Fonksiyon Yüklemesi *(Function Overloading)*

C++ dilinde, bir kapsamda *(scope)* aynı isimli birden fazla fonksiyon bildirilebilir ya da tanımlanabilir. Programcıların işini kolaylaştıran bu araç İngilizce'de "*function overloading*" (fonksiyon yüklemesi) olarak isimlendirilir. Önce şu sorulara yanıt aramakla başlayalım: Neden iki ya da daha fazla sayıda fonksiyonun isimlerinin aynı olmasını isteyelim? İki ayrı fonksiyona aynı isim vermenin nasıl bir faydası olabilir? Bir örnekle başlayalım:

C'nin aşağıdaki standart işlevlerini hatırlayalım:

```
int abs(int x);
double fabs(double x);
long labs(long x);
```

C'nin standart başlık dosyalarından biri olan *math.h* içinde bildirimleri yer alan yukarıdaki fonksiyonların hepsi aslında aynı işlemi yapar. Bu fonksiyonlar kendilerine argüman olarak gönderilen değerin mutlak değerini döndürür. Bu fonksiyonların çağıran koddan aldıkları değerlerin türleri farklıdır. Buna bağlı olarak geri dönüş değerlerinin türleri de farklıdır. C dilinde aynı kapsamda aynı isimli fonksiyonlar tanımlanamayacağı için bu işlevlere ayrı isimler verilmiştir. Bu fonksiyonların hepsinin isminin aynı olması bir fayda sağlar mıydı? Örneğin hepsinin isimleri *abs* olsaydı, mutlak değer alma işlemi yapacak programcının birden fazla fonksiyon ismini bilmesi ya da anımsaması gerekmezdi, değil mi? Aslında dilin temel operatörlerini düşündüğünüz zaman benzer bir aracın kullanıma hazır bir biçimde sunulduğunu görebilirsiniz.

Toplama operatörünü ele alalım. Toplama operatörü ne iş yapar? İki değerin toplanmasını sağlar, değil mi?

```
i1 + i2
```

i1 ve *i2*'nin *int* türden değişkenler olduğunu düşünelim. *i1* ve *i2* toplama toplama operatörünün terimleri yapılmış. Toplama operatörü bir işlemin yapılmasını sağlıyor. Yapılan işlemin sonucunda bir tamsayı değeri üretiyor. Şimdi de aşağıdaki işleme bakalım:

```
d1 + d2
```

d1 ve *d*'nin double türden değişkenler olduğunu düşünelim. *d1* ve *d2* toplama toplama operatörünün terimleri yapılmış. Toplama operatörü yine bir işlemin yapılmasını sağlıyor. Yapılan işlemin sonucunda bir gerçek sayı değeri üretiliyor. Oysa işlemci düzeyinde bakıldığında, tamsayı türünden iki değerin birbiriyle toplanmasıyla, gerçek sayı türünden iki değerin birbiriyle toplanması bambaşka işlemlerdir. Bize çok doğal görünen bu iki örnekte yapılan işlemi, aslında ayrıntılarından soyutlayarak "toplama" olarak ifade ediyoruz. Matematikte olduğu gibi C dilinde de, bu işlemleri yapmak için bir soyutlama kullanılarak aynı operatör yani aynı simge kullanılıyor. İki işlem için farklı iki simge kullanılmış olsaydı algılama bu kadar kolay olur muydu? Şimdi de *m1* ve *m2* değişkenleri *Matrix* isimli bir sınıf türünden olsun.

```
m1 + m2
```

gibi bir ifade karşılığı iki matrisin toplanması işleminin yaptırılması şüphesiz kodu yazanın işini kolaylaştırırdı.

Fonksiyon yüklemesinden söz edilebilmesi için iki koşulun sağlanması gerekir:

- Aynı isimli birden fazla fazla fonksiyon aynı kapsamda (scope) var olmalı.
- Bu fonksiyonların parametrik yapıları birbirinden farklı olmalı.

