C++ DERS NOTLARI

Necati Ergin

C VE SISTEM PROGRAMCILARI DERNEĞI

TABLE OF CONTENTS

C DİLİ İLE C++ DİLİ ARASINDAKİ FARKLILIKLAR

Bu bölümde ANSI C dili (C89 ISO/IEC 9889:1990) ile C++ dili (ISO/IEC 14882:1998) arasındaki sözdizim (sentaks) farklılıklarını ele alacağız.

Bir C++ derleyicisi ile aynı zamanda C'de yazılmış kaynak kodlar da derlenebilir. C++ derleyicisinin ayarları (settings) değiştirilerek yazılacak kaynak kodların doğrudan C kaynak dosyası olarak ele alınması sağlanabilir. Ayrıca derleyicilerin hemen hepsi, yaratılan kaynak dosyanın uzantısına bakarak -örneğin uzantının .c ya da .cpp olmasına göre- kaynak kodu hangi dilin kurallarına göre derleyeceklerini anlar.

C++ dilinin temel sözdizimsel yapısı C dili sözdiziminin üzerine birtakım eklemelerin yapılmasıyla oluşturulmuştur. Bu durumda C dili sözdizimsel açıdan C++ dilinin bir alt kümesi olarak görülebilir. Bazı noktalarda da C ve C++ dilleri sözdizim kuralları açısından farklılıklar gösterir.

C ile C++ arasındaki sözdizimsel farklılıklar iki ana grupta ele alınabilir:

1. C alt programlama tekniğine destek veren bir dilken, C++ birden fazla programlama tekniğine destek verir (multiparadigm). İki dil arasındaki farklılıkların önemli kısmı, C++‟ ın diğer programlama tekniklerine destek verebilmek için eklediği yeni araçlara ilişkindir.

2. Diğer farklılıkların doğrudan programlama teknikleriyle ilgisi yoktur. Bu farklar C++'ı daha iyi bir C yapma amacına yönelik yapılan değişiklikler ve eklemeleri kapsar.

C‟de yazılan bir dosya C++ diline, C++'da yazılan bir dosya da C diline taşınmak istenebilir. ANSI C dilinin kurallarına göre yazılan ve geçerli olan bir kaynak dosya C++ diline taşındığında, C++ dilinin kurallarına göre geçersiz durumlar oluşabilir.

Şüphesiz bunun tersi de söz konusudur. Yani C++‟da bir sözdizim hatası içermeyen bir kod parçası C dilinin kurallarına göre geçersiz olabilir

Bazı durumlarda da yazılan kod her iki dilin kurallarına göre geçerli olarak değerlendirilse de, yazılan kod iki dilin kurallarına göre farklı anlamlara sahip olabilir. Bir kaynak dosyanın bir dilden diğerine taşınmasında kural değişiklikleri yüzünden uyumsuzluk sorunları ile karşılaşılabilir.

C ve C++ dilleri arasındaki temel sözdizime ilişkin farklılıkları üç ayrı başlık altında ele alacağız:

C'de geçerli C++'da geçersiz olan durumlar C++'da geçerli C'de geçersiz olan durumlar C ile C++ arasındaki kural farklılıkları

C'de Geçerli C++'da Geçersiz Olan Durumlar

1. C'de anahtar sözcük olmayan bazı sözcükler C++'da anahtar sözcüktür. Standart ANSI C dilinin (C89) 32 anahtar sözcüğü vardır. Bu sözcükler aynı zamanda C++ dilinin de anahtar sözcükleridir. Ancak C++ bunlara ek olarak yeni 42 anahtar sözcük daha tanımlamıştır. Aşağıdaki sözcükler C++'da anahtar sözcük iken C'de anahtar sözcük değildir:

```
and/and_eq/asm /bitand /bitor /bool /catch /class /compl /const_cast
/delete /dynamic_cast /explicit /export /false /friend /inline/ mutable
/namespace/ new /not /not_eq /operator /or /or_eq /private /protected
/public /reinterpret_cast/ static_cast /template /this /throw /true /try
/typeid /typename /using /virtual /wchar_t /xor/ xor_eq
```

Geçerli bir C kodunda yukarıdaki sözcüklerden herhangi biri bir isim (*identifier*) olarak kullanılmış olabilir. Ancak aynı kod C++'da derlendiğinde geçerli değildir:

```
void foo()
{
    int new;
    /***/
}
```

Yukarıda *int* türden *new* isimli bir değişken tanımlanıyor. Tanımlama C"de geçerli C++"da geçersizdir.

2. C'de geri dönüş değeri üreten bir işlevin tanımı içinde, *return* deyimi ile bir geri dönüş değeri üretilmez ise bu durum derleyiciler tarafından bir sözdizim hatası olarak değerlendirilmez. Bu durumda çağrılan işlev bir çöp değere *(garbage value)* geri döner. Şüphesiz bu durum bir programlama hatasıdır.

Oysa C++'da geri dönüş değeri üreten bir işlevin tanımı içinde mutlaka *return* deyimi yazılarak bir geri dönüş değeri üretilmelidir. Eğer bir *return* deyimi ile geri dönüş değeri üretilmez ise kod geçersizdir.

```
int func()
{
}
```

Yukarıdaki işlev tanımı C'de geçerli, C++'da geçersizdir.

C++'da geri dönüş değeri üreten işlevlerin tanımı içinde yalın *return* deyimi kullanılamaz:

```
int func()
{
    /***/
    return;
}
```

Yukarıdaki örnekte yer alan return deyimi C'de geçerli ancak C++ geçersizdir. Şüphesiz yukarıda return deyiminin kullanılma biçimi bir programlama hatasıdır. C derleyicilerinin hemen hemen hepsi bu durumu mantıksal bir uyarı iletisi ile bildirir.

Ancak main işlevi C++'da bu kuralın dışında tutulur. main işlevi int türden bir geri dönüş değeri üretecek şekilde tanımlanmış olsa da, main işlevi içinde bir return deyimi yazılmayabilir. Bu durumda derleyici otomatik olarak main işlevinin ana bloğunun sonlanmasından önce, kaynak kod içinde

```
return 0;
```

deyimi yazılmış gibi kaynak kodu ele alır. Kod geçerlidir.

Aşağıdaki kod hem C‟de hem de C++'da geçerlidir. C++ derleyicileri için mantıksal bir uyarı gerektirecek bir durum söz konusu değildir.

```
int main()

{

}
```

3. C'de bir işlevin geri dönüş değerinin türü bilgisi yazılmaz ise, işlevin *int* türden değer döndürdüğü anlaşılır (*implicit int*). Oysa C++'da işlev bildirimlerinde ve tanımlarında geri dönüş değerinin türünü yazmak zorunludur:

```
foo(void);
func(double x)

{

    /***/

}
```

Yukarıda *foo* işlevinin bildirimi C‟de geçerli C++‟da geçersizdir. Yukarıda *func* işlevinin tanımı C‟de geçerli C++‟da geçersizdir.

4. C'de bir işlevin tanımlanmasına ilişkin iki ayrı sözdizim kuralı vardır. Bu sözdizim kuralları eski biçim *(old style)* ve yeni biçim *(new style)* olarak isimlendirilir. Eski biçim işlev tanımlaması programcılar tarafından artık hiç tercih edilmemesine karşın, C dilinin kurallarına göre halen geçerlidir. Oysa C++ dili standartlarına göre eski biçim işlev tanımlaması geçerli değildir:

```
int foo(a, b, c)
int a;

double b, c;

{

    /***/

}
```

Yukarıdaki işlev tanımı C‟de geçerli, C++‟da geçersizdir.

5. C'de işlev bildirimi zorunlu değildir. C derleyicisi bildirimi yapılmayan bir işlevin çağrısıyla karşılaştığında işlevin *int* türden geri dönüş değeri ürettiğini varsayarak kod üretir. Oysa C++'da işlev bildirimi zorunludur:

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

Yukarıdaki örnekte *foo* işlevine yapılan çağrı C'de geçerli C++'da geçersizdir:

Ancak bir işlevin bildirim yapılmadan çağrılması C'de geçerli olmasına karşın doğru kabul edilmez.

6. C'de işlev bildiriminde işlev parametre ayracının içinin boş bırakılması ile parametre ayracının içine *void* anahtar sözcüğünün yazılması farklı anlam taşır:

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

Yukarıdaki örnekte *func* işlevi için yapılan bildirimde parametre ayracı içine bir şey yazılmamıştır. C‟de bu bildirimin anlamı şudur: Bildirilen *func* işlevinin parametre değişkenleri hakkında derleyiciye bir bilgi verilmemiştir. Yani *func* işlevine yapılan çağrıda derleyici işleve gönderilen argüman sayısı ile işlevin parametre değişken sayısının aynı olup olmadığına bakmaz. Geçmişe doğru uyumluluk için bu bildirime özel bir anlam yüklenmiştir. C‟de bu bildirim

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

biçiminde yapılsaydı, derleyiciye *func* işlevinin parametre değişkeni olmadığı bilgisi açıkça verilmiş olurdu.

Oysa C++'da her iki bildirimin de anlamı aynıdır:

```
int func();
int foo(void)
```

C++'da işlev bildiriminde parametre ayracının içinin boş bırakılmasıyla ayracın içine *void* anahtar sözcüğünün yazılması arasında bir fark yoktur. Her ikisinde de işlevin parametre değişkenine sahip olmadığı bilgisi verilmiş olur.

7. C de main işlevinin adresi alınabilir. main işlevi kendi kendini çağırabilir. Ancak C++'da bu duruma izin verilmemiştir.

C++'da main işlevi kendi kendini çağıramaz. main işlevinin adresi alınamaz.

8. C'de belirleyiciler (*auto, register, static, extern, const, volatile*), değişken tanımlamalarında tür belirten anahtar sözcük olmaksızın doğrudan değişken ismiyle kullanılabilir. Bu durumda derleyici, tanımlanan değişkenin *int* türden olduğunu kabul eder. Ancak C++'da belirleyicilerin böyle kullanılması geçerli değildir.

```
void foo()
{
    const i = 0;
}
```

Yukarıdaki kod C‟de geçerli C++‟da geçersizdir.

9. C'de const bir değişkenin ilk değer verilmeden tanımlanması tamamen geçerlidir. Şüphesiz otomatik ömürlü bir nesne için bu durum bir programlama hatasıdır. C++'da const değişkenlere ilk değer vermek zorunludur:

```
void foo()
{
    const int x;
    char *const ptr;

    /***/
}
```

Yukarıdaki tanımlamalar C‟de geçerli, C++‟da geçersizdir.

C++‟da statik ömürlü *const* değişkenlere de ilk değer vermek zorunludur:

```
const int x;
void foo(){}
```

Yukarıdaki örnekte x değişkeninin tanımı C‟de geçerli, C++‟da geçersizdir. Ancak aşağıdaki

tanımlama iki dilde de geçerlidir:

```
const int *ptr;
```

Burada *ptr* değişkeninin kendisi *const* değildir. *ptr* değişkeninin gösterdiği yer *const*

olarak ele alınır.

10. C'de *void* türden bir adresin *void* türden olmayan bir gösterici değişkene, tür dönüştürme işlemi yapılmaksızın atanması geçerlidir. Bazı C derleyicileri bu durumda bir mantıksal uyarı iletisi verebilir. Ama bu durum bir sözdizim hatası olarak değerlendirilmez. Ancak bu durum C++'da bir sözdizim hatasıdır. C++'da *void* türden adresler tür dönüştürme işleci kullanılmadan, başka türden bir göstericiye atanamaz:

```
#include <stdlib.h>

void foo()

{

    int *ptr;

    ptr = malloc(sizeof(int));

    /***/

}
```

Yukarıdaki örnekte *ptr* değişkenine yapılan atama C‟de geçerli, C++‟da geçersizdir.

11. C'de bir gösterici değişkene adres bilgisi dışındaki bir değerin atanması ya da bir gösterici değişkene farklı türden bir adresin atanması yanlıştır. C derleyicilerinin çoğu bu durumu bir mantıksal uyarı iletisiyle bildirir. C‟de doğal türler ile adres türleri arasında otomatik tür dönüşümü vardır.

C++'da bir adres değeri olmayan bir ifadenin bir gösterici değişkene atanması ya da bir adresin farklı türden bir gösterici değişkene atanması durumu doğrudan sözdizim hatasıdır.

```
void foo()

{

    int x = 10;
    double *dp;
    int *ip;

    dp = &x;
    ip = 20;

    /***/

}
```

Yukarıdaki kod parçasında *dp* ve *ip* isimli gösterici değişkenlere yapılan atamalar C++"da geçerli değildir.

12. C'de *const* bir değişkenin adresinin, gösterdiği yer *const* olmayan bir göstericiye atanması yanlıştır. C derleyicilerinin çoğu böyle bir durumda yalnızca bir mantıksal uyarı iletisi verme eğilimindedir. C++'da bir *const* değişkenin adresinin gösterdiği yer *const* olmayan bir göstericiye tür dönüşümü yapılmadan atanması geçersizdir.

C++'da *const* nesnelerin adresleri ayrı bir adres türünden kabul edilir. Böyle adresler ancak gösterdiği yer *const* olan gösterici değişkenlere atanabilir:

```
void foo()

{

    const int x = 20;
    volatile int y = 50;
    int *ptr;

    ptr = &x;
    ptr = &y

}
```

Yukarıdaki kod parçasında *ptr* göstericisine yapılan atamalar C++'da geçerli değildir. Benzer şekilde C++'da gösterdiği yer *const* olan bir göstericinin değeri de, gösterdiği yer *const* olmayan bir göstericiye atanamaz:

```
void func(const char *ptr)
{
    char *temp = ptr;
    /***/
}
```

Yukarıdaki kod parçasında *temp* göstericisine yapılan atama C++'da geçerli değildir.

13. C'de *const* ya da *volatile* bir nesnenin adresi *void* türden bir göstericiye atanabilir. C++'da *void* türden bir göstericiye *const* ya da *volatile* anahtar sözcüğü ile tanımlanmış bir nesnenin adresi tür dönüşümü yapılmadan atanamaz.

void türden göstericilere her türden adres atanabilir. Ancak C++'da *const* ya da *volatile* anahtar sözcüğü ile tanımlanmış bir nesnenin adresi tür dönüştürmesi yapılmaksızın *void* türden bir göstericiye atanamaz.

```
void foo()
{
    void *vptr1, *vptr2;
    const int x = 10;
    volatile int v = 20;
    vptr1 = &x;
    vptr2 = &v;
    /***/
}
```

Yukarıdaki kod C'de geçerlidir. Ancak *vptr1* ve *vptr2* değişkenlerine yapılan atamalar C++'da geçerli değildir.

14. C'de işlev tanımlarında ya da bildirimlerinde, geri dönüş değerinin yazıldığı yerde, parametre değişkenlerinin bildirildiği yerde programcı tarafından tanımlanan türler bildirilebilir. C++'da böyle bildirimler geçerli değildir.

```
struct A{int a1, int a2;} foo();
```

Yukarıdaki bildirim C'de geçerlidir. *foo* işlevinin geri dönüş değeri bildirilen *struct A* yapısı türündendir. Ancak C++'da bu bildirim geçersizdir.

```
void func(struct B{int b1, int b2;} x);
```

Yukarıdaki bildirim de C'de geçerlidir. func işlevinin parametre değişkeni, parametre ayracı içinde bildirilen struct B yapısı türündendir. C++'da bu bildirim geçersizdir.

15. C'de doğal türlerden bir değerin bir numaralandırma türünden değişkene atanması geçerlidir. C++'da bir numaralandırma türünden bir nesneye ancak söz konusu numaralandırma türüne ilişkin bir "numaralandırma değişmezi" (enumaration constant) atanabilir.

```
enum Position {ON, OFF, HOLD};

void foo()

{

    enum Position pos = 1;

    /***/

}
```

pos değişkenine yapılan atama C‟de geçerli iken C++‟da geçersizdir.

16. C'de global bir değişken *extern* anahtar sözcüğü olmadan birden fazla kez bildirilebilir. C++ bir kaynak dosya içinde bir değişken *extern* anahtar sözcüğü kullanılmadan yalnızca bir kez bildirilebilir.

```
int x;
int x;


void foo()
{
}
```

Yukarıdaki kod C'de geçerli iken C++'da geçersizdir.

17. C'de programcı tarafından tanımlanan bir tür için kullanılan etiket-isim (*tag*) aynı bilinirlik alanı içinde kullanılan bir *typedef* ismiyle aynı olabilir. Bu durum bir sözdizim hatası değildir.

```
struct Word {
    char hb, lb;
};


typedef int Word;
```

Yukarıdaki kod parçası C'de geçerli, C++'da geçersizdir.

18. C'de içsel bir yapı bildirimi ait olduğu bilinirlik alanında doğrudan görülebilir.C++'da içsel bildirilen türler doğrudan kullanılamaz:

```
struct Outer {
    struct Inner {
        /***/
    }
```

```
    /***/
}

struct Inner i;
```

Yukarıdaki örnekte *struct Inner* türünün bildirimi *struct Outer* türünün bildirimi içinde yapılıyor. *struct Inner* türünden *i* isimli değişkenin tanımı C'de geçerlidir. Çünkü bu tanımlama noktasında *struct Inner* türü bilinir. Ancak yukarıdaki kod C++'da geçersizdir. C++'da *struct Inner* türü doğrudan değil *Outer::Inner* biçiminde kullanılabilir.

19. C'de *char* türden bir diziye bir dizgeyle ilkdeğer verilirken, dizge içinde yazılan karakterlerin sayısı, tanımlanan dizinin belirtilen boyutuna eşit olabilir. Bu durumda diziye yerleştirilen yazının sonunda sonlandırıcı karakter bulunmaz.

C++'da böyle bir ilkdeğer verme işlemi geçersizdir.

```
void foo()
{
    char message[4] = "hata";
    /***/
}
```

Yukarıdaki kod parçası C'de geçerli, C++'da geçersizdir.

20. C++'da dizgeler bir işleme sokulduğunda *char* türden adrese değil *const char* türden adrese dönüştürülür.

C'de kaynak kod içinde bir dizge gören derleyici, önce bu dizge için bellekte güvenilir bir yer ayarlar. Daha sonra bellekte ayarlanan yerin (bloğun) başlangıç adresini dizgenin yerine yerleştirir. C'de göstericiler yoluyla bir dizgeyi değiştirmeye çalışmak, bir programlama hatası olmasına karşın, bir sözdizim hatası oluşturmaz.

Oysa C++'da dizgeler işleme sokulduklarında *const char* türden bir adres olarak ele alınır. Ancak geçmişte yazılan çok sayıda C++ kodunun geçersiz hale gelmemesi için *const char* türden birer adres olan dizgelerin, *const* olmayan *char* türden göstericilere atanmasına 1998 Standartlarıyla izin verilmiştir. Ancak bu durum *"deprecated"*[(2)] ilan edilmiştir.

Bu durumda 1998 C++ standartlarına göre aşağıdaki kod geçerlidir:

```
char *p = "Necati Ergin";
```

Burada gösterdiği yer *const* olmayan *p* göstericisine *const* bir adres atanması geçerlidir. Ama C++ derleyicilerinin çoğu bu durumda mantıksal uyarı iletisi verir.

Ancak dizgeler herhangi bir ifade içinde, *const* olmayan *char* türden (*char **) bir adres gereken yerde kullanılırsa C++ da bu geçersizdir.

```
void foo()
{
    char *p = "Necati Ergin" + 1;
    char *str = x > 1 ? "Books" : "Book";
    *"selam" = 'k';
}
```

Yukarıdaki deyimler C'de geçerli C++'da geçersizdir.

Yazılacak yeni kodlarda artık dizgeler *const char* türden bir adres olarak görülmeli ve ancak okuma amaçlı gösterici değişkenlere, yani *const char ** türden değişkenlere atanmalıdır.

21. C"de bilinirlik alanı içinde bir sıçramayı sağlayan *goto* ve *switch* kontrol deyimleri bir tanımlama deyiminin bulunduğu kod bölgesinin sonrasına sıçramayı sağlayabilir. C++"da bu durum yasaklanmıştır.

C++'Da Geçerli C'de Hatalı Olan Durumlar

1. C89 standartlarına göre yerel her türlü bildirim blokların başında yapılmak zorundadır. Yani bir blok içinde yapılan bildirimden önce yürütülebilir bir deyim (executable statement) bulunamaz. [C99 standartlarına göre yerel bir blok içinde her yerde bildirim yapılabilir.] C'de böyle bir tasarımın yapılmış olmasının nedeni, programcının yapılmış olan bildirimin yerini kolayca bulabilmesini sağlamaktır.

Oysa C++'da bildirimler ve tanımlamalar blok içinde herhangi bir yerde yapılabilir:

```
void foo()

{

    int y;
    y = 10;

    int x;

    /***/

}
```

Yukarıda *x* değişkeninin tanımlanması C"de geçersiz C++'da geçerlidir.

C++'da bir blok içinde yapılan bildirim ya da tanımlamanın, bildirimin ilk kullanılacağı yere yakın bir yerde yapılabilmesi, kaynak kodun okunabilirliğini artırıcı bir etken olarak düşünülmüştür. Ancak asıl önemli neden, programcı tarafından tanımlanan türlere ait değişkenlere ilişkindir. C++'da böyle nesneler bir çok durumda tanımlanmalarıyla birlikte dışsal bazı kaynakları kullanmaya başlar. Eğer bu değişkenler bloğun en başında tanımlanmak zorunda olsalardı, ilk kullanıldıkları yere kadar gereksiz bir biçimde kaynak kullanmak zorunda kalırlardı.

Bir ismin derleyici tarafından tanınabildiği kaynak kod aralığına bilinirlik alanı (*scope*) denir. C'de yerel değişkenler blok bilinirlik alanına (*block scope*) uyar. Yani bildirilmiş oldukları bloğun başından sonuna kadar herhangi bir yerde bilinirler. C++'da bildirimler blok başlarında yapılmak zorunda olmadığından, bilinirlik alanı kuralını şu biçimde değiştirmek gerekir: C++'da yerel değişkenlerin bilinirlik alanı bildirildikleri yerden ilgili bloğun sonuna kadar olan kaynak kod bölgesidir. C++'da da C'de olduğu gibi, bir blok içinde aynı isimli birden fazla değişkenin bildirimi yapılamaz.

2. C++'da *bool* türü doğal bir veri türüdür.

C'de bool önceden tanımlanmış bir veri türü değildir. (C99 standartları ile C diline de bool türü eklenmiştir.)

```
void foo()

{
```

```
    bool flag;

    /***/

}
```

Yukarıdaki kod C++'da geçerli C'de geçersizdir.

C'de böyle bir gereksinim duyulduğunda bazı araçlar kullanılabilir.

```
#define BOOL       int         /* önişlemci komutuyla */
typedef int BOOL               /* typedef bildirimiyle */

enum BOOL {TRUE, FALSE}        /* numaralandırma türü olarak */
```

Oysa C++'da bool doğal bir türün ismidir ve bool bir anahtar sözcüktür. Bu türden değerleri gösteren true ve false sözcükleri de C++'ın anahtar sözcük kümesine eklenmiştir.

C++'da *bool* türden bir nesne *true* ve *false* değerlerini alabilir.

bool türden bir nesneye başka bir türden değer atanması durumunda bu değerler atama öncesinde otomatik olarak *true* ya da *false* değerlerine dönüştürülür. Yani diğer doğal türlerden *bool* türüne otomatik tür dönüşümü vardır. *bool* türden nesneye atanan 0 dışı

bir değerse *true* değerine 0 değeri ise *false* değerine dönüşüm gerçekleşir. *bool* türden bir nesneye bir adres bilgisi de atanabilir. *NULL* adresi atama öncesi *false* değerine dönüştürülürken diğer adres bilgileri *true* değerine dönüştürülürler.

bool türden bir değer de, diğer doğal türlere atanabilir. Bu durumda atama öncesi *true*

değeri *int* türden *1* değerine *false* değeri ise *int* türden *0* değerine dönüştürülür.

C'de karşılaştırma işleçleri ve mantıksal işleçler *int* türden *1* ya da *0* değerini üretir. Oysa C++'da bu işleçler *bool* türden *true* ya da *false* değerlerini üretir.

3. C++'da değişmez ifadesiyle ilkdeğerini alan *const* değişkenler değişmez ifadesi gereken yerlerde kullanılabilir.

C'de *const* değişkenler, değişmez ifadesi olarak kabul edilmez. *const* değişkenleri içeren ifadeler değişmez ifadesi olarak ele alınmaz. Ancak C++'da değişmez ifadeleri ile ilkdeğerini alan *const* değişkenler, "değişmez ifadesi" olarak kullanılabilir.

```
void foo()

{

    const int size = 10;
    int a[size];

    /***/

}
```

Yukarıdaki kod C++‟da geçerli C‟de geçersizdir.

const nesne tanımlamasında C ile C++ dilleri arasındaki en önemli fark, C++'da *const* anahtar sözcüğü ile tanımlanan değişkenin simgesel değişmez gibi ele alınmasıdır. Yani C++'da *const* anahtar sözcüğü *#define* işleminin derleme modülü tarafından yapılan biçimi gibidir. *const* bir değişkenin kullanıldığını gören C++ derleyicisi, *const* değişken yerine ona verilen ilk değeri değişmez olarak işleme sokar.

4. C++'da yapı, birlik, numaralandırma etiketleri (tags) aynı zamanda tür ismi olarak kabul edilir.

```
struct Complex {

    /*....*/

};
```

```
enum POS {OFF, ON};
```

```
Complex func(Complex, Complex);
POS position;
```

Yukarıdaki örnekte *func* işlevinin bildirimi ve *position* değişkeninin tanımı C++"da geçerli C"de geçersizdir.

C"de bildirilen bu türlerin ismi *struct Complex, enum Pos*"tur. Eğer *struct* ve *enum* sözcükleri kullanılmadan tür bilgisinin ifade edilmesi istenirse, *C*"de bir *typedef* bildirimi yapılmalıdır.

5. C++'da // atomuyla başlayan yorum satırı kullanım olanağı vardır. Bu yorumsatırı biçimi C'de geçerli değildir.

C'de /* ve */ atomları arasında kalan kaynak kod bölgeleri C derleyicileri tarafından yorum satırları olarak ele alınır ve derleme işlemine sokulmaz.

```
x = func(); /* x"e func işlevinin geri dönüş değeri atanıyor */
```

/* ve */ atomları arasında kalan bölgenin uzunluğu bir satırdan daha büyük olabilir.

```
x = func();

/* x değişkenine func işlevinin geri dönüş değeri atanıyor. Ama gerçek
programlarda yorum satırları ancak gereken durumlarda kullanılmalı, yorum
satırlarının gereksiz bir biçimde kullanılması programların
okunabilirliğini zorlaştırır */
```

C++'da bu tür yorum satırları tamamen geçerli olmakla birlikte, ikinci bir yorum satırı biçimi daha eklenmiştir:

C++'da **//** atomunun bulunduğu noktadan satır sonuna kadar olan kısım derleme işlemine sokulmaz. Bu yorumlama biçiminin özellikle kısa açıklamalar için daha kullanışlı olduğu söylenebilir. (C++ bu yorum satırını *BCPL* dilinden almıştır.)

Ancak // yorum satırı alt satırda devam edemez. Eğer açıklama ya da yorum bir satırdan daha uzunsa, her satırın başına tekrar // atomu yerleştirilmelidir.

C'de yorum satırının kapatılmaması zor bulunan hatalara yol açabilir:

```
int a, b;

a = 10;     /* a nesnesine 10 değeri atandı.

b = 20;     /* b nesnesine 20 değeri atanmadı! */
```

Yukarıdaki örnekte birinci yorum satırı kapatılmadığı için `b = 20`

ataması derleme işlemine sokulmaz.

C++ biçimi yorum satırı tek bir atomla oluşturulduğu için yukarıdaki yanlışlığın yapılması olasılığı yoktur:

```
void foo()

{

    int a, b;

    a = 10;     // a nesnesine 10 değeri atandı.
    b = 20;     // b nesnesine 20 değeri atandı.

    /***/

}
```

Yukarıdaki kod C++'da geçerli C'de geçersizdir.

Ancak C derleyicilerinin çoğu // yorum satırlarının kullanımına izin verir. // yorum satırları C99 standartlarıyla C diline de eklenmiştir.

6. C++'da statik ömürlü değişkenlere ilkdeğer verilirken değişmez ifadesi kullanma zorunluluğu yoktur. C'de statik ömürlü değişkenlere değişmez ifadeleriyle ilkdeğer vermek zorunludur.

```
int foo();
int x = 5;
int y = x;


void func()
{
    static int z = foo();
    /***/
}
```

Yukarıdaki örnekte global *y* değişkenine ve statik yerel *z* değişkenine, değişmez ifadesi olmayan ifadelerle ilkdeğer verilmiştir. Bu deyimler C++'da geçerli iken C'de geçersizdir.

7. *C++'da for* döngü deyimi ayracının birinci kısmında, *if*, *while*, *switch* deyimlerinin ayracı içinde değişken tanımlanabilir.

