Değişken sayıda Parametreli Şablonlar (Variadic Templates)

C++ türden bağımsız programlamaya `(generic programming)` en güçlü desteği veren programlama dili. C++11 standartları ile dile eklenen en önemli araçlardan biri olan değişken sayıda parametreli şablonlar `(variadic templates)` türden bağımsız programlamanın gücünü daha da arttırıyor. Standart kütüphane artık birçok yerde böyle şablonlar kullanıyor. Dile eklenen bu araç ile hem sınıf şablonları `(class templates)` hem de işlev şablonları `(function templates)` artık istenen sayıda parametreye sahip olabiliyor. Değişken sayıda parametreye "parametre paketi" `(parameter pack)` deniyor. İki ayrı parametre paketi oluşturabiliyoruz: Şablon parametre paketi `(template parameter pack)`, sıfır ya da daha fazla sayıda şablon parametresini temsil ediyor. İşlev parametre paketi `(function parametre pack)` sıfır ya da daha fazla sayıda işlev parametresini temsil ediyor.

Bir şablon parametre listesinde class ya da typename anahtar sözcüklerini izleyen üç nokta atomundan `(ellipsis)` sonra gelen isim, söz konusu şablon parametresinin sıfır ya da daha fazla sayıda türe karşılık geldiğine işaret ediyor:

```
template <class... Ts>
```

Yukarıdaki sözdizimde `Ts` tür parametre listesini yani şablon parametre paketini temsil ediyor. Diğer şablonlarda olduğu gibi burada da `class` yerine `typename` anahtar sözcüğü kullanılabiliyoruz:

```
template <typename... Ts>
```

`Ts` yalnızca bir isim. Burada herhangi bir ismi kullanabiliriz. Ancak çoğunlukla `Args`, `Types`, `Ts` gibi isimler tercih ediliyor.

```
template <unsigned... Ns>
```

Burada da şablon parametre paketi `Ns` olarak isimlendirilmiş. Bu paket `unsigned int` türden olan şablon sabit parametrelerini `(non-type parameter)` temsil ediyor. Normal şablonlarda olduğu gibi değişken sayıda parametreli şablonlar da sınıflara ya da işlevlere ilişkin olabiliyor ve özelleştirilebiliyorlar `(specialization)`.

Parametre paketi diğer şablon parametreleri ile birlikte kullanılabiliyor. Yalnızca iki kısıtlama söz konusu: Parametre paketi son şablon parametresi olmalı ve yalnızca tek bir paramete paketi olmalı:

```
template <typename X, int C, class... Ts>
```

Bu şablondan gerçek bir işlevin kodunu yazarken `X` yerine gerçek bir türü kullanacak. C ise şablonun sabit parametresi. Derleyici şablondan bir gerçek işlevin kodunu yazaraken `C` yerine `int` türden bir sabit kullanacak. `Ts` ismi ise bir tür parametre paketine işaret ediyor. `Ts`, sıfır ya da birden fazla türe karşılık gelen bir isim.

İşlev parametre listesinde yer alan, şablon parametre paketi türünden olan parametre ise "işlev parametre paketi". Aşağıdaki örneğe bakalım:

```
template <typename T, typename... Args>
void func(const T &t, Args ... rest);
```

Yukarıdaki bildirimde yer alan `Args` ismi şablon parametre paketini rest ismi ise işlev parametre paketini temsil ediyor. `Args` sıfır ya da daha fazla sayıda şablon tür parametresini `rest` ise yine sıfır ya da daha fazla sayıda işlev parametre değişkenini içerecek.

Derleyici diğer şablonlarda olduğu gibi şablon tür parametrelerinin çıkarımını da işleve gönderilen argüman olan ifadelerin türlerinden hareketle gerçekleştiriyor. Değişken sayıda parametreye sahip şablonlarda, derleyici paket içindeki kaç tane parametre yer aldığının da çıkarımını yapıyor. Bir örnekle gösterelim:

```
#include <string>

template <typename T, typename... Args>
void func(const T &t, Args... rest);

using namespace std;

int main()
{
 int ival = 10;
 double dval = 5.67;
 string name{ "necati" };
 func(ival, name, 34U, dval); // pakette 3 parametre var
 func(name, 19, "ali"); // pakette 2 parametre var
 func(dval, name); // pakette bir parametre var
 func("murat"); // boş paket
}
```

`func` ismiyle yapılan her çağrıda derleyici sırasıyla aşağıdaki parametrik yapıya sahip işlevler oluşturacak:

```
void func(const int&, string, unsigned int, double);
void func(const string&, int, const char *);
void func(const double&, string);
void func(const char(&)[5]);
```