Anlatım kolaylığı olsun diye, fonksiyonların parametre değişkenlerinin sayısı ile her bir parametre değişkeninin türünü kapsayan bilgiye "*fonksiyonun imzası*" diyelim. Fonksiyonun geri dönüş değerinin türünü fonksiyonun imzasının bir parçası olarak görmeyeceğiz. Aynı kapsamda aynı isimli iki ya da daha fazla sayıda fonksiyona ilişkin bildirim varsa aşağıdaki üç durumdan biri söz konusudur:

- Aynı kapsamdaki aynı isimli iki fonksiyonun hem parametre değişkeni sayısı aynı hem de parametre değişkenlerinin türleri aynı ise bu durum fonksiyonun yeniden bildirimidir *(redeclaration)*. C'de olduğu gibi C++'ta da bir fonksiyonun bildirimi özdeş olması koşuluyla birden fazla kez yapılabilir. Aşağıda aynı fonksiyon bildirimi iki kez yapılıyor:

```
int func(int);
int func(int);
```

- Fonksiyonların parametrik yapıları (imzaları) tamamen aynı, fakat bildirilen geri dönüş değerlerinin türleri farklı ise bu durum sentaks hatasıdır.

```
int func(int)
double func(int); //geçersiz - fonksiyon bildirimi çelişkili biçimde yineleniyor
```

Geri dönüş değeri türü dışında tüm parametrik yapısı aynı olan aynı iki fonksiyonun var olduğunu düşünelim. Bu isimle bir fonksiyon çağrısı yapıldığında derleyici hangi fonksiyonun çağrıldığını anlayabilir miydi? Yukarıdaki her iki bildirim de geçerli olsaydı hangi foo fonksiyonunun çağrıldığı nasıl anlaşılırdı?

- Aynı kapsamdaki aynı isimli fonksiyonların parametik yapıları birbirinden farklı ise (yani fonksiyonların imzaları farklı ise) bu durum fonksiyon yüklemesidir. bu Durumda fonksiyonların geri dönüş türlerinin bir önemi yoktur:

```
//function overloading (fonksiyon yüklenmesi)
int func(int);
int func(int, int);
```

Aynı isimli iki fonksiyonun bildiriminde parametrik yapı aynı olabilir ve bildirimlerin birinde varsayılan argüman belirtilebilir. Bu durum yine "yeniden bildirim"dir, fonksiyon yüklemesi değildir:

```
int max(int *ptr, int size);
int max(int *, int = 10); //fonksiyon bildirimi yineleniyor (function redeclaration)
```

Tür eş isim *(type alias)* bildirimleri ile *(typedef ya da using bildirimi)* ile yeni bir tür oluşturulmuş olmaz. *typedef* ya da *using* bildirimi ile var olan bir türe yeni bir isim *(tür eş ismi)* verilebilir. Aynı isimli iki fonksiyonun bildirimlerinde parametre değişkenlerinin türlerinin yazımında aynı türe ilişkin eş isim kullanılşması durumu fonksiyon yüklemesi değildir. Fonksiyonun yeniden bildirimidir.

Aşağıdaki örneği inceleyin:

```
typedef unsigned char Byte;

unsigned char foo(unsigned char);
Byte foo(Byte); //yeniden bildirim
```

Peki C++'ta aynı kapsamda parametrik yapıları birbirlerinden farklı aynı isimli fonksiyonlar bildirilebildiğine göre derleyici aynı isimli işlevlerden hangisinin çağrıldığını nasıl anlar? Bir fonksiyon çağrısının hangi fonksiyona ilişkin olduğunun saptanması işlemine "*function overload resolution*" (Yüklenmiş fonksiyonun belirlenmesi) denir. "*function overload resolution*" derleyici tarafından üç aşamada yapılan bir işlemdir:

- Birinci aşamada derleyici söz konusu fonksiyon çağrısı için ele alınacak aynı isimli işlevleri saptayarak bu fonksiyonların parametrik yapısı hakkında bilgi edinir. Çağrılması söz konusu olan aynı isimli işlevlere aday fonksiyonlar *(candidate functions)* denir. Aday fonksiyonlar, fonksiyon çağrı ifadesinde kullanılan isimle aynı isme sahip olan ve fonksiyon çağrı ifadesinin bulunduğu yerde görülebilen *(visible)* işlevlerdir. Birinci aşamada derleyici aday fonksiyonlar hakkında gerekli bilgiyi de edinir. Aday fonksiyonların parametre değişkeni sayılarını, parametre değişkenlerinin türlerini öğrenir.

- İkinci aşamada fonksiyon çağrı ifadesinde yer alan argümanlar kullanılarak hangi aday fonksiyonların geçerli biçimde çağrılabileceği saptanır.Yapılan fonksiyon çağrısıyla geçerli biçimde çağrılabilen işlevlere "uygun fonksiyonlar" *(viable functions)* denir. Bir fonksiyonun uygun fonksiyon olarak belirlenebilmesi için aşağıdaki koşulları sağlaması gerekir:

1. Fonksiyon çağrı ifadesindeki argüman sayısı ile, fonksiyonun parametre değişkeni sayısının aynı olması zorunludur.

2. Fonksiyon çağrı ifadesindeki argüman sayısı, fonksiyonun parametre değişkeni sayısından daha az ise, bu durumda fonksiyonun argüman gönderilmeyen parametre değişkenleri varsayılan argüman almalıdır. Argüman olan ifadenin türünden parametre değişkeninin türüne geçerli bir tür dönüşümünün *(type conversion)* yapılabilmesi gerekir. Bir fonksiyonun uygun olup olmadığını pratik olarak şöyle anlayabiliriz. Bu fonksiyon tek başına olsaydı (aynı isimli başka fonksiyon olmasaydı) çağrı geçerli olur muydu? Örnek olarak, aşağıda bildirimleri verilen işlevleri inceleyelim:

```
void foo();     //1
void foo(int);     //2
void foo(double, double = 3.4)   //3
void foo(char *);    //4

void func()
{
 foo(5);
}
```

func fonksiyonu içinde *foo* isimli fonksiyona yapılan çağrı hangi fonksiyona bağlanır?

1. aşama

Fonksiyonun çağrıldığı noktada, tüm fonksiyon bildirimleri görülebilir *(visible)*. Bu aşamada dört fonksiyon da aday olarak belirlenir. Derleyici bu fonksiyonların parametrik yapıları hakkında bilgi edinir.

2. aşama

- 1 numaralı fonksiyonun parametre değişkeni sayısı *(0)* ile fonksiyon çağrısındaki argüman sayısı *(1)* birbirine eşit değildir. Bu fonksiyon uygun *(viable function)* değildir. Bu fonksiyon tek başına olsaydı sentaks hatası oluşurdu.

- 2 numaralı fonksiyon uygundur. Parametre değişkeni ile argüman sayısı uyumludur. Argümandan parametre değişkenine geçerli bir dönüşüm söz konusudur. Bu fonksiyon tek başına bulunsaydı çağrı geçerli olurdu.

- 3 numaralı fonksiyon uygundur. onksiyonun iki parametre değişkeni vardır. Ancak ikinci parametre değişkeni varsayılan argüman aldığından fonksiyon tek argüman ile çağrılabilir. Argüman olan ifade *int* türden bu argümanın kopyalanacağı parametre değişkeni double türdendir. *int* türünden *doubl*e türüne geçerli dönüşüm vardır.