C'de yerel değişkenler yalnızca blok başlarında ve işlev parametre ayraçlarının içinde tanımlanabilir. C++'da for döngülerinin birinci kısmında, while döngülerinin ayraçları içinde ve if deyiminin ayracı içinde değişken tanımlama olanağı getirilmiştir.

```
#include <cstdio>

int main()

{

    for (int i = 0; i < 100; ++i)
        printf("%d\n", i);


    return 0;

}
```

Yukarıdaki for deyimi C++"da geçerli, C"de geçersizdir.

for döngü deyimi ayracının birinci kısmında tanımlanan nesne herhangi bir türden olabilir. Burada virgül işleci kullanılarak aynı türden birden fazla nesne de tanımlanabilir:

1998 ISO standartları öncesinde *if*, *while* ve *switch* ayraçları içinde bildirim yapılmasına izin verilmiyordu. Ancak standartlar ile bu durumlar da mümkün kılındı. Eski derleyicilerin bu deyimlerin ayraçları içinde değişken tanımlamasına izin vermeyebileceğini hatırlatalım. Çünkü bu durum 1998 standartları ile C++ diline eklenmiştir.

Peki *for* ayracının birinci kısmında tanımlanan değişkenin bilinirlik alanı nedir? Yukarıdaki programda, *for* döngüsünün birinci kısmında tanımlanan *i* değişkeni nerelerde kullanılabilir?

Bu tür değişkenlerin bilinirlik alanına ilişkin kural standartlar ile değiştirilmiştir. C++ standartları öncesinde –Bazı derleyiciler halen bu kurala göre derlemektedir- bilinirlik alanına ilişkin kural şu şekildeydi:

for ayracı içinde tanımlanan değişken, *for* döngüsünün içinde bulunduğu bloğun sonuna kadar, yani kapanan küme ayracına kadar bilinir.

Ancak 1998 standartları, *for* ayracı içinde tanımlanan değişkenlerin bilinirlik alanını döngü gövdesi ile *(loop body)* sınırlamıştır. Yani aşağıdaki kod parçası daha önceki kurallara göre geçerliyken, C++ standartlarına göre geçersizdir.

```
#include <cstdio>

int main()

{

    for (int i = 0; i < 100; ++i)
        printf("%d\n", i);


    printf("i = %d\n", i);     //Geçersiz

    return 0;

}
```

Döngü değişkeninin bilinirlik alanına ilişkin bu değişiklik tehlikeli böceklerin kaynağı olabilir.

```
#include <cstdio>

int i = 0;


int main()

{

    for (int i = 0; i < 100; ++i)
        printf("%d\n", i);

    i = 100;    // global değişken olan i'ye atama yapılıyor.

    //...

}
```

Yukarıda programda

```
i = 100;
```

atama deyimiyle, hangi *i* değişkenine atama yapılmaktadır? Eski derleyicilere göre *for* deyimi ayracı içinde tanımlanan *i* değişkenine atama yapılmaktadır. Çünkü dar bilinirlik alanında olan aynı isimli değişken geniş bilinirlik alanındaki aynı isimli değişkeni maskeler! Oysa standart *ISO C++* dilinin kurallarına göre yalnızca global değişken olan i'ye erişilebildiği için global değişken olan *i*'ye atama yapılır.

Diğer denetim deyimlerinin ayraçları içinde değişken tanımlanması uygulamada bir fayda getirir mi? Ana tema ayraç içinde tanımlanan değişkene bir işlev çağrısının ürettiği geri dönüş değerinin atanması ve bu değişkenin yalnızca denetim deyiminin gövdesinde kullanılmasıdır:

```
if (int x = get_value()) {
    //...
}
```

8. C++'da statik ömürlü değişkenlere değişmez ifadesi ile ilk değer vermek zorunlu değildir.

C'de statik ömürlü değişkenlere, yani global değişkenler ve *static* anahtar sözcüğü ile tanımlanmış yerel değişkenlere ilk değer verilmesi durumunda, ilk değer veren ifadenin (*initializer*) değişmez ifadesi olması gerekir. Yani ilkdeğer veren ifade içinde daha önce tanımlanmış bir değişken yer alamaz. Bunun nedeni C'de, statik ömürlü nesnelerin *amaç* kod içine ilk değerleriyle birlikte yazılmasıdır. Bunun mümkün olabilmesi için verilen ilkdeğerlerin derleme aşamasında belirlenmiş olması gerekir. Derleme aşamasında saptanabilmesi için ifadenin değişmez ifadesi olmalıdır. Oysa C++'da statik ömürlü nesnelere her türden bir ifadeyle ilk değer verilebilir. Bu değişkenlere ilk değer verilmemiş olsa da, bu değişkenler *0* değeriyle amaç kod içine yazılır. Programın çalışma zamanı sırasında ve *main* işlevinden önce ilk değerlerini alırlar.

9. C++ da iki karakterlik ayıraç atomları tanımlanmıştır. Derleyici ve önişlemci program bu kakarakter çiftelerini gördüğü yerde bunlara eşdeğer karakterlerinin bulunduğunu varsayar:

<: [

:>]

<% {

%> }

%: #

C'de bu karakter çifteleri geçerli değildir:

```
%:include <stdio.h>

void copy(char dest<::>, const char source<::>, size_t nbytes)

<%

    while (nbytes--)

        *dest++ = *source++;

%>
```

Yukarıdaki program C++'da geçerli, C'de geçersizdir.

10. C++'da sonek konumundaki ++ işleci ile oluşturulan ifade sol taraf değeridir. C'de böyle ifadeler sol taraf değeri değildir:

```
#include <stdio.h>

int main()

{

    int x = 5;


    ++x = 10;

    printf("x = %d\n", x);


    return 0;

}
```

Yukarıdaki programda yapılan

```
++x = 10;
```

ataması C++'da geçerli, C'de geçerli değildir.

11. C++'da koşul işleci ile oluşturulmuş bir ifade, koşul işlecinin ikinci ve/veya üçüncü terimleri nesne ise, bir sol taraf değeridir. C'de koşul işleciyle yazılan bir ifade hiç bir zaman sol taraf değeri değildir:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>


int main()

{

    int x, y;

    x = y = 0;


    srand((unsigned int)time(0));

    (rand() % 2 ? x : y) = 1;


    printf("x = %d\n", x);

    printf("y = %d\n", y);


    return 0;

}
```

Yukarıdaki programda yapılan

```
(rand() % 2 ? x : y) = 1;
```

deyimi C++'da geçerli, C'de geçersizdir.

12. C++'da virgül işleci ile oluşturulmuş bir ifade, virgül işlecinin sağ terimi bir nesne ise bu nesneye karşılık gelir. C'de virgül işleciyle yazılan bir ifade hiç bir zaman sol taraf değeri değildir:

```
#include <stdio.h>

int main()

{

    int x = 10;
    int y;

    (x, y) = 30;

    printf("x = %d\n", x);

    printf("y = %d\n", y);


    return 0;

}
```

Yukarıdaki programda yazılan

```
(x, y) = 30;
```

deyimi C++'da geçerli iken C'de geçersizdir.

13. C++'da bir *typedef* bildiriminin özdeş olarak yinelenmesi geçerlidir. C'de bir *typedef*

bildiriminin ikinci kez yazılması bir sözdizim hatasıdır.

```
typedef double dollar;
typedef double dollar;
```

Yukarıdaki bildirimler C++'da geçerli, C'de geçersizdir.

14. C++'da tür dönüştürme işlecinin işlevsel biçimi vardır. işlecin böyle kullanımı C'de geçerli değildir.

```
void foo()

{

    double d = 4.5;
    int x;

    x = int (d);

    /***/

}
```

Yukarıdaki örnekte *double* türünden *d* değişkeninin değeri atama öncesinde tür dönüştürme işlecinin işlevsel biçimi kullanılarak *int* türüne dönüştürülüyor.

Yukarıdaki kod C++'da geçerli olmakla birlikte C'de geçerli değildir.

15. C++'da geri dönüş değeri üretmeyen bir işlevin *return* deyiminde *return* anahtar sözcüğünü *void* türünden bir ifade izleyebilir. C'de böyle işlevlerde *return* anahtar sözcüğünü bir ifade izleyemez:

```
void foo();


void func()

{

    /***/

    return foo();

}
```

Yukarıdaki örnekte kullanılan *return* deyimi C++'da geçerli, C'de geçersizdir.

16. C++'da bir işlev tanımında parametre değişkenine isim vermek zorunlu değildir:

```
void func(int)

{

    /***/

}
```

Yukarıdaki örnekte *func* işlevinin tanımı C++'da geçerli C'de geçersizdir. Şüphesiz böyle bir işlev tanımı yapıldığında isim verilmeyen parametreyi işlev içinde kullanmak mümkün değildir. Zaten C++'da da böyle işlevlerin tanımlanmasının nedeni "işlev yüklemesi" ismi verilen araçla ilgilidir.

17. C++'da dizilere, yapı ya da birlik nesnelerine ilkdeğer verilirken, değişmez ifadeleri kullanmak zorunluluğu yoktur. Ancak C'de bu durumlarda ilk değer verici ifade olarak yalnızca değişmez ifadeleri kullanılabilir.

```
void func(int x)
{
    int a[3] = {x, x + 1, x + 2};
    /***/
}
```

Yukarıdaki örnekte *a* dizisinin tanımı C++'da geçerli, C'de geçersizdir.

C ile C++ Arasındaki Kural Farklılıkları

1. C'de bir numaralandırma türü derleyici açısından *signed int* türünden farklı değildir. Bu yüzden bir numaralandırma türünden nesnenin değeri *<= INT_MAX >=INT_MIN* (="ten sonra boşluk var mı yok mu) olabilir. *BLUE* bir numaralandırma değişmezi olmak üzere C'de

```
sizeof(int) == sizeof(BLUE)
```

ifadesi her zaman doğrudur.

Oysa C++'da, derleyici numaralandırma türü için kullanacağı tamsayı türünün seçiminde serbesttir. Numaralandırma türü için *signed int*, *unsigned int*, *signed long*, *unsigned long* türlerinden herhangi biri seçilebilir. Bir çok sistem söz konusu olduğunda C++ da C'ye göre daha büyük numaralandırma değişmezi tanımlanabilir.

(*boş satır*)

C++ derleyicisi bir numaralandırma türünün bildirimini gördüğünde bildirmde kullanılan numaralandırma değişmezlerinin değerine bakar. Eğer tüm değerler *int* türü sayı sınırlarında ise, numaralandırma türü olarak *int* türü seçilir. Numaralandırma değişmezlerinin değer(ler)inin *signed int* türünde tutulamaması durumunda sırasıyla *unsigned int, signed long, unsigned long* türlerinin uygunluğu sınanır.

Yukarıdaki örnek söz konusu olduğunda

`sizeof(int) == sizeof(BLUE)` ifadesi

doğru olmak zorunda değildir.

2. C++'da karakter değişmezlerinin *char* türünden olduğu varsayılır. Karakter değişmez ifadeleri *char* türdendir. C++"da

```
sizeof('A') == sizeof(char) == 1
```

olduğu güvence altına alınmıştır.

C"de karakter değişmezlerinin *int* türünden olduğu varsayılır ve karakter değişmez ifadeleri *int* türdendir. C"de

```
sizeof('A') == sizeof(int)
```

iken C++'da

```
sizeof('A') == sizeof(char)
```

ifadesi doğrudur.

3. C'de *const* anahtar sözcüğü ile tanımlanmış global nesneler doğrudan dış bağlantıya (*external linkage*)sahiptir. Oysa C++'da *const* global nesneler iç bağlantıya sahiptir. Programı oluşturan başka modüllerde bu nesnelere *extern* bildirimi ile erişilemez. Başka bir deyişle C++"da *const* global nesneler, *static* anahtar sözcüğü ile tanımlanmasalar da *static* anahtar sözcüğü ile tanımlanmış gibi ele alınır.

(*boş satır*)

C++'da global *const* bir nesnenin dış bağlantıya sahip olması isteniyorsa bu nesneler

extern anahtar sözcüğü ile tanımlanmalıdır:

```
const int size = 10;              //size C++'da iç bağlantıya sahip.
extern const int value = 20;      //value C++'da dış bağlantıya sahip.
```

Yukarıdaki örnekte tanımlanan değişkenlerden *size* iç bağlantıya sahipken *value* değişkeni dış bağlantıya sahiptir.

(*boş satır*)

4. C'de bir yapının etiket ismi (*structure tag*) ait olduğu yalanına katılmaz. Bir yapının etiket ismi hiç bir zaman daha geniş bir bilinirlik alanındaki aynı ismi maskelemez.Ancak C++'da programcı tarafından tanımlanan türlere ilişkin etiket isim, bildirimin yapıldığı bilinirlik alanına katılır.

```
int a[100];

void foo()

{

    struct a{int i;}
    int n;


    printf("sizeof(a) = %d\n", sizeof(a));

}
```

Yukarıdaki örnekte *sizeof(a)* ifadesinde kullanılan *a* C'de global diziye ilişkin iken C++'da

foo işlevi içinde bildirilen *a* isimli yapıya ilişkindir.

REFERANSLAR

Referanslar yazılımsal bazı amaçlara sağlamak için kullanılan düzeyi yükseltilmiş göstericiler olarak düşünülebilir.

Referanslar Nasıl Tanımlanır

Nasıl bir gösterici tanımlanırken gösterici isminin önünde '*' atomu kullanılıyorsa, bir referans tanımlanırken de, referans isminden önce „&" atomu yazılır. Örneğin:

```
int i = 20;
double d = 10.2;

int &ri = i;    // ri int türden bir referanstır.
double &rd = d;//rd double türden bir referanstır.
```

Gösterici değişkenlerden farklı olarak, bir referans aynı türden bir nesne ile ilkdeğer verilerek tanımlanmalıdır. Yukarıdaki örnekte, *ri* referansı *int* türünden *i* nesnesi ile, *rd* referansı ise *double* türünden *d* nesnesiyle ilkdeğer verilerek tanımlanıyor. Yukarıdaki tanımlamalarda kullanılan '&' bir işleç değildir. Yalnızca tanımlanan nesnenin bir referans olduğunu derleyiciye bildirir. Referansın ilkdeğer verilmeden tanımlanması geçersizdir.

Derleme zamanı hatası olarak değerlendirilir. Örneğin:

```
int &a;
```

tanımlaması geçersizdir.

Bir referans ilk değer olarak ona atanan nesnenin yerine geçer.

```
int &r = b;
```

gibi bir tanımlamadan sonra artık *r* nesnesi, görülebilir olduğu her kaynak kod noktasında *b* değişkenin yerine geçer, onun yerine kullanılabilir. Artık *b* değişkenine ulaşmanın bir başka yolu da, *b* ismini kullanmak yerine doğrudan *r* ismini kullanmaktır. Bu durum şöyle de ifade edilebilir:

r referansı *b* değişkenine bağlanmıştır.

Tanımlamadan Sonra Referansın Kullanılması

Tanımlama işleminden sonra referans değişkeni kullanılırsa, referansın bağlandığı değişken kullanılmış olur:

```
#include <iostream>

int main()

{

    int a = 10;
    int &r = a;
    r = 20;


    std::cout << "a = " << a << std::endl;

    return 0;

}
```

Yukarıdaki *main* işlevini inceleyelim:

r referansı daha önce tanımlanan *a* değişkenine bağlanıyor. Böylece *r* referansı *a* değişkeninin yerine geçiyor. Daha sonra *r* referansına yapılan atama, aslında *a* değişkenine yapılmış olur.

Bu işlemler referans değil de gösterici değişken kullanılarak yapılmış olsaydı, aşağıdaki gibi bir kod yazılabilirdi:

```
#include <iostream>

int main()

{

    int a = 10;
    int *ptr = &a;

    *ptr = 20;

    std::cout << "a = " << a << std::endl;


    return 0;

}
```

Gerçekten de derleyiciler her iki program için de aynı makine kodlarını üretir.

İlk değerini alan bir referans, yani bir nesnenin yerine geçen bir referans, bir işlecin terimi olduğunda, işleç referansı değil, referansın yerine geçtiği nesneyi işleme sokar:

```
#include <iostream>


using namespace std;

int main()

{

    int a = 10;
    int &r = a;



    cout << "a = "  << a << endl;
    r += 2;

    cout << "a = "  << a << endl;
    r = -r;

    cout << "a = "  << a << endl;



    return 0;

}
```

```
r += 2
```

deyimiyle *a* değişkeninin değeri *2* artırılıyor. Artık *a*'nın yeni değeri *12* olur.

```
r = -r;
```

deyimiyle *a* değişkeninin değeri *-12* yapılıyor.

Bir referansın işlevini iyi anlayabilmek için, kodun gösterici değişkenlerle oluşturulmuş eşdeğer karşılıkları göz önüne alınabilir. Örneğin yukarıdaki program parçasının gösterici değişken ile oluşturulmuş eşdeğer karşılığı aşağıdaki gibidir:

```
#include <iostream>


using namespace std;

int main()

{
```

```
    int a = 10;
    int *ptr = &a;


    cout << "a = "  << a << endl;

    *ptr += 2;

    cout << "a = "  << a << endl;

    *ptr = -*ptr;

    cout << "a = "  << a << endl;


    return 0;

}
```

Tanımlanmış Bir Referans Bir Başka Nesnenin Yerine Geçemez

Bir referans, tanımlanmasıyla birlikte bir nesnenin yerine geçer. Artık söz konusu referans, bilinirlik alanı içinde hep aynı nesnenin yerine geçer. Referansın daha sonra bir başka nesneye bağlanması mümkün değildir:

```
#include <iostream>

int main()

{
    int a = 10;
    int b = 20;
    int &r = a;
    r = b;

    std::cout << "a = " << a << std::endl;


    return 0;

}
```

Yukarıdaki *main* işlevinde tanımlanan *r* referansı, *a* değişkeninin yerine geçiyor. Daha sonra yapılan

```
r = b;
```

ataması, *r* referansının *b* değişkeninin yerine geçtiği anlamına gelmez. Bu atamanın anlamı şudur: *r* referansının bağlandığı nesneye *b* değişkeninin değeri atanmaktadır. Derleyicilerin ürettiği kod açısından bu durum şu

şekilde de değerlendirilebilir: Referanslar kendisi *const* olan göstericilere karşılık gelir. Yukarıdaki kodun bir referans ile değil de bir gösteri değişken kullanılarak yazıldığını düşünelim:

```
#include <iostream>

int main()

{

    int a = 10;
    int b = 20;

    int *const ptr = &a;

    *ptr = b;

    std::cout << "a = "  << a << std::endl;
    ptr = &b;  //Geçersiz



    return 0;

}
```

Yukarıdaki örnekte *ptr,* kendisi *const* olan bir gösterici değişkendir. *ptr* değişkeninin tanımından sonra, *ptr*'ye artık başka bir nesnenin adresi atanamaz, değil mi?

Referansların const Olarak Bildirilmesi

Bir referans *const* anahtar sözcüğü ile tanımlanabilir. *const* anahtar sözcüğü „*&*" atomundan önce yazılır. Bu şekilde tanımlanmış bir referans değişkeninin yerine geçtiği nesne, referans yoluyla değiştirilemez. Örneğin:

```
int main()

{

    int a = 10;



    const int &r = a;

    r = 20;     // Geçersiz!



    return 0;

}
```

main işlevi içinde yer alan

```
r = 20
```

ifadesi geçerli değildir. Atama aslında *a* değişkenine yapılır. *const* referans kullanılarak, referansın yerine geçtiği nesne değiştirilemez. Ancak tabii, *const* referans sağ taraf değeri olarak işlemlere sokulabilir. Yukarıdaki program parçasının göstericilerle oluşturulan eşdeğer C karşılığı şöyle olur:

```
int main()
{
    int a = 10;
    const int *const ptr = &a;

    *ptr = 20;     //Geçersiz
    return 0;
}
```

Burada *ptr,* hem kendisi *const* hem de gösterdiği yer *const* olan bir gösterici değişkendir. *ptr*'nin kendisine yapılan atamalar geçersiz olduğu gibi *ptr*'nin gösterdiği nesneye yapılan atamalar da geçersizdir.

Peki *const* anahtar sözcüğü *„&"* atomundan sonra, referansın isminden önce kullanabilir mi? Gösterici isminden önce *„*"* atomundan sonra *const* anahtar sözcüğünü kullanıldığında, bu durum göstericinin kendisinin *const* olduğu anlamına geliyordu.

```
int x;
int *const ptr = &x;
int &const r = x;      //Geçersiz!
```

Yukarıdaki *r* isimli referansın tanımı geçersizdir. Çünkü referanslar zaten tanımları gereği yalnızca belirli bir nesnenin yerine geçmek üzere yaratılır. Yani bir referans, zaten bir nesne ile ilkdeğerini aldıktan sonra artık başka bir nesnenin yerine geçemez. Referanslar bu anlamlarıyla zaten kendileri *const* nesnedir. Dolayısıyla, *const* anahtar sözcüğünün yukarıdaki biçimde kullanılmasına gerek yoktur.

const bir nesne adresinin, ancak gösterdiği yer *const* olan bir göstericiye atanabileceğini anımsayın:

```
const int x = 10;
```

```
int *p1 = &x;       //Geçersiz
const int *p2 = &x;    //Geçerli
```

Benzer şekilde, *const* olmayan bir referans *const* bir nesnenin yerine geçemez. *const* bir nesnenin yerine ancak *const* bir referans geçebilir.

```
const int x = 10;

int &r1 = x;        //Geçersiz!
const int &r2 = x; //Geçerli
```

const Referanslar Neden Kullanılır

Gösterdiği yer *const* olan göstericiler ne amaçla kullanılıyorlarsa, *const* referanslar da aynı amaçla kullanılır. *const* referanslar ile bunların bağlandıkları nesneler değiştirilemez. Böyle referanslar, yalnızca okuma amacıyla nesnelerin yerine geçer.

Referansların Parametre Değişkeni Olarak Kullanılması C++"da referanslar genellikle işlev tanımlamalarında parametre değişkeni olarak kullanılır. Parametre değişkeni referans olan işlevler, aynı türden bir nesne ile

çağrılmalıdır. Böyle bir çağrı sonucunda parametre değişkeni olan referans argüman olan nesnenin yerine geçer:

```
#include <iostream>

void func(int &r)

{

    r = 20;

}


int main()

{

    int a = 10;


    func(a);

    std::cout << a << std::endl;


    return 0;

}
```

Parametre değişkenlerinin işlev çağrıldığında otomatik olarak yaratıldığını hatırlayalım.

Örneğin *func* isimli işleve

```
func(a);
```

biçiminde yapılan çağrı ile aslında *a* değişkeni, *func* isimli işlevin referans parametresine ilkdeğer olarak atanır. Yani işleve yapılan çağrı ile parametre değişkeni olan *r* referansı *a* değişkeninin yerine geçer. *func* işlevine gönderilen *a* değişkeninin kendisidir.

Aslında derleyicinin ürettiği kod söz konusu olduğunda, gerçekte bir adres aktarımı yapılmaktadır. Parametresi referans olan bir işlevin bir adres bilgisiyle değil, bir nesnenin kendisiyle çağrıldığına dikkat edin. Çağrı bu biçimde yapılsa da aslında bir adres aktarımı söz konusu olur.

Aşağıdaki örneği inceleyin:

```
#include <iostream>

using namespace std;


void swap(int &a, int &b)

{

    int temp = a;
    a = b;

    b = temp;

}


int main()

{

    int x = 10;
    int y = 20;

    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    swap(x, y);


    cout << "x = " << x << endl;
    cout << "y = " << y << endl;


    return 0;
```

```
}
```

Burada *swap* isimli işlev iki değişkenin değerini takas eder. İşlev

```
swap(x, y);
```

biçiminde çağrılıyor. *swap* işlevinin parametre değişkenleri olan referanslar *x* ve *y* nesnelerinin yerine geçer. Şüphesiz, işleve aslında *x* ve *y* değişkenlerinin adresleri aktarılmaktadır. İşlev içinde kullanılan *a* aslında *x* in, *b* ise *y*"nin yerine geçer. Yukarıdaki programın gösterici değişkenlerle oluşturulmuş eşdeğer C karşılığı da şöyledir:

```
#include <iostream>

using namespace std;


void swap(int *a, int *b)

{

    int temp;
    temp = *a;

    *a = *b;

    *b = temp;

}


int main()

{

    int x = 10;
    int y = 20;



    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    swap(&x, &y);

    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
```

```
    return 0;

}
```

const Referans Parametreli İşlevler

Bir işlevin parametresi *const* referans da olabilir. Böyle bir işlev bir nesne üzerinde değişiklik yapmayan, yalnızca o nesnenin değerini kullanan bir işlevdir.

```
void access(const T &r);
```

Yukarıda bildirimde yer alan *T* isminin, bir tür bilgisi olduğunu düşünelim. *access* isimli işlev, *T* türünden bir nesne üzerinde işlem yapabilir, ama söz konusu nesneyi değiştiremez. *access* işlevi *T* türünden bir nesnenin yalnızca değerinden faydalanır. `void mutate(T &r);`

mutate isimli işlev de *T* türünden bir nesne üzerinde işlem yapar ama söz konusu nesneyi değiştirebilir. Sözdizim kuralları açısından bakıldığında şüphesiz *mutate* işlevinin kendisine gelen *T* türünden nesneyi değiştirmesi konusunda bir zorunluluk yoktur. Ancak böyle bir işlevin kendisine gelen nesneyi değiştireceği kabul edilmelidir. Eğer nesneyi değiştirmeseydi işlevin parametresi *const* olarak seçilirdi.

Bu durumda şunu da söyleyebiliriz: Bir işlev referans yoluyla adresini aldığı nesne üzerinde bir değişiklik yapmayacak ise, işlevin parametresi *const* referans yapılmalıdır. Bir işlevin parametresi *const* olmayan bir referans ise ve işlev aldığı nesne üzerinde değişiklik yapmıyor ise, bu durum bir programlama hatası olarak değerlendirmelidir.

İşlevin arayüzünü yani bildirimini gören programcılar, işlevin gönderilen nesne üzerinde değişiklik yapacağını düşünürler.

Yapı Nesnelerinin Referans Yoluyla İşlevlere Geçirilmesi

Bir yapı nesnesinin bir işleve iki şekilde geçirilebileceğini biliyorsunuz.

1. Yapı nesnesinin değerinin geçirilmesi durumu *(call by value)*. Bu durumda işlevin parametre değişkeni bir yapı değişkenidir. İşlev başka bir yapı değişkenin kendisi ile çağrılır. Aynı türden iki yapı değişkeni birbirine atanabildiğine göre, bu çağrı biçimi de geçerlidir. Aşağıdaki örneği inceleyin:

```
#include <iostream>

struct Person {

    char name[30];
    int no;

};
```

```
void display_person(Person y)
{
    std::cout <<  y.name << '\n' <<  y.no << std::endl;
}

int main()
{
    Person per = {"Necati Ergin",  123};
    display_person(per);

    return 0;
}
```

Bu tür bir aktarımda, yapının karşılıklı elemanları birbirlerine blok olarak kopyalandığı için göreli bir zaman kaybı söz konusudur. Bu nedenle bu tür aktarım biçimi C dilinde kötü bir teknik olarak kabul edilir ve pek kullanılmaz. Tabii çok küçük yapı nesnelerinin bu biçimde aktarılması söz konusu olabilir. İşlev değerle çağrıldığı için, yani işlev çağrı ifadesindeki argüman olan yapı nesnesi işlevin parametre değişkenine kopyalanarak aktarıldığı için, bu yolla argüman olan yapı nesnesinin işlev tarafından değiştirilebilmesi olası değildir.

2. Yapı nesnesinin adresinin işleve geçirilmesi durumu (call by reference). Bu durumda işlev yapı değişkenin adresi ile çağrılır. İşlevin parametre değişkeni de aynı türden bir yapı göstericisi olur. Bu aktarım biçiminde yapı nesnesi ne kadar büyük olursa olsun gerçekte aktarılan tek bir adres bilgisidir. İşlev tanımı içinde adresi alınan yapı nesnesinin elemanlarına erişmek için parametre değişkeni olan gösterici -> işlecinin terimi olur. Yapı değişkenleri çoğunlukla işlevlere bu biçimde geçirilir. Aşağıdaki örneği inceleyin:

```
void display_person(const Person *ptr)
{
    std::cout <<  ptr->name << '\n' <<  ptr->no << std::endl;
}

int main()
{
    Person per = {"Necati Ergin",  123};
    display_person(&per);
```

```
    return 0;

}
```

Madem C++ dilinde referanslar bir çeşit göstericidir, o halde yapıların da verimli bir biçimde referans yöntemiyle aktarımları da söz konusu olabilir. Böyle bir aktarım biçiminde, işlev yapı nesnesinin kendisiyle çağrılır. İşlevin parametre değişkeni aynı yapı türünden bir referans olur. Böyle bir çağrıda işlevin parametre değişkeni olan referans, işleve argüman olarak gönderilen yapı nesnesinin yerine geçer. Peki işlev içinde yapı nesnesinin elemanlarına nasıl erişilir? Nokta işleci ile mi, ok işleci ile mi? Evet nokta işleci ile. Çünkü artık yapı referansı kullanıldığında, referans dışarıdan gönderilen nesnenin yerine geçeceğine göre parametre olan referans dışarıdan gönderilen yapı nesnesinin yerine geçer. Yani referansın kullanımı bir adres belirtmez. O nedenle yapı referansı ile yapı elemanlarına erişimde nokta işleci kullanılır. Aşağıdaki örneği inceleyin:

```
#include <iostream>


using namespace std;

struct Person {

    char name[30];
    int no;

};


 void display_person(const Person &r)

{

    cout <<  r.name << '\n' <<  r.no << endl;

}


int main()

{

    Person per = {"Necati Ergin",  123};
    display_person(per);



    return 0;

}
```

İşlevin Parametre Değişkeni Gösterici mi Referans mıOlmalı

C++ dilinde argüman aktarımında gösterici mi yoksa referans mı kullanılmalıdır? Nesne yönelimli programlama söz konusu olduğunda, gösterici kullanmak yerine mümkün olduğu kadar referans kullanmak gerektiği söylenebilir. Bir çok programcı şu ilkeyi benimser:

*"*Kullanabildiğin her yerde referans kullan, ancak zorunlu olduğun yerde gösterici kullan*!" (use references wherever you can, use pointers when you have to!).*

Ancak C++ dili yalnızca nesne yönelimli programlama tekniği ile program yazmak için kullanılmak zorunda değildir. C++ ile şüphesiz "prosedürel" programlama tekniği kullanılarak da program yazılabilir. Eğer amacınız C++"ı daha iyi bir C olarak kullanmak ve prosedürel programlama tekniği ile program yazmak ise, makina düzeyinde olanları daha iyi betimlediği için gösterici kullanmayı tercih edebilirsiniz.

C++"da, *a* bir nesne olmak üzere aşağıdaki gibi bir işlev çağrısının yapıldığını düşünelim:

```
func(a);
```

Argüman olan ifadede *a* nesnesinin ismi yazılmıştır. Bu durumda iki olasılık söz konusudur:

1. İşlevin parametre değişkeni aynı türden bir nesnedir. Bu durumda derleyici argüman olan nesnenin değerini parametre değişkenine kopyalayacak bir kod üretir. Örneğin:

```
void func(int val)
{
    val = 20;
}

int main()
{
    int a = 5;
    func(a);

    return 0;
}
```

Yukarıdaki örnekte *a* değişkeninin değeri, işlev çağrıldığında yaratılan, parametre değişkeni *val*"e kopyalanır. Bu yüzden *func* isimli işlevde yapılan

```
val = 20;
```

atamasının *main* işlevinin *a*'sı ile bir ilgisi yoktur. Atama *func* işlevinin parametresi olan

val değişkenine yapılır. *func* değerle çağrılan (*call by value)* bir işlevdir.

2. İşlevin parametre değişkeni aynı türden bir referanstır. Bu durumda gizli bir adres aktarımı söz konusudur. Yani derleyici, çağrı ifadesinde argüman olarak kullanılan nesnenin adresini çağrılan işlevin referans parametresine kopyalayacak bir kod üretir.