Her bir çağrı için şablonun birinci tür parametresi olan `T` türünün çıkarımı işleve gönderilen ilk argümana göre yapılıyor. İşleve gönderilen diğer argümanlar parametre paketine dahil ediliyor. İşlev parametre paketi için `&` ya da `&&` bildirgeçleriyle paketteki türlerin referans ya da gönderim referansı `(forwarding reference)` olmalarını da sağlayabiliyoruz. Aşağıdaki koda bakalım:

```
template<typename... Args>
void f1(Args... args)
{
 //
}

template<typename... Args>
void f2(Args &... args)
{
 //
}

template<typename... Args>
void f3(Args &&... args)
{
 //
}

int main()
{
 int ival = 12;
 double dval = 4.5;

 f1(ival, dval);
 f2(ival, dval);
 f3(ival, dval);
 f3(10, 5.6);

}
```

`main` işlevi içinde yapılan değişken sayıda parametreye sahip şablonlardan üretilecek işlevlere çağrılar yapılıyor. Derleyici yapılan çağrılar için aşağıdaki işlevleri yazacak:

```
void f1(int, double);
void f2(int &, double &);
void f3(int &, double &);
void f3(int, double);
```

Gönderim parametreli `f3` işlevine yapılan çağrılardan biri sol taraf değerlerini diğeri sağ taraf değerlerini kullandığından derleyicinin bu çağrılar karşılığı üreteceği işlevler de farklı olacak.

sizeof... işleci Değişken sayıda parametreye sahip bir şablon içinde kullanılabilen `sizeof...` işleci pakette yer alan parametre sayısına eşdeğer bir sabit ifadesi oluşturuyor. `sizeof...` işlecinde de C dilinden gelen `sizeof` işlecinde olduğu gibi argüman olan ifadeler için bir işlem kodu üretilmiyor. Aşağıdaki koda bakalım:

```
#include <iostream>

template<typename... Args>
void func(Args... args)
{
 std::cout << sizeof...(Args) << std::endl;
 std::cout << sizeof...(args) << std::endl;
}

int main()
{
 func(1, 2.5, "necati"); //3 3

}
```

paket açılımı `(pack expansion)`

Bir parametre paketini şablon kod içinde kullanmanın tek yolu o paketi açmak. Bir paketin açılmasıyla derleyici paketi, paketteki her bir öğeyi içeren virgüllerle ayrılmış bir listeye dönüştürüyor. En basit paket açılımı, paketi temsil eden ismin yanına üç nokta atomumun yazılmasıyla oluşturuluyor. Aşağıdaki örneğe bakalım:

```
#include <tuple>

template<typename...Args>
auto makeTuple(Args... rest)
{
 return std::tuple<Args...>(rest...);
}

int main()
{
 auto t = makeTuple(10, 4.5, 3L);
 ///
}
```

main işlevinde yer alan

```
makeTuple(10, 4.5, 3L)
```

çağrısından hareketle derleyici şablondan üreteceği kodda iki ayrı paket açılımı gerçekleştirecek.

`Args...`

ifadesinin yerine işleve gönderilen argümanların çıkarımla elde edilmiş türlerini içeren virgüllerle ayrılmış bir listeyi

`rest...`

ifadesinin yerine ise işleve gönderilen argüman olan ifadeleri içeren virgüllerle ayrılmış bir listeyi kullanacak. Yani paket açılımından sonra şöyle bir işlev kodu oluşturulacağını düşünebiliriz:

```
auto makeTuple(int args_1, double args_2, long args_3)
{
 return std::tuple<int, double, long>(10, 4.5, 3L);
}
```

İşlev parametre paketini, şablon işlev kodu içinde kullanmanın temel olarak iki yöntemi var.

- Parametre paketindeki tüm argümanları bir defada kullanmak.
- Aynı isimli işleve çağrı yaparak her defasında paketteki argümanlardan birini işleyeek tüketmek.

`makeTuple` örneğinde parametre paketindeki argümanların hepsini bir defada kullandık, değil mi?

Paket açılımına ilişkin daha karmaşık senaryolar da var. Bu senaryoları da ileride teker teker ele alacağız. Şimdi değişken sayıda parametreye sahip, iş gören gerçek bir şablon yazmanın zamanı geldi.

