std::bind işlev uyumlandırıcısı

`C++11` ile standart kütüphaneye eklenen ve `functional` başlık dosyasında yer alan `bind` işlev şablonu genel amaçlı bir işlev uyumlandırıcısı `(function adapter)`. `std::bind`, standart kütüphanede önceden beri var olan, kullanımı daha zahmetli ve biraz da sorunlu olan, `ptr_fun`, `mem_fun`, `mem_fun_ref`, `bind1st` ve `bind2nd` uyumlandırıcılarının yerine geldi. `bind` uyumlandırıcısı bunların görevlerinin tamamını tek başına yerine getirebiliyor. `bind` işlevinin dile eklenmesiyle bu uyumlandırıcılar artık standartlar tarafından eskimiş `(deprecated)` kabul edildi. `bind` uyumlandırıcısı, bir işlev göstericisi `(function pointer)` ve bu işlev göstericisinin gösterdiği işleve yapılacak çağrıda arguman olarak kullanılacak değerleri veri öğelerinde tutan bir işlev nesnesi `(function object)` oluşturuyor. `bind`'ın oluşturduğu sınıf nesnesinin işlev çağrı operatörü, veri öğesi olan işlev göstericisinde adresi tuturulan işlevi, yine veri öğelerinde tutulan değerlerle çağırıyor. Basit bir örnekle başlayalım:

```
#include <iostream>
#include <functional>

int sum(int a, int b, int c)
{
 int sum = a + b + c;
 std::cout << a << '+' << b << '+' << c << '=' << sum << "\n";

 return sum;
}

int main()
{
 int x = 10, y = 20, z = 30;
 auto f = std::bind(sum, x, y, z);

 std::cout << f() << "\n";
}
```

Yukarıdaki kodda `func`, `3` parametre değişkenine ve `int` türden bir geri dönüş değerine sahip bir işlev.

```
auto f = bind(sum, x, y, z)
```

deyimi ile `bind` tarafından oluşturulan işlev nesnesi ile `f` değişkenine ilk değer vermiş oluyoruz. Yani `f` aslında `bind` işlevi tarafından oluşturulan işlev nesnesi. `f` işlev nesnesi kendi veri öğelerinden birinde `sum` işlevinin adresini tutuyor. Yine `f` işlev nesnesi `3` veri öğesinde ise `x`, `y` ve `z` değişkenlerinin değerlerini tutuyor.

```
f()
```

ifadesi ile `f` işlev nesnesi işlev çağrı işlecinin terimi yapılıyor. Bu durumda çağrılan `f` nesnesinin işlev çağrı işlevi, adresi tutulan sum işlevini `x`, `y` ve `z` değişkenlerinin değerleriyle çağırıyor. Bir başka deyişle `x, y` ve `z` değişkenlerinin değerleri `f` nesnesi tarafından `sum` işlevine argüman olarak gönderilmek üzere bağlanmış oluyor. Yani yukarıdaki çağrı ile

```
sum(x, y, z)
```

arasında sonuç olarak bir fark yok. `bind` işlevinin geri döndürdüğü işlev nesnesi ile bir değişkene ilk değer vermek yerine, bu işlev nesnesini doğrudan işlev çağrı işlecinin terimi yaparak da `sum` işlevinin çağrılmasını sağlayabilirdik:

```
std::cout << bind(sum, x, y, z)() << "\n";
```

`bind` işlevine gönderilen argümanlar `bind` işlevi tarafından oluşturulan işlev nesnesine kopyalanıyor ve `bind` nesnesinin işlev çağrı işlecinin terimi yapılmasıyla, bu nesnelerin değerlerinin `bind` tarafından adresi tutulan sarmalanan işleve argüman olarak gönderilmesi sağlanıyor. `bind` uyumlandırıcısının adresini tuttuğu işlevin bir parametre değişkeni bir referans ise bu parametreye bu yolla bir nesnenin kendisini göndermemiz mümkün değil:

```
#include <iostream>
#include <functional>


void func(int a, int &b, int &c)
{
 b *= a;
 c += b;
}

int main()
{
 int x = 10, y = 20, z = 30;
 auto f = std::bind(func, x, y, z);
 f();
 std::cout << "y = " << y << "\n";
 std::cout << "z = " << z << "\n";

}
```