```
#include <iostream>

using namespace std;

void func(int &r)

{

    r = 20;

}


int main()

{

    int a = 5;


    func(a);

    cout << "a = " << a << endl;


    return 0;

}

r = 20
```

atamasıyla gerçekte *main* işlevi içinde tanımlanan *a* değişkeni değiştirilir. *func* isimli işlevin içinde kullanılan *r* *a* nesnesinin yerine geçer. Bu durumu, *a* nesnesinin kendisinin *func* işlevine gönderilmesi olarak görebilirsiniz.

C++"da bir işlevin

```
func(a);
```

biçiminde çağrıldığını gören programcı, eğer işlevin tanımını ya da bildirimini görmemişse standart C"deki alışkanlıkla işlevin *a* değişkenini değiştiremeyeceğini düşünerek yanlış bir fikir edinebilir. Nesnenin değerinin mi, yoksa adresinin mi kopyalanacağının net olarak bilinmemesi bazı durumlarda okunabilirliği azaltabilir. Bu durumu engellemek için referans kullanımına dikkat edilmelidir.

Ancak Nesne Yönelimli Programlama Tekniği (*NYPT*), programın programlama dilinin çalışma düzleminde değil de, problemin kendi düzleminde yazılmasını hedefler. Bu durumda nesne yönelimli olarak yazılmış bir programda, programı okuyan *a* nesnesinin değerinin mi adresinin mi işleve gönderildiğini merak etmez, *func* işlevinin problem düzlemindeki anlamıyla ilgilenir. Yani C dilinde okunabilirlik açısından bir eksiklik olarak görünen bu durum C++ dilinde bir çeşit *"veri saklama" (information hiding)* olarak görülür. C programcılarının C++"a geçişte karşılaştıkları temel zorluklardan biri budur. C dili ile programlamada, genel ilke ortada gizliliğin bulunmamasıdır. Ancak C++ dilinin *NYPT* için kullanılmasında durum böyle değildir. Çünkü ana amaç problemin kendi düzlemine yaklaşmaktır. Programı yazan, program hakkında düşünürken programlama dilinin araçları ile değil de, programın yazılma amacı olan problemin parçaları ile düşünür.

Parametrenin değiştirilmeyeceği durumlarda referans kullanılabileceğinden söz ettik. Madem böyle bir değiştirme söz konusu değil, o halde okunabilirliği artırmak amacıyla referans da *const* olarak tanımlanmalıdır.

Referanslar ile Göstericilerin Benzerlikleri ve Farklılıkları

Hem gösterici değişkenler hem de referanslar adres bilgileri tutan nesnelerdir. Ancak gösterici değişkenler kullanıldığında içlerindeki adres bilgisi ele alındığı halde, referans kullanıldığında artık referansın bağlandığı nesne ele alınır. Aşağıdaki kodları inceleyelim:

```
#include <iostream>

int main()

{
    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    int *p = &a[0];    //int *p = a;


    *p = 10;

    ++p;

    *p = 20;
```

```
    std::cout << a[0] << " " << a[1];


    return 0;
}
```

Şimdi de aşağıdaki kodu inceleyelim:

```
#include <iostream>
using namespace std;

int main()
{
    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int &r = a[0];


    r = 10;
    ++r;
    r = 20;
    cout << a[0] << " " << a[1];
    return 0;
}
```

Referanslar tek bir elemanın adresine ilişkindir. Bir referansa adres ilkdeğer verilirken yerleştirilir. Daha sonra bu adres bilgisi değiştirilemez. Örneğin:

```
int a = 10;
int x = 20;
int &r = a;
r = x;
```

işlemleri yapıldığında, *x* değeri referansa değil, referansın yerine geçtiği nesneye yani *a*"ya aktarılır. Referanslar arka planda kendisi *const* olan gösterici değişkenlere karşılık gelir. Oysa bir gösterici değişken eğer kendisi *const* değilse program içinde dinamik olarak farklı nesneleri gösterebilir.

Bir diziyle ilgili genel işlem yapan işlev tanımlanabiliyordu. Böyle işlevlere dizinin başlangıç adresi ve boyutu argüman olarak gönderiliyordu. Dizinin başlangıç adresini işleve göndermek için gösterici kullanıyordu. Peki, böyle bir işlevin parametresi bir referans olabilir mi? Hayır! referanslarla bu iş göstericilerle olduğu gibi yapılamaz. Ancak, örneğin *10* elemanlı *int* türden bir diziyi gösteren gösterici olduğu gibi *10* elemanlı *int* türden bir dizinin yerine geçecek bir referans da tanımlanabilir. Aşağıdaki kodu inceleyin:

```
#include <iostream>

using namespace std;


void display(int(&r)[10])

{

    int k;



    for (k = 0; k < 10; ++k)
        cout << r[k] << " ";

    cout << endl;

}


int main()

{

    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};


    display(a);

    return 0;

}
```

Referanslar daha çok, tek bir nesneyi adres yöntemiyle işleve geçirmek amacıyla kullanılabilir. Örneğin tek bir *int* değer ya da tek bir yapı değişkeni referans yoluyla işleve geçirilebilir. Ancak *int* türden bir dizi ya da bir yapı dizisi bu yöntemle işleve doğal bir biçimde geçirilemez.

Sözdizimsel açıdan referansların göstericilere göre kullanım alanları daha dardır. Gösterici dizileri olur ama referans dizileri olamaz. Göstericileri gösteren göstericiler olabilir ama referansların yerine geçen referanslar olamaz.

Ancak şüphesiz bir göstericinin yerine geçen bir referans olabilir. Aşağıdaki kodu inceleyin:

```
#include <iostream>

int main()
```

```
{

    int x = 10;
    int *ptr = &x;
    int *&r = ptr;



    *r = 20;

    std::cout << "x = " << x << std::endl;



    return 0;

}
```

İşlev göstericileri olduğu gibi işlev referansları da olabilir. İşlev referanslarını ileride ayrıntılı olarak inceleyeceğiz.

NULL Adresi ve Referanslar

Değeri *NULL* adresi olan bir gösterici değişken hiçbir yeri göstermeyen bir gösterici değişkendir. *NULL* adresinin C"de ne kadar yaygın bir biçimde kullanıldığını biliyorsunuz: Bir işlev bir adrese geri döndüğünde başarısızlık belirtmek amacıyla *NULL* adresine geri dönebilir. Örneğin standart bir C işlevi olan *strchr* işlevinde bir yazı içinde bir karakter arar. Aranan karakteri yazıda bulursa, bulduğu yerin adresi ile bulamazsa *NULL* adresi ile geri döner.

Dışarıdan adres alan bir işleve özel bir bilgi iletmek amacıyla *NULL* adresi geçilebilir. Örneğin standart *time* işlevi kendisine *NULL* adresi gönderilirse takvim zamanı değerini yani *time_t* türünden değeri bir adrese yazmaz yalnızca geri dönüş değeri olarak üretir. *NULL* adresi bir gösterici için bir bayrak değeri olarak kullanılabilir. Örneğin bir kontrol deyiminde bir göstericinin değerinin *NULL* adresi olup olmamasına göre farklı işler yapılabilir.

Oysa hiç bir nesnenin yerine geçmeyen bir referans tanımlanamaz. Bir referans tanımlandığı zaman bir nesnenin yerine geçmelidir.

C++"da referansların eklenmesiyle göstericilere olan gereksinim tamamen ortadan kalkmamıştır. Ancak özellikle "Nesne Yönelimli Programlama Tekniği"nin kolay ve doğal bir biçimde uygulanması referans kullanımını gerektirir.

Referanslar Üzerinde Adres İşlemleri

Bir referansın adres işleci ile adresi alınabilir. Bu durumda referansın yerine geçtiği nesnenin adres değeri elde edilir. Örneğin:

```
#include <iostream>
```

```
int main()

{

    int a = 10;
    int &r = a;
    int *p = &r;



    *p = 20;

    std::cout << "a = " << a << std::endl;



    return 0;

}
```

işlemlerini inceleyin. Burada:

```
p = &r;
```

deyiminde referans adres işlecinin terimi olmuştur. Bu durumda referansın yerine geçtiği nesnenin adresi değeri elde edilir. Örneğimizde bu adres aslında *a* değişkeninin adresidir. Elde edilen adresin referans türü ile aynı türden olduğuna dikkat edin. Nihayet,

```
*p = 20;
```

ile aslında *a* nesnesine *20* değeri atanır.

Yukarıdaki program parçasının gösterici değişkenlerle oluşturulmuş eşdeğer C karşılığı da şöyledir:

```
#include <iostream>

int main()

{

    int a = 10;
    int *p = &a;

    int *ptr = &*p;



    *ptr = 20;

    std::cout << "a = " << a << std::endl;
```

```
    return 0;
}
```

Referanslara Farklı Türden Bir Nesne ile İlkdeğer Verilmesi Durumu

Bir referansın aynı türden bir değişkenle ilkdeğer verilerek tanımlanması gerektiğini belirtmiştik. Referansıns farklı türden bir değişkenle ilkdeğer verilerek tanımlanması geçersizdir. Aşağıdaki örneği inceleyin:

```
void func
{
    double d = 10.5;
    int &r = d;    //Geçersiz
    //...
}
```

Ancak *const* bir referansa başka türden bir nesne ile ilk değer verilmesi geçerlidir:

```
void func()
{
    double d = 10.5;
    const int &r = d;  // Geçerli
    //...
}
```

Bu durumda önce *const* referansa bağlanan farklı türden nesnenin değeri, referansın türünden yaratılacak geçici bir nesneye atanır. Referans da bu geçici nesneye bağlanır. Yani derleyici aşağıdaki gibi bir kod üretir:

```
int main()
{
    double d = 10.5;

    int temp = (int)d;
    const int &r = temp;
```

```
    return 0;
}
```

Referanslara Değişmezlerle İlkdeğer Verilmesi

Bir referansa bir değişmez ile ilk değer verilmesi de geçersizdir. Ancak bir *const* referansa bir değişmez ile ilk değer verilebilir:

```
int &r = 10;           // Geçersiz!
const int &r = 10;     // Geçerli
```

Bu durumda derleyici önce geçici bir nesne yaratır. Geçici nesneye değişmezi atar, referansı geçici nesneye bağlayan bir kod üretir. Yukarıdaki atamada aslında geri planda şunlar yapılır:

```
int temp = 10;     // geçici nesne 10 değeri ile yaratılıyor.
const int &r = temp;   //r referansı geçici nesneye bağlanıyor.
```

Referansa Geri Dönen İşlevler

İşlevlerin geri dönüş değerlerinin derleyici tarafından önce geçici bir bölgeye alındığını, buradan çekilerek kullanıldığını anımsayın. Örneğin:

```
x = func();
```

gibi bir çağrıda, önce *func* isimli işlev çağrılır. İşlevin geri dönüş değeri geçici bir nesnede saklanır. Daha sonra bu geçici nesneden çekilerek kullanılır:

Bunu kaba kod olarak aşağıdaki biçimde gösterebiliriz:

```
temp = return ifadesi;
x = temp;
```

Aslında işlevin geri dönüş değerinin türü, bu geçici nesnenin türünü belirtir.

```
double func()
{
```

```
    // …
    return ifade;
}
```

Örneğin yukarıdaki *func* isimli işlevin geri dönüş değerinin yerleştirileceği geçici bölge *double* türdendir. Yani aslında *return* deyimi ile bu geçici nesneye ilkdeğer verilmesi söz konusudur.

İşlevin geri dönüş değerinin referans olması geçici bölgenin referans olması anlamına gelir. Bu durumda *return* ifadesi bir referansa ilk değerini verir, değil mi? Aşağıdaki örneği inceleyin:

```
#include <iostream>
int x = 10;

int &func()
{
    return x;
}

int main()
{
    func() = 20;
    std::cout << "x = " << x << std::endl;

    return 0;
}
```

Burada geçici bölge *int* türden bir referanstır. Yani aslında *return* deyimiyle yapılan:

```
int &temp = x;
```

gibi bir işlemdir. Bu durumda geçici nesne olan referans *x* nesnesine bağlanır. Böylece işlev çağrı ifadesi artık bir nesne belirtir duruma gelmiştir. Bu durumda

```
func() = 20;
```

ataması gerçekte global *x* değişkenine yapılmış olur.

Geri dönüş değeri bir referans olan işlevlerin, kendilerini çağıran kod parçasına, doğrudan bir nesnenin kendisini ilettiğini düşünebilirsiniz. Böyle işlevlere yapılan çağrı ifadeleri nesne belirtir. Yani sol taraf değeri olarak kullanılabilirler. Geri dönüş değeri referans olan işlevlerin çağrı ifadeleri return ifadesi ile belirtilen nesne anlamına gelir. Böyle işlevlerin *return* ifadelerinin de aynı türden bir nesne belirtmelidir.

Yukarıdaki programın gösterici değişkenlerle oluşturulmuş karşılığı aşağıdaki gibi olur:

```
#include <iostream>
int x = 10;

int *func()
{
    return &x;
}

int main()
{
    *func() = 20;
    std::cout << "x = " << x << std::endl;

    return 0;
}
```

Her iki program için üretilecek makina kodları tamamen eşdeğerdir. Geri dönüş değeri referans olan işlevlerin aslında adresle geri döndüğünü ama bu adres yoluyla içeriklerinin alınarak kullanıldığını görebiliyor musunuz?

Yukarıdaki programda

```
*func() = 20;
```

deyimini inceleyelim. Bu ifadede üç işleç vardır. Önce işlev çağrı işleci ele alınır, *func* işlevi çağrılır. Bu işlev bir adres değeri döndürür. Daha sonra * işleciyle bu adresteki nesneye erişilir. Erişilen nesne *x* nesnesinin kendisidir. Nihayet atama işleciyle *20* değeri işlevin adresini döndürdüğü nesneye, yani *x* değişkenine atanır.

Referansa Geri Dönen İşlev Yerel Nesne İle Geri Dönmemeli

Yukarıdaki örnekte *x* değişkeni yerel olabilir miydi? Referansa geri dönen bir işlevin yerel bir nesne ile geri dönmesi, adrese geri dönen bir işlevin yerel bir değişkenin adresi ile dönmesine eşdeğer bir yanlışlıktır. Yani bir gösterici hatasıdır. İşlevin geri dönüş değeri ile kendisine çağıran kod parçasına ilettiği yerel nesnenin ömrü, işlevin kodunun yürütülmesinin tamamlanmasıyla sona erer. Ancak bu durum derleme zamanında bir hata oluşumuna neden olmaz. Derleyicilerin çoğu bu durumu mantıksal bir uyarı iletisi ile bildirir.

Referanslar Neden Kullanılır

Referanslar temel olarak adres işlemlerinin daha yalın bir yazım biçimiyle ifade edilmesi amacıyla kullanılır. Örneğin bir işlevin parametre değişkeni gösterici yerine referans olsa, işlev içinde adres yardımıyla nesneye erişileceği zaman * ya da -> işlecini kullanmak gerekmez. Böylece ifadeler daha yalın bir biçimde yazılabilir. & ya da * işlecinin kullanılması programın yazımını programlama dilinin düzlemine yaklaştırırken, problemin kendi düzleminden uzaklaştırıcı bir etki yapar.

Adres ya da içerik işlecinin kullanılmaması üretilen makina kodunda bir küçülme yaratmaz. Çünkü derleyici referans yoluyla yapılan erişimleri tıpkı göstericilerde olduğu gibi yine adres işlemleriyle gerçekleştirir. Bunun dışında C++ dilinde nesne yönelimli programlama tekniğinin uygulanabilmesi için çeşitli biçimlerde referans türüne gereksinim duyulmuştur. Yani referanslar C++"da ileride göreceğimiz pek çok konunun uygulaması için gerekmektedir.

PARAMETRE DEĞİŞKENLERİNE VARSAYILAN DEĞERLERİN AKTARILMASI

C"de bir işlevin kaç tane parametre değişkeni varsa işleve o kadar argüman geçilmelidir. Oysa C++"da bir işlev, parametre değişkeni sayısından daha az sayıda argümanla çağrılabilir. Bir işlevin bir ya da birden fazla parametre değişkeni varsayılan argüman *(default argument)* alabilir. Bunun anlamı şudur: İşlev çağrısı ile işlevin parametre değişkenine bir değer aktarılmaz ise otomatik olarak daha önceden belirlenmiş bir değer aktarılır.

Bir parametre değişkeninin önceden belirlenmiş bir değer alacağı işlevin bildiriminde ya da işlevin tanımında, parametre değişkeninden sonra eşittir *(=)* işlecinden yazılan bir ifadeyle belirtilir. Aşağıdaki örneği inceleyin:

```
#include <iostream>

void foo(int x = 10, int y = 20);

void foo(int x, int y)

{

    std::cout <<"x = " << x << '\t' << "y = " << y << std::endl;

}


int main()

{

    foo();                  //  x = 10  y =20

    foo(100);               //  x = 100 y = 20

    foo(100, 200);          //  x = 100 y = 200


    return 0;

}
```

foo işlevinin bildiriminde varsayılan argümanlar kullanılıyor. *main* işlevi içinde yapılan ilk çağrıda *foo* işlevine hiç bir argüman gönderilmiyor. Bu durumda parametre değişkenleri olan *x* ve *y*"ye varsayılan değerler olan *10* ve *20* değerleri aktarılır. *main* işlevi içinde yapılan ikinci çağrıda, *foo* işlevine yalnızca *100* değeri gönderiliyor. Bu durumda birinci parametre değişkenine *100* değeri kopyalanırken, ikinci parametre değişkenine varsayılan değer olan *20* değeri kopyalanır. *foo* işlevine yapılan üçüncü çağrıda ise işleve *100* ve *200* değerleri gönderiliyor. Bu durumda, parametre değişkenlerinden hiçbiri varsayılan bir değer almaz. Görüldüğü gibi varsayılan değerler işlev çağrılırken işleve gönderilmeyen değerlerdir.

Daha soldaki parametre değişkenlerine işlev çağrısıyla argüman gönderilip, daha sağdaki diğer argümanlar yazılmadan işlev çağrılırsa varsayılan değerler kullanılabilir. Ancak bunun tersi geçerli değildir. Aşağıdaki çağrı biçimi her durumda geçersizdir:

```
foo(, 10);
```

Bir parametre değişkeni için varsayılan bir değer belirlenmişse, bu parametre değişkeninin daha sağında bulunan parametre değişkenlerinin hepsi varsayılan değerler almak zorundadır:

```
void func(int x = 10, int y);      //Geçersiz!
void foo(int x, int y = 20);       //Geçerli
```

Varsayılan değer almamış olan bütün parametre değişkenlerine çağrı ifadesi ile gereken argümanlar gönderilmek zorundadır.

Bir işlevin parametre değişkeni olan gösterici de varsayılan değer alabilir. Aşağıdaki örneği inceleyin:

```
#include <iostream>


void put_message(const char *p = "Success!")
{
    std::cout << p;
}


int main()
{
    put_message("Failed!");
    put_message();


    return 0;
}
```

Yukarıdaki programda *put_message* isimli işlevin gösterici olan parametre değişkeni *p* için varsayılan argüman olarak *"Success!"* dizgesi geçiliyor. *main* işlevi içinde yapılan ilk çağrıda, işleve *"Failed!"* dizgesi argüman olarak geçilirken, ikinci çağrıda işleve herhangi bir argüman gönderilmiyor. Yapılan ikinci çağrıyla ekrana *"Success!"* yazısı yazdırılır.

Varsayılan argüman olarak belirlenen ifade, bir değişmez ifadesi olmak zorunda değildir. Değişken içeren ifadeler de varsayılan argüman olarak kullanılabilir. Varsayılan argüman olarak kullanılan ifadelerde daha önce bildirimi yapılmış global değişkenler kullanılabileceği gibi işlev çağrıları da yer alabilir.

```
int func1();
int func2(int);

int func3(double  = 3.14);


int g = 10;

int func4(int a = func1(), int b = func2(g), int c = func3());
```

Yukarıda yapılan tüm işlev bildirimleri geçerlidir. *func4* işlevinin her üç parametresi de varsayılan argüman alıyor. Birinci parametre olan *a* değişkenine, işlev çağrı ifadesi ile bir değer atanmaz ise, *func1* işlevinin geri dönüş değeri atanır. İkinci parametre değişkeni olan *b*'ye bir değer geçilmez ise, *func*2 işlevinin geri dönüş değeri atanır. Bu arada çağrılan *func2* işlevine, global değişken olan *g* değişkeninin değeri geçilir. Son parametre değişkenine değer geçilmediği zaman ise, bu parametre değişkenine *func3* işlevinin geri dönüş değeri atanır ki, çağrılacak *func3* işlevi de kendisine argüman gönderilmediği için varsayılan değer olarak belirlenen *3.14* değerini alır.

Varsayılan argümanlara ilişkin ifadelerin değerlendirilmesi işlevin çağrıldığı noktada gerçekleşir. *Yani func4* işlevi eğer çağrılmaz ise *func1, func2* ve *func3* işlevleri de çağrılmaz.

Varsayılan argüman olarak belirlenen değer yalnızca bir kez yazılmalıdır. Aynı kaynak dosyada varsayılan argümanın ikinci bir bildirimde yeniden yazılması geçersizdir.

Aşağıdaki örneği inceleyin:

```
///////   file1.h

int func(int a, int b, int c = 0);


///////   file2.h
#include "file1.h"


int func(int a, int b, int c = 0);        //Geçersiz
```

Ancak ikinci kez yapılan bildirimde, bu kez daha önce varsayılan argüman belirlenmemiş bir parametre değişkeni için varsayılan değer bildirilebilir. Tabii yine bu durumda daha sağda kalan tüm parametre değişkenleri için mutlaka daha önce varsayılan değerlerin belirlenmesi gerekir. *file2.h* dosyası aşağıdaki gibi olsaydı, derleme zamanında bir hata oluşmazdı:

```
///////   file2.h   ////////////////////////////

#include "file1.h"

int func(int a, int b = 1, int c);
int func(int a = 2, int b, int c);
```

Varsayılan argüman olarak belirlenen değerler yalnızca bir kez yazılmalıdır. Bu değerler işlevin arayüzünün bir parçasıdır. Doğal olan varsayılan argümanların işlev bildiriminde belirtilmesidir. Varsayılan argüman değerin hem işlev bildiriminde, hem de işlev tanımında yer alması geçersizdir:

```
void func(int x = 10, int y = 20);


void func(int x = 10, int y = 20)     // Geçersiz.
{
}
```

Hatırlayacağınız gibi, işlev bildirimlerinde parametre değişkenlerinin isimleri yazılmak zorunda değildir. Varsayılan bir değer alacak parametre değişkeni için de isim yazılması zorunlu değildir. Aşağıdaki iki bildirim de geçerlidir:

```
void func(int a = 10, int b  = 20);
void func(int = 10, int = 20);
```

Ancak işlevin varsayılan değer alacak parametre değişkeni bir gösterici ise ve işlev bildiriminde parametre değişkeni olan göstericiye isim verilmiyorsa dikkatli olunmalıdır:

```
void func(char *= "Ahmet");    //  Geçersiz!
```

Derleyici burada ***=*** karakterlerini tek bir atom olarak ele alıp *işlemli atama işleci* olarak değerlendirir (*En uzun atom kuralı - maximum munch*). Bildirim geçersizdir. Bu durumda varsayılan argümana ilişkin bu iki karakter bitişik yazılmamalıdır:

```
void func(char * = "Ahmet");
```

Gösterici parametre değişkenleri varsayılan argüman alabildiği gibi, referans parametre değişkenleri de varsayılan değerler alabilir:

```
#include <iostream>

int g = 20;

void func(int &r = g);


int main()

{

    int y = 30;
    func();

    func(y);
    return 0;

}

void func(int &r)

{

    std::cout << r << std::endl;

}
```

Varsayılan Argümanlar Neden Kullanılır?

Bazı işlevlerin çok sayıda parametre değişkeni vardır. Özellikle böyle işlevler söz konusu olduğunda ve bu parametre değişkenlerinin belli bölümüne işleve yapılan çağrıda çoğunlukla aynı değerler gönderiliyorsa, varsayılan argümanların kullanılması büyük bir yazım kolaylığı sağlar. İşleve gönderilen argüman sayısının azaltılması hem programcının iş yükünü azaltır, hem de kodun okunabilirliğini arttırır.

Parametre değişkenleri için varsayılan argüman belirlenirken dikkat edilmelidir. Bir işlev çoğunlukla aynı argüman değerleriyle çağrılıyorsa varsayılan argüman alan parametre değişkeni kullanılabilir. Bir örnek verelim:

Standart olmasa da, C derleyicilerinin çoğunda *stdlib* kütüphanesi içinde *itoa* isimli bir işlev bulunur. İşlev verilen bir tamsayı değerini verilen bir sayı sisteminde bir yazıya dönüştürerek, adresini aldığı bir diziye yazar:

```
char *itoa(int, char *, int);
```

İşlevin birinci parametresi yazıya dönüştürülecek değerdir. İşlevin 2. parametresi yazının yerleştirileceği adrestir.

İşlevin 3. parametresi dönüşümün yapılacağı sayı sistemidir. İşlevin geri dönüş değeri yazının yerleştirildiği adrestir.

itoa işlevi çoğunlukla bir tamsayıyı onluk sayı sistemine göre oluşturulmuş bir yazıya dönüştürmek için kullanıldığından, işlevin üçüncü parametresine çoğunlukla *10* değeri geçilir. Şimdi aynı işi yapan ancak varsayılan argüman alan *itoa_d* isimli bir işlev yazalım:

```
#include <iostream>
#include <cstdlib>


char *itoa_d(int n, char *str, int base = 10)
{
    return itoa(n, str, base);
}


int main()
{
    char s[100];


    itoa_d(123, s);
    std::cout << s;


    return 0;
}
```

Yazılan *itoa_d* isimli işlevin *base* isimli parametre değişkeninin varsayılan argüman alarak

10 değeri aldığını görüyorsunuz. İşlev iki argümanla çağrılırsa üçüncü parametre olan *base* isimli parametre değişkenine *10* değeri geçilir. İşlevin kendi içinde *itoa* işlevini çağırdığını görüyorsunuz. *itoa* işlevi bu durumda varsayılan argüman alan *itoa_d* isimli işlevi tarafından sarmalanmıştır.

Varsayılan argümanlar *"Hiçbir değer almayacağına bari şu değeri alsın"* fikriyle kullanılmamalıdır. Böylesi kullanımlar kaynak kodu inceleyen kişiyi yanıltır.

Bazen parametre değişkenine verilen varsayılan değerin özel bir anlamı olmaz. Bu varsayılan değer yalnızca işlevin varsayılan argümanla çağrılıp çağrılmadığını saptamak amacıyla kullanılır. Gerçek varsayılan değerler işlevin içinde ve bir dizi işlemle elde edilebilir. Aşağıdaki örneği inceleyin:

```
#include <cstdio>


#define DEFAULT_CALL        (-1)


void write_file(const void *ptr, unsigned nbytes, FILE *fp, long offset =
DEFAULT_CALL)

{

    if (offset != DEFAULT_CALL)
        fseek(fp, offset, SEEK_SET);

    fwrite(ptr, 1, nbytes, fp);

}


int main()

{

    double d = 10.2;
    FILE *f;

    /*...*/

    write_file(&d, sizeof(double), f);

    /*...*/
    return 0;

}
```

write_file isimli işlevin tanımını inceleyelim: İşlev birinci parametresine geçilen adresten başlayarak *nbytes* kadar byte'ı bir dosyaya yazar. İşlevin *offset* isimli son parametresine bir *ofset* değeri geçilirse, işlev dosya konum göstericisini bu ofset değerine konumlandırır ve yazma işlemini bu konumdan başlatır. İşlevin son parametresine bir değer geçilmez ise, yazma işlemi dosya konum göstericisinin gösterdiği konumdan başlayarak yapılır.

İŞLEV YÜKLEMESİ

C++ dilinde, bir kaynak dosyada aynı isimli birden fazla işlev tanımlanabilir. Nesne yönelimli programlama tekniğinin uygulanmasını kolaylaştıran bu araç İngilizce'de *"function overloading" (işlev yüklemesi)* olarak isimlendirilir. Şu sorulara yanıt aramakla başlayalım:

Neden iki ya da daha fazla sayıda işlevin isimleri aynı olsun? İki ayrı işleve aynı isim vermenin nasıl bir faydası olabilir? Bir örnekle başlayalım:

Aynı Arayüz Farklı İşlemler

C'nin aşağıdaki standart işlevlerini

```
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 işlevlerin hepsi aslında aynı işlemi yapar. Bu işlevler, kendilerine argüman olarak gönderilen ifadenin mutlak değerini geri dönüş değeri olarak üretir. Bu işlevlerin dışarıdan aldıkları değerlerin türleri farklıdır, dolayısıyla ürettikleri geri dönüş değerlerinin türleri de farklıdır. C dilinde aynı isimli işlevler tanımlanamayacağı için, bu işlevlere ayrı isimler verilmiştir. Bu işlevlerin hepsinin ismi aynı olsaydı, örneğin hepsinin isimleri *abs* olsaydı, mutlak değer alma işlemi yapan programcının, birden fazla işlev ismini bilmesi ya da anımsaması gerekmezdi, değil mi?