- 4 numaralı fonksiyon uygun değildir. Fonksiyonun parametre değişkeni sayısı ile fonksiyon çağrısındaki argüman sayısı birbirine eşittir. Ancak int türünden *char** türüne geçerli bir tür dönüşümü yoktur. İkinci aşamada uygun bir fonksiyon bulunmaz ise fonksiyon çağrısı geçersiz kabul edilir. Bu duruma İngilizcede "*no match*" durumu *(çağrılacak uygun bir fonksiyonun bulunmaması)* denir.

Üçüncü aşama

Üçüncü yani son aşamada uygun fonksiyonlar içinde en uygun olan fonksiyon belirlenir. Bu aşamada ya çağrıya karşı bir fonksiyon seçilir ya da sentaks hatası oluşur. Uygun fonksiyonlar içinden seçilerek çağrılacak fonksiyona "best viable function" ya da "*best match* function" *(en uygun fonksiyon)* denir. Dilin kurallarına göre iki ya da daha fazla fonksiyon arasında seçim yapılaması durumu sentaks hatasıdır. Üçüncü aşamada en uygun fonksiyonun seçilebilmesi için aşağıdaki koşulun sağlanması gerekir:

- Argümandan fonksiyonların parametre değişkenlerine yapılan dönüşüm belirli kalitelere ayrılmıştır. Her bir fonksiyon için argümandan o fonksiyonun parametre değişkenine yapılacak dönüşümün kalitesine bakılır. Kalitesi en yüksek olan fonksiyon seçilecektir.

- Birden fazla uygun fonksiyon içinden, en uygun fonksiyonun seçilememesi durumunda *ambiguity (çift anlamlılık hatası)* denilen bir hata durumu oluşur.

variadic dönüşüm

Kalitesi en düşük olan dönüşüm *variadic* dönüşümdür. "*variadic*" fonksiyonlara farlı sayıda argüman gönderilebileceğini hatırlayınız. Aşağıdaki fonksiyona bakalım:

```
void func(int, ...);
```

Fonksiyonun son parametresinin üç nokta atomu *(ellipsis)* ile belirtildiğini görüyorsunuz. Bu fonksiyonun birinci parametresine bir argüman gönderilmek zorundadır. *variadic* parametre *(ellipsis)* için istenilen sayıda argüman gönderilebilir. Yani *func* fonksiyonu bir ya da birden fazla argüman ile çağrılabilir.

Programcının tanımladığı dönüşümler *(user-defined conversions)*

Programcının tanımladığı dönüşümlerin kalitesi "*variadic*" dönüşümün kalitesine göre daha iyidir. C'den farklı olarak C++ dilinde normalde geçerli olmayan tür dönüşümleri bazı fonksiyonların tanımlanması ile geçerli hale gelir. Derleyici bildirilen bir fonksiyona çağrı yaparak bir dönüşümü gerçekleştirir. Yani dönüşümü gerçekleştiren aslında derleyicin çağıracağı bir fonksiyondur. Böyle fonksiyonlara çağrı yapılması ile, sınıf türleri ile temel türler arasında ya da farklı sınıf türleri arasında tür dönüşümleri yapılabilir. Böyle dönüşümlerin yapılmasını sağlayan fonksiyonlar şunlardır:

- Dönüştüren kurucu fonksiyon "*(conversion constructor)*
- (Tür dönüştürme operatör fonksiyonları *(type-cast operator functions)*

Bu fonksiyonları ileride ayrıntılı olarak inceleyeceğiz.

standart dönüşümler

Bu dönüşümler dilin kurallarına göre örtülü *(implicit)* olarak yapılabilen dönüşümlerdir. Standart dönüşümlerin kalitesi programcı tarafından tanımlanan dönüşümlerin kalitesinden daha yüksektir. Standart dönüşümler de keni içinde 3 kategoriye ayrılır.