Yukarıdaki programı derleyip çalıştırdığınızda `y` ve `z` değişkenlerinin değerlerinin değişmediğini göreceksiniz. `bind` tarafından oluşturulan işlev nesnesinin değişkenlerimizin değerini değil de kendilerini kullanmasını istiyorsak bunları `bind` işlevine gönderirken standart kütüphanenin `ref` ya da `cref` işlevlerini kullanmamız gerekiyor:

```
#include <iostream>
#include <functional>

int main()
{
 using namespace std;

 int x = 10, y = 20, z = 30;
 auto f = bind(func, x, ref(y), ref(z));
 f();
 cout << "y = " << y << "\n";
 cout << "z = " << z << "\n";

}
```

Kopyalanamayan sınıf nesnelerini de `bind` işlevine geçebilmek için yine `ref` işlevini kullanmak zorundayız. Aşağıdaki koda bakalım:

```
#include <iostream>
#include <functional>

void dprint(std::ostream& os, double d)
{
 os << d << "\n";
}

int main()
{
        using namespace std;

 //auto f = bind(dprint, cout, 3.4);  gecersiz
 auto f = bind(dprint, ref(cout), 3.4);
 f();
}
```

`bind` işlevine `cout` nesnesini argüman olarak göndermemiz sentaks hatası oluştururdu. `std::ostream` sınıfının kopyalayan kurucu işlevinin `delete` edilmiş olduğunu hatırlayalım. (`ref` ve `cref` işlevlerini ve bu işlevin geri dönüş değeri olan `reference_wrapper` sınıfını bir başka yazımızda ele alacağız.)

placeholders nesneleri

`bind` işlevine geçilen bağlanmış argümanlar listesinde yer tutucu `(placeholders)` denilen özel değişkenleri kullanabiliyoruz. Yer tutucu değişkenler isim çakışmasından kaçınmak için `std` isim alanı içinde yer alan `placeholders` isimli bir içsel isim alanında `(nested namespace)` tanımlanmış ve kodun kolay anlaşılmasını sağlamak için `_1, _2, _3, ...` biçiminde isimlendirilmişler.

Bu değişkenleri

```
std::placeholders::_1
```

yerine doğrudan isimleriyle yani

```
_1
```

biçiminde kullanabilmek için bir `"using namespace"` bildirimi yapabiliriz:

```
using namespace std::placeholders;
```

Yer tutucu nesnelerinin kullanım notasyonunu anlamak başlangıçta biraz zor olabilir. Bir yer tutucu değişkenin ismi, bağlanacak argüman listesinde hangi sırada kullanılmış ise, işlev nesnesi ile yapılan çağrıda bu ismin işaret ettiği sırada kullanılan argümanın, sarmalanan işlevin bu sıradaki parametresine argüman olarak geçileceği anlamına geliyor. Örneğin, `_1` ismi `bind`'a geçilen bağlanacak argümanlar listesinde üçüncü sırada kullanılmış ise oluşturulan işlev nesnesi ile yapılan çağrıda kullanılan birinci argüman sarmalanan işlevin üçüncü parametresine aktarılır. `_2` ismi `bind`'a geçilen bağlanacak argümanlar listesinde birinci sırada kullanılmış ise oluşturulan işlev nesnesi ile yapılan çağrıda kullanılan ikinci argüman sarmalanan işlevin birinci parametresine aktarılır.

```
#include <iostream>
#include <functional>

using namespace std;
using namespace placeholders;

void func(int x, int y, int z)
{
 cout << x << "  " << y << " " << z << "\n";
}

int main()
{
 auto f1 = bind(func, 10, 20, 30);
 f1(); //10 20 30
 auto f2 = bind(func, _1, _2, 5);
 f2(10, 20); //  10 20 5
 auto f3 = bind(func, 10, _2, _1);
 f3(3, 7); //10 7  3
 auto f4 = bind(func, _3, _1, _2);
 f4(4, 6, 8); // 8 4 6
}
```

`f1` işlev nesnesi için yapılan `bind` çağrısında, `func` işlevine bağlanacak bağlanmış argümanlar listesinde `10, 20, 30` değerleri kullanılmış. Bu durumda `f1` işlev nesnesi ile çağrı yapıldığında, `func` işlevinin birinci parametresine `10`, ikinci parametresine `20` ve üçüncü parametresine `30` değerleri gönderilecek. Bu örnekte bir yer tutucu kullanılmamış. Bu durumda eğer `f1` ile yapılan çağrıda bir ya da birden fazla sayıda argüman kullanılsaydı sentaks hatası oluşurdu.