Aslında dilin temel işleçlerini düşündüğünüz zaman benzeri bir aracın kullanıma hazır bir biçimde sunulduğunu görebilirsiniz. Toplama işlecini ele alalım. Toplama işleci ne iş yapar? İki değerin toplanmasını sağlar, değil mi?

```
10 + 20
```

Yukarıda, *10* ile *20* değerlerinin toplama işlecinin terimleri olduğunu görüyorsunuz. Toplama işleci bir işlemin yapılmasını sağlıyor, yapılan işlemin sonucunda *30* değeri üretiliyor. Şimdi de aşağıdaki işleme bakalım:

```
1,5 + 3.5
```

Yukarıdaki örnekte ise *1.5* ve *3.5* değerlerinin toplama işlecinin terimleri olduğunu görüyorsunuz. Toplama işleci yine bir işlemin yapılmasını sağlıyor, yapılan işlemin sonucunda *5.0* değeri üretiliyor. Oysa makina açısından 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 işleçler yani aynı simgeler kullanılıyor. İki işlem için farklı iki simge kullanılmış olsaydı, algılama bu kadar kolay olur muydu?

C dilinde aynı isimli işlevlerin bir arada bulunamamasının nedeni nedir? C derleyicileri, bir işlev çağrı ifadesi ile karşılaştığında, hedef koda *(object code)* yalnızca çağrılan işlevin ismini yazar. Örneğin *Borland* derleyicileri, çağrılan işlevin isminin başına alttire karakteri ekleyerek oluşturdukları ismi hedef koda yazar. Aşağıda bildirimi verilen işlevi örnek olarak ele alalım:

```
int foo(int, int);
```

Derleyici bu işlevin çağrılması durumunda hedef koda

```
_foo
```

gibi bir isim yazar. Oysa *foo* işlevinin bildirimi farklı olsa da, C derleyicisinin hedef koda yazacağı isim aynı olur. Aynı isimli fakat parametrik yapıları farklı, birden fazla *foo* işlevi olsaydı, derleyici bu işlevlerin hepsi için amaç kodda aynı ismi kullanırdı. Bu durumda da, bağlayıcı program aynı isimli işlevlerden hangisinin çağrılmış olduğunu anlayamazdı.

C++ dilinde aynı isimli işlevlerin tanımlanabilmesi nasıl mümkün oluyor? C++ dilinde farklı olan nedir?

C++ derleyicisi, bir işlev çağrı ifadesinin karşılığı olarak hedef koda, C derleyicisi gibi yalnızca işlevin ismini yazmaz. C++ derleyicisi çağrılan işlevin ismini, işlevin parametre değişkenlerinin türleri ile birleştirerek bir isim oluşturur, bu ismi amaç koda yazar. Amaç koda yazılmak üzere oluşturulan bu ismin oluşturulma biçimi, standartlarca kesin olarak belirtilmemiş, derleyiciyi yazanlara bırakılmıştır. Örneğin bir C++ derleyicisi yukarıda bildirimi verilen *foo* işlevini hedef koda aşağıdaki gibi yazabilir:

```
_foo@i@i
```

C++derleyicisinin, işlev isimlerinin sonuna @ karakterini izleyecek biçimde, parametre değişkeninin türünün baş harfini yazdığını varsayalım. Bu durumda

```
int foo(double, double);
```

gibi ikinci bir *foo* işlevi var olsaydı, derleyici bu işlevin çağrı ifadesi karşılığı hedef koda bu kez

_foo@d@d

yazardı.

Böylece sıra bağlayıcı programa geldiğinde, bağlayıcı program aynı isimli birden fazla

işlev olmasına karşın, hedef koda yazılan ismi görerek, aynı isimli işlevlerden hangisinin çağrılmış olduğunu anlar.

Aynı isimli işlevlerin var olabilmesi için gerekli koşul da ortaya çıkıyor:

Aynı isimli iki işlevin, aynı bilinirlik alanında var olabilmesi için, işlevlerin parametrik yapılarının farklı olması gerekir. Aynı isimli işlevlerin parametre değişkeni sayısı *ve/veya* parametre değişkenlerinin türleri bir farklılık göstermelidir. Eğer aynı isimli iki işlevin, hem parametre değişkeni sayısı aynı hem de parametre değişkenlerinin türleri aynı ise, bu durumda her iki işlev de hedef koda aynı şekilde yazılacağından, bağlayıcı program yazılan isimlerden, hangi işlevin çağrılmak istendiğini anlayamaz.

Yukarıdaki açıklamalardan, C++derleyicilerinin çağrılan işlevi hedef koda yazarken kullandıkları ismin oluşturulmasında, işlevin geri dönüş değeri türünün hiçbir katkısı olmadığını anlamalısınız. Bu durumda, imzaları birbirleriyle tamamen aynı, ancak geri dönüş değerleri farklı türlerden olan, aynı isimli iki işlev, aynı bilinirlik alanında bildirilemez. Geri dönüş değeri türü dışında tüm parametrik yapısı aynı olan iki işlevin var olduğunu düşünelim. Bu isimle bir işlev çağrısı yapıldığında, derleyici hangi işlevin çağrıldığını anlayabilir miydi?

```
int foo(int);

double foo(int);   //Geçersiz!


int main()

{

    foo();


    return 0;

}
```

Yukarıdaki her iki bildirim de geçerli olsaydı, hangi *foo* işlevinin çağrıldığı nasıl anlaşılırdı?

İşlevlerin parametre değişkenlerinin sayısı ile, her bir parametre değişkeninin türünü kapsayan bilgiye "işlevin imzası" denir. İşlevin geri dönüş değerinin türü bilgisi, işlevin imzasının bir parçası değildir.

Aynı bilinirlik alanında aynı isimli iki ya da daha fazla sayıda işleve ilişkin bildirim varsa, aşağıdaki üç durumdan biri söz konusu olur:

İ. Hem işlevlerin imzaları tamamen aynı hem de işlevlerin geri dönüş değerlerinin türleri aynı ise, derleyici bu durumu, aynı işlevin bildiriminin birden fazla kez yapıldığı biçiminde yorumlar. C'de olduğu gibi C++'ta da, bir işlevin bildirimi özdeş olması koşuluyla birden fazla yapılabilir:

```
//işlev bildirimi iki kez yapılıyor (function redeclaration)
int func(int);

int func(int);
```

ii. Aynı isimli işlevlerin parametik yapıları birbirinden farklı ise, yani işlevlerin imzaları farklı ise, derleyici bu durumu, aynı isimli fakat farklı işlevlerin bulunduğu *(function overloading)* biçiminde yorumlar. Bu durumda işlevlerin geri dönüş türlerinin bir önemi yoktur:

```
//function overloading (İşlev yüklenmesi)
int func(int);

int func(int, int);
```

Aynı isimli iki işlevin parametrik yapısı, yalnızca işlevin birinde varsayılan argüman kullanılması biçiminde farklılık gösteriyorsa, bu durum yine işlev bildiriminin yinelenmesi olarak ele alınır:

```
//işlev bildirimi yineleniyor(function redeclaration)

int max(int *ptr, int size);

int max(int *, int = 10);
```

iii. İşlevlerin parametrik yapıları tamamen aynı, fakat işlevlerin bildirilen geri dönüş değerlerinin türleri farklı ise, bu durumda derleyici, işlev bildiriminin özdeş olmayan biçimde yinelendiği sonucunu çıkartır. Bu durumu bir sözdizim hatası olarak belirler.

```
//işlev bildirimi çelişkili biçimde yineleniyor

int func(int)

double  func(int);     //Geçersiz!
```

typedef bildirimi ile yeni bir tür yaratılmış olmaz. *typedef* bildirimiyle var olan bir türe yeni bir isim verilebilir. Eğer aynı isimli iki işlevde, parametre değişkenlerinin türlerinin yazılmasında birinde *typedef* ismi kullanılırken, diğerinde türün gerçek ismi kullanılmış ise bu durum *"işlev yüklemesi"* olarak kabul edilmez. Aşağıdaki örneği inceleyin:

```
typedef unsigned char BYTE;

BYTE foo(BYTE);

int  foo(unsigned char);
```

Yukarıdaki iki bildirimin bir arada bulunması geçersizdir. Bildirilen her iki işlevin de imzası aynı, geri dönüş değerlerinin türleri farklıdır.

Peki C++'da, parametrik yapıları birbirlerinden farklı, aynı isimli işlevler tanımlanabildiğine göre, derleyici aynı isimli işlevlerden hangisinin çağrıldığınınasıl anlar?

Bir işlev çağrısının, hangi işleve ilişkin olduğunun saptanması işlemine *"*Yüklenmiş İşlevin Saptanması*"* *(function overload resolution)* denir.

"Yüklenmiş işlevin saptanması", derleyici tarafından üç aşamada yapılan bir işlemdir.

Birinci aşamada, derleyici söz konusu işlev çağrısı için ele alınacak aynı isimli işlevleri saptayarak bu işlevlerin parametrik yapısı hakkında bilgi edinir. Çağrılması söz konusu olan aynı isimli işlevlere "aday işlevler" *(candidate functions)* denir.

Aday işlevler, işlev çağrı ifadesinde kullanılan isimle aynı isme sahip olan, işlev çağrı ifadesinin bulunduğu yerde görülebilen *(visible)* işlevlerdir.

Birinci aşamada derleyici, aday işlevler hakkında gerekli bilgiyi de edinir. Aday işlevlerin parametre değişkeni sayılarını, parametre değişkenlerinin türlerini öğrenir.

İkinci aşamada, işlev çağrı ifadesinde yer alan argümanlar kullanılarak, hangi aday işlevlerin geçerli biçimde çağrılabileceği saptanır. Söz konusu işlev çağrısıyla, geçerli biçimde çağrılabilen işlevlere, uygun işlevler *(viable functions)* denir.

Bir işlevin uygun işlev olarak belirlenebilmesi için, aşağıdaki koşulları sağlaması gerekir:

i) İşlev çağrı ifadesindeki argüman sayısı ile, işlevin parametre değişkeni sayısının aynı olması zorunludur. İşlev çağrı ifadesindeki argüman sayısı, işlevin parametre değişkeni sayısından daha az ise, bu durumda işlevin argüman gönderilmeyen parametre değişkenleri varsayılan argüman almalıdır.

ii) Argüman olan ifadenin, parametre değişkeni olan nesneye uygun bir şekilde dönüştürülebilmesi gerekir:

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 işlevi içinde *foo* isimli işleve yapılan çağrının, işlevlerden hangisine ait olduğunun saptanması aşamalarını inceleyelim:

1. aşama

İşlevin çağrıldığı noktada, tüm işlev bildirimleri görülebilir (*visible*) olduğundan bu aşamada, dört işlev de aday olarak belirlenir, parametrik yapıları hakkında bilgi edinilir.

2. aşama

1 numaralı işlevin parametre değişkeni sayısı (*0*) ile, işlev çağrısındaki argüman sayısı (*1*) birbirine eşit olmadığından, bu işlev uygun *(viable function)* olarak ele alınmaz.

2 numaralı işlev uygundur. Parametre değişkeni ile argüman sayısı uyumludur, argümandan parametre değişkenine geçerli bir dönüşüm söz konusudur.

3 numaralı işlev uygundur. İşlevin iki parametre değişkeni vardır. Ancak ikinci parametre değişkeni varsayılan argüman aldığı için, işlev tek bir argüman ile çağrılabilir. Argüman olan ifade *int* türden, bu argümanın kopyalanacağı parametre değişkeni *double* türdendir. Ancak atama öncesinde *int* türü, geçerli olarak *double* türüne dönüştürülebilir.
4 numaralı işlev uygun değildir. İşlevin parametre değişkeni sayısı ile, işlev çağrısındaki argüman sayısı birbirine eşit olmasına karşın, *int* türden ifade, dilin kurallarına göre *char** türüne dönüştürülemez.

İkinci aşamada uygun bir işlev bulunmaz ise, işlev çağrısı geçersiz kabul edilir. Bu duruma İngilizcede *"no match"* durumu *(çağrılacak uygun bir işlevin bulunmaması)* denir.

Üçüncü aşama

Üçüncü yani son aşamada, uygun işlevler içinde en uygun olan işlev belirlenir. Uygun işlevler içinden seçilerek, çağrı ifadesi ile eşlenecek işleve *"en uygun işlev" (best viable function / best match function)* denir.

üçüncü aşamada, belirli sayıda uygun işlev içinden, bir işlevin en uygun işlev olarak seçilebilmesi için aşağıdaki koşulları yerine getirmesi gerekir:

İşlev çağrı ifadesindeki argümanlardan işlevin ilgili parametre değişkenine yapılan dönüşümün derecesi diğer uygun işlevlere göre daha kötü olmaması gerekir. En az bir argüman için yapılacak dönüşümün diğerlerine göre daha iyi olması gerekir.

Birden fazla uygun işlev içinden, en uygun işlevin seçilememesi durumunda "çift anlamlılık hatası" *(ambiguity)* denilen bir hata durumu oluşur. Bu durumda, derleyici için kurallara uygun olarak çağrılabilecek birden fazla işlev söz konusudur.

"Argümanlardan parametre değişkenlerine yapılacak dönüşümün derecesi" ne anlama gelir?

C++ standartları, argümanlardan parametre değişkenlerine yapılabilecek otomatik dönüşümleri 4 ayrı dereceye ayırmıştır.

1. Tam uyum *(exact match)*
2. Yükseltme *(promotion)*
3. Standart dönüşüm (*standard conversion*)
4. Programcının tanımladığı dönüşüm (*user defined conversion*)

Kurallara göre, "tam uyum" durumu "yükseltme"den, "yükseltme" durumu standart dönüşümden, standart dönüşüm de programcının tanımladığı dönüşümden daha iyi olarak kabul edilir.

Yukarıdaki derecelendirmeleri ayrıntılı biçimde inceleyelim:

1. Tam uyum durumu

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

i) Argüman olan nesne bir sol taraf değeri, yani bir nesne ise, parametre değişkenine kopyalanacak değerin, bu nesneden alınması. Bu duruma sol taraf değerinden sağtaraf değerine dönüşüm denir *(L-value to R-value transformation)*.

ii) Parametre değişkeninin bir gösterici olması, işlevin de aynı türden bir dizinin ismi ile çağrılması. Dizi isimlerinin bir işleme sokulduğunda, işlem öncesinde otomatik olarak dizinin ilk elemenının başlangıç adresine dönüştürüldüğünü *(array to pointer conversion)* biliyorsunuz.

iii) Parametre değişkeninin bir işlev göstericisi (*function pointer*) olması, işlevin de aynı türden bir işlevin ismi ile çağrılması. İşlev isimlerinin bir işleme sokulduğunda, işlem öncesinde otomatik olarak işlev bloğunun başlangıç adresine dönüştürüldüğünü biliyorsunuz.
iv) İşlev parametre değişkeninin, gösterdiği yer *const* olan bir gösterici olması, işlevin aynı türden ancak *const* olmayan bir adres ile çağrılması *(qualification conversion)*.

2. Yükseltme durumu

Yükseltme (*promotion),* aşağıdaki durumları kapsar

i. *char, unsigned char, short, unsigned short* , *bool* türlerinden *int* türüne yapılacak dönüşüm. Bu duruma "int türüne yükseltme" (*integral promotion*) denir. Argüman olan ifade, *int* türünden küçük türlerden ise, *int* türüne yükseltme kuralı gereği, işlevin parametre değişkenlerine yapılan bir atama söz konusu ise, bu durum yükseltme olarak ele alınır.

```
void func(int);

int main()

{

    func('A'); //Yükseltme (integral promotion)
    func(true); // Yükseltme (integral promotion)

    //...

}
```

ii) Argüman olan ifade *float* türden ise, işlevin bu argümana karşılık gelen parametre değişkeni *double* türden ise, bu durumda, *float* türünden *double* türüne yapılacak dönüşüm de yükseltme olarak değerlendirilir:

```
void func(double);

int main()

{

    float fx;

    //...

    func(fx);       //yükseltme

    //...

}
```

iii. Bir *numaralandırma* türünden, o *numaralandırma* türüne baz olan *(underlying type)*

türe yapılan dönüşüm de, yükseltme olarak değerlendirilir. Aşağıdaki örneği inceleyin:

```
enum POS {OFF, ON, HOLD, STAND_BY};

int func(int);

int main()

{

    POS position = OFF;

    func(position);        // Numaralandırma türünden int türüne yükseltme)
    func(STAND_BY);        // Numaralandırma türünden int türüne yükseltme)

    //...

}
```

3. Standart dönüşümler

Standart dönüşüm (*standard conversions*) başlığı altında toplanan, 5 grup dönüşüm söz konusudur:

i. Tamsayı türlerine ilişkin dönüşümler

Bir tamsayı türünden ya da bir *numaralandırma* türünden, başka bir tamsayı türüne yapılan dönüşümler.

ii. Gerçek sayı dönüşümleri

Bir gerçek sayı türünden başka bir gerçek sayı türüne yapılan dönüşümler

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

iv. Adres türlerine ilişkin dönüşümler

0 değerinin herhangi türden bir göstericiye atanması ya da *void* türden olmayan herhangi bir adres bilgisinin *void* türden bir göstericiye atanması durumu.

v. *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 sample(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)



    sample(7.5);    //standart dönüşüm (double türden float türüne)

    pf(0);          //standart dönüşüm (0 değerinin bir göstericiye atanması
    vfunc(&x)       //standart dönüşüm (int * türden void * türüne)

    //...
    return 0;

}
```

4. Programcının tanımladığı dönüşümler.

Bu durumu sınıflara giriş yaptıktan sonra ele alacağız. Ancak şimdilik şu kadarını söyleyebiliriz: Bu dönüşümler, bir sınıf türünden nesnenin başka bir sınıf türüne ya da doğal bir veri türüne dönüştürülmesine ilişkindir. Derleyicinin bu tür dönüşümleri gerçekleştirebilmesi için, programcının özel dönüşüm işlevleri tanımlamış olması gerekir. İsimleri *"Dönüştürme kurucu işlevleri"* ya da *"tür dönüştürme işlevleri"* olan bu işlevleri ilerde ayrıntılı bir şekilde inceleyeceğiz.

Hangi işlevin çağrılmış olduğunun saptanması konusunda örnekler 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 işlevi içinde yapılan işlev çağrılarını teker teker ele alalım:

```
foo(10);
```

Çağrısı için

1, 2, 3, 4, 5, 6, 7 numaralı işlevler adaydır.

1, 2, 3, 4 numaralı işlevler uygundur.

Tam uyum sağladığı için, 1 numaralı işlev en uygun olanıdır.

```
foo(3.4F)
```

Çağrısı için

1, 2, 3, 4, 5, 6, 7 numaralı işlevler adaydır.

1, 2, 3, 4 numaralı işlevler uygundur.

Yükseltme durumu olarak değerlendirildiğinden, 2 numaralı işlev en uygun olanıdır.

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

Çağrısı için

1, 2, 3, 4, 5, 6, 7 numaralı işlevler aday işlevlerdir. Uygun işlev
yoktur *(no match)*.

İşlev çağrısı geçersizdir.

```
foo(6U)
```

Çağrısı için

1, 2, 3, 4, 5, 6, 7 numaralı işlevler adaydır.

1, 2, 3, 4 numaralı işlevler uygundur.

1, 2, 3 ve 4 numaralı işlevler için standart dönüşüm uygulanabilir. Çift anlamlılık hatası (*ambiguity*) söz konusudur. İşlev çağrısı geçersizdir.

const Yüklemesi

Bir işlevin parametre değişkeni gösterdiği yer *const* olan bir gösterici iken, aynı isimli bir başka işlevin parametre değişkeni, gösterdiği yer *const* olmayan bir gösterici olabilir:

```
void foo(const int *);
void foo(int *);
```

Bu durum da işlev yüklemesi olarak ele alınır. Yani iki ayrı işlev söz konusudur. *foo*

işlevine *int* türden bir nesne ile çağrı yapıldığında, hangi işlev çağrılmış olur?

```
void func()

{

    int x = 20;

    foo(&x); //hangi işlev çağrılır?

}
```

Bu durum "*const yüklemesi" (const overloading)* olarak bilinir. Hangi işlevin çağrılacağı, işleve gönderilen nesnenin *const* olup olmamasına göre belirlenir. Eğer işlev çağrısı *const* bir nesne adresi ile yapılırsa, parametre değişkeni gösterdiği yer *const* gösterici olan işlev çağrılırken, işlev çağrısı *const* olmayan bir nesnenin adresi ile yapılırsa, çağrılan diğer işlev olur:

```
void foo(const int *);
void foo(int *);


void func()

{

    int x = 20;

    const int y = 30;

    foo(&x);        // void foo(int *);
    foo(&y);        // void foo(const int *);

}
```

Şüphesiz *const* yüklemesi, parametre değişkeninin referans olması durumunda da geçerlidir:

```
void foo(const int &);
void foo(int &);


void func()

{

    int x = 20;

    const int y = 30;

    foo(x);     // void foo(int  &);
    foo(y);     // void foo(const int &);

}
```

Aşağıda yer alan programda, çift anlamlılık hatasının olduğu bir başka tipik durum gösteriliyor:

```
#include <iostream>


using namespace std;

void foo(int &r)

{

    cout << "void foo(int &)" << endl;

}


void foo(int x)

{

    cout << "void foo(int)" << endl;

}


int main()

{

    int a = 20;

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

```
    return 0;
}
```

Yukarıdaki programda, ilk tanımlanan *foo* işlevinin parametre değişkeni *int* türden bir referans iken, ikinci *foo* işlevinin parametre değişkeni *int* türden bir nesnedir. Bunlar farklı işlevdir. *main* işlevi içinde, yorum satırı içine alınmış birinci işlev çağrısı çift anlamlılık hatasına neden olur. Yani bu durumda, değerle çağırma(*call by value*) ya da adresle çağırma (*call by reference*) birbirine göre öncelikli değildir. Ancak ikinci işlev çağrısında, argüman olan ifade bir değişmezdir. Bir referansa bir değişmez ifadesi ile ilk değer verilemeyeceğine göre, çağrılabilecek tek bir işlev vardır. Şunu da ekleyelim:

Birinci işlevin parametre değişkeni *const* referans olsaydı, ikinci işlev çağrısı da çift anlamlılık hatası durumuna düşerdi.

Hangi işlevin çağrılmış olduğunu saptamak, derleme zamanında yapılan bir işlemdir. Yani kaynak kod derlenip hedef dosya haline getirildiğinde, artık hangi işlevin çağrılmış olduğu bilinir. Çünkü çağrılan işlevin kimliği bir şekilde hedef koda yazılmış olur. Başka bir deyişle "işlev yüklemesi" aracının, programın çalışma zamanı açısından bakıldığında bir ek maliyeti 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 işlevler gereksiz yere tanımlanmamalıdır. İşlev yüklemesinin ana amacı, çalışacak kodları gerçekte birbirinden farklı olan işlemleri, aynı isim altında soyutlamaktır. Farklı işler gören işlevlerin, aynı ismi taşıması okunabilirliği bozar. Birden fazla işlevin aynı ismi taşıması, kullanıcı kodların işini kolaylaştırmaya yöneliktir.

SINIFLAR

Sınıflar, nesne yönelimli programlama tekniğinin temel yapı taşıdır. Nesne yönelimli programlama tekniğine "sınıfları kullanarak program yazma" tekniği diyebiliriz. Bu bölümde sınıflara bir giriş yapacak, sınıflara ilişkin temel kavramları açıklayacağız. Bundan sonraki bölümlerde ise ağırlıklı olarak sınıfların kullanılması üzerinde duracağız.

Sınıf Nedir

Sınıfları öncelikle sözdizim açısından ele alacağız. *Sınıf* C++ dilinin, programcının yeni bir tür yaratmasına olanak veren bir aracıdır. C'de, programcının yeni bir tür yaratması, yapı (*struct),* birlik (*union)* ve *enum* araçlarıyla mümkün oluyordu. C++ dilinde bu araçlara bir de sınıf (*class)* eklenmiştir.

Sınıf *(class)* nesne yönelimli programlama tekniğinin uygulanmasına olanak sağlayan, *C* dilinde olmayan yeni bir yazılımsal birimdir. Sınıflar, *C*"deki yapılara benzetilebilir. Ancak *C*"deki yapılar yalnızca eleman *(member)* içerirken, *C*++"da sınıflar yapılardan fazla olarak hem veri elemanı hem de üye işlevleri *(member function)* içerir. Sınıflar, yapılara göre ek bir çok özelliğe sahiptir. Bu özelliklerin çoğu *Nesne Yönelimli Programlama Tekniği*"ni destekleme amacıyla eklenmiştir.

Nasıl bir "yapı" türü programcı tarafından tanımlanmış bir tür *(user defined type)* ise, sınıflar da programcının tanımlamış olduğu türdür. Programcı, önce yeni bir türü derleyiciye tanıtır, daha sonra bu yeni türden nesne, gösterici, referans tanımlayabilir. Sınıfları kullanabilmek için ilk yapılması gereken işlem, bir sınıfın tanımını yapmaktır. Bir sınıfın tanımını yapmak, bu sınıf hakkında derleyiciye bilgi vermek anlamına gelir.

Derleyici aldığı bilginin sonucunda, bu sınıf türünden bir nesne tanımlanması durumunda, hem bellekte ne kadar yer ayıracağını bilir, hem de programcının yazmış olduğu koda ilişkin bazı kontrol işlemlerini yapma olanağına kavuşur.

Sınıflar yapılara göre temel olarak iki önemli farklılığa sahiptir:

i. Sınıf bildirimi içinde sınıfların elemanları, ismine *public*, *protected* ya da *private* denilen üç ayrı bölgede yer alabilir.

ii. Sınıflar yalnızca veri elemanları değil işlevler de içerir. Bu işlevlere sınıfın üye işlevleri

(member functions) denir.

Önce bu iki yeni özellik üstünde ayrıntılı bir biçimde duracağız:

Sınıf Tanımı

Bir sınıfın tanımı, yani derleyiciye tanıtılması özel bir sözdizim ile olur: Sınıf bildiriminin genel biçimi şöyledir:

```
class [ sınıf_ismi] {
    [private:]

        //…
    [protected:]

        //…
    [public:]

        //…

};
```

class bir anahtar sözcüktür. Türkçe "sınıf" anlamına gelir. Genel biçimdeki *sınıf_ ismi(class tag*), bildirimi yapılan sınıfın ismidir. Sınıf ismi, isimlendirme kurallarına uygun herhangi bir isim olabilir.

Bir sınıf tanımında yer alan blok, *private*, *protected* ve *public* isimli üç bölümden oluşur. *public, private* ve *protected* C++"ın anahtar sözcükleridir. Bu sözcüklere bundan sonra *erişim belirteci* diyeceğiz. Bir bölüm, erişim belirtecini izleyen iki nokta üst üste *(:)* ayracı ile başlatılır, diğer bir erişim belirtecinin kullanılmasına kadar sürer. Üç bölümün, hepsinin bir sınıf bildiriminde bulunması zorunlu değildir. Hiç bir erişim belirtecinin kullanılmaması durumunda sınıfın *private* bölümü anlaşılır. Yani sınıf bildirimi içinde varsayılan (*default*) bölüm *private* bölümdür.

Üye işlevlerin yalnızca bildirimleri sınıf bildirimi içine yazılır. Bu işlevlerin tanımlamaları, normal bir işlev gibi ancak farklı bir sözdizim kuralı ile sınıf bildiriminin dışında yapılır. Ancak üye işlevlerin tanımı sınıf bildiriminin içinde de yapılabilir. Bu durumu ileride, sınıf içi *inline* işlevler başlığıyla inceleyeceğiz.