- Tam uyum *(exact match)*
- Yükseltme *(promotion)*
- Diğer dönüşümler *(conversion)*

Kurallara göre, "tam uyum"'un seçilebilirliği "yükseltme"den, "yükseltme" durumu standart dönüşümden, standart dönüşüm de programcının tanımladığı dönüşümden daha iyi olarak kabul edilir. Yukarıdaki derecelendirmeleri ayrıntılı biçimde inceleyelim:

Tam uyum durumu

Argüman olan ifadenin türü ile bu argümanın kopyalanacağı parametre değişkeninin türü tamamen aynı ise, bu durum tam uyum *(exact match)* olarak ele alınır. Ancak aşağıdaki durumlar da tam uyum olarak ele alınır:

- Argüman olan nesne bir sol taraf değeri yani bir nesne ise, parametre değişkenine kopyalanacak değerin bu nesneden alınması. Bu duruma sol taraf değerinden sağ taraf değerine dönüşüm denir. *(L-value to R-value transformation)*
- Parametre değişkeninin bir gösterici olması ve fonksiyonun da aynı türden bir dizinin ismi ile çağrılması. Dizi isimlerinin bir ifade içinde kullanılmaları durumunda otomatik olarak dizinin ilk elemenının başlangıç adresine dönüştürüldüğünü *(array to pointer conversion* biliyorsunuz.
- Parametre değişkeninin bir fonksiyon göstericisi *(function pointer)* olması ve fonksiyonun da aynı türden bir fonksiyonun ismi ile çağrılması. Fonksiyon isimlerinin bir işleme sokulduğunda işlem öncesinde otomatik olarak fonksiyonun adresine dönüştürüldüğünü hatırlayın. *(function to pointer conversion)*
- Fonksiyon parametre değişkeninin, *const* nesne göstericisi olması ve fonksiyonun aynı türden ancak *const* olmayan bir adres ile çağrılması *(const qualification conversion)*.

Yükseltme *(promotion)* durumu

Yükseltme aşağıdaki durumları kapsar:

char, signed char, unsigned char, bool, short, unsigned short türlerinden int türüne yapılacak dönüşüm. Bu duruma "*int* türüne yükseltme" *(integral promotion)* denir. *int* türünden küçük türler *int* türüne yükseltilirler.

```
void func(int);

int main()
{
 func('A'); *// yükseltme (integral promotion)*
 func(true);  *// yükseltme (integral promotion)*
 //...
}
```

float türünden double türüne yapılan dönüşüm.

```
void func(double);

int main()
{
 float fx = 2.3f;
 //...
 func(fx); *//yükseltme*
 //...
}
```

float türünden *long double* türüne ya da *double* türünden *long double* türüne yapılan dönüşümler "yükseltme" değildir. Bir numaralandırma *(enum)* türünden o numaralandırma türüne baz olan *(underlying type)* türe yapılan dönüşüm de yükseltme olarak değerlendirilir. Aşağıdaki örneği inceleyin:

```
enum Color
{
 Blue, Red, Green, Yellow
};

void f(int); //1
void f(long); //2
void f(double); //3

int main()
{
 f(Blue);
}
```

Color türünün baz türü *int* türüdür. Bu durumda *1* numaralı fonksiyon çağrılacaktır.

Diğer standart dönüşümler

Dilin kurallarınca geçerli olan ve örtülü *(implicit)* olarak yapılabilen diğer dönüşümlerdir. Standart dönüşüm *(standard conversions)* başlığı altında toplanan 5 grup dönüşüm söz konusudur:

1. Tamsayı türlerine ilişkin dönüşümler Bir tamsayı türünden ya da bir numaralandırma (enum) türünden başka bir tamsayı türüne yapılan dönüşümler.

2. Gerçek sayı dönüşümleri Bir gerçek sayı türünden başka bir gerçek sayı türüne yapılan dönüşümler

3. Gerçek sayı türleri ile tamsayı türleri arasında yapılan dönüşümler.

4. Adres türlerine ilişkin dönüşümler

0 tamsayı sabitinin herhangi türden bir adres türüne kopyalanması için *nullptr* sabitine dönüştürülmesi. *void* türden olmayan herhangi bir adresin *void* türden bir adrese dönüştürülmesi.