`f2` işlev nesnesi için yapılan `bind` çağrısında, `func` işlevine bağlanacak argümanlar listesinde birinci sırada `_1` ismi yer alıyor. Bu, `f2` işlev nesnesi ile yapılacak işlev çağrısında kullanılan birinci argümanın `func` işlevinin birinci parametresine gönderileceği anlamına geliyor. İkinci sırada ise `_2` isminin kullanıldığını görüyorsunuz. Bu da `f2` işlev nesnesi ile yapılacak işlev çağrısında kullanılan ikinci argümanın `func` işlevinin ikinci parametresine gönderileceği anlamına geliyor. `f2` ile yapılacak çağrıda `func` işlevinin üçüncü parametresine ise `5` değeri gönderilecek. `f2` işlev nesnesi ile yapılan çağrıda `2`'den az ya da `2`'den fazla sayıda argüman kullanılsaydı sentaks hatası oluşurdu.

`f3` işlev nesnesi için yapılan `bind` çağrısında, `func` işlevine bağlanacak argümanlar listesinde birinci sırada `10` değeri yer alıyor. Bu, `f1` ile yapılacak işlev çağrısında `func` işlevinin birinci parametresine `10` değerinin gönderileceği anlamına geliyor. İkinci sırada ise `_2` isminin kullanıldığını görüyorsunuz. Bu da `f3` işlev nesnesi ile yapılacak işlev çağrısında kullanılan ikinci argümanın `func` işlevinin ikinci parametresine gönderileceği anlamına geliyor. Bağlanacak argümanlar listesinde üçüncü sırada ise `_1` isminin yer aldığını görüyorsunuz. Bu da `f3` işlev nesnesi ile yapılacak işlev çağrısında kullanılan ilk argümanın `func` işlevinin üçüncü parametresine gönderileceği anlamına geliyor.

`f4` işlev nesnesi için yapılan `bind` çağrısında, `func` işlevine bağlanacak argümanlar listesinde birinci sırada `_3` ismi yer alıyor. Bu, `f4` ile yapılacak işlev çağrısında kullanılan üçüncü argümanın `func` işlevinin

birinci parametresine gönderileceği anlamına geliyor. İkinci sırada ise `_1` isminin kullanıldığını görüyorsunuz. Bu da `f4` işlev nesnesi ile yapılacak işlev çağrısında kullanılan birinci argümanın `func` işlevinin ikinci parametresine gönderileceği anlamına geliyor. Son olarak bağlanacak argümanlar listesinde üçüncü sırada ise `_2` isminin yer aldığını görüyorsunuz. Bu da `f4` işlev nesnesi ile yapılacak işlev çağrısında kullanılan ikinci argümanın `func` işlevinin üçüncü parametresine gönderileceği anlamına geliyor. `f4` işlev nesnesi ile yapılan çağrıda `3` 'den az sayıda argüman kullanılsaydı sentaks hatası oluşurdu.

`place_holders` nesneleri bağlanacak argümanlar listesinde birden fazla yerde de kullanılabilir:

```
#include <iostream>
#include <functional>

double func(double a, double b, double c)
{
 return a * b * c;
}

int main()
{
 using namespace std::placeholders;
 auto f = bind(func, _1, _1, _1);

 std::cout << f(1.5) << "\n";
}
```

Yukarıdaki `main` işlevinde `bind` işlevine çağrıda kullanılan bağlanacak argüman listesinde hem birinci, hem ikinci hem de üçüncü sırada `_1` isminin kullanıldığını görüyorsunuz. Bu durumda `f` işlev nesnesiyle yapılan çağrıda kullanılan argüman olan `1.5` değeri `func` işlevinin `3` parametresinin her birine argüman olarak gönderiliyor.

iç içe bind çağrıları

Bir `bind` çağrısına başka bir `bind` çağrısından elde ettiğimiz bir işlev nesnesini geçebiliriz. Aşağıdaki kodu inceleyelim:

```
#include <iostream>
#include <functional>

using namespace std;
using namespace placeholders;

int main()
{
 auto f = bind(plus<int>(), bind(multiplies<int>(), _1, 10), 20);

 cout << f(5) << "\n";
}
```

Bu durumda derleyicinin ürettiği kod ile, içteki `bind` çağrısının ürettiği işlev nesnesinin operatör işlevinin çağrılmasıyla elde edilen değer dıştaki `bind` çağrısının bağlanacak argümanlar listesinde kullanılıyor. Yukarıdaki kodda

```
f(5)
```