C++"da, sınıfın tanımı içinde bildirilen işlevlerle sınıfın dışında bildirimleri yapılan işlevleri birbirlerinden ayırabilmek için yeni terimler kullanılır. Sınıfın içinde bildirilen bir işleve, o sınıfın üye işlevi *(member function)* denirken, diğer işlevlere global işlevler (*global functions*) denir. Yani C dilinde daha önce tanımlamaya alışmış olduğumuz işlevlere artık bundan böyle *global işlevler* diyeceğiz.

Sınıfın elemanları, yapılarda olduğu gibi sınıf içinde tür bilgileri ile bildirilir. Aşağıdaki örneği inceleyin:

```
class A {
private:

    int a;
```

```
    void func1();
protected:

    long b;

    int func2(int);
public:

    double c;
    double func3();

};
```

Yukarıdaki sınıf bildiriminde, sınıfın üç bölümünün de kullanılmış olduğunu görüyorsunuz. Sınıfın her bölümünde, birer eleman ile birer işlev bildiriliyor. Sınıf bildirimi içinde, birden fazla erişim belirteci kullanılabilir. Örneğin yukarıdaki bildirim aşağıdaki gibi de yapılabilirdi:

```
class A {
protected:

    long b;
private:

    int a;
public:

    double func3();
protected:

    int func2(int);
public:

    double c;
private:

    void func1();

};
```

Sınıf bildirimi bir erişim belirteci ile başlatılmamışsa, *private* bölüm üzerinde işlem yapıldığı anlaşılır. Aşağıdaki örneği inceleyin:

```
class A {

    long b;
    int a;

public:

    double func3();
protected:
```

```
    int func2(int);
public:

    double c;
private:

    void func1();

};
```

Yukarıdaki örnekte, *A* isimli sınıfın *a* ve *b* elemanları sınıfın *private* bölümünde bildiriliyor. Şimdi de aşağıdaki sınıf bildirimini inceleyin:

```
class Date {

    int day, mon, year;
    bool verify_date();

public:

    void set_date(int , int, int);
    void display_date();

};
```

Yukarıda, ismi *Date* olan bir sınıf tanımlanıyor. Yukarıdaki tanımla, *Date* isimli yeni bir veri türü yaratılmış olur. C ile C++'ın bu noktadaki farkını da hatırlayın: C++'da bu veri türünün ismi hem *class Dat*e hem de *Date'*dir. Yani bir *typedef* bildirimi yapılmaksızın *Date* ismi bu türün ismi olarak kullanılabilir. Oysa C dilinde yapılar söz konusu olduğunda, bir yapı ismini *(structure tag)* bir tür ismi olarak kullanmak için bir *typedef* bildirimi yapmak gerekir.

Bir sınıf ismi *(class tag),* isimlendirme kurallarına uygun olmak koşuluyla, istenildiği gibi seçilebilir. Ancak programcıların çoğu, yalnızca ilk harfi büyük diğer harfleri küçük olan isimleri seçerler.

Date isimli sınıfın tanımını incelemeyi sürdürelim:

int türden *day*, *mon*, *year* isimli elemanlar sınıfın *private* bölümünde bildirilmiş. Bir erişim belirteci kullanılmadığı zaman sınıfın *private* bölümünün anlaşıldığı söylenmişti. Başka bir deyişle, *int*, *mon* ve *year*, *Date* sınıfının *private* elemanlarıdır. *Date* sınıfının bildirimi içinde, *verify_date*, *set_date* ve *display_date* isimli üç işlevin bildiriminin yapıldığını görüyoruz. Bu işlevler, *Date* sınıfının *üye* işlevidir. *verify_date* isimli işlevin bildirimi sınıfın *private* bölümünde yapılmış iken, *set_dat*e ve *display_date* işlevlerinin bildirimleri *Date* sınıfının *public* bölümünde yapılmış. Şöyle de söylenebilirdi: *set_date* ve *display_date*, *Date* sınıfının *public* üye işlevleridir. *verify_date Date* sınıfının *private* üye işlevidir. Peki, sınıfın üye işlevleri ile bizim daha önceden bildiğimiz işlevler arasında bir fark var mı?

Sınıfların üye işlevleri ne işe yarıyor, nasıl tanımlanıyor? Bütün bu konuları ayrıntılı bir şekilde ele alacağız.

Mantıksal olarak bir sınıf ile ilişkilendirilmiş işlevlere "üye işlevler*" (member functions)* denir. Her üye işlev, sınıfın elemanlarına doğrudan erişebilir. Sınıfın elemanları, üye işlevler tarafından ortaklaşa kullanılan değişkenlerdir. Sınıfın üye işlevleri, bir konuya ilişkin çeşitli alt işlemleri yapar. Bu işlemleri yaparken de sınıfın elemanlarını ortaklaşa olarak kullanırlar. Bir işi gerçekleştiren bir dizi işlevin sınıf adı altında ele alınması, algılamayı ve tasarımı kolaylaştırır, derleyicinin bazı denetimleri yapmasına olanak verir.

Yapılarla olduğu gibi, sınıflarla da çalışmak için önce sınıf tanımını yapmak, sonra da bu sınıf türünden nesneler tanımlamak gerekir.

Sınıf tanımı kaynak kodun neresinde yapılmalıdır? Bir sınıfın tanımı kaynak kodun herhangi bir yerinde yapılabilir. Ancak tanımın bilinirlik alanının (*scope*), dosya bilinirlik alanında (*file scope*) ya da blok bilinirlik (*block scope*) alanında olması, farklı anlamlar içerir. Örneğin *Date* sınıfının tanımı, bütün blokların dışında yani global düzeyde yapılırsa, dosya bilinirlik alanındaki bu bildirim sonucunda *Date* sınıfını, dosyadaki tüm işlevler bilir. Yani sınıf bildiriminden sonra dosyanın sonuna kadar her noktada *Date* sınıfı kullanılabilir. Oysa bildirim yerel düzeyde yani bir blok içinde yapılırsa, ilgili sınıf yalnızca bildirimin yapıldığı blok içinde bilinir.

Yapılar gibi sınıflar da çoğunlukla başlık dosyaları içinde tanımlanır. Çoğu zaman bir sınıfın tanımı hizmet veren kodların arayüzünün bir parçasıdır. Ancak, C++ dilinde başlık dosyalarının kullanılması konusunu daha sonraki bölümlere bırakıyoruz.

Sınıf tanımı, bir sınıfın elemanları ile üye işlevlerinin derleyiciye tanıtılması işlemidir. Sınıf tanımı ile derleyici bellekte herhangi bir yer ayırmaz, yalnızca sınıf hakkında bilgi edinir. Sınıflar üzerinde bazı işlemlerin yapılabilmesi için, sınıf türünden nesnelerinin tanımlanması gerekir. Şimdi sınıf nesnelerinin tanımlanmasını ele alacağız:

Sınıf Türünden Değişkenlerin Tanımlanması

Daha önce bildirimi yapılmış bir sınıf türünden, bir değişken tanımlanabilir. Doğal türlerden değişkenler nasıl tanımlanırsa sınıf türünden değişkenler de aynı biçimde tanımlanır. Bildirimde önce tür bildiren sözcük ya da sözcükler, daha sonra tanımlanan değişkenin ismi gelir. Daha önceki örneklerde tanımlanan *A* ve *Date* isimli sınıfları düşünelim.

```
A a;
```

a değişkeni *A* sınıfı türünden bir sınıf değişkenidir.

```
Date date1, date2;
```

date1 ve *date2* isimli değişkenler *Date* sınıfı türündendir. *C++*"da yapı ve sınıf türünden değişkenler tanımlarken, *struct* ve *class* anahtar sözcüklerinin yazılmasına gerek olmadığını biliyorsunuz. Bu anahtar sözcüklerin kullanılması herhangi bir soruna yol açmaz. Yani yukarıdaki bildirim

```
class Date date1, date2;
```

biçiminde de yapılabilirdi. Ancak *class* anahtar sözcüğünün gereksiz bir şekilde kullanılması tercih edilen bir biçim değildir.

Üye İşlevlerin Tanımlanması

Bir sınıfın üye işlevi, belirlenmiş bir sözdizim ile sınıfın dışında tanımlanır:

```
[geri dönüş değerinin türü] <sınıf_ismi> :: <işlev_ismi> ([parametre
değişkenleri])
{
    // …
}
```

Sözdizimi inceleyelim:

Önce işlevin geri dönüş değerinin türünün yazıldığını görüyorsunuz. Sonra sınıfın ismi yazılıyor. Daha sonra gelen *::* atomunu, işlevin ismi izliyor. İki tane "iki nokta üst üste"nin yan yana getirilmesiyle oluşturulan *::* atomuna, bundan sonra "çözünürlük atomu" diyeceğiz. Bu atom bir işleç olarak kullanıldığında "çözünürlük işleci" *(scope resolution operator)* ismini alır.

Date sınıfının, parametre değişkeni olmayan, geri dönüş değeri üretmeyen *display_date*

isimli üye işlevi şöyle tanımlanabilir:

```
void Date::display_date()
{
    // …
}
```

Sınıf Nesneleri Yoluyla Sınıfın Elemanlarına ve Üye İşlevlerine EriŞim

Yapılarda olduğu gibi, bir sınıfa ilişkin nesne yoluyla sınıfın veri elemanlarına ve üye işlevlerine nokta işleci ile erişilebilir. Örneğin *object* bir sınıf türünden nesne, *m* ise bu sınıfın bir elemanı ve *func* da bu sınıfa ilişkin bir üye işlev olsun. Erişim aşağıdaki gibi gerçekleştirilir:

```
object.m
```

Bir üye işlev ancak bir sınıf nesnesi için ile çağrılabilir. Global işlevlerde olduğu gibi dışarıdan doğrudan çağrılamaz. Örneğin *object* isimli nesnenin ait olduğu sınıfın *func* isimli bir üye işlevi varsa, bu işlev

```
func();
```

biçiminde çağrılamaz. Eğer böyle bir çağrı yapılırsa derleyici global bir *func* işlevinin çağrıldığını anlar.
Üye işlev çağrısı aşağıdaki gibi yapılabilir:

```
object.func();
```

Yukarıdaki deyimde *object* bir sınıf türünden nesne ve *func* da *object*" in ait olduğu sınıfın üye işlevlerinden birinin ismidir.

Bir sınıfın üye işlevi, sınıf türünden bir gösterici ya da referans aracılığıyla da çağrılabilir. Bu durumları biraz daha ileride ele alacağız.

Üye İşlevlere Yapılan Çağrıların Amaç Kod İçine Yazılmaları

C++"da global bir işleve yapılan çağrının amaç kod içine nasıl yazıldığına daha önce değinmiştik. İşlev ismi parametre değişkenlerinin sayısı ve türleriyle birleştirilerek bir isim elde ediliyor amaç koda bu isim yazılıyordu. Bir sınıfın üye işlevi çağrıldığında derleyici bu kez üye işlev ismini, parametre değişkenlerinin sayısı ve türlerinin yanı sıra sınıfın ismiyle de birleştirerek amaç kod içine yazar. Bu durumda bir program içinde, aynı isimli ve aynı parametrik yapıya sahip global bir işlev ile bir sınıfa ilişkin üye işlev birlikte bulunabilir.

Bu iki işlevin amaç koda yazılmalarında bir sorun ortaya çıkmaz. Örneğin *Borland C++*

3.1 derleyicisinde

```
void func() ;
```

global işlevi

```
@func$qv
```

biçiminde, *X* sınıfına ilişkin

```
X::void func()
```

işlevi ise

```
@X@func$qv
```

biçiminde amaç kod içine yazılır. Derleyiciler, aynı isimli global işlevlerle üye işlevleri çağrı biçimlerine bakarak ayırabilir. Örneğin:

```
a.func();
```

gibi bir çağrı, sınıf nesnesi ile yapıldığına göre, derleyici *a* hangi sınıf türünden bir nesne ise *func* isimli işlevi de o sınıfın bir üye işlevi olacak biçimde arayacaktır.. Oysa çağrı

```
func();
```

biçiminde yapılırsa, derleyici global olan *func* işlevinin çağrıldığını anlardı.

C++"da işlevlerin amaç kod içine yazılmalarında herhangi bir standart söz konusu değildir. Her derleyici üye işlevlerin isimlerini sınıf ismiyle ve parametre türleriyle kombine ederek amaç kod içine yazar. Ancak bu yazma işlemine ilişkin kesin bir notasyon(yazım şekli) standart olarak belirlenmemiştir. Bu nedenle farklı *C++* derleyicileriyle derlenmiş modüllerin birleştirilmesinde sorunlar çıkabilir.

Peki, bir sınıf nesnesinin elemanları belleğe ne şekilde yerleştirilir?

Bir sınıf nesnesi tanımlandığında derleyici yalnızca sınıfın elemanları için bellekte yer ayırır. Sınıfın üye işlevleri sınıf nesnesi içinde herhangi bir yer kaplamaz. Üye işlevler yalnızca mantıksal bakımdan sınıf ile ilişkilendirilmiştir. Sınıfın iki bölüm belirten anahtar sözcüğü arasına yazılan veri elemanları, ilk yazılan düşük adreste olacak biçimde bitişik yerleştirilir. Bölümlerin birbirlerine göre yerleşim biçimleri standart olarak belirlenmemiştir, derleyiciden derleyiciye değişebilir. Aşağıdaki örneği inceleyin.

```
class A {
private:
```

```
    int a;
    int b;

public:

    int c;
    int d;
protected:

    int e;
int f;
private:

    int g;
    int h;

};
```

Burada *A* sınıfı türünden bir sınıf nesnesinin bellekte kapladığı alan *8 * sizeof(int)* kadar olur. Nesne içinde, *a* ile *b, c* ile *d, e* ile *f ve g* ile *h* veri elemanları bitişik olarak yerleştirilir. Ancak bu grupların kendi aralarındaki yerleşimleri derleyiciden derleyiciye değişebilir.

Bölümler arasındaki yerleşim standart bir biçimde belirlenmemiş olsa da derleyicilerin hemen hepsi daha yukarı yazılan bölümleri düşük adrese yerleştirir. Yani yukarıdaki örnekte derleyicilerin büyük bölümü veri elemanlarını sırasıyla düşük adresten başlayarak yukarıdan aşağıya doğru yerleştirir. Bölümler arası yerleşim standart olmadığına göre, nesnenin elemanlarının yerleşimine ilişkin yazılan kodlarda taşınabilirlik sorunları ortaya çıkabilir.

Nesnenin bellekte kapladığı alan, derleyicinin hizalama (*alignment*) işlemleri etkin hale getirilmişse veri elemanlarının toplam uzunluğundan fazla olabilir.

Sınıflarda Temel Erişim Kuralı

Global bir işlev içinde bir sınıf nesnesi, göstericisi ya da referansı ile sınıfın her bölümüne erişilemez. Hangi elemanlara ya da hangi üye işleve erişebileceği, sınıfın elemanlarının ya da üye işlevlerin yerleştirilmiş oldukları yer ile ilgilidir. Erişim kurallarını iki bölüme ayırarak inceleyeceğiz:

Global İşlevlerin Erişim Kuralı

Bir sınıf nesnesi, göstericisi ya da referansı ile, global bir işlevden ancak sınıfın *public* bölümündeki elemanlara ve üye işlevlerine erişilebilir. Global bir işlev içinde sınıfın *private* ya da *protected* bölümlerindeki elemanlara ve üye işlevlere sınıf nesnesi yoluyla erişilemez. Aşağıdaki örnekleri inceleyin:

```
#include <iostream>
#include <cstdlib>


class Date {
```

```
    int day, month, year;
    bool verify_date();

public:

    void set_date(int, int, int);
    void display_date();

};


// Üye işlev tanımlamaları...

using namespace std;


int main()

{

    Date date;

    date.day = 10;                // Geçersiz!


    if (!date.verify_date()) {    // Geçersiz!
        cout << "Geçersiz tarih…" << endl;
        exit(EXIT_FAILURE);

    }


    date.set_date(10, 10, 1997);  // Geçerli!
    date.display_date();          // Geçerli!


    return 0;

}
```

Bir sınıf nesnesi referansı ya da göstericisi yoluyla sınıfın yalnızca *public* bölümündeki veri elemanları ile işlevlerine doğrudan erişilebilir: Yukarıdaki örnekte *date.day* veri erişimi ile *date.verify_date()* üye işlevinin çağrısı bu nedenle geçerli değildir. Fakat sınıfın *public* bölümünde bulunan *set_date* ve *display_date* üye işlevlerine yapılan çağrılar geçerlidir.

Bu işlev çağrıları derleme zamanında herhangi bir soruna yol açmaz. Sınıflardaki erişim kuralına uyulmamasından dolayı oluşan hatalara ilişkin iletiler *Microsoft* derleyicilerinde,

```
„xxx" cannot access private member declared in Class „YYY"
```

Borland derleyicilerinde ise,

```
„YYY“::xxx not accessible
```

biçimindedir.

Üye İşlevlerin Erişim Kuralı

Bir sınıfın üye işlevi, hangi bölümde bildirilmiş olursa olsun sınıfın her bölümündeki elemanlarına doğrudan erişebilir. Sınıfın üye işlevlerini doğrudan çağırabilir. Örneğin, bildirimini yukarıda verdiğimiz *Date* sınıfının *set_date* isimli üye işlevi şöyle tanımlanmış olsun:

```
void Date::set_date(int d, int m, int y)
{
    date  =  d;  //  Geçerli.
    month  =  m;  //  Geçerli.
    year = y; // Geçerli.

    if (!verify_date(d, m, y)) {      // Geçerli.
        cout << "Geçersiz tarih…" << endl;
        exit(EXIT_FAILURE);
    }
}
```

Görüldüğü gibi *set_date* isimli üye işlevin tanımı içinde, *verify_date* isimli üye işlev doğrudan çağrılıyor. *verify_date* üye işlevi, söz konusu tarihin geçerli bir tarih olup olmadığını sınıyor. Eğer tarih geçersiz ise program sonlandırılıyor. *set_date* üye işlevi içinde sınıfın *day*, *month*, *year* isimli *private* elemanlarına doğrudan erişiliyor. Yine *set_date* işlevi içinde sınıfın *private* üye işlevi olan *verify_date* isimli işlevi çağrılıyor. *private* bölümde olan bu isimlere erişim geçerlidir. Çünkü bir üye işlev kendi sınıfının her bölümüne erişebilir.

Sınıfın üye işlevleri içinde tanımlanan aynı sınıfa ilişkin nesneler, referanslar ya da gösterici değişkenler ile sınıfın her bölümüne erişebilir. Örneğin *Date* sınıfının *process_date* isimli bir üye işlevi daha olsa,

```
void Date::process_date()
{
    Date date;
    date.day = day;                //Geçerli.
```

```
    date.month = month;           //Geçerli.

    date.year = year;             //Geçerli.
    if (!date.verify_date()) {    //Geçerli.
        cout << "Geçersiz tarih: << endl;

        exit(EXIT_FAILURE);

    }

}
```

process_date üye işlevi içinde tanımlanan *date* sınıf değişkeni ile sınıfın her bölümüne erişilebilir. Çünkü *date* değişkeni sınıfın üye işlevi içinde tanımlanıyor. Bir işlevin parametre ayracının içi de işlevin içi kabul edilir. Bu durumda bir üye işlevin parametre değişkeni aynı sınıf türünden bir nesne, referans ya da gösterici ise, parametre değişkeni ile nokta ya da ok işlecinin kullanılmasıyla sınıfın *private* elemanlarına ya da üye işlevlerine erişilebilir.

Üye İşlevlerin Sınıfın Elemanlarına Erişmesi

Bir üye işlev hangi sınıf nesnesi için çağrılmışsa, üye işlevin içinde kullanılan veri elemanları da o nesneye ilişkindir. Aşağıdaki örneği inceleyin:

```
class A {
public:

    void set(int, int);
    void display();

private:

    int a, b;

};



void A::set(int x, int y)

{

    a = x;
    b = y;

}



#include <iostream>

using namespace std;


void A::display()
```

```
{
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

}


int main()
{
    A a1;


    a1.set(10, 20);
    a1.display();


    A a2;
    a2.set(30, 40);
    a2.display();


    return 0;
}
```

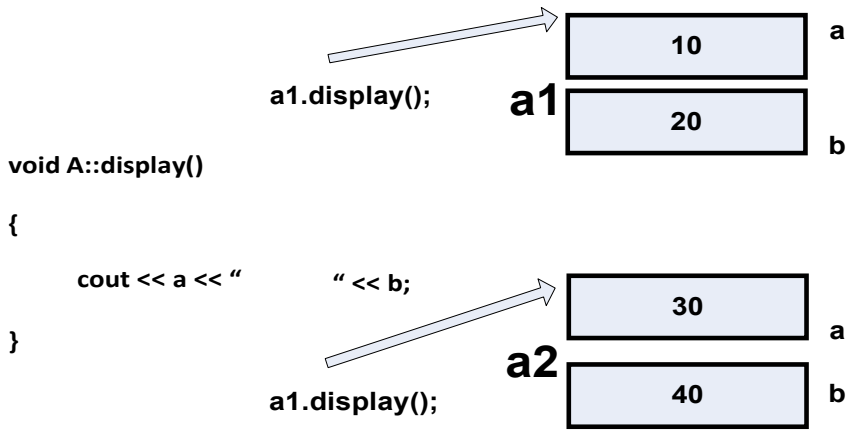
Burada *set* ve *display* üye işlevleri tanımları içinde doğrudan kullanılan *a* ve *b* elemanları, bu işlevler hangi nesne ile çağrılmışsa o nesnenin elemanlarıdır. *display* üye işlevine yapılan

```
a1.display()
```

çağrısında *a1* değişkeninin elemanları,

```
a2.display()
```

çağrısında ise *a2* değişkeninin elemanları kullanılır.



Bir üye işlevin başka bir üye işlevi çağırması durumunda, çağrılan üye işlev çağıran üye işlev ile aynı elemanları kullanır. Örneğin *A* sınıfının *set* üye işlevi *display* üye işlevini doğrudan çağırmış olsun:

```
void A::set(int x, int y)
{
    a = x;
    b = y;

    display();
}
```

Bu durumda *set* üye işlevi hangi sınıf nesnesi ile çağrılmışsa, *display* üye işlevi de onun elemanlarını kullanır.

```
int main()
{
    A a1;


    a1.set(10, 20);
    return 0;
}
```

Yukarıdaki *main* işlevinde çağrılan *set* ve *display* üye işlevleri *a1* sınıf nesnesinin elemanlarını kullanır. Yani *set* ve *display* işlevlerinin içindeki *a* ve *b* elemanları *a1* nesnesine ilişkindir.

Sınıf Elemanlarına Erişim ve this Göstericisi

Bir üye işlevin kendi sınıfının elemanlarına ve üye işlevlerine doğrudan erişebildiği söylendi. Peki, gerçekte bu nasıl oluyor? Yalnızca bir grup işlev tarafından erişilebilen bir nesne, *C* dilinde ne anlama gelir? Bir işlev kendi içinde tanımlanmayan bir yapı nesnesinin elemanlarına nasıl erişebilir? İşlevin bu yerel nesnenin adresini alması gerekir, değil mi?

C, C++'dan daha doğal bir dildir. Yani bilgisayarda olanları daha iyi betimler. Bir işlem pek çok programlama dilinde farklı bir biçimde yapılıyorsa, gerçekte işlem *C*'dekine(C dilindekine) benzer bir biçimde yapılır. Bir işlemin *C dilinde* neye karşılık geldiğini araştırmak onun gerçekte nasıl olduğunu yani makina dili düzeyinde neye karşılık geldiğini araştırmak anlamına gelir. Örneğin sınıf bilinirlik alanı diye bir kavram *C dilinde* olmadığına göre bu durum doğal değildir. Aslında *C*'deki(C dilindeki) doğal başka mekanizmalarla karşılanabilir.

Bir üye işlevin sınıfın elemanlarına erişmesi üye işleve gizlice geçirilen bir adres yoluyla yapılır. Bu adres, üye işlev hangi nesne ile çağrılmışsa o nesnenin adresidir. Örneğin bir üye işlevin hiç parametre değişkeni yoksa, aslında gizli bir parametre değişkeni vardır. O parametre değişkenine işlev çağrısıyla bir sınıf nesnesinin adresi aktarılır. Örneğin aşağıda tanımlanan *Myclass* sınıfının *display* isimli üye işlevini inceleyelim.

Görünüşe göre *display* işlevinin parametresi yoktur:

```
void Myclass::display()

{

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

}
```

Ancak aslında işleve gizlice bir argüman gönderilir. Bu argüman çağrı ifadesindeki sınıf nesnesinin adresidir. Aslında *display* işlevinin makina kodları incelenirse böyle bir argümanın geçirildiği görülebilir. Bu durumda, *display* işlevinin C karşılığı şöyle olur:

```
void A::display(A *const this)

{

    cout << "a = " << this->a << endl;
    cout << "b = " << this->b << endl;

}
```

Bir üye işlev içinde sınıfın bir elemanı kullanıldığında, derleyici o elemana işleve gizlice geçirilen bir adres ile erişir. Adres geçirme işleminin gizlice yapılması kolay yazımı sağlamak için düşünülmüştür. Örneğin bir üye işlevinin açıkça yazılmış dört parametresi olsa, gerçekte derleyici bu işleve beş argüman geçirir.

Üye işleve nesnenin adresi gizlice geçiriliyor olsa da, üye işlevler içinde bu adres *this*

anahtar sözcüğü ile kullanılabilir. Örneğin,

```
void Myclass::display()

{

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

}
```

tanımıyla,

```
void Myclass::display()

{

    cout << "a = " << this->a << endl;
    cout << "b = " << this->b << endl;

}
```

tanımı arasında hiç fark yoktur. Zaten örneğin, *this->a* denmese bile *a* elemanı kullanıldığında derleyici gizlice geçirilen sınıf nesnesi adresiyle *a* elemanına erişir. *this* bir anahtar sözcüktür ve hangi sınıfa ilişkin üye işlev içinde kullanılırsa o sınıf

türünden bir adres belirtir. Örneğin *this* anahtar sözcüğü *Myclass* sınıfına ilişkin bir üye işlev içinde kullanılırsa, *this Myclass* sınıfı türünden bir adres belirtir.

this aynı zamanda kendisi *const* bir göstericidir. İçindeki adres değiştirilemez. Üye işlev içinde

```
this = adr;
```

gibi bir atama geçersizdir.

Üye işlev hangi nesne için çağrılmışsa *this* göstericisine o nesnenin adresi geçirilir. Tabii eğer üye işlev bir adres yoluyla çağrılıyorsa *this* göstericisine üye işlevin çağrıldığı gösterici içindeki adres geçirilir. Örneğin, *func X* sınıfının bir üye işlevi, *a* bu sınıf türünden bir nesne ve *p* de bu sınıf türünden bir gösterici olsun,

```
a.func();
```

çağrısının *C* karşılığı,

```
Xfuncv(&a);
```

biçimindedir. Benzer biçimde,

```
ptr->func();
```

çağrısının *C* karşılığı ise,

```
Xfuncv(ptr);
```

biçimindedir. *C* karşılıklarını yazarken üye işlevlerin sınıf isimleriyle ve parametre türleriyle birleştirildiğini vurgulamak için başına ve sonuna ekler getirdik.

Aşağıdaki programı yazarak çalıştırın:

```
class X {
public:

    void display();
    void set(int);

private:

    int a;

};


#include <iostream>


using namespace std;
void X::set(int val)

{

    cout << "X::set()cagrildi" << endl;
    cout << "this = " << this << endl;
    a = val;

}
```

```
void X::display()

{

    cout << "X::display() cagrildi!" << endl;
    cout << "this = " << this << endl;

    cout << "a = " << a << endl;

}


int main()

{

    X myx;

    cout << "&myx = " << &myx << endl;
    myx.set(10);

    myx.display();
    X *ptr = &myx;
    ptr->display();



    return 0;

}
```

Elde edilen adres değerleri birbirinin aynısı, değil mi?

Bu durumda bir üye işlevin başka bir üye işlevi çağırmasının da ne anlama geldiği de açıktır. Üye işlev dışarıdan gizlice aldığı *this* adresini gizlice başka bir üye işleve geçer.

```
void A::set(A *const this, int x, int y)

{

    this->a = x;
    this->b = y;
    display(this);

}
```

Peki, bir sınıf göstericisi içinde rastgele bir değer varken o göstericiyle bir üye işlev çağrılırsa ne olur? Örneğin:

```
int main()

{
```

```
    X *p;

    p->func();
    return 0;

}
```

Burada *p,* yerel bir gösterici değişken olduğundan içinde bir çöp değer vardır. Bu durumda *func* işlevine *this* adresi olarak bir çöp değer geçilir. Yani *func* işlevi rastgele bir bellek bölgesini sınıf nesnesi gibi kullanır. Böyle bir durum bir gösterici hatasına yol açar.

Şimdi şöyle bir soru soralım: Madem bir sınıfın üye işlevi içinde sınıf nesnesinin elemanlarına doğrudan erişiliyor, o zaman neden *this* diye bir anahtar sözcük var? Üye işlev içinde sınıf nesnesinin adresinin elde edilebilmesinin ne gibi faydaları olabilir? *this*

adresi bir üye işlev içinde aşağıdaki amaçlar için kullanılabilir:

1. Üye işlev hangi sınıf nesnesi için çağrılmışsa, o sınıf nesnesinin adresi, üye işlev tanımı içinde bir başka işleve argüman olarak gönderilebilir:

```
class A {
public:

    //...

    void a_func();
private:

    //...

};


void g_func(A *);

void A::a_func()

{

    //...
    g_func(this);

}


int main()

{

    A a;
    a.a_func();
```

```
    return 0;
}
```

Yukarıdaki programda, *A* sınıfının *a_func* isimli üye işlevi içinde çağrılan *g_func* isimli global işleve, *this* adresi argüman olarak gönderiliyor. *main* işlevi içinde tanımlanan *A* sınıfı türünden *a* nesnesi ile *a_func* üye işlevi çağrıldığında, *a* nesnesinin adresi gizlice *a_func* işlevine geçirilir. Bu işlev içinden de, bu adres global *g_func* işlevine gönderilir.