5. bool türüne yapılan dönüşümler

Herhangi bir tamsayı, gerçek sayı, numaralandırma ya da adres türünden *bool* türüne yapılan dönüşümler.

Aşağıda standart dönüşümlere ilişkin bazı örnekler veriliyor:

```
void func(int);
void foo(long);
void f(float);
void pf(int *);
void vfunc(void *);

int main()
{
 int x = 10;
 foo(x);  //standart dönüşüm (int türden long türüne)
 foo('A'); //standart dönüşüm (char türden long türüne)
 func(20U); //standart dönüşüm (unsigned int türünden int türüne)
 f(7.5);  //standart dönüşüm (double türden float türüne)
 pf(0);  //standart dönüşüm (0 tamsayı sabitinin bir göstericiye kopyalanması
 vfunc(&x) //standart dönüşüm (int * türden void * türüne)
}
```

Yüklenmiş fonksiyon çözümlemesine ilişkin bir örnek verelim:

```
int foo(int);  //1
int foo(double); //2
void foo(char);  //3
long foo(long);  //4
void foo(int, int); //5
void foo(char *); //6
void foo(int *); //7

void func()
{
 foo(10);
 foo(3.4F);
 foo((double *)0x1FC0);
 foo(6U);
}
```

func fonksiyonu içinde yapılan fonksiyon çağrılarını teker teker ele alalım:

```
foo(10);
```

Çağrısı için

- 1, 2, 3, 4, 5, 6, 7 numaralı fonksiyonlar adaydır.
- 1, 2, 3, 4 numaralı fonksiyonlar uygundur.
- Tam uyum sağladığı için, *1* numaralı fonksiyon en uygun olanıdır.

```
foo(3.4F)
```

Çağrısı için

- 1, 2, 3, 4, 5, 6, 7 numaralı fonksiyonlar adaydır.
- 1, 2, 3, 4 numaralı fonksiyonlar uygundur.
- Yükseltme durumu olarak değerlendirildiğinden 2 numaralı fonksiyon en uygun olanıdır.

```
foo((double *) 0x1FC0)
```

Çağrısı için 1, 2, 3, 4, 5, 6, 7 numaralı fonksiyonlar aday işlevlerdir. Uygun fonksiyon yoktur *(no match)*. Fonksiyon çağrısı geçersizdir.

```
foo(6U)
```

Çağrısı için

- 1, 2, 3, 4, 5, 6, 7 numaralı fonksiyonlar adaydır.
- 1, 2, 3, 4 numaralı fonksiyonlar uygundur.
- 1, 2, 3 ve 4 numaralı fonksiyonlar için standart dönüşüm uygulanabilir. Çift anlamlılık hatası *(ambiguity)* söz oluşur. Fonksiyon çağrısı geçersizdir.

const* yüklemesi *(const overloading)

Aşağıdaki bildirimlere bakalım:

```
void func(int *ptr);  //1
void func(const int *ptr); //2
```

Bu bir fonksiyon yüklemesidir *(function overloading*). Bu şekilde yapılan bir yüklemeye "*const* yüklemesi" *(const overloading)* denir. Her iki işlevin de parametresi bir gösterici (pointer). Ancak ikinci işlevin parametresi gösterdiği yer *const* olan bir gösterici *(pointer to const object / low level const)*. *func* işlevi bir adresle çağrıldığında derleyici hangi işlevin çağrıldığını nasıl anlayacak? Eğer fonksiyon çağrısı bir *const* nesne adresi ile yapılırsa zaten birinci func işlevi uygun *(viable)* olmaktan çıkıyor. Bu durumda çağrılan ikinci fonksiyon olur. Çağrının const olmayan bir nesne adresi ile yapılması durumunda her iki fonksiyon da çağrıya uygun düşse de dilin kurallarına göre çağrılan birinci fonksiyon olur. Aşağıdaki koda bakalım:

```
void func(int *ptr);  //1
void func(const int *ptr); //2

int main()
{
 int x = 10;
 const int y = 20;

 func(&x); //1
 func(&y); //2
}
```