çağrısında kullanılan `5` değeri standart `multiplies` sınıfının işlevine birinci argüman olarak, `10` değeri de yine aynı işleve ikinci argüman olarak gönderilecek. Bu işlev çağrısından elde edilen `5o` değeri ise standart `plus` sınıfının işlevine birinci argüman olarak gönderilecek. İçiçe `bind` çağrılarıyla daha karmaşık işlev nesneleri oluşturabiliyoruz.

bind işlevinin algoritmalar ile kullanılması

`bind` işlevine çağrı yapılarak elde edilen bir işlev nesnesi, bir algoritmanın çağrılabilir öğe `(callable)` isteyen bir parametresine argüman olarak geçilebilir. Aşağıdaki örneği inceleyelim:

```
#include <functional>
#include <list>
#include <iostream>
#include <algorithm>

using namespace std;
using namespace placeholders;

int main()
{
 list<int> x{ 2, 12, 9, 2, 7, 8, 5, 3, 7, 1, 6, 4, 8, 9, 0 };

 auto n = count_if(x.begin(), x.end(), bind(greater<int>(), _1, 7));
 cout << n << "\n";
}
```

Yukarıdaki kodda `count_if` algoritmasının üçüncü parametresine

```
bind(greater<int>(), _1, 7)
```

çağrısı ile oluşturulmuş işlev nesnesi gönderiliyor. Bu işlev nesnesinin tek parametreli operator işlevi çağrıldığında bu işleve gönderilen değer, standart `greater<int>()` nesnesinin iki parametreli işlevinin ilk parametresine argüman olarak gönderilecek. `greater<int>()` işlevinin ikinci parametresine ise bağlanacak argüman listesinde kullanılan `7` değeri gönderilecek. Böylece `count_if` algoritması ile liste içinde tutulan öğelerden kaç tanesinin `7`'den büyük olduğunu saymış oluyoruz. Şimdi de aşağıdaki koda bakalım:

```
#include <functional>
#include <vector>
#include <iostream>
#include <algorithm>

using namespace std;
using namespace placeholders;

void dprint(ostream& os, double d)
{
 os << d << "\n";
}

int main()
{
 vector<double>dvec{ 1.2, 3.4, 8.8, 2.5, 6.7, 3.3, 2.7, 5.6 };

 for_each(begin(dvec), end(dvec), bind(dprint, ref(cout), _1));
}
```

`for_each` algoritmasının `bind` tarafından üretilen işlev nesnesinin işlevine gönderdiği değer, bu işlev tarafından `dprint` işlevinin ikinci parametreresine geçiliyor. Kopyalanamayan `cout` nesnesinin işlev nesnesinde tutulabilmesi için standart `ref` işlevi tarafından üretilen `reference_wrapper` nesnesi kullanılıyor.

bind işlevi ve sınıf nesneleri

Aşağıdaki gibi bir sınıfımız olsun:

```
#include <iostream>

class Val {
 int mx;
public:
 Val(int i) : mx{ i }{}
 void print()const { std::cout << mx << "\n"; }
 void set(int i) { mx = i;}
 int get()const  { return mx; }
};
```

Bu sınıfla ilgili `bind` işlevini kullanan bazı kodlar yazalım:

```
#include <functional>
#include <algorithm>

using namespace std;
using namespace placeholders;

int main()
{
 Val x{ 10 };

 auto f1 = bind(&Val::set, x, _1);
 f1(20);
 x.print();

 auto f2 = bind(&Val::set, ref(x), _1);
 f2(30);
 x.print();

 auto f3 = bind(&Val::set, _1, 100);
 f3(x);
 x.print();
}
```

`bind` işleviyle sınıfların `static` olmayan üye işlevlerini de oluşturulacak işlev nesnelerine bağlayabiliriz. Bu durumda çağrılacak üye işlev için kullanılacak `*this` nesnesi bağlanacak argümanlar listesinde birinci sırada kullanılmalıdır: `f1 bind` işlev nesnesinin oluşturulmasında bağlanacak argüman listesinde ilk olarak `Val` sınıfının set isimli üye işlevinin adresinin gönderildiğini görüyorsunuz. Bu işleve yapılacak çağrıda `*this` nesnesi olarak kullanılacak `x` nesnesi bağlanmış argümanlar listesinde ikinci sırada yer alıyor. Oluşturulan işlev nesnesinin işlevine gönderilecek argüman ise set üye işlevinin birinci parametresine geçilecek. Yalnız bu durumda `set` üye işlevi `x` nesnesinin kendisi için değil, değerini kopyalamayla `x` nesnesinden alan bir sınıf nesnesi için çağrılır.

```
f1(20);
```

çağrısı sonucunda `x` nesnesi değişmemiş olur.