2. Üye işlev içinde, üye işlevi çağıran sınıf nesnesinin kendisine erişilebilir. Madem ki *this* adresi üye işlevi çağıran sınıf nesnesinin adresidir, bu durumda

```
*this
```

ifadesi bu nesnenin kendisidir, değil mi? Aşağıdaki kodu inceleyin:

```
void g_func(A);

void A::a_func()
{
    A a;

    a = *this;

    //...
    g_func(*this);
}


int main()
{
    A x;
    x.a_func();


    return 0;
}
```

A sınıfının *a_func* işlevi içinde tanımlanan *A* sınıfı türünden *a* nesnesine **this* ifadesi atanıyor. *main* işlevi içinde *a_func* işlevi *x* nesnesi ile çağrıldığından, *a_func* işlevi içindeki *a* ya *x* nesnesinin değeri atanıyor. Bu kez global *g_func* isimli işlevin parametre değişkeninin *A* sınıfı türünden olduğunu görüyorsunuz. *a_func* üye işlevi içinde çağrılan global *g_func* işlevine böylelikle *main* işlevi içindeki yerel *x* nesnesinin değeri aktarılmış olur.

3. Çağrılan bir üye işlev, hangi sınıf nesnesi için çağrılmışsa, o sınıf nesnesini ya da o sınıf nesnesinin adresini geri döndürebilir. Bu durum ileride anlatılacak bazı araçlarda yoğun bir biçimde kullanılır. Aşağıdaki örneği dikkatle inceleyin:

```
class A {
public:

    //...

    A a_func1();

    A *a_func2();

    A &a_func3();

private:

    //...

};
```

A sınıfının üye işlevlerinden *a_func1*, *A* sınıf türünden bir değerle, *a_func2* işlevi *A* sınıfı türünden bir adresle, *a_func3* işlevi ise *A* sınıfı türünden bir referansla geri dönüyor.

Şimdi bu işlevlerin tanımlarına bir bakalım:

```
A A::a_func1()

{

    //...

    return *this;

}


A *A::m_func2()

{

    //...

    return this;

}
```

```
A &A::m_func3()
{
    //...
    return *this;
}
```

a_func1 üye işlevi hangi sınıf nesnesi için çağrılmışsa, o sınıf nesnesinin değerini geri döndürür.

a_func2 üye işlevi hangi sınıf nesnesi için çağrılmışsa, o sınıf nesnesinin adresini geri döndürür.

a_func3 üye işlevi hangi sınıf nesnesi için çağrılmışsa, o sınıf nesnesinin kendisini geri döndürür.

4. this göstericisi, sınıf elemanlarının yerel değişkenler tarafından maskelenmesi durumunda, sınıf elemanlarına erişmek amacıyla da kullanılır. Bu durum "sınıf bilinirlik alanı" başlığı altında incelenecek.

Sınıf Bilinirlik Alanı

C dilinde bilinirlik alanlarının dörde ayrıldığını anımsayın. Bunlar dardan genişe doğru "işlev bildirim bilinirlik alanı" (*function prototype scope), "*blok bilinirlik alanı" *(blok scope), "*işlev bilinirlik alanı "*(function scope)* ve "dosya bilinirlik alanı" *(file scope)* dır.

Blok bilinirlik alanı bir ismin yalnızca bir blok içinde, işlev bilinirlik alanı bir işlevin her yerinde, dosya bilinirlik alanı ise tüm işlevler içinde bilinmesi, kullanılabilmesi anlamına gelir. *C++*'da bu bilinirlik alanlarına ek olarak bir de sınıf bilinirlik alanı (*class scope)* tanımlanmıştır. Sınıf bilinirlik alanı, bir ismin hem sınıf bildirimi içinde, hem de sınıfın tüm üye işlevleri içinde bilinmesidir. Sınıfın elemanları ile üye işlevleri, sınıf bilinirlik alanı kuralına uyar. Sınıf bilinirlik alanının darlık – genişlik bakımından işlev bilinirlik alanı ile dosya bilinirlik alanı arasında bulunduğuna dikkat edin. Bu durumda *C++*'daki bilinirlik alanları dardan genişe doğru,

1. İşlev bildirimi bilinirlik alanı (*function prototype scope)*
2. Blok bilinirlik alanı (*block scope*)
3. İşlev bilinirlik alanı (*function scope*)
4. Sınıf bilinirlik alanı (*class scope*)
5. Dosya bilinirlik alanı (*file scope*)

biçimindedir. Şimdi de aynı isimli değişkenlerin durumuna bir göz atalım: Aynı isimli değişkenler konusunda şu kuralı anımsatalım: *C*'de(C dilinde) olduğu gibi *C++*'da aynı bilinirlik alanına ilişkin aynı isimli birden fazla değişken tanımlanamaz. Fakat farklı bilinirlik alanına ilişkin aynı isimli birden fazla değişken tanımlanabilir. Bir blok içinde aynı isimli birden fazla değişken etkinlik gösteriyorsa, o blok içinde dar bilinirlik alanına sahip olana erişilebilir.

Aşağıdaki örneği inceleyin:

```
#include <iostream>

void  func();  //işlev  bildirimi

int  a  =  50;  //global  değişken

class X {

public:

    void foo(int);
    void func();

private:

    int a;

};

using namespace std;

void X::func()

{

    cout << "X sınıfının func isimli üye işlevi…" << endl;

}


void func()

{

    cout << "Global func isimli işlev…" << endl;

}
```

Bu örnekte hem *a* isimli global bir değişken tanımlanıyor, hem de *X* sınıfının *a* isimli bir elemanı var. Sınıfın *foo* üye işlevi de şöyle tanımlanmış olsun:

```
void X::foo(int a)

{

    cout << a << endl;         // Parametre değişkeni olan a

    {

        int a = 30;

        cout << a << endl;     // Blok içindeki a

    }

    func();                // Üye işlev olan func

}
```

Şimdi dört tane *a* söz konusu. Global olan *a*, sınıfın elemanı olan *a*, parametre değişkeni olan *a* ve iç blokta tanımlanmış olan *a*. Dar bilinirlik alanına sahip olana erişme kuralına göre, iç blokta kullanılan *a* o blokta tanımlanan *a* değişkenidir. Dış bloktaki ise işlevin parametre değişkenidir. Çağrılan işlev üye işlev olan *func* isimli işlevdir. Çünkü işlev isimleri de değişken gibi ele alınır, aynı bilinirlik alanı kuralında uyar. Global işlevler dosya bilinirlik alanına, üye işlevler ise sınıf bilinirlik alanına sahiptir. Şimdi *foo* işlevinin parametre değişkeninin ismini değiştirelim:

```
void X::foo(int x)
{
    a = x;                              // Sınıfın elemanı olan a
    {
        int a = 30;                     // Blok içindeki a
        cout << a << endl;
    }
    func();                             // Üye işlev olan func
}
```

Şimdi dış bloktaki *a,* sınıfın elemanı olan *a* olarak ele alınır. Peki, bir üye işlev içinde sınıfın elemanları ya da üye işlevleriyle aynı isimli global değişkenlere ya da işlevlere erişmek mümkün olabilir mi? İşte çözünürlük işleci ile bu durum mümkün kılınmıştır.

Çözünürlük İşleci

Çözünürlük işleci iki „:" karakterinin *::* biçiminde yan yana getirilmesiyle elde edilir. Çözünürlük işlecinin tek terimli önek *(unary prefix)* ve iki terimli araek biçiminde *(binary infix)* iki kullanımı vardır. Önce tek terimli önek biçim üzerinde duracağız:

Çözünürlük İşlecinin Tek Terimli Önek Kullanımı

Bu kullanım biçiminde çözünürlük işlecinin tek terimi vardır. İşleç bu durumda her zaman global olan isme erişimi gerçekleştirir. Aşağıdaki örneği inceleyin:

```
#include <iostream>

using namespace std;

int a = 10;


int main()

{

    int a = 20;



    ::a = 50;                   // Global a

    cout << a  << endl;         // Yerel a

    {

        int a = 30;

        ::a = 100;              // Global a
        cout << a  << endl;     // Yerel a

    }

    cout << ::a  << endl;       // Global a



    return 0;

}
```

Çözünürlük işleci, aynı isimli hem yerel hem de global bir değişkenin tanımlı olduğu durumda global olana erişmek amacıyla kullanılabilir. Yukarıdaki örnekte blok içlerinde *a* değişkeninin :: işleci ile kullanılmasıyla global olan *a* değişkenine erişilir.

Burada bir de uyarı yapalım: :: işleci bir üst blokta tanımlı olana erişimi değil, her zaman global olana erişimi sağlar. Bir yukarıdaki bloğa erişmenin ciddi bir faydası yoktur. Oysa global değişkene erişim pek çok durumda gerekebilir.

Tek terimli önek kullanım sıklıkla bir sınıfın üye işlevleri içinde sınıfın elemanları ile aynı isimli global değişkenlerin bulunması durumunda, global olana erişimi sağlamak için kullanılır. Aşağıdaki örneği inceleyelim:

```
#include <iostream>


void func();           //Global işlev

class X {

public:
```

```
    void func();
    void foo();

};

using namespace std;

void func()

{

    cout <<"Global func isimli işlev…" << endl;

}


void X::func()

{

    cout <<  "X sınıfının func isimli işlevi…" << endl;

}


void X::foo()

{

    cout << "X sınıfının foo isimli işlevi…" << endl;

    func();            // X sınıfının üye işlevi olan çağrılıyor

    ::func();          // Global olan çağrılıyor

}
```

foo üye işlevinde,

```
func()
```

Biçiminde normal olarak çağrılan dar bilinirlik alanı kuralına göre *X* sınıfına ilişkin olan *func* işlevidir. Oysa

```
::func();
```

biçiminde çağrılmış olan global *func* işlevidir.

Bazı programcılar global olanla aynı isimli bir üye işlev olmasa bile, üye işlevler içinde global işlevleri okunabilirliği artırmak için *::* işleciyle çağırırlar. Örneğin,

```
CMyDialog:: CMyDialog()
{
    hProcess = ::GetProcessHeap();
}
```

GetProcessHeap işlevin *CMyDialog* sınıfının üye işlevi olmadığını düşünelim. Bu durumda işlevin :: işleci ile çağrılmasına gerek yoktur, değil mi? Çünkü bilinirlik alanlarının çakışması söz konusu olmadığı için nasıl olsa *GetProcessHeap* dendiğinde global olan anlaşılır. Ancak programcı kodu inceleyen kişiye yardımcı olmak için durumu vurgulamak istemiş olabilir. Sınıfların yoğun olarak kullanıldığı kütüphanelerde bu tür vurgulamalarla oldukça sık karşılaşabilirsiniz.

Çözünürlük İşlecinin İki Terimli Araek Kullanımı

Çözünürlük işleci iki terimli araek *(binary infix*) olarak da kullanılır. Böyle bir kullanımda sol terim bir sınıf ismi, sağ terim ise sınıfın bir elemanı ya da üye işlevi olmak zorundadır.

Bu kullanımla her zaman sınıfa ilişkin olan elemana ya da üye işleve erişilir. Örneğin *Date* isimli bir sınıfın *int* türden *day, month, year* isimli veri elemanları olsun. *set_date* bu sınıfın bir üye işlevi olmak üzere

```
class Date {
public:
    void set_date(int, int, int);
private:
    int day, month, year;
};

void Date::set_date(int day, int month, int year)
{
    Date::day = day;
    Date::month = month;
    Date::year = year;
}
```

Parametre değişkenlerinin isimleriyle sınıfın elemanlarının isimlerinin aynı olduğuna dikkat edin. Bu durumda üye işlev içinde sınıfın elemanlarına erişmek için iki terimli çözünürlük işleci kullanılıyor. Kullanım biçimini inceleyin:

```
Date::month = month;
```

Atama işlecinin sol tarafında kullanılan *month* ismi sınıfın elemanına ilişkin iken, atama işlecinin sağ tarafında bulunan *month* ismi parametre değişkenine ilişkindir.

Görüldüğü gibi çözünürlük işlecinin iki terimli ara ek biçimi, sınıfın elemanları ile aynı isimli yerel değişkenler ya da parametre değişkenlerinin olması durumunda sınıfın elemanlarına erişilmesi amacıyla kullanılır. Bunun dışında çözünürlük işlecinin sınıfların türetilmesi işlemlerinde de benzer amaçlarla kullanıldığını göreceksiniz.

Sınıfın Kurucu İşlevleri

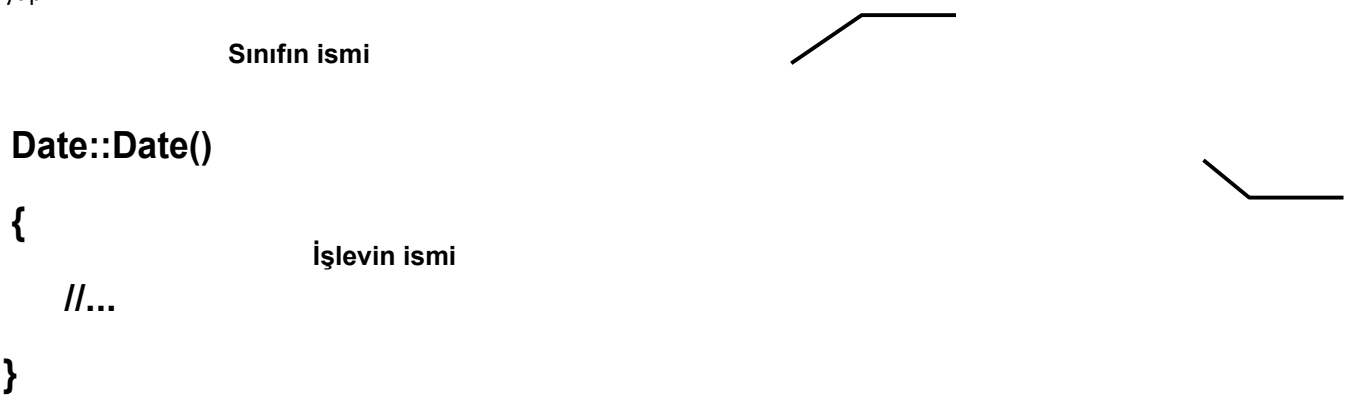
Bir sınıf nesnesi yaratıldığında ismine kurucu işlev *(constructor)* denilen bir üye işlev derleyici tarafından otomatik olarak çağrılır. Derleyici kurucu işlevleri isimlerine bakarak saptar. Kurucu işlevlerin ismi ait oldukları sınıfın ismidir. Kurucu işlevlerin geri dönüş değerleri yoktur. "Geri dönüş değerleri yoktur" demekle geri dönüş değerlerinin *void* olduğunu anlatmak istemiyoruz. Bu işlevlerin geri dönüş değerleri diye bir kavramları yoktur. Yazarken geri dönüş değeri yerine hiç birşey yazılmaz. Kurucu işlevlerinin parametrik yapısı herhangi bir biçimde olabilir.

Aşağıdaki sınıf bildirimini inceleyin:

```
class Date {
public:

    Date();

    void display();
private:

    int day, month, year;

};
```

```
Date();
```

biçiminde bildirilen işlev sınıfın kurucu işlevidir. Bu işlevin isminin sınıf ismiyle aynı olduğuna dikkat edin. Bildirimde geri dönüş değeri yerine hiçbir şey yazılmıyor. Bu durum yukarıda da belirttiğimiz gibi geri dönüş değerinin *int* ya da *void* olduğu anlamına gelmiyor. *Date* sınıfının kurucu işlevinin tanımı diğer üye işlevler gibi yapılır:



Aşağıdaki örnekte *Person* isimli sınıfın kurucu işlevi hangisidir?

```
class Person {
public:

    Person(const char *, int);     // Kurucu işlev
    void display();

    void set_name(const char *);
    void set_no(int );

    // …
private:

    char *name;
    int no;

};
```

Person sınıfının kurucu işlevi,

```
Person(const char *, int );
```

biçiminde bildirilen işlevdir. Bu işlevin tanımı aşağıdaki gibi olabilir:

```
Person::Person(const char *nm, int n)

{

    //…

}
```

Kurucu işlevlerin parametrik yapısı herhangi bir biçimde olabilir. *C++*'da imzaları farklı aynı isimli işlevler olabildiğine göre, bir sınıfın da farklı imzalara sahip birden fazla kurucu işlev de olabilir. Bir başka deyişle kurucu işlevler de yüklenebilir. Örneğin *Date* isimli bir sınıfın birden fazla kurucu işlevi olabilir:

```
class Date {
public:

    Date();

    Date(int, int, int);
    Date(const char *);
```

```
    // …
private:

    int day, int month, year;

};
```

Yukarıda tanımlanan *Date* sınıfı için üç kurucu işlev bildirimi yapılıyor: Parametre değişkeni olmayan, üç parametre değişkenli olan ve tek parametre değişkenli olan. Bir sınıfın parametre değişkeni olmayan, ya da tüm parametre değişkenleri varsayılan argüman alan kurucu işlevine "varsayılan kurucu işlev" *(default constructor)* denir.

Kurucu işlevler sınıf nesneleri yaratıldığında derleyici tarafından otomatik olarak çağrılır. Yani derleyici önce sınıf nesnesi için bellekte yer ayırır, daha sonra uygun olan kurucu işlevi çağırır. Yerel değişkenlerin programın akışının tanımlama noktasına geldiğinde global değişkenlerin ise programın belleğe yüklenmesiyle yaratıldığını anımsayın. Buna göre yerel bir sınıf nesnesine ilişkin kurucu işlev nesnenin tanımlandığı yerde, global bir sınıf nesnesine ilişkin kurucu işlev ise programın belleğe yüklenmesiyle, yani *main* işlevinden önce çağrılır. C++ dilinde *main* işlevinden önce çalışan bir kod da olabilir.

Bir sınıfın birden fazla kurucu işlevi olabildiğine göre, derleyici bunlardan hangisi çağıracağını nasıl saptar? Hangi kurucu işlevin çağrılacağı nesnenin tanımlanma ifadesiyle belirlenir. Eğer nesne isminden sonra ayraç açılmış, ayraç içine bir argüman listesi yazılmışsa, uygun imzaya sahip kurucu işlev çağrılır. Örneğin:

```
Date bdate(6, 1, 1966);
```

gibi bir tanımlamayla sınıfın

```
Date(int, int, int);
```

parametre yapısına sahip olan kurucu işlevi çağrılır. Burada, 6, 1 ve 1966 sınıfın kurucu işlevine yapılan çağrıda, işleve gönderilen argumanlardır. Benzer biçimde:

```
Complex c(10.2, 5.3);
```

gibi bir nesne tanımlamasında da parametre yapısı

```
Complex(double, double);
```

biçiminde olan kurucu işlev çağrılabilir. Nokta içeren ve sonuna ek almamış olan değişmezlerin *double* türden değişmez olarak ele alındığını anımsayın. Peki, aşağıdaki örnekte *Person* isimli sınıfın hangi kurucu işlevi çağrılır?

```
Person y("Necati Ergin");
```

Dizgelerin işleme sokulduğunda tür dönüştürme işlemiyle otomatik olarak *const char* * türüne dönüştürüldüğünü anımsayın. Bu durumda *y* isimli sınıf nesnesinin tanımlanmasıyla *Person* isimli sınıfın, *const char** türünden gösterici parametresine sahip olan kurucu işlevi çağrılır.

Eğer tanımlama işlemi diğer türden nesnelerde olduğu gibi ayraç açılmadan yapılmışsa, nesne yaratılırken varsayılan kurucu işlev yani parametresi olmayan kurucu işlev çağrılır. Örneğin:

```
Date date;

Person person;

Complex c;
```

Yukarıdaki *date*, *person* ve *c* nesneleri için varsayılan kurucu işlevler çağrılır.

Eğer bir sınıf nesnesi ilk değer verilerek tanımlanıyorsa, ilk değer olarak verilen türe uygun kurucu işlev çağrılır. *X* bir sınıf olmak üzere:

```
X a = b
```

tanımlamasıyla

```
X a(b)
```

tanımlaması tamamen eşdeğerdir. Bu durumda örneğin:

```
Person y = "Necati Ergin";
```

gibi bir tanımlamayla,

```
Person y("Necati Ergin");
```

tanımlaması aynı anlamdadır. Bu biçimde yalnızca tek parametreli kurucu işlevler çağrılabilir. Zaten sınıf nesnelerine birden fazla değerle ilkdeğer(ilk değer) verilemez. Sınıf nesnelerine yapılarda olduğu gibi küme ayraçları arasında ilkdeğer(ilk değer) verme de söz konusu değildir. Örneğin,

```
X a = {10, 20, 30};    // Geçersiz
```

a, *X* türünden bir sınıf nesnesi ise bu biçimde ilkdeğer(ilk değer) verilemez. Aslında istisna olarak özel bazı sınıflara küme ayraçları ile ilk değer verilebilir. Ancak bu konuyu ileride ele alacağız.

Aşağıdaki sınıf bildirimini inceleyin, izleyen örneği yazarak çalıştırın:

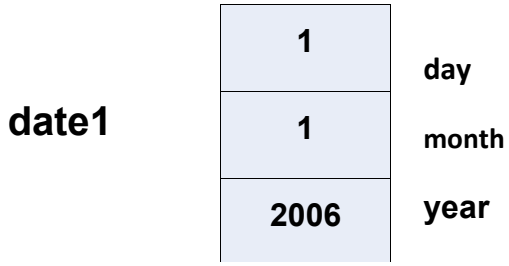
```
#include <iostream>
#include <ctime>


class Date {
public:

    Date();

    Date(int, int, int);
    void display();

private:

    int day, month, year;

};


using namespace std;

Date::Date()

{

    time_t timer = time(0);

    tm *tptr = localtime(&timer);
    day = tptr -> tm_mday;

    month = tptr -> tm_mon + 1;
    year = tptr -> tm_year + 1900;

}
```

```
Date::Date(int d, int m, int y)

{

    day = d;
    month = m;
    year = y;

}


void Date::display()

{

    cout << day <<'/' << month << '/' << year;

}


int main()

{

    Date date1;
    date1.display();



    Date date2(28, 10, 1998);
    cout << endl;
    date2.display();

    cout << endl;



    return 0;

}
```

Örneğimizde,

```
Date date1;
```

tanımlamasıyla sınıfın varsayılan kurucu işlevi çağrılıyor. Varsayılan kurucu işlev sistem tarihini alarak sınıfın elemanlarına yerleştiriyor. Örneğin, kurucu işlev çalıştırıldığında *date* nesnesinin elemanları şöyle doldurulmuş olsun.



Örneğimizde daha sonra,

```
date1.display();
```

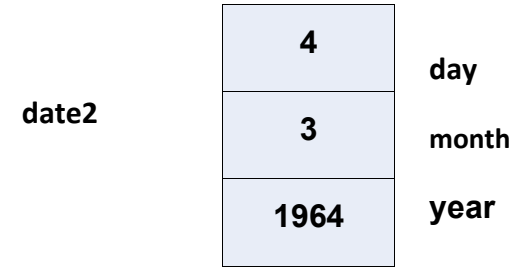
çağrısının yapıldığını görüyorsunuz. *display* işlevi ile ekrana yazdırılan *da*y, *month* ve *year, date1* nesnesinin elemanlarıdır değil mi? Yerel sınıf nesnelerine ilişkin kurucu işlevler, programın akışı nesnenin tanımlama noktasına geldiğinde çağrılacağına göre,

```
Date date2(4, 3, 1964);
```

gibi bir tanımlamayla

```
Date(int, int, int);
```

bildirimine uygun olan kurucu işlev çağrılır. Bu işlev argüman olarak gönderilen değerleri sınıfın elemanlarına yerleştirir. Bu durumda *date2* nesnesinin elemanları kurucu işlev çağrıldıktan sonra şöyle olur:



Örneğimizde daha sonra,

```
date2.display()
```

ile *date2* nesnesinin veri elemanlarının yazdırıldığını görüyorsunuz.

Global değişkenler programın belleğe yüklenmesiyle yaratıldıklarına göre, global sınıf nesnelerine ilişkin kurucu işlevler de *main* işlevinden önce çalıştırılır. Aşağıdaki örneği inceleyin:

```
class X {

    int a;
public:

    X(int);

    void display();

};


#include <iostream>

using namespace std;

X::X(int n)

{

    a = n;

    cout << "X sınıfının kurucu işlevi" << endl;

}


void X::display()

{

    cout << a << endl;

}


X g = 20;


int main()

{

    cout << "main işlevi başladı…" << endl;

    g.display();


    return 0;

}
```

Örneğimizde *g* sınıf nesnesine ilişkin kurucu işlev *main* işlevinden önce çağrılır. Ekranda şunları görmeniz gerekir:

```
X sınıfının kurucu işlevi
main işlevi başladı…

20
```

Eğer birden fazla global sınıf nesnesi tanımlanmışsa kurucu işlevlerin çağrılma sırası yukarıdan aşağıya doğrudur. Yani daha yukarıda tanımlanmış sınıf nesnesinin kurucu işlevi daha önce çağrılır.

Kurucu işlevler de varsayılan argümana sahip olabilir. Aşağıda örnekte verilen *Complex*

sınıfını inceleyin:

```
class Complex {
    double real, imag;

public:

    Complex(double, double = 0.);
    void display();

};


Complex::Complex(double r, double i)

{

    real = r;
    imag = i;

}

void Complex::display()

{

    cout << real;
    if (imag != 0)

        cout << "+" << imag;
    cout <<'\n';

}


int main()

{
```

```
    Complex c1(10);    // Complex x(10, 0)
    c1.display();


    Complex c2(10, 20);
    c2.display();


    return 0;
}
```

Bir kurucu işlevin tüm parametre değişkenleri varsayılan değer alıyorsa o kurucu işlev varsayılan kurucu işlev olarak da kullanılabilir. Örneğin, *Complex* sınıfının kurucu işlevi aşağıdaki gibi bildirilmiş olsun:

```
class Complex {
public:

    Complex(double = 0, double  = 0);

    // …

};
```

Aşağıdaki tanımlamaların hepsi geçerlidir:

```
Complex x;             //  Complex(0, 0) ile aynı anlamda
Complex y(10);         //  Complex(10, 0) ile aynı anlamda
Complex z(10, 20);
```

Bir sınıf için hiç bir kurucu işlev yazılmayabilir. Bu durumda derleyici varsayılan kurucu işlevi kendisi yazar. Yani bir sınıf için hiçbir kurucu işlev yazılmasa da, varsayılan kurucu işlev varmış gibi nesne tanımlanabilir. Örneğin:

```
class Point {
public:

    void set(int, int);
    void display();

private:

    int x, y;

};
```

Görüldüğü gibi, *Point* sınıfının hiç kurucu işlevi yok. Bu durumda varsayılan kurucu işlev varmış gibi nesne tanımlanması, herhangi bir hataya yol açmaz.

```
Point pt;       // Geçerli
```

Peki derleyicinin yazdığı varsayılan kurucu işlev tam olarak ne iş yapar? Bu konuyu ileride ele alacağız. Şimdilik şöyle düşünebiliriz: Bir kurucu işlevin yapması gereken bazı içsel işlemler söz konusudur. Bu işlemlerin her durumda yapılması gerekir. Derleyici bu içsel işlemler için bir kod yazar. Derleyicinin yazdığı bu kod, programcının yazdığı her kurucu işlevin en başına eklenir. Kurucu işlev programcı tarafından tanımlanmaz ise, derleyicinin yazdığı varsayılan kurucu işlev, yalnızca derleyicinin yazdığı kod parçasından oluşur.

Sınıfın en az bir kurucu işlevi varsa, ama varsayılan kurucu işlevi yoksa bu durumda varsayılan kurucu işlev çağrılacak biçimde nesne tanımlanması geçersizdir. Çünkü bu durumda artık derleyici otomatik olarak varsayılan kurucu işlevi yazmaz. Örneğin:

```
class Point {
public:
    Point(int, int);
    //...
private:
    int x, y;
};
```

gibi bir sınıf bildiriminden sonra,

```
Point pt;   //Geçersiz!
```

Biçiminde bir nesne tanımlanamaz. Fakat tabi,

```
Point pt(10, 20);
```

gibi bir nesne tanımlaması geçerlidir.

Kurucu İşlevin Sınıfın protected ya da private Bölümünde Bildirilmesi

Bir üye işlev nasıl sınıfın herhangi bir bölümünde bildirilebiliyorsa, sınıfın kurucu işlevi de sınıfın herhangi bir bölümünde bildirilebilir. Yani kurucu işlevin sınıfın *public* bölümünde bildirilmesi zorunda değildir.

Kurucu işlevin sınıfın *private* bölümünde bildirildiğini düşünelim. Peki, bu durumda ne olur? Global bir işlev içinde sınıfın *private* bölümüne erişilemeyeceğine göre, bir sınıf nesnesi yaratılması durumunda derleme zamanında hata oluşur, değil mi?

