std::optional Sınıfı

`C++17` standartları ile standart kütüphanemizin bir eksiği daha tamamlandı. Artık bizim de bir `optional` sınıfımız var. Bu yazımızda std::optional sınıfını ayrıntılı olarak ele alacağız.

Programlamada sıklıkla karşımıza çıkan bir durum var: Bir koşul sağlandığında bir nesne oluşturup o nesneyi kullanmamız gerekiyor. Ama bu koşul sağlanmadığında ise bir nesneye ihtiyacımız kalmıyor dolayısıyla bir nesne oluşturmamız gerekmiyor. İşte `std::optional` sınıfı böyle durumlarda kullanılıyor.

`std::optional<T>` türünden bir nesne, programın çalışma zamanının belirli bir noktasında `T` türünden bir değer tutuyor ya da tutmuyor durumda olabilir. optional nesnesinin bir değere sahip olması kadar bir değere sahip olmaması da son derece doğal bir durum. `std` isim alanı içinde tanımlanan `optional` sınıf şablonunun bildirimi şöyle:

```
template<typename T>
class optional;
```

Burada T, `optional` nesnesinin tutabileceği nesnenin türü.

Peki böyle bir sınıfın gerçekleştirimi nasıl yapılabilir dersiniz? `std::optional` sınıfı türünden bir nesne aslında `T` türünden bir nesne için kullanılacak bir bellek alanına ve bir de `bool` değişkeni tutacak bellek alanına sahip. `bool` türden değişken `std::optional` nesnesinin `T` türünden bir nesneye sahip olup olmadığını gösterecek bir bayrak olarak kullanılıyor.

Herhangi türden bir nesneyi `bool` türden bir bayrak değişken ile birlikte bir sınıf oluşturacak şekilde sarmalayabiliriz. Sarmalanmış yapı içindeki bayrak öğesi, kullanıcı kodlara bir değerin tutulduğu ya da tutulmadığı konusunda bilgi verebilir. İngilizcede bu şekilde oluşturulmuş türlere `"nullable types"` deniyor. Böyle bir türden bir nesnenin kullanıcıları, bir yorum satırına gerek kalmaksızın, nesnenin bir değer tutup tutmadığını sorgulayabilirler. Programlama dünyasında opsiyonel türler yeni değil. Örneğin `Haskel` dilinde yer alan `Data.maybe` opsiyonel türlerin en eskilerinden biri. `optional` sınıfı `2003` yılından bu yana `Boost` kütüphanesinin kullanılan öğelerinden biri. Standart kütüphanenin `optional` sınıfının tasarımında da `Boost` kütüphanesinin deneyiminden büyük ölçüde faydalanılmış.

`std::optional` bir değer türü `(value type)` oluşturuyor. Yani kopyalama işlemleri ile aynı değere sahip farklı

nesneler oluşturulabiliyor. Diğer taraftan bir `optional` nesnesi tutacağı değer için kendisi için ayrılmış bir bellek alanınını kullanıyor. Yani dinamik bir bellek alanı kullanmıyor. Aşağıdaki kodu derleyin ve ekran çıktısını yorumlamaya çalışın:

```
#include <optional>
#include <iostream>

template<size_t n>
class A {
 unsigned char buffer[n];
};

template<size_t n>
using optype = std::optional<A<n>>;


int main()
{
 std::cout << sizeof(optype<128>) << '\n';
 std::cout << sizeof(optype<256>) << '\n';
 std::cout << sizeof(optype<512>) << '\n';
 std::cout << sizeof(optype<1024>) << '\n';
 std::cout << sizeof(optype<2048>) << '\n';
}
```

`optional` nesnelerinin oluşturulması

`optional` nesnelerinin oluşturulması için birden fazla yol var. Bunlardan biri, bir nesne hayata getirmeyen yani değer tutmayan (boş) bir `optional` nesnesi oluşturmak:

```
#include <optional>

int main()
{
 std::optional<int> op1;
 std::optional<int> op2{};
 std::optional<int> op3{std::nullopt };
 //...
}
```