Öyle durumlar var ki bir işlevin kaç tane argüman üzerinde işlem yapacağını ve bu argümanların türlerinin neler olacağını bilmeden bir şablon kod oluşturmamız gerekiyor. Değişken sayıda parametreli şablonlar burada ideal bir çözüm oluşturuyor. İlk örnek olarak bir çıkış akımına istenen herhangi bir sayıda ve herhangi bir türden olabilecek değerleri yazdırabilecek bir işlev şablonu oluşturacağız. Değişken sayıda parametreli şablonlar tipik olarak özyinelemeli `(recursive)` bir karakterde. Tabi buradaki özyineleme yapısı çalışma zamanına değil derleme zamanına yönelik bir özyineleme.

```
#include <ostream>

template<typename T>
std::ostream &print(std::ostream &os, const T &t)
{
 return os << t;
}

template <typename T, typename... Ts>
std::ostream &print(std::ostream &os, const T &t, const Ts &... rest)
{
 os << t << ", ";
 return print(os, rest...);
}

#include <iostream>
#include <string>

using namespace std;

int main()
{
 string s{ "ali" };
 int ival = 23;
 double dval = 6.7;
 print(cout, ival, s, dval, "murat");
}
```

Şimdi kodu inceleyelim. Kodda daha yukarıda yer alan ve değişken sayıda parametreye sahip olmayan `print` işlev şablonuna bakalım:

```
template<typename T>
std::ostream &print(std::ostream &os, const T &t);
```

Bu işlev derleyicin özyinelemeli olarak kod üretmesini sonlandıracak `(base case)` ve işleve gönderilen son argümanı çıkış akımına yazdıracak. Daha aşağıda yer alan değişken sayıda parametreye yönelik yazılan print şablonu ise birinci şablon tür parametresi olan `T` türüne bağlanacak argümanı çıkış akımına yazdırıyor ve diğer argümanların yazdırılması için yine print isimli bir işleve çağrı yapıyor. Her işlev çağrısında kullanılan ilk argüman birinci şablon tür paamtresine bağlandığından işlev parametre paketindeki argüman sayısı birer azalıyor. Pakette yalnızca bir argüman kaldığında derleyici dilin kurallarına göre bu durumda üstteki şablondan kod üretiyor.Yükleme sıralama kurallarına göre bu durumda değişken sayıda parametreye sahip olmayan işlevin önceliği söz konusu. Aslında derleyici derleme zamanında toplamda 5 ayrı işlevin kodunu yazmış olacak. Derleyicinin yazacağı kodların şöyle olacağını düşünebiliriz:

```
std::ostream &print(std::ostream &os, const char(&t)[6])
{
 return os << t;
}

ostream &print(ostream &os, const double &arg_1, const char(&arg_2)[6])
{
 os << arg_1 << ", ";
 return print(os, arg_2);
}

ostream &print(ostream &os, const string &arg_1,
            const double &arg_2, const char(&arg_3)[6])
{
 os << arg_1 << ", ";
 return print(os, arg_2, arg_3);
}


ostream &print(ostream &os, const int &arg_1, const string &arg_2,
            const double &arg_3, const char(&arg_4)[6])
{
 os << arg_1 << ", ";
 return print(os, arg_2, arg_3, arg_4);
}
```

Şimdi de her bir çağrı için işlev parametre paketinde tutulan argümanlara bakalım:

```
çağrı                                t      paket

print(cout, ival, s, dval, "murat")    ival   s, dval, "murat"
print(cout, s, dval, "murat")        s      dval, "murat"
print(cout, dval, "murat")             dval, "murat"
print("murat")                        diğer şablondan kod üretilecek
```

Her bir çağrı için paketteki argüman sayısı bir azalıyor. Bunun nedeni her çağrıda ilk argümanın normal şablon tür parametresine bağlanıyor olması. Şimdi de kendisine gönderilen argümanların toplamını geri döndürecek değişken sayıda parametreli bir işlev şablonu yazalım:

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

template<typename T>
T sum(T x)
{
 return x;
}

template<typename T, typename... Args>
T sum(const T &t, const Args&... args)
{
 return t + sum(args...);
}

int main()
{
 int i1 = 20, i2 = 30;
 std::string s1{ "ali" }, s2{ "kagan" }, s3{ "necati" };
 std::cout << sum(i1, i2, 60) << std::endl;
 std::cout << sum(s1, s2, s3) << std::endl;

}
```

Yukarıdaki kodda yer alan

```
template<typename T>
T sum(T x)
```

işlevi derleme zamanında oluşturulan özyinelemeli kod yazımını durduruyor. İşlevin tek yaptığı kendisine gelen argümanı geri döndürmek.

Bir sonraki yazımızda değişken sayıda parametreli şablonlara ilişkin daha karmaşık yapıları ele alacağız.