`f2 bind` işlev nesnesinin oluşturulmasında ise bağlanacak arguman listesinde bu kez ikinci sırada `ref(x)` ifadesinin yer aldığını görüyorsunuz. Bu durumda

```
f2(30);
```

çağrısı `x` nesnesinin kendisi için yapılır ve `x` nesnesi işleve gönderilen `30` değerini alır.

`f3 bind` nesnesinin oluştulmasında ise bağlanacak argüman listesinde birinci sırada `_1` nesnesinin, ikinci sırada ise `100` sabitinin kullanıldığını görüyorsunuz. Bu durumda `f3` ile yapılacak çağrıya bir `Myclass` nesnesi

gönderilmeli. Sınıfın set işlevinde `*this` olarak bu nesne kullanılacak ve `set` üye işlevine bağlanmış argüman olarak `100` değeri gönderilecek.

```
f3(x);
```

çağrısı sonucunda `x` nesnesinin değeri `100` olur. Şimdi de `Val` sınıfı türünden nesnelerin bir kapta tutulduğunu ve kabın bir aralığı `(range)` için bir algoritmanın çağrıldığını düşünelim:

```
#include <iostream>

class Val {
 int mx;
public:
 Val(int i) : mx{ i }{}
 void print()const { std::cout << mx << "\n"; }
 void set(int i) { mx = i;}
 int get()const  { return mx; }
};

#include <functional>
#include <vector>
#include <algorithm>

using namespace std;
using namespace placeholders;

int main()
{
 vector<Val> vec(10, 0);

 for_each(vec.begin(), vec.end(), bind(&Val::set, _1, 10));
 for_each(vec.begin(), vec.end(), bind(&Val::print, _1));
}
```

`for_each` algoritmasına yapılan çağrıyla `vec` isimli `vector`'de tutulan `Val` sınıf nesnelerinin `set` ve `print` işlevleri çağrılıyor.

Son olarak aşağıdaki örneği inceleyelim:

```
#include <iostream>
#include <functional>
#include <list>
#include <vector>
#include <string>
#include <algorithm>
#include <iterator>

using namespace std;
using namespace placeholders;

int main()
{
 list<string> slist{ "nur", "mert", "mustafa", "hakan" };
 transform(slist.begin(), slist.end(),
                  ostream_iterator<size_t>(cout, " "), bind(&string::size, _1));
}
```

Yukarıdaki kodda `transform` algoritması ile slist isimli `list` nesnesi içinde tutulan `string` nesnelerinin size üye işlevleri çağrılarak bu işlev çağrılarından elde edilen değerler standart çıkış akımına yazdırılıyor. `C++11` öncesi kullanılan işlev uyumlandırıcılarına göre `bind`'ın bir avantajı daha var. Kapta sınıf nesnelerinin değil de sınıf

nesnelerinin adreslerinin tutulması durumunda da, bir algoritmanın adresleri tutulan sınıf nesnelerinin üye işlevlerini çağırması için yine aynı uyumlandırıcıyı yine aynı şekilde kullanıyoruz. Derleyici gereken kodu uygun şekilde derleme zamanında oluşturuyor:

```
#include <iostream>
#include <functional>
#include <list>
#include <string>
#include <algorithm>
#include <iterator>

using namespace std;
using namespace placeholders;

int main()
{
 list<string *> slist;
 slist.push_back(new string{ "ali" });
 slist.push_back(new string{ "onur" });
 slist.push_back(new string{ "metin" });

 transform(slist.begin(), slist.end(),
  ostream_iterator<size_t>(cout, " "), bind(&string::size, _1));
 //
}
```

Özellikle `C++14` ve `C++17` ile lambda ifadelerine getirilen ek özelliklerle, artık `bind` işlev uyumlandırıcısı ile yapabildiğimiz her işi `lambda` ifadeleri ile de gerçekleştirebiliyoruz. Bir başka yazımızda `lambda` ifadeleri ile `bind` uyumlandırıcısını karşılaştıracağız.