Yukarıdaki kodda `optional<int>` türünden `op1, op2` ve `op3` isimli nesneler hayata boş olarak getiriliyorlar. `op3` için çağrılan kurucu işleve argüman olarak `"std::nullopt"` ifadesinin gönderildiğini görüyorsunuz. `<optional>` başlık dosyasında `nullopt_t` isimli bir boş sınıf `(empty class)` tanımlanmış. `nullopt`, bu boş sınıf türünden oluşturulan ve sabit ifadesi olarak kullanılabilen `constexpr` bir sınıf nesnesi. `optional` sınıfının `nullopt_t` türünden kurucu işlevi, `nullopt` sabiti ile çağrıldığında bu kurucu işlev boş bir optional nesnesi hayata getiriyor. Yine bir `optional` değişkenine bu sabitin atanması `optional` nesnesinin sarmaladığı değişkenin hayatını sonlandırıyor, böylece `optional` nesnesi boşaltılmış oluyor:

```
#include <optional>

int main()
{
 std::optional<int> op{ 12 };
 //op nesnesi boş değil

 op = std::nullopt;
 //op nesnesi boş durumda
}
```

Bir `optional` nesnesini dolu olarak da hayata getirebiliriz. Aşağıdaki koda bakalım:

```
#include <optional>
#include <string>

int main()
{
 using namespace std::literals;

 std::optional op1{ 34.5 };
 std::optional op2{"necati"}; //optional<const char *>
 std::optional op3{ "oguz"s }; //optional<string>
 //...
}
```

`op1, op2`, ve `op3` nesnelerinin tanımında şablon tür argümanının kullanılmadığını görüyorsunuz. Burada `C++17` standartları ile dile eklenen ve popüler olarak `CTAD (constructor template argument deduction)` diye isimlendirilen özellik kullanılıyor. Bu mekanizma ile derleyici sınıfın kurucu işlevine gönderilen argümanın türünden hareketle şablon tür argümanının çıkarımını yapabiliyor. Yukarıdaki kodda derleyici `op1` nesnesine ilk değer veren ifadeden hareketle `op1` nesnesinin türünün çıkarımını

```
std::optional<int>
```

olarak yapıyor. Benzer şekilde `op2` nesnesinin türü için

```
std::optional<const char *>
```

`op3` nesnesinin türü için de

```
std::optional<std::string>>
```

çıkarımları yapılıyor. Şüphesiz `optional` nesnesini oluştururken şablon tür argümanını istediğimiz gibi belirleyebiliriz:

```
#include <optional>
#include <string>
#include <complex>
#include <vector>

int main()
{
 std::optional<int> op1{ 12 };
 std::optional<std::string> op2{"irfan"};
 std::optional<std::complex<double>> op3{ std::complex{1., 2.} };
 std::optional<std::vector<int>> ivec{ {1, 3, 5, 7, 9} };
 //...
}
```

Ancak `optional` nesnesinin kurucu işlevine birden fazla değer gönderilecek ise bu durumda şablon tür argümanı belirtilse dahi çıkarım yapılamıyor. Bu yüzden optional sınıfının kurucu işlevine birden fazla argümanın gönderilmesi durumunda, tür çıkarımın yapılabilmesi için ilk argüman olarak `in_place` ifadesinin gönderilmesi gerekiyor. `std::in_place` standart `<utility>` başlık dosyasında tanımlanmış olan `in_place_t` isimli bir boş sınıf türünden `constexpr` bir nesnenin ismi. Bu tür boş sınıfların ve boş sınıf nesnelerinin varlık nedeni derleyicinin çıkarım yapmasına olanak sağlamak. Aşağıdaki koda bakalım:

```
#include <optional>
#include <complex>
#include <set>
#include <cctype>

int main()
{
 //std::optional<std::complex<double>> op1{1, 2}; //gecersiz
 std::optional<std::complex<double>> op2{std::in_place, 1, 2 };
 auto f = [](char x, char y) {
  return std::toupper(x) > std::toupper(y);
 };
 std::optional<std::set<char, decltype(f)>> op3{ std::in_place,
                                                 {'c', 'T', 'a', 'B'}, f };
}
```

