std::initializer_list Sınıf Şablonu

`C++11` standartları ile standart kütüphanemize eklenen çok faydalı sınıf şablonlarından biri initializer_list. `STL`, artık bir çok yerde kendi arayüzünde bu sınıfı kullanıyor. Örneğin birçok kap `(container)` sınıfının bu türden parametresi olan kurucu işlevleri var. Yine bazı kap sınıflarının bu türden parametresi olan ekleme `(insert)` ve atama işlevleri var. Sınıfımız aynı türden istenen sayıda değeri bellekte ardışık olarak bir arada tutuyor. Bu sınıfı kullanarak, aynı türden belirli sayıda değeri, kaynak kodda küme parantezleri içinde, virgüllerle ayrılan bir listeyle oluşturarak bir işleve argüman olarak geçebiliyor ya da bir işlevden geri dönüş değeri olarak onu çağıran işleve iletebiliyoruz. Sınıf şablonumuz `C++11` ile dile eklenen `initializer_list` isimli başlık dosyasında:

```
#include<initializer_list>
```

Şablonumuzun bildirimi şöyle

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

Şablondaki `T` listede tutulacak öğelerin türü. Bir `initializer_list` nesnesini aşağıdaki biçimlerde oluşturabiliyoruz:

```
std::initializer_list<int> x{ 1, 5, 7, 10, 23 };
std::initializer_list<int> y({ 1, 5, 7, 10, 23 });
std::initializer_list<int> z = { 1, 5, 7, 10, 23 };
```

Sınıfın temel varlık nedeni aynı türden birden fazla değeri bir işleve (düşük maliyetle) argüman olarak göndermek. Kap sınıflarının kurucu işlecleri, ekleme yapan işlevleri ve atama işlevleri `initializer_list` sınıfının kullanımına tipik örnek olarak verilebilir. `initializer_list` sınıfının kullanımına ilişkin ilk örneğimizi standart `vector` sınıfından verelim:

```
#include <vector>
#include <iostream>

using namespace std;

void display(const vector<int> &vec)
{
 for (int i : vec)
  cout << i << " ";
 cout << endl;
}

int main()
{
 vector<int> ivec = { 4, 6, 7 };
 display(ivec);
 ivec.insert(ivec.begin(), { 1, 2, 3 });
 display(ivec);
 ivec = { 10, 20, 30 };
 display(ivec);

 return 0;
}
```

`main` işlevi içinde `vector` sınıfının `initializer_list<T>` parametreli üç ayrı işlevi çağrılıyor:

```
vector<int> ivec = { 4, 6, 7 };
```

Burada çağrılan `vector` sınıfının `initializer_list` parametreli kurucu işlevi. Bu işlev `vector` kap nesnemizi küme parantezi ile verilen listedeki öğelerle başlatıyor.

```
ivec.insert(ivec.begin(), { 1, 2, 3 });
```

Burada çağrılan işlev `vector` sınıfının ikinci parametresi `initializer_list` türünden olan insert üye işlevi. Bu işlev küme parantezi ile verilen listedeki değerleri birinci parametreye geçilen konuma ekliyor.

```
ivec = { 10, 20, 30 };
```

Burada çağrılan ise `vector` sınıfının `initializer_list` parametreli atama işlevi. Bu işlev `vector` nesnesine küme parantezi ile verilen listedeki değerleri atıyor.

Modern `C++`'da sınıfların parametresi initializer_list türünden olan olan kurucu işlevlerine `"initializer_list constructor"` deniyor. array dışındaki standart `STL` kaplarının hepsinin `initializer_list` parametreli kurucu işlevleri var. Böylece `STL` kaplarını kendi belirlediğimiz değerlerle başlatabiliyoruz:

```
#include <map>
#include <string>

std::map<std::string, int> simap = { { "ali", 34 }, { "can", 12 },{ "kaya", 21 } };
```

Yukarıdaki kodda `simap` isimli `map` nesnesi sınıfın `initializer_list` parametreli kurucu işleviyle hayata başlatılıyor:

Şimdi de aşağıdaki koda bakalım:

```
#include <set>
#include <iostream>
#include <initializer_list>

template<typename T>
size_t erase_vals(std::multiset<T> &s, std::initializer_list<T> list)
{
 size_t cnt = 0;
 for (const auto &val : list)
  cnt += s.erase(val);

 return cnt;
}

using namespace std;

int main()
{
 multiset<int> iset{ 3, 7, 3, 4, 5, 6, 1, 8, 1, 9, 2, 7, 9, 2, };

 auto n = erase_vals(iset, { 1, 3, 7, 9 });
 cout << "n = " << n << endl;

 return 0;
}
```

Yukarıdaki kodda `erase_vals` isimli bir işlev şablonunun tanımlandığını görüyorsunuz. İşlev bir `multiset` kabından seçilmiş belirli değerlerin silinmesini sağlıyor. İşlevin ikinci parametresinin `std::initializer_list` türünden olduğunu görüyorsunuz. `main` işlevi içinde `iset` kabından silinmesi istenen değerler `erase_vals` işlevine küme parantezi içinde verilen bir listeyle iletiliyor. İşlevin geri dönüş değeri `multiset` kabından silinen öğe sayısı.