Bu durumu pratik olarak şöyle ifade edebiliriz: Fonksiyon const nesne adresi ile çağrılırsa parametresi *const T** olan Fonksiyon const olmayan nesne adresi ile çağrılırsa parametresi *T** olan fonksiyon çağrılır.

const yüklemesi gösterici semantiği ile yapılabildiği gibi referans semantiği ile de gerçekleştirilebilir. *T* bir tür olmak üzere:

```
void func(T &);  //1
void func(const T &); //2
```

Yukarıdaki bildirimlere göre *func* işlevine *T* türünden *const* olmayan bir nesne gönderildiğinde birinci fonksiyon *const* bir nesne gönderildiğinde ise ikinci fonksiyon çağrılır.

İşlevlerin parametreleri gösterici ya da referans değilse bildirimde kullanılan const anahtar sözcüğü bir anlam farklılığı yaratmaz:

```
void func(int x);  //1
void func(const int x);  //2  yeniden bildirim (function redeclaration)
```

Yukarıdaki bildirimler bir fonksiyon yüklemesi oluşturmuyor. Derleyici ikinci deyimi bir "yeniden bildirim" *(function redeclaration)* kabul eder. Eğer her iki fonksiyon da tanımlansaydı sentaks hatası oluşurdu. Aşağıda yer alan programda çift anlamlılık hatasının olduğu bir başka tipik durum gösteriliyor:

```
#include <iostream>

void foo(int &r)
{
 std::cout << "void foo(int &)\n";
}

void foo(int x)
{
 std::cout << "void foo(int)\n";
}

int main()
{
 int a = 20;

 //foo(a); çift anlamlılık hatası
 foo(5); //void foo(int);
}
```

Yukarıdaki kodda tanımlanan birinci *foo* fonksiyonunun parametre değişkeni *int &* türünden iken ikinci *foo* fonksiyonunun parametre değişkeni *int* türdendir. Bunlar farklı işlevdir. *main* fonksiyonu içinde yorum satırı içine alınmış birinci fonksiyon çağrısı çift anlamlılık hatasına neden olur. Yani bu durumda değerle çağrı *(call by value)* ya da referansla çağrı *(call by reference)* birbirine göre öncelikli değildir. Ancak ikinci fonksiyon çağrısında argüman olan ifade bir sabittir yani bir sağ taraf değeri ifadesidir. Bir sol taraf referansına bir sağ taraf değeri ifadesi ile ilk değer verilemeyeceğine göre çağrılabilecek tek bir fonksiyon vardır. İkinci fonksiyon çağrılacaktır. Birinci fonksiyonun parametre değişkeni *const* sol taraf referansı olsaydı ikinci fonksiyon çağrısı da çift anlamlılık hatası *(ambiguity)* durumuna düşerdi.

Yüklemeye konu iki fonksiyonun parametresi için de standart dönüşümün olduğu durumlarda oluşan çift anlamlılık hatası programcıları yanıltabilmektedir. Aşağıdaki koda bakalım:

```
void f(long double);
void f(char);

void func()
{
 f(12.5); //geçersiz
}
```

Yukarıdaki kodda *func* fonksiyonu içinde yapılan çağrıda çift anlamlılık hatası oluşur. *long double* parametreli fonksiyonun çağrılması gerektiğini düşünebilirsiniz. Argüman olan ifade *double* türündendir. *double* türünden *long double* türüne ve *char* türüne yapılacak dönüşümlerin her ikisi de standart dönüşümdür. Aralarında bir seçilebilirlik farkı yoktur. Çözümlemeye ilişkin bazı özel durumları da inceleyelim:

```
void f(void *);  //1
void f(bool);  //2

void func()
{
 int x = 10;
 f(&x);
 //...
}
```

Yukarıdaki kodda *f* isimli fonksiyona yapılan çağrıda her iki fonksiyon için de yapılacak dönüşümün kalitesi aynı. Bu durumda parametresi *void* olan fonksiyon (birinci fonksiyon) çağrılır.