`make_optional` işlevi

`optional` nesnelerini oluşturmanın bir başka yolu da `make_optional` isimli global yardımcı işlevi çağırmak. Bu işleve birden fazla argüman geçsek de artık `in_place` nesnesini işleve göndermek zorunda değiliz:

```
#include <optional>
#include <complex>

int main()
{
 auto op1 = std::make_optional(12); //optional<int>
 auto op2 = std::make_optional("neco"); //optional<const char *>
 auto op3 = std::make_optional<std::complex<double>>(1.2, 5.6);
        //op3 nesnesinin turu : optional<complex<double>>
}
```

optional nesnelerinin boş olup olmadığını sınamak

Bir `optional` nesnesinin boş olup olmadığını yani bir değer tutup tutmadığını sınıfın `operator bool` ya da `has_value` isimli işlevleriyle sınayabiliriz:

```
constexpr explicit operator bool() const noexcept;
constexpr bool has_value()const noexcept;
```

Bir `optional` nesnesini `nullopt` değeriyle eşitlik/eşitsizlik karşılaştırmasına da sokabiliriz:

```
#include <optional>

std::optional<int> func();

int main()
{
 auto op = func();

 if (op) {/**/} //dolu ise
 if (op.has_value()) {/**/ } //dolu ise
 if (op != std::nullopt) {/**/ } //dolu ise

 if (!op) {/**/ } //bos ise
 if (!op.has_value()) {/**/ } //bos ise
 if (op == std::nullopt) {/**/ } //bos ise
 //...
}
```

optional nesnesinin tuttuğu değere erişmek

`optional` nesnesinin tuttuğu değere erişmenin yine birden fazla yolu var. Sınıfın içerik operatör ve ok operatör fonksiyonları ile tutulan nesneye ya da o nesnenin öğelerine erişebiliriz. Ancak bu operatörlerin terimi olan optional nesnesinin boş olması durumunda tanımsız davranış `(undefined behavior)` oluşuyor. Böyle bir erişimde bir hata nesnesi gönderilmiyor `(exception throw edilmiyor)`. Aşağıdaki koda bakalım:

```
#include <optional>
#include <string>
#include <iostream>

int main()
{
 std::optional<std::string> op{ "kaan" };
 std::cout << *op << "\n";
 *op += " aslan";
 std::cout << *op << "\n";
 std::cout << "uzunluk : " << op->size() << "\n";
 op = std::nullopt;
 std::cout << *op << "\n"; //tanımsız davranış
 //...
}
```

Yukarıdaki koddan da görüldüğü gibi operator `"*"` işlevi referans döndürüyor.

Tutulan nesneye güvenli bir şekilde erişim gerçekleştirmek için öncelikle `optional` nesnesinin boş olmadığından emin olmalıyız:

```
#include <optional>
#include <iostream>

std::optional<int> func();

int main()
{
 if (auto op = func())
  std::cout << *op << "\n";

 //...
 if (auto op = func(); op)
  std::cout << *op << "\n";
 //...

 if (auto op = func(); op.has_value())
  std::cout << *op << "\n";

 if (auto op = func(); op.has_value())
  std::cout << *op << "\n";
 //...

}
```

`value` işlevi

Tutulan nesneye erişmenin bir başka yolu da sınıfın value isimli üye işlevini çağırmak. `operator *` işlevi gibi `value` işlevi de tutulan nesneye referans döndürüyor. Boş bir `optional` nesnesi için value işlevinin çağrılması durumunda, `std::exception` sınıfınından kalıtım yoluyla elde edilen `std::bad_optional_access` türünden bir hata nesnesi gönderiliyor:

```
#include <optional>
#include <string>
#include <iostream>

int main()
{
 std::optional<std::string> op{ "oguz karan" };

 std::cout << op.value() << "\n";
 op.value().assign(5, 'A');
 std::cout << op.value() << "\n";
 op = std::nullopt;

 try {
  std::cout << op.value() << "\n";
 }
 catch (const std::bad_optional_access &ex) {
  std::cout << "hata yakalandi : " << ex.what();
 }
}
```