`initializer_list` sınıfında tutulacak değerler için daraltıcı dönüşümler `(narrowing conversions)` geçerli değil:

```
#include <vector>

using namespace std;

int main()
{
 vector<int> ivec1 = { 1, 4, 6, 'A' }; //geçerli
 vector<int> ivec2 = { 2, 5, 7.8}; //geçersiz

}
```

`ivec1` nesnesine ilk değer veren listede son öğe olarak bir karakter sabiti yani `char` türden bir değer kullanılmış. `char` türünden `int` türüne yapılan dönüşüm bir veri kaybına neden olmadığından kod geçerli. Ancak, `ivec2` isimli `vector` kap nesnesine ilk değer olarak verilen listedeki son öğe `double` türden. `double` türünden `int` türüne yapılan dönüşüm daraltıcı (veri kaybına yol açan) olduğundan kod geçerli değil.

Bir sınıf nesnesinin küme parantezi içinde sağlanan değerlerle başlatılması durumunda, `initializer_list` parametreli kurucu işlevin diğer kurucu işlevlere seçilebilirlik açısından üstünlüğü var. Aşağıdaki kodu inceleyin:

```
#include <initializer_list>
#include <iostream>

class A {
public:
 A(int) { std::cout << "A(int)\n"; }
 A(int, int) { std::cout << "A(int, int)\n"; }
 A(std::initializer_list<int>) {std::cout << "A(init_list)\n"; }
};

int main()
{
 A a1(1); //A(int)
 A a2{1}; //A(std::initializer_list)
 A a3(3, 5); //A(int, int)
 A a4{3, 5}; //A(std::initializer_list)
 A a5(4, 5, 6); //gecersiz
}
```

`vector`, `deque`, `list` ve `forward_list` sınıfları hem `size_t` parametreli hem de `initializer_list` parametreli kurucu işlevlere sahip olduğundan bu duruma dikkat edilmeli:

```
#include <vector>
#include <iostream>

using namespace std;

void print(std::vector<int> &ivec)
{
 for (auto i : ivec)
  std::cout << i << " ";
 std::cout << std::endl;
}

int main()
{
 vector<int> ivec1(5);
 print(ivec1); //0 0 0 0 0
 vector<int> ivec2{ 5 };
 print(ivec2); //5
 vector<int> ivec3(4, 3);
 print(ivec3); //3 3 3 3
 vector<int> ivec4{ 4, 3 };
 print(ivec4); //4, 3

}
```

Benzer bir durum da `string` sınıfı için söz konusu:

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

using namespace std;

int main()
{
 string s1(10, 'A');
 string s2{ 10, 'A' };

 cout << s1.length() << endl; //10
 cout << s2.length() << endl; //2
}
```

Sınıfın `begin` ve `end` işlevlerine çağrı yaparak listede tutulan değerlere ilişkin aralığın `(range)` `iterator` değerlerini elde edebiliriz. Aynı `iterator` değerleri global `begin` ve `end` işlevlerinden de elde edilebilir:

```
#include <iostream>
#include <initializer_list>

using namespace std;

int main()
{
 initializer_list<int> ilist{ 1, 4, 6, 9, 3 };

 for (auto iter = ilist.begin(); iter != ilist.end(); ++iter) {
  cout << *iter << " ";
 }
 cout << "\n";
 for (auto iter = begin(ilist); iter != end(ilist); ++iter) {
  cout << *iter << " ";
 }
 cout << "\n";
}
```

`C++14` standartları ile sınıfa `reverse_iterator` döndüren global `rbegin` ve `rend` işlevleri de eklendi:

```
#include <iostream>
#include <initializer_list>

int main()
{
 initializer_list<int> ilist{ 1, 4, 6, 9, 3 };

 for (auto iter = rbegin(ilist); iter != rend(ilist); ++iter) {
  cout << *iter << " ";
 }
}
```

`initializer_list` içinde tutulan değerlere bir aralık tabanlı `for` döngüsüyle de `(range based for loop)` erişilebiliyoruz:

```
#include <initializer_list>

void func(int);

int main()
{
 for (auto x : { 2, 3, 5, 7, 11 })
  func(x);
}
```

Yukarıdaki kodda aralık tabanlı döngünün her turunda global `func` işlevine listede verilen değerlerden biri gönderiliyor.

`initializer_list` nesneleri tarafından tutulan öğeler `const` nesneler. Listede tutulan öğeleri değiştirmeye yönelik kodlar geçerli değil.

