto up = make_unique<string>(10, 'A');
 cout << *up << "\n";
 string *ptr = up.get();
 cout << ptr->size() << "\n";
}
```

Bu işlev dikkatli kullanılmalı. Örneğin bu işlevden elde edilen adresle yeni bir `unique_ptr` nesnesinin oluşturulması ya da bu adresteki nesnenin `delete` edilmesi çalışma zamanı hatasına neden olurdu:

```
int main()
{
 auto up1 = make_unique<string>(10, 'A');
 string *ptr = up1.get();
 unique_ptr<string> up2(ptr);

}
```

Yukarıdaki kodda hem `up1` hem de `up2` `unique_ptr` nesneleri aynı dinamik `string` nesnesini kontrol ediyor hale geliyor, böylece tek sahiplik `(exclusive ownwership)` ilkesi çiğneniyor. Her iki nesnenin de sonlandırıcı işlevi aynı dinamik `string` nesnesini `delete` edecek `(double deletion)` ve bu durumda çalışma zamanı hatası oluşacak. Benzer hataya aşağıdaki gibi bir kodla da düşülebilir:

```
int main()
{
 auto up = make_unique<string>(10, 'A');
 string *ptr = up.get();
 delete ptr;
 //...
}
```

Yukarıdaki kodda `get` işlevinden adresi alınan dinamik nesne `delete` ediliyor. `up` nesnesinin sonlandırıcı işlevi çağrıldığında aynı nesneyi yeniden `delete` edecek. Bu tanımsız davranış oluşturacak.

**deleter şablon tür parametresi**

`unique_ptr` sınıf şablonunun tanımını bir hatırlayalım:

```
template<
    typename T,
    typename Deleter = std::default_delete<T>
> class unique_ptr;
```

Şablonumuzun ikinci tür parametresi olan `Deleter` türüne bağlı işleve yapılan çağrı hayatı kontrol edilen nesnenin `delete` edilmesini sağlıyor. Bu şablon tür parametresinin varsayılan tür argumanı olarak standart `default_delete` şablonunu aldığını görüyorsunuz. Bu durumda

```
std::unique_ptr<std::string> up;
```

gibi bir tanımlama

```
std::unique_ptr<std::string, std::default_delete<std::string>> up;
```

biçiminde bir tanımlamaya eşdeğer.

`default_delete` sınıf şablonunun (basitleştirilmiş) kodunun şu şekilde olduğunu düşünebilirsiniz:

```
template <typename T>
class default_delete {
public:
 void operator()(T* p) const
 {
  delete p;
 }
};
```

`default_delete` sınıf şablonu dizi türleri için kısmi özelleştirmeye `(partial specialization)` tabi tutulmuş. Bu özelleştirmeye ilişkin kodun aşağıdaki gibi olduğunu düşünebilirsiniz:

```
template <typename T>
class default_delete<T[]> {
public:
 void operator()(T* p) const
 {
  delete[]p;
 }
 //...
};
```

Bu da şu anlama geliyor: `default_delete` sınıf şablonlarının kullanılmasıyla, `unique_ptr` nesnelerinin kontrol ettiği nesnelerin hayatları uygun biçimde `delete` ve `delete[]` işleçleriyle sonlandırılıyor. Eğer yaşamı kontrol edilen dinamik nesnenin hayatının sonlandırılması özelleştirilmiş bir şekilde gerçekleşecek ise bu durumda kendi deleter türümüzü şablon tür argümanı olarak kullanmak zorundayız. Bu amaçla global bir işlev, bir işlev sınıfı `(functor class)`, bir `lambda`, ya da bir işlev nesnesi kullanabiliriz. İlk örneğimizde global bir işlev kullanıyoruz:

```
#include <iostream>
#include <memory>
#include <string>

class A {
public:
 A() { std::cout << "A ctor\n"; }
 ~A() { std::cout << "A dtor\n"; }
 //..
};

void fdel(A *p)
{
 std::cout << p << " adresindeki nesne delete ediliyor\n";
 delete p;
}

int main()
{
 using namespace std;

 {
  unique_ptr<A, void(*)(A *)> up(new A, &fdel);
 }

 cout << "main devam ediyor\n";
 //...
}
```

Yukarıdaki kodda `deleter` olarak kullandığımız global `fdel` isimli işlev, `delete` işleminden önce `delete` edilecek nesnenin adresini standart çıkış akımına yazdırıyor. Şimdi de aynı iş için bir `lambda` ifadesi kullanıyoruz:

```
int main()
{
 using namespace std;
 auto f = [](A *p) {
                 cout << p << " adresindeki nesne delete ediliyor\n"; delete p; };
 {
  unique_ptr<A, decltype(f)> up(new A, f);
 }

 cout << "main devam ediyor\n";
 //...
}
```

`lambda`'ya ilişkin kapanış sınıfının `(closure)` tür bilgisi için `decltype` işlecinin kullanımına dikkat ediniz. Şüphesiz bu iş için kendimiz de bir `functor` sınıf oluşturabilirdik:

```
struct ADeleter {
 void operator()(A *p)const
 {
  std::cout << p << " adresindeki nesne delete ediliyor\n";
  delete p;
 }
};

int main()
{
 using namespace std;
 {
  unique_ptr<A, ADeleter> up(new A);
 }

 cout << "main devam ediyor\n";
 //...
}
```

Şimdi de `deleter` olarak `std::function` sınıf şablonunu kullanıyoruz:

```
#include <iostream>
#include <memory>
#include <string>
#include <functional>

class A {
public:
 A() { std::cout << "A ctor\n"; }
 ~A() { std::cout << "A dtor\n"; }
 //..
};

struct ADeleter {
 void operator()(A *p)const
 {
  std::cout << p << " adresindeki nesne delete ediliyor\n";
  delete p;
 }
};

void fdel(A *p)
{
 std::cout << p << " adresindeki nesne delete ediliyor\n";
 delete p;
}

auto f = [](A *p) {
 std::cout << p << " adresindeki nesne delete ediliyor\n"; delete p; };

template<typename T>
using UniquePtr = std::unique_ptr<T, std::function<void(T *)>>;

int main()
{
 using namespace std;
 {
  UniquePtr<A> uptr1(new A, fdel);
  UniquePtr<A> uptr2(new A, f);
  UniquePtr<A> uptr3(new A, ADeleter());
 }

 cout << "main devam ediyor\n";
 //...
}
```

Yukarıdaki kodda yapılan şablon eş isim bildirimine dikkat ediniz. Bu bildirim ile `T` bir tür olmak üzere,

```
UniquePtr<T>
```

açılımı

```
std::unique_ptr<T, std::function<void(T *)>>
```

açılımına karşılık geliyor. "deleter" olarak

```
std::function<void (A *)>
```

sınıfının kullanılması ile artık `deleter` olarak uygun parametrik yapıda işlev sağlayan herhangi bir çağrılabilir varlık `(callable)` kullanılabilir hale geliyor.

Bir `unique_ptr` nesnesinin yaşamını kontrol ettiği kaynağın `new` işleciyle oluşturulması zorunlu değil. Aşağıdaki kodda `unique_ptr` nesneleri standart `fopen` işleviyle oluşturulan dosyaları kontrol ediyor:

```
#include <cstdio>
#include <cstdlib>
#include <memory>

using namespace std;

FILE *fopen_write(const char *pfname)
{
 FILE *f = fopen(pfname, "w");
 if (!f) {
  fprintf(stderr, "%s dosyasi olusturulamadi\n", pfname);
  exit(EXIT_FAILURE);
 }
 return f;
}

struct FileCloser {
 void operator()(FILE *f)
 {
  fclose(f);
 }
};

void file_close(FILE *f)
{
 fclose(f);
}

int main()
{
 unique_ptr<FILE, void(*)(FILE *)> up1(fopen_write("necati.txt"),
                                       &file_close);
 auto fc = [](FILE *f) {fclose(f); };
 unique_ptr<FILE, decltype(fc)> up2(fopen_write("kaan.txt"), fc);
 unique_ptr<FILE, FileCloser> up3(fopen_write("oguz.txt"));

 fprintf(up1.get(), "necati ergin");
 fprintf(up2.get(), "kaan aslan");
 fprintf(up3.get(), "oguz karan");

}
```

Yukarıdaki kodda `up1` nesnesi için `deleter` olarak standart `fclose` işlevini sarmalayan global file_close işlevini kullanılıyor. `up2` nesnesi için ise `deleter` olarak bir `lambda`'nın kullanıldığını görüyorsunuz. `up3` nesnesi ise `deleter` türü olarak `FileCloser` `functor` sınıfını kullanıyor.

**unique_ptr nesnelerinin kaplarda tutulması**

Dinamik ömre sahip nesneleri `STL` kaplarında tutmanın bir yolu da `unique_ptr` sınıf şablonunu kullanmak. Aşağıdaki kodu inceleyelim:

```
#include <iostream>
#include <memory>
#include <string>
#include <vector>

using UpStrvec = std::vector<std::unique_ptr<std::string>>;

int main()
{
 using namespace std;

 unique_ptr<string> up{ new string {"kayhancan"} };
 UpStrvec myvec;

 myvec.push_back(move(up));
 myvec.emplace_back(new string{ "necati" });
 myvec.push_back(unique_ptr<string>{new string{ "kaan" }});
 myvec.push_back(make_unique<string>(*myvec[0], 1, 5));

 for (auto &up : myvec)
  cout << *up << "\n";

}
```

Yukarıdaki kodda önce içinde `std::unique_ptr<std::string>` sınıf nesnleri tutacak `std::vector` sınıf şablonu açılımına `UpStrvec` eş ismi veriliyor. `main` işlevi içinde `myvec` isimli bir `vector` nesnesinin oluşturulduğunu görüyorsunuz. `vector` sınıfının `push_back` ve `emplace_back` işlevleriyle `vector`'e sırasıyla kayhancan, `necati`, `kaan` ve `ayhan` isimleri ekleniyor. `myvec` nesnesi için sonlandırıcı işlev çağrıldığında kapta tutulmakta olan `unique_ptr` nesnelerinin de sonlandırıcı işlevleri çağrılacak ve böylece dinamik `string` nesneleri `delete` edilecek.

**unique_ptr<> nesnelerinin karşılaştırılması**

`unique_ptr<>` nesneleri karşılaşırma operatörleriyle karşılaştırılabilir. Bu durumda karşılaştırılan `unique_ptr<>` nesnelerinin sarmaladığı adreslerdir:

```
#include <memory>
#include <iostream>

int main()
{
 using namespace std;

 auto up1 = make_unique<int>();
 auto up2 = make_unique<int>();

 auto ptr1 = up1.get();
 auto ptr2 = up2.get();

 cout << "ptr1 = " << ptr1 << "\n";
 cout << "ptr2 = " << ptr2 << "\n";
 cout << (ptr1 == ptr2) << "\n";
 cout << (ptr1 == ptr2) << (up1 == up2) << "\n";
 cout << (ptr1 != ptr2) << (up1 != up2) << "\n";
 cout << (ptr1 < ptr2) << (up1 < up2) << "\n";
 cout << (ptr1 > ptr2) << (up1 > up2) << "\n";
}
```

© 2020 GitHub, Inc.