`value_or` işlevi

Tutulan değere erişmenin bir başka yolu da sınıfın `value_or` isimli işlevini çağırmak. Bu işlev `value` işlevinden farklı olarak bir argüman alıyor ve `optional` nesnesinin boş olması durumunda kendisine gelen bu değeri döndürüyor:

```
#include <optional>
#include <string>
#include <iostream>

void display_e_mail(const std::optional<std::string> &op)
{
 std::cout << "e posta adresi : " << op.value_or("belirtilmemis") << "\n";
}

int main()
{
 std::optional<std::string> e_mail_address{ "necati@gmail.com" };
 display_e_mail(e_mail_address);
 e_mail_address = std::nullopt;
 display_e_mail(e_mail_address);
 //...
}
```

`value` işlevinden farklı olarak `value_or` işlevi referans döndürmüyor:

```
#include <optional>

int main()
{
 std::optional<int> op{ 10 };

 op.value() = 20;
 op.value_or(0) = 30; //gecersiz
 //...
}
```

`optional` nesnelerinin değerlerini değiştirmek

`std::optional<T>` sınıfı türünden bir nesnenin değerini değiştirmek için sınıfın atama operatörlerini kullanabiliriz. Atama operatörünün sağ terimi olan ifade

- `optional<T>` türünden olabilir.
- `optional<U>` türünden olabilir. (`U` türünden `T` türüne dönüşüm var ise)
- `T` türünden olabilir.
- `U` türünden olabilir. (`U` türünden `T` türüne dönüşüm var ise)
- `std::nullopt` değeri olabilir.
- `{}` ifadesi olabilir.

Aşağıdaki kodu inceleyelim:

```
#include <optional>
#include <iostream>


int main()
{
 std::optional<double> d1{ 2.56 };
 std::optional<double> d2{ 0.78};
 std::optional<int> i{ 40 };

 std::cout << *d1 << '\n';
 d1 = d2;
 std::cout << *d1 << '\n';
 d1 = i;
 std::cout << *d1 << '\n';
 d1 = 5.69;
 std::cout << *d1 << '\n';
 d1 = 13;
 std::cout << *d1 << '\n';
 d1 = std::nullopt;
 d2 = {};

 if (!d1 && !d2)
  std::cout << "her iki nesnenin de degeri yok\n";
}
```

`emplace` işlevi

`optional` sınıfının en önemli işlevlerinden biri `emplace`. Bu işlev ile bir nesneyi kopyalama olmadan doğrudan `optional` nesnesi içinde yer alan bellek alanında hayata başlatabiliyoruz. `emplace` işlevi standart kütüphanedeki kap sınıflarının `emplace` işlevlerinde olduğu gibi mükemmel gönderim `(perfect forwarding)` mekanizmasından faydalanıyor. Dolu bir `optional` nesnesi için emplace işlevi çağrıldığında `optional` nesnesi tutmakta olduğu nesnesin sonlandırıcı işlevini `(destructor)` çağırıyor:

```
#include <iostream>
#include <optional>

class A {
public:
 ~A() { std::cout << "~A() this: " << this << "\n"; }
 A() { std::cout << "A() this : " << this << "\n"; }
 A(int) { std::cout << "A(int) this : " << this << "\n"; }
 A(int, int) { std::cout << "A(int, int) this : " << this << "\n"; }
 A(double) { std::cout << "A(double) this : " << this << "\n"; }
 A(std::string) { std::cout << "A(string) this : " << this << "\n"; }
};

int main()
{
 std::optional<A> os;
 std::cout << "&os = " << &os << "\n";
 os.emplace();
 os.emplace(10);
 os.emplace(10, 20);
 os.emplace(4.5);
 os.emplace("necati");
}
```

Yukarıdaki kodu derleyip çalıştırın. `optional` nesnesi dolu iken `emplace` işlevi her çağrıldığında önce `A` sınıfının sonlandırıcı işlevinin çağrıldığını daha sonra ise `A` sınıfının uygun kurucu işlevinin çağrıldığını göreceksiniz.