Aşağıdaki örneği inceleyin:

```
class A {
private:

    A();

    void foo();

};


int main()

{

    A a;    //Geçersiz!

}
```

Bir üye işlev içinden, *private* bölümde bildirilen bir başka üye işlev çağrılabilir, değil mi? Bu durumda, örneğin *A* sınıfının bir başka üye işlevi olan *foo* işlevi içinde *A* sınıfı türünden bir nesne tanımlanabilir:

```
void A::foo()

{

    A a;    //Geçerli

}
```

İleride, kurucu işlevin *private* ya da *protected* bölümde bildirilmesinden fayda sağlanabilecek bazı kod kalıpları da inceleyeceğiz.

Kurucu işlevleri diğer üye işlevlerden ayıran bir başka özellik de, bu işlevlerin programcı tarafından doğrudan çağrılamamasıdır:

```
class X {
public:
```

```
    X();
};


Void foo()
{
    X object;
    X.X();      // Geçersiz!
}
```

Sınıfın Sonlandırıcı İşlevi

Bir sınıf nesnesi bellekten boşaltılacağı zaman derleyici tarafından çağrılan üye işleve sınıfın sonlandırıcı işlevi (*destructor*) denir. Nasıl kurucu işlev nesne yaratıldığında çağrılıyorsa, sonlandırıcı işlev de nesnenin bellekten boşaltılacağı zaman, yani nesnenin ömrü sona erdiğinde çağrılır.

Sonlandırıcı işlevin ismi sınıf ismiyle aynıdır. Ancak ismin önüne ~ atomu eklenmiştir. Yani sonlandırıcı işlevin ismi ~*Sınıfİsmi* biçimindedir. Aşağıdaki sınıf bildirimini inceleyin:

```
class Person {
public:
    Person(const char *);      // Kurucu işlev
    //Diğer üye işlevler
    ~Person();                 // Sonlandırıcı işlev
private:
    char *name;
    int no;
};
```

Person sınıfının sonlandırıcı işlevi,

```
~Person();
```

bildirimi ile belirtilmiş olandır. Bu işlevin tanımı da kurala uygun olarak,

```
Person::~Person()

{

    //...

}
```

biçiminde yapılır. Sonlandırıcı işlevlerin de geri dönüş değerleri yoktur. Geri dönüş değerinin türü yerine hiç birşey yazılmaz. Bu durum onların *int* ya da *void* geri dönüş değerine sahip olduğu anlamına gelmez.

Sınıfın sonlandırıcı işlevi tektir. Parametresi *void* olmak zorundadır. Yani parametresi olmamak zorundadır. *C++*"da parametre ayracı içine *void* yazmakla hiçbir şey yazmamanın aynı anlama geldiğini anımsayın. Bu durumda sınıfın birden fazla sonlandırıcı işlevi de olamaz.

Sınıfın sonlandırıcı işlevi, sınıf nesnesi bellekten boşaltılacağı zaman çağrılır. Yerel değişkenlerin program akışının tanımlanma bloğunu bitirmesiyle, global değişkenlerin ise programın sonlanmasıyla bellekten boşaltıldığını anımsayın. Bu durumda yerel sınıf nesnelerine ilişkin sonlandırıcı işlevler değişkenin tanımlandığı bloğunun sonunda, global sınıf nesnelerine ilişkin sonlandırıcı işlevler de *main* işlevi sonlandığında çağrılır.

Aşağıdaki örneği inceleyin:

```
#include <iostream>
#include <cstdio>
#include <cstdlib>


class File {
public:

    File(const char *);
    void type();

     ~File();
private:

    FILE *fp;

};


using namespace std;

File::File(const char *fname)

{
```

```
    if ((fp = fopen(fname, "r")) == NULL) {
        cout << "Cannot open file..." << endl;
        exit(EXIT_FAILURE);

    }

}


void File::type()

{

    int ch;



    fseek(fp, 0L, SEEK_SET);

    while ((ch = fgetc(fp)) != EOF)
        putchar(ch);

}


File::~File ()

{

    fclose(fp);

}


int main()

{

    File file = "c:\\autoexec.bat";

    {

        File file = "c:\\config.sys";
        file.type();

    }

    file.type();



    return 0;

}
```

Verilen örnekte *File* sınıfının kurucu işlevi ismini aldığı dosyayı açıyor. Açılan dosyaya ilişkin *FILE* türünden adres sınıfın *fp* isimli elemanında saklanıyor. Sonlandırıcı işlev de açılmış olan dosyayı kapatıyor. Dosyanın açılması ve kapatılması işlemlerinin kurucu ve sonlandırıcı işlevler tarafından otomatik olarak yapıldığını

görüyorsunuz. Örneğimizdeki kurucu ve sonlandırıcı işlevlerin çağrılma yerlerine dikkat edin. Yerel sınıf nesneleri için sonlandırıcı işlevler tanımlandıkları blokların sonlarında çağrılır.

C++"da kurucu ve sonlandırıcı işlevlere ilişkin her zaman geçerli olan şöyle bir kural vardır: Sonlandırıcı işlevler kurucu işlevler ile ters sırada çağrılır. Yani kurucu işlevi daha önce çalıştırılan sınıf nesnesinin sonlandırıcı işlevi deha sonra çalıştırılır. Örneğin *a* nesnesinin kurucu işlevi *b* nesnesinin kurucu işlevinden daha önce çağrılmışsa *a* nesnesinin sonlandırıcı işlevi de b nesnesinin sonlandırıcı işlevinden daha sonra çağrılır. Yukarıdaki örnekte gördüğünüz gibi, *x* nesnesinin kurucu işlevi akış dikkate alındığında *y* nesnesinin kurucu işlevinden daha önce çağrılmıştır.

Bu özellik aynı bilinirlik alanı içindeki sınıf nesneleri için daha önemlidir. Örneğin,

```
{
    Date date1, date2;
    //…
}
```

Burada kurucu işlevler önce *date1* sonra *date2* sırasıyla, sonlandırıcı işlevler ise ters sırada yani önce *date2* sonra *date1* sırasıyla çağrılır.

Global sınıf nesnelerine ilişkin sonlandırıcı işlevler *main* işlevinden sonra çağrılır. Yukarıda verilen kurala göre, birden fazla global sınıf nesnesi varsa kurucu işlevlerin çağrılma sırasına ters sırada sonlandırıcı işlevler çağrılır. Yani kaynak kod içinde en aşağıda tanımlanan nesnenin sonlandırıcı işlevi en önce çağrılır.

Aşağıdaki programda kurucu ve sonlandırıcı işlevler hangi sırada çağrılır?

```
class X {
    // …
};

X a;
X b;

int main()
{
    X c;
    X d;
    // …
    {
```

```
        X e;
        // ..
    }
}
```

Burada programın akışı dikkate alındığında kurucu işlevlerin çağrılma sırası a*, b, c, d, e*

biçiminde, sonlandırıcı işlevlerin çağrılma sırası ise *e, d, c, b, a* biçiminde olur.

Bir işlev *return* deyimi ile sonlandırılırsa, işlev sonlandırılmadan önce, o noktaya kadar tanımlanmış bütün yerel sınıf nesneleri için sonlandırıcı işlevler çağrılır. *X* bir sınıf olmak üzere örneğin:

```
int foo()
{
    X a;
    // …
    {
        X b;
        // …
        if (func())
            return -1;
        // …
    }
    X c;
    // …
    return 0;
}
```

Yukarıdaki *foo* isimli işlevde, iç blokta *func* işlevi çağrılarak geri dönüş değeri sınanıyor. Bu işlevin geri dönüş değeri sıfır dışı bir değerse, işlev sonlandırılıyor. Derleyici işlev sonlanmadan önce, yaratılmış olan *a* ve *b* nesneleri için sonlandırıcı işlevleri de çağırır. *c* nesnesinin programın akışının *return* işleminin yapıldığı noktaya gelene kadar tanımlanmadığını görüyorsunuz. Tabii bu durumda *c* nesnesi için sonlandırıcı işlev çağrılmaz.

Bir sınıf için sonlandırıcı işlev yazmak zorunlu değildir. Sonlandırıcı işlev yazılmamışsa, derleyici sınıfın sonlandırıcı işlevini kendisi yazar. Derleyicinin yazacağı sonlandırıcı işlevin şimdilik söyle olduğunu varsayabilirsiniz:

```
Myclass::~Myclass()
```

```
{

}
```

Zaten her sınıfın sonlandırıcı işleve sahip olması gerekmez. Fakat kurucu işlev ile yapılanların otomatik olarak geri alınması isteniyorsa sonlandırıcı işlevin yazılması anlamlıdır. Genel olarak dışsal bir kaynak kullanan sınıflar, kurucu işlevleriyle kendilerine bağlanan bir kaynağı, sonlandırıcı işlevler ile serbest bırakırlar ya da geri verirler.

Son olarak herhangi bir işlev içinde tüm programın *exit* işlevi ile sonlandırılması üzerinde duralım. *exit* işlevi nerede çağrılmış olursa olsun, program içinde tanımlanmış tüm global sınıf nesneleri için eğer varsa sonlandırıcı işlevler çağrılır. Tabii *exit* işlevi ile programın sonlandırılması durumunda yerel sınıf nesneleri için sonlandırıcı işlevler çağrılmaz.

Kurucu ve Sonlandırıcı İşlevler Ne Amaçla Kullanılır

Kurucu işlevler sınıfın elemanlarına belirli ilkdeğerleri vermek ve çeşitli ilk işlemleri yapmak amacıyla tanımlanır. Bir sınıf nesnesi tanımlandığında çeşitli ilk işlemlerin kurucu işlevler tarafından gizli bir biçimde yapılması algılamayı kolaylaştırır. Böylece sınıf nesnesini kullanan kodlar (*client)*, sınıf nesnesini kullanmadan önce herhangi bir işlem yapmak zorunda kalmazl. Gerekli işlemler sınıfın kendi kodu tarafından yapılır. Bu şekilde bir sınıf nesnesinin kararlı bir şekilde ilk değerini alması sağlanır. İlkdeğer verme işlemi otomatik olarak çağrılan bir işlev tarafından değil de, sınıfı kullanan kod parçası tarafından açıkça çağrılması gereken bir işlev tarafından yapılsaydı, hata oluşma riski çok daha fazla olurdu, değil mi? Kurucu işlevler, veri gizleme (data hiding) prensibinin gerçekleştirilmesini sağlayan temel araçlardan biridir.

Aynı şekilde sonlandırıcı işlev ise kurucu işlev ile yapılan ilk işlemlerin geri alınması için kullanılır. Örneğin, kurucu işlev bir dosyayı açmışsa, sonlandırıcı işlevi dosyayı kapatabilir. Ya da kurucu işlev seri portu çeşitli değerlerle ayarlamış olabilir. Bu durumda sonlandırıcı işlevi de bu ayarları eski haline getirebilir. Ancak en sık karşılaşılan durum, kurucu işlevinin(işlevin) *new* işleci ile dinamik olarak bellekte bir yer ayırması ve sonlandırıcı işlevin de ayrılan yeri *delete* işleci ile bu alanı geri vermesidir. Aşağıdaki örneği inceleyin:

```
#include <iostream>
#include <cstdlib>


class Array {
public:

    Array(int);
    void display();

    int  get_item(int);
    void set_item(int,int);
    int get_max();

    ~Array();
private:

    int *pArray;
    int size;
```

```
};


using namespace std;



Array::Array(int length)
{
    pArray = new int[size = length];
}

Array::~Array()
{
    delete [] pArray;
}

void Array::display()
{
    cout << '(';
    for (int i = 0; i < size; ++i)
        cout << pArray[i] << ' ';
    cout << ')';
}

int Array::get_item(int index)
{
    if (index < 0 || index >= size) {
        cout << "Geçersiz index: " <<  index << endl;
        exit(1);
    }
    return pArray[index];
}

void Array::set_item(int index, int val)
{
```

```
    if (index < 0 || index >= size) {
        cout << "Geçersiz index: " <<  index << endl;
        exit(EXIT_FAILURE);
    }

    pArray[index] = val;
}

int Array::get_max()
{
    int max = pArray[0];

    for (int i = 1; i < size; ++i)
        if (max < pArray[i])
            max = pArray[i];
    return max;
}

const int size = 10;



int main()
{
    Array x = size;

    for (int i = 0; i < size; ++i)
        x.set_item(i, rand());
    x.display();
    cout << "max = " << x.get_max() << endl;

    return 0;
}
```

Örneğimizde *int* türden bir dizi bir sınıfla temsil ediliyor. Sınıfın kurucu işlev bir dizi için *free store* alanından bir blok ayırmış sonlandırıcı işlevi de bu dinamik bloğu geri veriyor. *get_item* ve *set_item* işlevlerinin sınır kontrolü yaparak diziye eleman yerleştirme ve dizinin elemanının değerini alma işlemlerini yaptığını görüyorsunuz. Ayrıca, *get_max* isimli işlev dizinin en büyük elemanını buluyor. *display* işlevi ise dizinin bütün elemanlarını ekrana yazdırıyor.

Sınıf Türünden Göstericiler ve Adresler

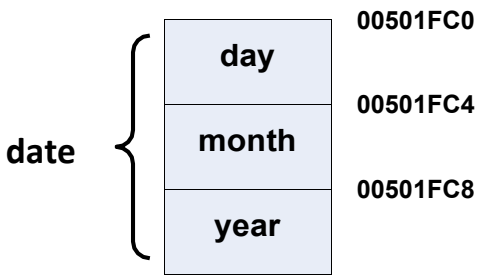
Bir sınıf nesnesinin adresi alınabilir. Elde edilen adres nesnenin ilişkin olduğu sınıfın türündendir, aynı türden bir gösterici değişkene atanmalıdır. Daha önceki örneklerde bildirilen *Date* isimli sınıfa ilişkin bir gösterici değişken tanımlanmış olsun:

```
Date *p;
```

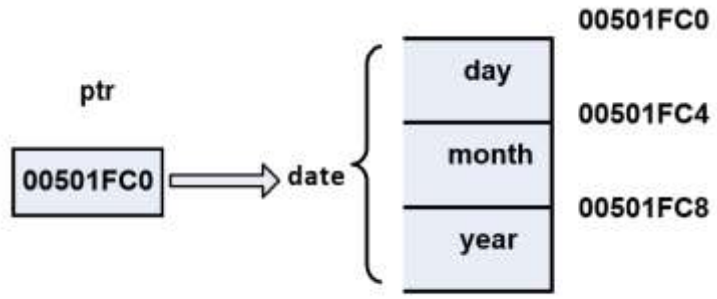
ile *p* gösterici değişkeninin gösterdiği yer, yani **p*, *Date* sınıfı türündendir. Başka bir deyişle, *p* göstericisinin gösterdiği yerde *Date* sınıfı türünden bir nesne vardır. Derleyici bu adresten başlayarak ilk *sizeof(int)* kadar *byte*'ı sınıfın *day* elemanı, sonraki iki *sizeof(int)* kadar *byte*'ı ise sırasıyla *month* ve *year* elemanları olarak ele alır. Şimdi bir sınıf nesnesinin adresinin bir sınıf göstericisine atandığında neler olduğunu adım adım inceleyelim:

Aşağıdaki şekil *int* türünün *4 byte* olduğu varsayımıyla çizilmiştir. *00501FC0* yalnızca konuya açıklık getirmek amacıyla kullanılan rastgele bir adrestir.

```
Date date;      // Date sınıfının veri elemanları için yer ayrılıyor.
```



```
Date *ptr;
ptr = &date;
```



Artık *ptr* adresinden başlayan bilgiler *date* sınıf nesnesinin elemanları olarak yorumlanır. Bu durumda **ptr* ifadesi ile *date* aynı nesneleri belirtir.

Sınıf türünden bir gösterici yoluyla sınıfın elemanlarına ve üye işlevlerine -> işleci ile erişilebilir. Örneğin artık:

```
ptr->display()
```

ile *display* üye işlevi çağrılabilir. Bu durumda *display* üye işlevi *ptr* adresindeki yani

ptr"nin değeri olan adresten başlayan elemanları kullanır. Daha açık bir anlatımla,

```
date.display();
```

gibi bir çağrıda *display* üye işlevi *date* nesnesinin veri elemanlarını kullanıyor. Yani *display*

işlevi içinde kullanılan *day*, *month*, *year* değişkenleri *Date* sınıfının elemanlarıdır. İşte,

```
ptr->display();
```

çağrısıyla da *display* üye işlevi *p* adresindeki nesnenin elemanlarını kullanır. *ptr*

adresinde

```
ptr = &date
```

ataması ile *date* nesnesinin adresi olduğuna göre, *display* işlevi de *date*"in elemanlarını kullanır, değil mi? Yani iki işlem eşdeğerdir.

Tabii -> işleci yerine öncelik ayracı ve nokta işleçlerini de kullanılabilir:

```
(*ptr).display();
```

ile

```
ptr->display();
```

eşdeğerdir. Sınıf göstericisi yoluyla sınıfın elemanlarına yine -> işleci ile erişilebilir. Ancak şüphesiz elemanların erişilebilir olması yani sınıfın *public* bölümünde bildirilmiş olmaları gerekir. Örneğin:

```
ptr->day = 10;
```

gibi bir ifadenin geçerli olması için *day* elemanının sınıfın *public* bölümünde olması gerekir. Aşağıdaki kodu yazarak sınayın:

```
int main()
{
    Date x;
    Date *p;
    p = &x;

    p->display();


    return 0;
}
```

Sınıfın kurucu işlevi yalnızca sınıf nesnesi yaratılırken çağrılır. Bir sınıf türünden gösterici değişken tanımlandığında kurucu işlevi çağrılmaz. Örneğin,

```
Date date;      // Kurucu işlev çağrılır, çünkü nesne yaratılmış!
Date *p;        // Kurucu işlev çağrılmaz, yalnızca gösterici yaratılmış!
```

Sınıf Türünden Referanslar

Bir referans değişkeni bir sınıf nesnesinin de yerine geçebilir. Sınıf türünden referanslar aynı türden bir sınıf nesnesi ile ilkdeğer vererek tanımlanır:

```
Complex a(10, 20);
Complex  &r = a;
```

Burada *Complex* sınıfı türünden *r* isimli referansa yine *Complex* sınıfı türünden *a* nesnesiyle ilkdeğer veriliyor. Böylece artık *r* referansı *a* nesnesinin yerine geçer. *r*"nin kullanıldığı her yerde *a* nesnesi kullanılmış olur. *r* referansı *a* nesnesinin yerine geçtiğine göre, referans yoluyla sınıfın elemanlarına ve üye işlevlerine -tıpkı yapılarda olduğu gibi- *nokta işleci* ile erişilir. Örneğin *Complex* sınıfının *display* isimli bir üye işlevi olduğunu varsayalım. Bu üye işlev *r* referansı ile,

```
r.display();
```

biçiminde çağrılabilir. Bir referans ile bir üye işlev çağrıldığında, üye işlev referansın yerine geçtiği nesnenin elemanlarını kullanır. Örneğin yukarıdaki çağrıda *display* üye işlevi *r* referansının yerine geçtiği nesnenin elemanlarını kullanır.

Sınıf türünden bir referansa bir değişmez ile ya da başka türden bir nesneyle ilkdeğer verme işlemi bazı durumlarda geçerli olabilir. Bu durumu "dönüştüren kurucu işlev" (*conversion constructor*) başlığı altında ele alacağız. Aşağıdaki örneği yazarak çalıştırın:

```
#include <iostream>


class Complex {
public:

    Complex(double = 0, double = 0);
    void display();

private:

    double real, imag;

};


using namespace std;
Complex::Complex(double r, double i)

{

    real = r;
    imag = i;

}
```

```
void Complex::display()

{

    cout << real;
    if (imag > 0)

        cout << "+ ";
    else

        cout << " ";
    cout << imag << "i";

}


int main()

{

    Complex a(10, -20);
    Complex &r = a;
    r.display();



    return 0;

}
```

Öncelikle bir nokta üzerinde açıklama yapalım. *Complex* sınıfının kurucu işlevinin parametre değişkenleri varsayılan değerler alıyor. Bu kurucu işlev, hem tek parametreli kurucu işlev hem de varsayılan kurucu işlev olarak kullanılabilir. Örneğimizde *r* isimli referansa *a* nesnesi ile ilkdeğer verildiğini görüyorsunuz. Bu durumda *r* referansı *a* değişkeninin yerine geçer. Artık, *r.display()* çağrısında, *display* üye işlevi *r* referansının yerine geçtiği elemanları kullanır.

Sınıf Nesnelerinin İşlevlere Geçirilmesi

Bir sınıf nesnesiyle ilgili işlerin, bir sınıfın üye işlevleri tarafından yapılması zorunlu değildir. Şüphesiz global işlevler de sınıf nesneleri üzerinde işlemler yapabilir. Şimdi sınıf nesnelerinin işlevlere geçirilmesi üzerinde duracağız. Sınıf nesnelerinin işlevlere geçirilmesi, tıpkı yapılarda olduğu gibi üç yolla yapılabilir.

Sınıf Nesnesinin Değerinin İşleve Aktarılması

Bu yöntem bildiğimiz "değerle çağırma" (*call by value*) yönteminden başka bir şey değildir. Bu yöntemde işlevin parametre değişkeni sınıf türünden bir nesne olur. İşlev de başka bir sınıf nesnesinin değeri ile çağrılır. Çünkü *C++*"da tıpkı yapılar gibi aynı türden sınıf nesneleri de birbirlerine doğrudan atanabilir. Böyle bir atama sırasında sınıfın elemanları karşılıklı olarak birbirine kopyalanır.

C++"da bir sınıf nesnesi başka bir sınıf nesnesi ile ilk değer verilerek tanımlanırsa, tanımlanan sınıf nesnesi için özel bir kurucu işlev çağrılır. Bu kurucu işleve "kopyalayan kurucu işlev" *(copy constructor)* denir. Programcı kopyalayan kurucu işlevi tanımlamaz ise derleyici bu işlevi kendi tanımlar. Derleyicinin yazdığı

kopyalayan kurucu işlev yaratılan sınıf nesnesinin tüm elemanlarına, ilkdeğer veren sınıf nesnesinin elemanlarının değerini atar. Kopyalayan kurucu işlevi ileride ayrı bir başlık altında ele alacağız.

Daha önce verdiğimiz *Date* sınıfınının kullanıldığı aşağıdaki örneği inceleyin:

```
void func(Date d)

{

    // …
    a.display();

}


int main()

{

    Date date;
    func(date);


   return 0;

}
```

Bu örnekte *func* işlevi çağrısında kullanılan *date* değişkeni *func* işlevinin parametre değişkeni olan *a* değişkenine ilkdeğerini verir. Bu durumda *date* değişkeninin tüm elemanları bire bir *d*"nin elemanlarına kopyalanır. Ancak böyle bir çağrı, nesnenin bütün elemanlarının kopyalanmasını gerektirdiğinden çoğunlukla tercih edilmez. Bu durum aktarım sırasında gereksiz bir zaman kaybı oluşturur. Çağrılan işlev yerel bir sınıf nesnesi üzerinde değişiklik yapamaz.

Nesnenin Adresinin İşleve Geçirilmesi

Bu çağrı biçiminde işleve bir sınıf nesnesinin adresi gönderilir. İşlevin parametre değişkeni de aynı sınıf türünden bir gösterici değişken olur. Bu durumda yalnızca sınıf nesnesine ilişkin veri bloğunun adresi işleve geçirilir. İşlev içinde nesnenin elemanlarına ve sınıfın üye işlevlerine -> işleci ile erişilebilir.

Aşağıdaki örneği inceleyin:

```
void func(Date *pdate)

{

    // …

   pdate -> display();

}
```

```
int main()

{

    Date date;
    func(&date);



    return 0;

}
```

Şüphesiz, bir sınıf nesnesinin adres yoluyla işlevlere geçirilmesi, sınıfın elemanlarının bellekte bitişik bir biçimde bulunması ile mümkün olur. Bu yöntemle işlev parametre olarak aldığı adresteki nesneyi de değiştirebilir.

Nesnenin Referans Yoluyla İşleve Geçirilmesi

Referanslar da aslında adres tutan değişkenler olduğuna göre, nesnenin adresi referans yoluyla da işlevlere geçirilebilir. Bu yöntemde işlev sınıf nesnesinin kendisi ile çağrılır.

İşlevin parametre değişkeni de sınıf türünden bir referans olur. İşlevin parametre değişkeni olan referans işleve gönderilen argüman olan nesnenin yerine geçer. Artık işlev içinde nesnenin elemanlarına ve sınıfın üye işlevlerine nokta işlecinin terimi olan referans ile erişilir.

Aşağıdaki örneği de inceleyin:

```
void func(Date &rdate)

{

    //…
    rdate.display();

}


int main()

{

    Date date;
    func(date);



    return 0;

}
```

Sınıf nesnesinin referans yoluyla aktarımı gösterici yoluyla aktarımı ile eşdeğer verimliliktedir. Ancak referanslar konusunda da belirtildiği gibi, bir işlev aldığı adresteki bilgiyi değiştirmiyorsa okunabilirlik bakımından *const* referans ya da *const* gösterici tercih edilmelidir.

Sınıf Bölümlerinin ve Erişim Kurallarının Anlamı

Daha önce de açıkladığımız gibi sınıfın üye işlevleri ve elemanları sınıfın istenilen bir bölümüne yerleştirilebilir. Yani programcı bu konuda özgürdür. Ancak genellikle sınıfın dışarıya hizmet veren üye işlevlerinin bildirimleri sınıfın *public* bölümüne, sınıfın veri elemanları ise sınıfın *private* bölümüne yerleştirilir. Yukarıda verilen örnekleri incelediğinizde bu yönde hareket edildiğini göreceksiniz.

Sınıfın dışarıya hizmet veren üye işlevleri sınıfın *public*, sınıfın elemanları da sınıfın *private* bölüme yerleştirilirse, dışarıdan sınıf nesnesi ya da göstericisi yoluyla üye işlevlere erişilebilir fakat sınıfın elemanlarına erişilemez. Bu durumda elemanlar doğrudan değil ancak sınıfın üye işlevleri yardımıyla kullanılabilir.

Sınıfın kodlarını yazan kişi böyle bir durumda sınıfın elemanlarına erişimi sağlamak istiyorsa sınıfın *private* elemanlarının değerlerini alan ve onlara değer yerleştiren bir grup üye işlev yazabilir: Bu üye işlevler *İngilizce* genellikle *GetXXX* ya da *SetXXX* biçiminde isimlendirilir. Örneğin:

```
class Date {
private:

    int day, month, year;(mDay, mMonth, mYear)
public:

    Date();

    Date(int d, int m, int y);
    void display();

//Sınıfın elemanlarının değerlerini alan üye işlevler

    int get_day ();
    int get_month();
    int get_year();

//Sınıfın elemanlarına değer yerleştiren üye işlevler

    void set(int, int, int);
    void set_day(int);

    void set_month(int);
void set_year(int);

// Diğer üye işlevler…
```

```
};
```

Yukarıda verilen *Date* sınıfının *day*, *month* ve *year* elemanları *private* bölüme yerleştirilmiş. Bu yüzden bu elemanlara doğrudan erişmek mümkün olmaz. Örneğin *date, Date* türünden bir nesne olsun. *date* nesnesinin *year* elemanının değerini almak ya da *year* elemanına bir değer verilmek istensin.

```
x = date.year;          //Geçersiz!
date. year = 2000;      //Geçersiz!
```

gibi işlemler erişim kurallarına göre geçerli değildir. Ancak bu işlemler *get_year* ve

set_year üye işlevleri çağrılarak gerçekleştirilebilir:

```
x = date.get_year();
date. set_year(2000);
```

Çoğunlukla bir sınıf başka kod parçalarına hizmet verir. Sınıf kodlarından hizmet alan kodlar *(client codes)* sınıfın *public* arayüzüne göre yazılır. Zaten daha önce de açıklandığı gibi sınıftan hizmet alan bir kod parçasının sınıfın *private* bölümüne erişmesi mümkün değildir. Sınıfın *public* bölümü ya da sınıfın *public* arayüzü, sınıf bildiriminde *public* erişim belirteci altından bildirilen bölümdür.