C++11 standartları ile, dile eklenen sağ taraf referanslarına ilişkin de yeni çözümleme kuralları getirilmiştir.

```
void f(const T &);
void f(T &&);
```

Yukarıdaki fonksiyonlar farklı fonksiyonlardır. Fonksiyon yüklemesi söz konusudur. *f* fonksiyonuna çağrı yapıldığında hangisinin seçileceği çağrıda kullanılacak argüman olan ifadenin değer kategorisine *(value category)* bağlıdır. Fonksiyon bir sol taraf değeri ifadesi ile çağrılırsa *T&&* parametreli fonksiyon zaten uygun olmayacağından const *T&* parametreli olan çağrılır. Fonksiyon bir sağ taraf değeri ifadesi ile çağrılırsa her iki fonksiyon da uygundur. Ancak bu durumda *T&&* parametreli fonksiyonun seçilir.

Yüklenen fonksiyonların birden fazla parametre değişkenine sahip olması durumunda bir fonksiyonun seçilebilmesi için şu şartları sağlaması gerekir:

- Seçilecek fonksiyon az bir parametrede diğer fonksiyonlardan daha iyi olmalıdır.
- Seçilecek fonksiyon kalan parametrelerde diğer fonksiyonlardan daha kötü olmamalıdır.
- Fonksiyonlardan hiçbiri bu koşulu sağlamıyorsa çift anlamlılık hatası oluşur.

Aşağıdaki örneğe bakalım:

```
void f(int, double, float);   //1
void f(int, int, char);    //2
void f(double, long, long double);  //3

void func()
{
 f(12u, 'C', 8.6);
}
```

f fonksiyonu yapılan çağrı *2* numaralı fonksiyona bağlanır. Fonksiyonların parametrelerine yapılan dönüşümler söz konusu olduğunda *2*. fonksiyonun *2*. parametresine yapılan dönüşüm diğerlerine göre daha iyidir. *2*. fonksiyonun diğer parametrelerine yapılan dönüşümler de diğerlerinden daha kötü değildir. Bir de şu örneğe bakalım:

```
void f(double, double);  //1
void f(int, int);  //2

void func()
{
 f(1.2, 'A');
}
```

f fonksiyonuna yapılan çağrı çift anlamlılık hatası oluşturur. Birinci parametre için birinci fonksiyon ikinci parametre için ikinci fonksiyon daha iyidir.

Hangi fonksiyonun çağrılmış olduğunu saptamak derleme zamanında yapılan bir işlemdir. Yani kaynak kod derlenip hedef kod *(object code)* haline getirildiğinde artık hangi fonksiyonun çağrılmış olduğu bilinmektedir. Çünkü çağrılan fonksiyonun kimliği bir şekilde hedef koda yazılmış olur. Başka bir deyişle "*fonksiyon yüklemesi*" için programın çalışma zamanı açısından bir ek maliyet söz konusu değildir. Ancak böyle bir araç derleyici üzerindeki yükü de artırır. Derleyici programın boyutunun büyümesine neden olur. Küçük bir dil olarak tasarlanan C dilinde bu aracın bulunmamasının önemli bir nedeni de budur. Aynı isimli fonksiyonlar gereksiz yere tanımlanmamalıdır. Fonksiyon yüklemesinin ana amacı özünde aynı işi yapan ancak kodları farklı fonksiyonları aynı isim altında soyutlamaktır. Birden fazla fonksiyonun aynı ismi taşıması kullanıcı kodların işini kolaylaştırmaya yöneliktir. Farklı işler gören fonksiyonların, aynı ismi taşıması kodun okunmasını zorlaştırır.