`optional` nesneleri tarafından kontrol edilen nesnelerin ömürleri

Bir `optional` nesnesinin hayatı bittiğinde `optional` nesnesi dolu ise hayata getirilmiş nesnenin sonlandırıcı işlevi çağrılır. Ancak aşağıdaki durumlarda da `optional` nesnesinin kontrol ettiği nesnenin sonlandırıcı işlevi çağrılır:

- `optional` nesnesinin `emplace` isimli işlevinin çağrılması
- `optional` nesnesine `nullopt` değerinin atanması
- `optional` nesnesine `{}` ifadesinin atanması
- `optional` nesnesinin reset işlevinin çağrılması (sınıfın `reset` isimli işlevinin çağrılmasıyla eğer `optional` nesnesi boş değil ise kontrol edilen nesnenin ömrü sonlandırılır.)

```
#include <iostream>
#include <optional>

class A {
 int mx;
public:
 ~A() { std::cout << "~A() mx = " << mx << "\n"; }
 A(int x) : mx{ x } {}
};

int main()
{
 std::optional<A> os(50);
 os.reset();
 os = 20;
 os = std::nullopt;
 os = 30;
 os = {};
 os = 40;
 os.emplace(50);
 os.emplace(60);
}
```

optional sınıfı ve taşıma semantiği

`optional` sınıfı taşıma semantiğini de destekliyor. Bir `optional` nesnesini başka bir `optional` nesnesine taşıyabiliyoruz. Bu durumda içerilen bir nesne var ise o da taşınıyor. Aşağıdaki kodu derleyip çalıştırın:

```
#include <optional>
#include <iostream>

struct A {
 A() { std::cout << "A default ctor.()\n"; }
 A(const A &) = delete;
 A& operator=(const A &) = delete;
 A(A &&) { std::cout << "A move ctor()\n"; }
 //...
};

int main()
{
 std::optional<A> op1;
 op1.emplace(); //A default ctor.
 std::optional<A> op2{ std::move(op1) }; //A's move ctor.
 std::cout << std::boolalpha;
 std::cout << op1.has_value() << "\n";
 std::cout << op2.has_value() << "\n";
}
```

Yukarıdaki kodda, `op1` nesnesinin taşınması ile `op1` 'in içerdiği `A` nesnesi taşınmış oluyor. Taşıma işleminden sonra `op1` nesnesi dolu olsa da içerdiği nesne taşınmış durumda `(moved-from state)`. İçerilen nesneye yeniden bir değer atamadan bu nesne yeniden kullanılmamalı. İçerilecek nesneyi dışarıdan içeriye ya da içerilen nesneyi içeriden dışarıya taşımak da mümkün. Aşağıdaki koda bakalım:

```
int main()
{
 A ax;

 std::optional<A> op1{ std::move(ax) };
 std::optional<A> op2{ std::move(*op1) };
}
```

Yukarıdaki kodu derleyip çalıştırdığınızda `A` sınıfının taşıyan kurucu işlevinin `(move constructor)` iki kez çağrıldığını göreceksiniz.

optional nesneleri ve karşılaştırma işlemleri

optional türünden bir nesne a) optional türünden bir nesne ile b) optional türünden bir nesne ile (eğer T ve U karşılaştırılabilir türler ise) b) T türünden bir ifade ile d) U türünden bir ifade ile (eğer T ve U karşılaştırılabilir türler ise) c) std::nullopt değeri ile

karşılaştırılabilir. Karşılaştırılan değerler optional nesnelerinin tuttuğu değerlerdir. Boş bir optional nesnesi değeri ne olursa olsun dolu bir optional nesnesinden daha küçük kabul edilir. İki boş optional nesnesinin karşılaştırılmasından true değeri elde edilir. Aşağıdaki kodda yapılan karşılaştırma işlemlerini inceleyiniz:

```
#include <optional>
#include <iostream>

using namespace std;

int main()
{
 optional<int> oe;
 optional<int> ox{ 10 };
 optional<int> oy{ 20 };

 cout.setf(ios::boolalpha);

 cout << (oe == ox) << '\n'; //false
 cout << (oe == nullopt) << '\n'; //true
 cout << (oe < ox) << '\n'; //true
 cout << (ox > oy) << '\n'; //false
 cout << (ox == 10) << '\n'; //true

 optional<unsigned> oz;
 optional<unsigned> omin{ 0 };
 cout << (oz < omin) << '\n'; //true
}
```