```
#include <iostream>
#include <initializer_list>

using namespace std;

int main()
{
 initializer_list<int> x{ 1, 3, 5, 7, 8 };

 for (auto iter = x.begin(); iter != x.end(); ++iter)
  ++*iter; //geçersiz

 return 0;
}
```

`initializer_list` sınıfı listedeki öğeleri silmeye ya da listeye yeni öğeler eklemeye yönelik de bir arayüz sağlamıyor. `initializer_list` sınıfının `[ ]` işlevi de yok. Yani bir `initializer_list` nesnesini `[ ]` işlecinin terimi yaparak listedeki öğelere erişemiyoruz. Sınıfın `size` üye işlevi listede tutulan öğe sayısını döndürüyor:

```
#include <initializer_list>
#include <iostream>

using namespace std;

int main()
{
 initializer_list<double> dlist{ 1.1, 2.2, 3.3, 4.4 };
 cout << dlist.size() << endl; //4
}
```

Oluşturabileceğimiz boş bir listeyi de bir işleve geçebiliyoruz:

```
#include <initializer_list>
#include <iostream>

using namespace std;

void func(initializer_list<int> y)
{
 cout << y.size() << endl;
}

int main()
{
 func({});
}
```

Bir `initializer_list` nesneninin bir yerden bir yere kopyalanması durumunda listede tutulan öğelerin kendileri kopyalanmıyor. Derleyici yalnızca arka planda oluşturduğu dizide tutulan ilk öğenin ve dizinin bittiği yerin adreslerini kopyalayacak bir kod üretiyor:

```
#include <initializer_list>
#include <iostream>

using namespace std;

class Myclass {
public:
 Myclass(int) {
  std::cout << "Myclass(int)\n";
 }

 Myclass(const Myclass &) {
  std::cout << "Myclass(const Myclass &)\n";
 }
};

void func(initializer_list<Myclass> y)
{
 //
}

int main()
{
 initializer_list<Myclass> x{ Myclass{ 12 }, Myclass{ 20 } };
 func(x);
}
```

Yukarıdaki kodu derleyip çalıştırdığınızda `Myclass` sınıfının kopyalayan kurucu işlevinin çağrılmayacağını göreceksiniz.

tür çıkarımı ve `initializer_list`

Bir işlev şablonu söz konusu olduğunda, işlev şablonunun tür parametresinin ne olduğu, bu türden bir paramete değişkenine küme parantezi içinde değerler gönderilmesi yoluyla anlaşılamıyor.

```
#include <initializer_list>

template<typename T>
void func(T x)
{
 ///
}

int main()
{
 func({ 1, 2, 5 }); //geçersiz
}
```

Yukarıdaki kodda `func` işlevine argüman olarak

```
{ 1, 2, 5 }
```

geçildiğini görüyorsunuz. Derleyici bu durumda `T` türünün ne olduğu çıkarımını yapamıyor. Ancak `auto` belirteci kullanıldığında durum farklı:

```
auto y = {12};
```

Yukarıdaki kodda tanımlanan `x` değişkeninin `std::initializer_list<int>` türünden. Küme parantezi içinde tek bir değerin olması durumunda da bir farklılık söz konusu değil. `y` değişkeninin türü yine `std::initializer_list<int>` türü. Ancak ilk değer verme doğrudan yapılırsa `(direct initialization)` durum farklı. `C++17` standartlarından önce bu durumda da tür çıkarımı aynı şekilde yapılıyordu:

```
auto x {1, 4, 5};
auto y{12};
```

Yukarıdaki tanımlamalarda `C++11` ve `C++14` standartlarına göre `x` ve `y` değişkenlerinin türlerinin çıkarımı derleyici tarafından `std::initializer_list<int>` olarak yapılıyordu. Ancak `C++17` standartları ile bu konuda ciddi bir değişiklik yapıldı: `auto` belirteci ile tanımlanan değişkene doğrudan ilk değer verme durumunda küme parantezi içinde yalnızca tek bir değer olabiliyor. Yani yeni standartlara göre yukarıdaki kodda `x` değişkeninin tanımı geçerli değil. Eğer küme parantezi içinde tek bir değer var ise bu durumda `auto` belirteci ile tanıtılan değişkenin türü artık küme parantezi içindeki ifadenin türü kabul ediliyor. Yani yukarıdaki kodda `y` değişkenin türü artık `int`.