`optional<bool>` nesneleri ile yapılan karşılaştırmalara özellikle dikkat edilmeli. Aşağıdaki koda da bakalım:

```
#include <optional>
#include <iostream>

using namespace std;

int main()
{
 optional<bool> oe{ nullopt };
 optional<bool> ox{ false };
 optional<bool> oy{ true};


 cout.setf(ios::boolalpha);

 cout << (oe == ox) << '\n';  //false
 cout << (oe == oy) << '\n';  //false
 cout << (oe < ox) << '\n';  //true
 cout << (oe < oy) << '\n';  //true
 cout << (oe == true) << '\n';  //false
 cout << (oe == false) << '\n';  //false
 cout << (ox == oy) << '\n';  //false
}
```

optional sınıfının kullanıldığı tipik durumlar

- Bir işlevin `optional` sınıfı türünden bir değer döndürmesi. Bazı işlevler bir koşula bağlı olarak bir değer döndürebilirler. Ancak koşul sağlanmadığında döndürecek değerleri olmayabilir. Yani işlevin bir değer döndürmesi kadar döndürmemesi de doğaldır. Bu tür durumlarda işlevin geri dönüş değeri optional sınıfı türünden olabilir. Aşağıdaki kodu inceleyelim:

```
#include <optional>
#include <string>
#include <iostream>

std::optional<int> to_int(const std::string& s)
{
 try {
  return std::stoi(s);
 }
 catch (...) {
  return std::nullopt;
 }
}

int main()
{
 for (auto ptr : { "987", "007bond", "ali", "23haziran", "2312126" })
  if (auto op = to_int(ptr))
   std::cout << ptr << " icin tamsayi degeri: " << *op << "\n";
  else
   std::cout << ptr << " gecerli tamsayi icermiyor\n";
}
```

Yukarıdaki kodda tanımlanan `to_int` isimli işlev, bir `std::string` nesnesini bir tamsayıya dönüştürüyor. Ancak işleve gönderilen yazının geçerli bir tamsayı ifade etmemesi durumunda işlevimizin geri döndüreceği bir tamsayı olamayacak. Bu yüzden işlevin geri dönüş değeri türünün

```
std::optional<int>
```

olarak seçildiğini görüyorsunuz. İşlev gelen yazıdan bir tamsayı elde edilemesi durumunda boş bir `optional<int>` nesnesi döndürüyor. `to_int` isimli işlevi aşağıdaki gibi de tanımlayabilirdik:

```
std::optional<int> to_int(const std::string& s)
{
 std::optional<int> retval;

 try {
  retval = std::stoi(s);
 }
 catch (...) {}

 return retval;
}
```

- Bir işlevin parametre değişkeninin `optional` sınıfı türünden olması. Bir işlevin bir parametresine, müşteri kodun bir değer göndermesi kadar değer göndermemesi de doğal bir durum ise işlevin parametre değişkeni `optional` sınıfı türünden yapılabilir. Bu durumda işlevi çağıracak kod bu parametreye ya bir değer ya da `std::nullopt` sabitini gönderebilir.
- Bir sınıfın bir veri öğesinin optional sınıfı türünden olması.

Aşağıdaki kodda hem bir işlevin parametresinin hem de bir sınıfın bir veri öğesinin `std::optional` türünden olduğunu göreceksiniz:

```
#include <optional>
#include <iostream>
#include <string>

class Name
{
private:
 std::string m_first;
 std::optional<std::string> m_middle;
 std::string m_last;
public:
 Name(std::string first,
  std::optional<std::string> middle,
  std::string last)
  : m_first{ std::move(first) }, m_middle{ std::move(middle) },
  m_last{ std::move(last) } {
 }

 friend std::ostream& operator << (std::ostream& os, const Name& name) {
  os << name.m_first << ' ';
  if (name.m_middle) {
   os << *name.m_middle << ' ';
  }
  return os << name.m_last;
 }
};
```

Yukarıdaki kodda kişilerin isimlerini, temsil etmek amacıyla `Name` isimli bir sınıfın tanımlandığını görüyorsunuz. Sınıfın `std::optional<std::string` türünden `m_middle` isimli veri öğesi kişilerin sahip olabileceği ya da sahip olmayacağı ikinci isimlerini tutması için tanımlanmış.