Sınıfın elemanlarının *private* bölüme yerleştirilmesinin en önemli faydası sınıfın elemanlarının genel yapısı değiştirildiğinde, sınıftan hizmet alan kodların bundan etkilenmesinin engellenmesidir. Yani sınıfın *public* arayüzüne bağlı kalmak koşuluyla sınıfın *private* kısmında yapılacak değişmeler sonucunda, sınıftan hizmet alan kodlarda bir değişiklik yapılması gerekmez. Aslında yapılan iş, sınıfın arayüzüyle *(interface*) sınıfın kendi kodlarının *(implementation)* birbirinden ayrılması olarak tanımlanabilir. Sınıfın arayüzünde bir değişiklik yapmadan sınıfın kendi kodlarını değiştirmek sınıfı kullanan kodlarda bir değişiklik yapılmasını engeller. Büyük projeler söz konu olduğunda kodlarda bir değişiklik yapılmasının maliyeti çok azaltılmış olur. Aşağıdaki *Date* sınıfının kodlarını ve onu kullanan örnek bir kodu inceleyin:

```
#include <iostream>
#include <ctime>

using namespace std;

Date::Date()

{

    time_t timer = time(0);;

    tm *tptr = localtime(&timer);
```

```
    day = tptr -> tm_mday;
    month = tptr -> tm_mon + 1;

    year = tptr -> tm_year + 1900;

}



Date::Date(int d, int m, int y)

{

    set(d, m, y);

}


void Date::display()

{

    cout << day << " " << month << " " << year;

}


int Date::get_day()

{

    return day;

}


int Date::get_month()

{

    return month;

}


int Date::get_year()

{

    return year;

}


void Date::set(int d, int m, int y)

{
```

```
    day = d;
    month = m;
    year = y;

}


void Date::set_day(int d)

{

    day = d;

}


void Date::set_month(int m)

{

    month = m;

}


void Date::set_year(int y)

{

    year = y;
}
```

Bu örnekteki *day*, *month* ve *year* elemanlarının sınıfın *public* bölümünde bildirilmiş olduğunu görüyorsunuz. *day*, *month* ve *year* elemanlarına erişmek için üye işlevleri kullanmaya gerek yok, değil mi? Erişim doğrudan gerçekleştirilebilir. Şimdi erişimin doğrudan yapıldığı bir kod örneği verelim:

```
int main()

{

    Date date;
    int d, m, y;



    d = (date.day > 15) ? date.day –1 : date.day + 1;

    m = (date.month > 6) ? date.month – 1 : date.month + 1;
    date.set_date (d, m, y);

    // …
    return 0;

}
```

Burada da *Date* sınıfı türünden bir nesne tanımlanmış. *Date* sınıf nesnesinin elemanlarına sınıfın kurucu işlevi tarafından o anki tarih bilgisi yerleştirilir. Daha sonra bu tarih bilgisinin gün ve ay değerlerine bakılmış ve duruma göre bu değerler bir artırılmış ya da azaltılmış. Sonra da elde edilen değerin yeniden sınıfın veri elemanlarına *set_date* üye işlevi ile yerleştirildiğini görüyorsunuz. Buradaki kodu çok anlamlı bulmayabilirsiniz; Örnek yalnızca sınıfı kullanan koda örnek olarak verildi. Tabii siz sınıfı kullanan kodun yalnızca *main* olduğunu düşünmemelisiniz.

Sınıfın elemanlarını *private* bölüme yerleştirip onlara üye işlevler yoluyla erişilmesinin bir faydası olabilir mi? Şimdi buna da bir örnek verelim: Yukarıdaki *Date* sınıfının veri elemanlarının genel yapısının değiştirildiğini düşünelim. Örneğin artık tarih bilgisinin int türden *day*, *month year* veri elemanlarında değil *"dd/mm/yyyy"* biçiminde *11* elemanlı *char* türden bir dizi içinde tutulmasına karar verilmiş olsun. Bu durumda sınıfın elemanlarını doğrudan kullanan kodun bir geçerliliği kalacak mı? Hayır! Örneğin,

```
d = (x.day > 15) ? x.day –1 : x.day + 1;

m = (x.month > 6) ? x.month – 1 : x.month + 1;
```

Artık *Date* sınıfının *day* ve *month* içiminde bir elemanı olmadığına göre bu kodlar da geçersiz hale gelir. O halde sınıfı kullanan bütün kodlar yeniden yazılmalıdır. İşte eğer bu elemanlar sınıfın *public* bölümüne yerleştirmeseydi o zaman bu elemanlara doğrudan erişilmeyecekti. Onlara erişmek için zorunlu olarak sınıfın *public* üye işlevleri kullanılacaktı. Örneğin, sınıfın *day*, *month*, *year* elemanları sınıfın *private* bölümünde olsaydı, kullanıcı kodlar da şöyle yazılmış olurdu:

```
int main()

{

    Date x;

    int d, m, y;
```

```
    d = (x.get_day() > 15) ? x.get_day() -1 : x.get_day() + 1;
    m = (x.get_month() > 6) ? x.get_month() - 1 : x.get_month() + 1;
    x.set_day(d, m, y);


    return 0;
}
```

Şimdi sınıfın veri yapısı değiştirildiğinde sınıfı kullanan kodların yeniden düzenlenmesine gerek kalmaz. Tek yapılacak şey, üye işlevlerin yeni veri yapısına uygun olarak yeniden yazılmalarıdır. Örneğin, tarih bilgisi karakter dizisi biçiminde tutulacak olduğunda sınıfın üye işlevleri aşağıdaki gibi değiştirilirse, sınıfı kullanan kodların hiç biri değiştirilmek zorunda kalmaz:

```
class Date {
public:
    //…
private:
    char date[11];
};

using namespace std;

Date::Date()
{
    time_t timer = time(NULL);
    struct tm *tptr = localtime(&timer);
    sprintf(date, "%02d/%02d/%04d", tptr->tm_mday, tptr->tm_mon + 1, tptr-
>tm_year + 1900);
}


Date::Date(int d, int m, int y)
{
    sprintf(date, "%02d/%02d/%04d", d, m, y);
}

void Date::display()
{
    puts(date);
```

```
}

int Date::get_day()
{
    return atoi(date);
}

int Date::get_month()
{
    return atoi(date + 3);
}

int Date::get_year()
{
    return atoi(date + 6);
}

void Date::set_date(int d, int m, int y)
{
    day = d;
    month = m;
    year = y;
}

void Date::set_day(int d)
{
    sprintf(date, "%02d", d);
    date[2] = '/';
}
void Date::set_month(int m)
{
    sprintf(date + 3, "%02d", d);
    date[5] = '/';
}
```

```
void Date::set_year(int y)

{

    sprintf(date + 5, "%02d", d);
    date[5] = '/';

}


int main()

{

    Date date;
    int d, m, y;

    d = (date.get_day() > 15) ? date.get_day() -1 : date.get_day() + 1;

    m = (date.get_month() > 6) ? date.get_month() - 1 : date.get_month() +

1;

    date.set_day(d, m, y);



    return 0;

}
```

Gelin sonucu özetleyelim: Sınıfın içsel yapısı değiştirildiğinde sınıfı kullanan kodların bu değişimden etkilenmesi istenmiyorsa, sınıfın elemanları *private* bölüme yerleştirilmeli ve erişimler üye işlevlerle yapılmalıdır. Böylece sınıfın veri yapısı değiştirildiğinde yalnızca üye işlevleri yeniden yazmak yeterli olur.

Dinamik Sınıf Nesneleri

new işleciyle yalnızca *C+*‟ ın doğal veri türleri için değil, sınıflar ve yapılar için de dinamik nesneler yaratılabilir: Örneğin:

```
class Date {
public:

    Date();

    Date (int, int, int);

    // Diğer üye işlevler
private:

    int day, month, year;

};
```

```
Date *p = new Date;
```

deyimiyle *sizeof(Date)* uzunluğunda dinamik bir blok elde ediliyor. Elde edilen dinamik bloğun başlangıç adresi *p* gösterici değişkenine atanıyor. Böylelikle *p* gösterici değişkeni *Date* nesnesini gösteriyor.

Artık *p* gösterici değişkeni ok işleci ile kullanılarak üye işlevler çağrılabilir. Örneğin,

```
p->display();
```

gibi bir işlemde *display* üye işlevi *p* göstericisinin içindeki adreste bulunan yani dinamik olarak elde edilmiş olan nesnenin elemanlarını kullanır.

Sınıfın kurucu işlevi yalnızca tanımlama yöntemiyle nesne yaratıldığında değil, dinamik olarak nesne yaratıldığında da çağrılır. Örneğin,

```
p = new Date;       //Kurucu işlev çağrılır
```

işlemiyle *Date* türünden dinamik bir nesne yaratılıyor. Yaratılan bu dinamik nesne için de kurucu işlev çağrılır. Yani yukarıdaki işlemde sırasıyla şunlar yapılır:

1. *sizeof(Date)* uzunluğunda *free store* üzerinde dinamik bir blok elde edilir.
2. Elde edilen dinamik blok için kurucu işlev çağrılır. Çağrılan kurucu işleve *this* göstericisi olarak elde edilen bloğun başlangıç adresi geçirilir.
3. Yaratılan nesnenin adresi *p* göstericisine atanır.

Peki, dinamik olarak yaratılan sınıf nesneleri için hangi kurucu işlev çağrılır? İşte eğer sınıf isminden sonra ayraç açılıp parametre listesi yazılmamışsa varsayılan kurucu işlevi (*default constructor*) çağrılır. Örnek verelim:

```
p = new Date;
```

burada yaratılan alan için varsayılan varsayılan kurucu işlev çağrılır. Eğer sınıf isminden sonra ayraç açılıp bir parametre listesi yazılırsa, o zaman parametre değişkeni listesine uygun olan kurucu işlevi çağrılır. Örneğin:

```
p = new Date(2, 2, 1989);
```

gibi bir yer ayırma işlemiyle parametrik yapısı

```
Date(int, int, int);
```

biçiminde olan kurucu işlev çağrılır. Peki aşağıdaki gibi bir yer ayırma işleminde *Person*

sınıfının hangi kurucu işlevi çağrılır?

```
p = new Person("Erdem Eker");
```

Dizgeler *const char* türden bir adres belirttiklerine göre, parametresi *char* türden gösterici olan kurucu işlev çağrılır? Ayrıca, sınıf isminden sonra ayraç içine birşey yazılmaması da geçerli bir durumdur. Bu durum, yaratılan dinamik nesne için varsayılan kurucu işlevin çağrılacağı anlamına gelir.

```
p = new Date();
```

ile

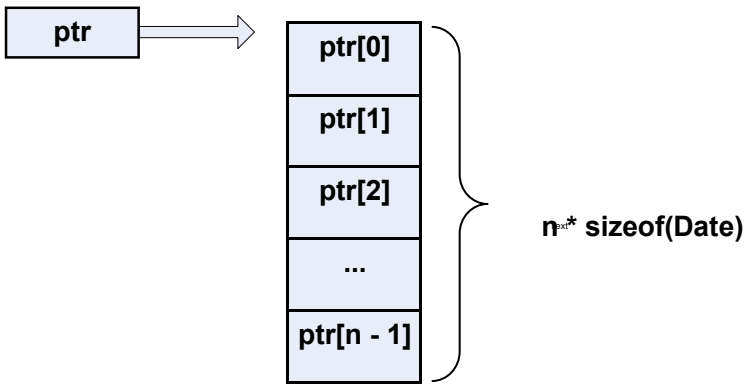
```
p = new Date;
```

tamamen aynı anlamdadır.

new[] işleci ile birden fazla dinamik nesne yaratılabilir. Örneğin:

```
p = new Date[n];
```

burada *n* tane *Date* türünden nesnenin sığabileceği büyüklükte bir bellek bloğu elde ediliyor. Elde edilen bloğun başlangıç adresi *p* gösterici değişkenine atanıyor.



Bu biçimde birden fazla sınıf nesnesi için yer ayrıldığında her sınıf nesnesi için tek tek varsayılan kurucu işlev çağrılır. Sınıf nesneleri için başka bir kurucu işlevi çağırmanın yolu yoktur. Aşağıdaki gibi bir sözdizimin geçerli olabileceğini düşünebilirsiniz:

```
p = new Date[10](10, 10, 20);
```

Ancak böyle bir sözdizim geçerli değildir!

C++"da dinamik bellek yönetimi için neden *malloc* işlevi yerine *new* gibi bir işlecin kullanıldığını şimdi belki de daha iyi anlayabilirsiniz. Dinamik yer elde etme işleminden sonra derleyicinin kurucu işlevi çağırabilmesi için, yer ayırma işleminin hangi türden nesne için yapıldığının derleme zamanında bilinmesi gerekir. Eğer böyle bir anahtar sözcük olmasaydı, yer ayırma işlemleri yine *malloc* işleviyle yapılıyor olsaydı, derleyici bir sınıf nesnesi için yer ayırma işlemi yapıldığını nasıl anlayabilirdi?

Bir sınıf nesnesi için dinamik olarak ayrılan bellek bloğu *delete* işleci ile *free store* alanına geri verilebilir. Bu durumda derleyici, dinamik bellek alanını geri vermeden önce sınıfın sonlandırıcı işlevini *(destructor)* çağırır. Örneğin *p, Date* türünden bir adres olsun:

```
delete  p;
```

işlemiyle önce *p* adresindeki nesne için sonlandırıcı işlevi çağrılır. Daha sonra bu nesne için ayrılmış olan alan geri verilir.

Eğer bir dizi sınıf nesnesi için dinamik olarak yer ayrılmışsa, derleyici tüm dinamik nesneler için tek tek sonlandırıcı işlevleri çağırır:

```
#include <iostream>

class A {

public:
```

```
    A();
    ~A();
};

A::A()
{
    std::cout << "A::A()" << std::endl;
}

A::~A()
{
    std::cout << "A::~A()" << std::endl;
}

int main()
{
    A *pd = new A[5];
    delete[] pd;

    return 0;
}
```

Dinamik bir sınıf dizisi için ayrılan alanın boşaltılması durumunda *[]* unutulmamalıdır. Eğer *[]* unutulursa, sorunlu durumlarla karşılaşılabilir. Bu durumu *new* ve *delete* işleç işlevlerinin ayrıntılı olarak anlatıldığı bölümde ele alacağız.

Dinamik Sınıf Nesneleri Ne Amaçla Kullanılır

Bildiğimiz gibi, kurucu işlevler nesne yaratıldığında, sonlandırıcı işlevler ise nesne bellekten boşaltılacağı zaman çağrılır. Bu durumda yerel sınıf nesnelerinin yaratılması ve bellekten boşaltılması tanımlama noktasına bağlıdır. Bir sınıf nesnesinin istenilen bir noktada yaratılmasını ve istenilen bir noktada yok edilmesi ancak dinamik bellek yönetimiyle sağlanabilir. Örneğin:

```
{
    p = new X;
}
```

```
delete p;
```

Nesnelerin yaratılıp boşaltılma yerlerine bakın. Nesne yerel olarak yaratılsaydı bu blok yapısıyla böyle bir sonucu elde edilebilir miydi? Bir sınıfın bir üye işlevi içinde başka bir sınıf nesnesinin yaratılması ve başka bir üye işlev içinde de yok edilmesi durumlarına sıklıkla rastlanır. Örneğin:

```
void Wnd::initialize()
{
    pScr = new Scr;
    //...
}

void Wnd::close()
{
    //...
    delete pScr;
}
```

Burada *Scr* sınıfı türünden nesne *Wnd* sınıfının *initialize* üye işlevi içinde dinamik olarak yaratılıyor, *close* üye işlevi içinde yok ediliyor. Bu işlem ancak nesnenin dinamik olarak yaratılmasıyla gerçekleştirilebilir. Bu örneklerden de görüldüğü gibi nesnenin dinamik olarak yaratılmasının asıl nedeni dinamik bellek kullanmak değildir. Sınıfın kurucu işlevinin ve sınıfın sonlandırıcı işlevin istenilen yerlerde çağrılmasını sağlamaktır.

const Üye İşlevler ve const Sınıf Nesneleri

Öncelikle *const* anahtar sözcüğünün daha önce öğrendiğimiz anlamlarını hatırlayalım: *const* anahtar sözcüğü bir değişkenin tanımlanmasında kullanıldığında, tanımlanan değişkenin değerinin değiştirilemeyeceğini gösteriyordu:

```
const int x = 5;    // x nesnesinin değeri değiştirilemez.
x = 10;             //Geçersiz
```

Değişkenin bir gösterici olması durumunda ise, *const* anahtar sözcüğünün verdiği anlam anahtar sözcüğün kullanıldığı yere göre değişiyordu:

```
int x = 5;
const int *ptr = &x;
```

```
*ptr = 1;   // Geçersiz!
```

Yukarıdaki örnekte *ptr* gösterdiği yer *const* olan bir gösterici değişkendir. *ptr* değişkenine başka bir adres atanabilir ama **ptr*, yani *ptr*"nin gösterdiği nesne atama yoluyla değiştirilemez.

```
int x = 5, y = 10;
int * const ptr = &x;

ptr = &y;   // Geçersiz!
```

Yukarıdaki örnekte *ptr* kendisi *const* olan bir göstericidir. *ptr* göstericisinin gösterdiği nesneye, yani **ptr* ifadesine atama yapılabilir. Ancak *ptr* nesnesine bir atama yapılamaz. Yani *ptr* göstericisinin bir başka nesneyi göstermesi sağlanamaz.

```
int x = 5, y = 10;

const int *const ptr = &x;

*ptr = 1;          //Geçersiz!

ptr = &y;          //Geçersiz!
```

Yukarıdaki örnekte ise *const* anahtar sözcüğünün her iki anlamı birleştiriliyor. Yani *ptr* hem kendisi hem de gösterdiği yer *const* olan bir gösterici değişkendir. Ne *ptr* nesnesine ne de *ptr*"nin gösterdiği nesneye, yani **ptr* nesnesine atama yapılabilir.

const anahtar sözcüğü bir işlev bildiriminde ya da tanımında işlevin geri dönüş değerinin tür bilgisinden önce de yazılabilir :

```
const char *func();
```

Yukarıdaki bildirimden *func* işlevin geri dönüş değeri olan adresteki nesnenin değiştirilemeyeceği sonucu çıkar:

```
*func() = 'm';  //Geçersiz!
```

Şüphesiz yukarıdaki kuralların hepsi referanslar için de geçerlidir.

```
int x = 5;

const int &r = x;

r = 20;         //Geçersiz!
const char &func();

func() = 5;    //Geçersiz!
```

const Anahtar Sözcüğünün Yeni Bir Anlamı

C++ dilinde bir üye işlev *const* anahtar sözcüğü ile bildirilebilir. Bunun için *const* anahtar sözcüğünü hem işlevin bildiriminde hem de işlevin tanımında, parametre ayracının sağına yerleştirmek gerekir. Örneğin:

```
class Date {
public:

    Date(int, int, int);
    void set(int, int, int);
    int get_day()const;

    int get_mon()const;
    int get_year()const;
    void display()const;

    //…
private:

    int day, mon, year;

};
```

Yukarıdaki örnekte *Date* isimli sınıfın *get_day*, *get_mon*, *get_year* ve *display* isimli üye işlevleri *const* üye işlevler olarak bildiriliyor. Bu işlevlerin tanımları da aşağıdaki gibi yapılabilir:

```
#include <iostream>


int Date::get_day() const

{

    return day;

}


int Date::get_mon() const
```

```
{
    return mon;
}

int Date::get_year() const
{
    return year;
}

void Date::display() const
{
    std::cout << day << " " << mon << " " << year;
}
```

const anahtar sözcüğünün yalnızca işlevin bildiriminde değil, işlevin tanımında da yazılması zorunludur. Yalnızca birinde yazılmış olması geçersizdir. Ayrıca *const* anahtar sözcüğünün yerleştirildiği yere de dikkat edilmelidir. Eğer bu anahtar sözcük en sola konulmuş olsaydı, işlevin geri dönüş değerinin *const* olduğu, yani değiştirilemeyeceği anlamına gelirdi.

const Üye İşlev ne Anlama Gelir

const bir üye işlev ait olduğu sınıfın elemanlarını değiştiremez. Yukarıdaki örnekte *get_day*, *get_mon*, *get_year* ve *display* işlevleri *Date* sınıfının *day*, *mon*, *year* elemanlarının değerlerini kullanabilir. Ancak içlerindeki değerleri değiştiremez. *day*, *mon*, *year* isimli elemanlar sınıfın *public* bölümünde olsalardı da bu işlevler tarafından değiştirilemezlerdi. *const* anahtar sözcüğü bu kullanımı ile okunabilirliği artırır. Kodu inceleyen kişi bu işlevlerin elemanlarını değiştirmeyeceğini bilirse, kod hakkında daha fazla bilgi edinmiş olur. İyi bir tasarımda *const* anahtar sözcüğü kararlı bir biçimde kullanılmalıdır. Eğer kararlı bir biçimde kullanılırsa artık *const* olmayan üye işlevlerin sınıfın elemanlarını değiştireceği düşünülebilir.

Yalnızca üye işlevler *const* olabilir. Global bir işlev *const* anahtar sözcüğü ile tanımlanamaz.

const Üye İşlevlerin Sınıfın Diğer Üye İşlevlerini Çağırması

Bir üye işlev içinde başka bir üye işlevin doğrudan çağrılabildiğini, bu durumda çağrılan üye işlevin çağıran üye işlev ile aynı sınıf nesnesinin elemanlarını kullandığını hatırlayalım. *const* bir üye işlev içinde *const* olmayan bir üye işlevin çağrılması geçerli değildir. Yukarıdaki örnekte *Date* sınıfının *display* üye işlevi *const* bir üye işlevdir. Bu işlevin *const* olmayan bir üye işlevi çağırması geçersizdir. Eğer bu duruma derleyici tarafından izin verilseydi *display* işlevi örneğin *set* işlevini çağırarak sınıfın elemanlarını dolaylı bir biçimde değiştirebilirdi:

```
void Date::set(int d, int m, int y)
{
    day = d;
    mon = m;
    year = y;
}

int Date::display()const
{
    set(3, 5, 1980);   //Gecersiz.
    cout << day << " " << mon << " " << year;
}
```

Yukarıdaki örnekte *Date* sınıfının *display* işlevi içinde yine aynı sınıfın *set* isimli işlevine yapılan çağrı geçersizdir. *const* bir üye işlev *const* olmayan bir üye işlevi çağıramaz.

Sınıf Elemanlarının Adresine Geri Dönen const Üye İşlevler

const bir üye işlev içinde sınıfın elemanlarının *const* olduğu varsayılır. Eğer *const* bir üye işlev sınıfın bir elemanın adresi ile geri dönecekse, elemanının *const* olduğu varsayıldığından işlevin geri dönüş değerinin de *const* bir gösterici ya da referans olması gerekir.

```
class A {
    int a;
public:
    //...
    int *get_adr() const;
};

int *A::get_adr() const
{
    return &a; //Geçersiz
}
```

Yukarıdaki örnekte *A* sınıfının *get_adr* işlevi *const* bir üye işlevdir. İşlevin tanımı içinde, hangi sınıf nesnesi için çağrılmışsa, o nesnenin elemanları da *const* olarak ele alınır. Bu durumda yukarıdaki işlevde *const* bir nesnenin adresi gösterdiği yer *const* olmayan bir göstericiye atanır. Hata durumunun ortadan kaldırılması için, ya işlevin geri dönüş değeri *const int ** olmalı ya da *get_adr* işlevi *const* üye işlev olmamalıdır.

Şüphesiz bir referansa geri dönen üye işlevler için de yukarıdaki anlatılan durum geçerlidir. Yani *const* bir üye işlev sınıfın elamanının kendisi ile dönecekse bu durumda işlevin geri dönüş değeri *const* referans olmalıdır.

Kurucu ve Sonlandırıcı İşlevler const Olamaz

Sınıfın kurucu ve sonlandırıcı işlevleri *const* olamaz. Kurucu işlevler sınıfın elemanlarına ilkdeğer vermek amacıyla kullanıldığına göre *const* olmalarının zaten bir anlamı olmaz. Benzer biçimde sonlandırıcı işlevlerin de sınıfın elemanlarını değiştirmesi gerekebildiği için *const* olarak tanımlanması yasaklanmıştır.

Sınıf Nesnelerinin Bitsel ve Soyut Durumları

Bir sınıf nesnesi bellekte bir yer kaplar, bu yönüyle nesne *byte*"lar ve *byte*"ları oluşturan bitler topluluğudur. Ancak sınıf nesnesi problem düzleminde bir varlığa karşılık gelir.

Nesnenin elemanlarının bitsel değerleri değiştiğinde nesnenin gözlenebilir durumunda bir değişiklik olmayabilir. Diğer taraftan nesnenin elemanlarının bitsel değerlerini değiştirmeksizin, nesnenin gözlenebilir durumunda bir değişiklik yaratmak da olasıdır.

Bir sınıf nesnesi bellekte kendi kapladığı yerin dışında bir ya da birden fazla kaynağı kontrol ediyor olabilir. Örneğin bazı sınıf nesneleri, dinamik olarak elde edilmiş bir bellek alanını kontrol eder. Aşağıdaki örneği inceleyin:

```
class Name {

    char *pstr;
    int len;

public:

    Name(const char *);
    void set_at(int, int);

    //...

};
```

```
#include <iostream>
#include <cstring>
```

```
Name::Name(const char *str)

{

    len = strlen(str);

    pstr = new char[len + 1];
    strcpy(pstr, str);

}


void Name::set_at(int index, int ch)

{

    pstr[index] = ch;

}
```

Yukarıdaki örnekte *Name* sınıfının kurucu işlevi, sınıf nesnesi için dinamik bir yer elde ediyor. Elde edilen dinamik bloğun başlangıç adresi, *char* türden bir gösterici olan *pstr* isimli elemanda tutuluyor. *set_at* işlevi ise elde edilen dinamik bloktaki *index* indisli nesneye, ikinci parametre değişkenine gönderilen değeri atıyor.

set_at işlevi *const* olarak tanımlanmalı mıdır? *set_at* işlevi sınıfın elemanlarını, yani *len* ve *pstr* elemanlarını değiştirmese de, *const* olarak tanımlanmamalıdır. Sınıf nesnesinin yer aldığı bellek bloğuna bir veri yazılmasa da, sınıf nesnesinin değerinde bir değişiklik yapıldığı için, *set_at* işlevinin *const* olmaması gerekir. Yani bir üye işlevin *const* olup olmaması kararında, nesnenin fiziksel durumundaki değişiklik değil, soyut durumundaki değişiklikler dikkate alınmalıdır. Nesne yalnızca fiziksel ya da somut durumu ile yani bellekte kapladığı yer ile değerlendirilmemelidir. Bazen bir üye işlev sınıf nesnesinin bellekte kapladığı yere bir veri aktarmamasına karşın bir sınıf nesnesinin durumunu değiştirebilir. Bir *Name* nesnesinin görevi bir isim tutmaktır, değil mi? Bir *Name* nesnesinin tuttuğu ismin değişmesi demek, nesnenin soyutlanmış durumunun da değişmesi demektir. Böyle bir değişiklik *Name* nesnesinin fiziksel durumunda bir değişiklik yapmadan gerçekleştirilebilir.

Yukarıdaki *set_at* işlevi bu durum için örnek olabilir. Bu durumda işlevin *const* olmayan üye işlev olması gerekir.

const Üye İşlevlerin Derleyici Açısından Anlamı

Derleyici açısından bakıldığında bir üye işlevin *const* olması ne anlama gelir? Üye işlevlerin aslında gizli bir parametre değişkenine sahip olduğunu biliyorsunuz. Bir üye işlev hangi sınıf nesnesi ile çağrılıyorsa, üye işlevin gizli parametre değişkenine o nesnenin adresi aktarılır, değil mi? Bir üye işlev içinde bu göstericiye *this* anahtar sözcüğü ile erişilebildiğini hatırlıyorsunuz.

```
class A {

    int x;
public:

    void member_func();

};
```

*Y*ukarıdaki sınıf bildirimine *A* sınıfının *public* üye işlevi olan *member_func* isimli üye işlevin parametre değişkenine sahip olmadığını görüyorsunuz. Oysa derleyici açısından bakıldığında bu işlevin bir parametre değişkeni vardır:

```
void A::member_func(A *const this);
```

this göstericisinin kendisi *const* bir gösterici olduğunu anımsayalım. Bu durumda *this*

göstericisine işlev içinde bir atama yapılamaz. Örneğin aşağıdaki kod geçersizdir:

```
A a;

void A::member_func()
{
    this = &a; //Geçersiz!
}
```

Ancak *this* göstericisi gösterdiği yer *const* olan bir gösterici değildir. Yani *this* göstericisinin gösterdiği nesneye atama yapılabilir. Ya da *this* göstericisinin gösterdiği nesnenin elemanlarına atama yapılabilir.

Peki dışarıdan gizlice adresi alınan sınıf nesnesinin üye işlev içinde atama yoluyla değiştirilmesi engellenmek istenseydi, *this* göstericisinin, gösterdiği yer *const* olan bir gösterici olması tercih edilirdi, değil mi?

```
void A::member_func(const A*const this);
```

Böyle bir durumda *this* göstericisinin gösterdiği nesneye yani ***this* nesnesine atama yapılmayacağı gibi, *this* göstericisinin gösterdiği nesnenin elemanlarına da atama yapılması engellenmiş olurdu. İşte derleyici açısından bakıldığında *const* üye işlev tam bu anlama gelir. *const* olarak bildirilmiş bir işlevin *this* göstericisi gösterdiği yer *const* olan bir göstericidir.

Üye işlev içinde sınıfın elemanlarına doğrudan erişilebilir. Ama gerçekte derleyici bu elemanlara ulaşmak için *this* göstericisini kullanır.

```
void A::member_func()
{
```

```
    x = 10;     //this->x = 10;
}
```

Üye işlev *const* olduğunda *this* göstericisi de gösterdiği yer *const* olan bir gösterici olduğundan bu atama artık geçerli olmaz:

```
void A::foo() const
{
    x = 10;     //Geçersiz!
}
```

Şimdi de bir üye işlevin tanımı içinde sınıfın başka bir üye işlevinin çağrılması durumunu ele alalım. Bu duruma derleyici açısından bakalım. Çağıran üye işlev dışarıdan gizlice aldığı adresi, yani *this* göstericisinin değerini, çağırdığı işleve gizlice argüman olarak geçer. Yani çağıran üye işlevin *this* göstericisinin çağrılan üye işlevin *this* göstericisine atanması, kopyalama yoluyla aktarılması söz konusudur.

C++'da gösterdiği yer *const* olmayan bir göstericiye gösterdiği yer *const* olan bir göstericinin değerinin atanması geçerli değildir:

```
const int  *cptr;
int *p;

p = ptr;    //Geçersiz!
```

mutable Anahtar Sözcüğü

Bir üye işlev kendisini çağıran sınıf nesnesinin bir elemanının değerini değiştirmiş olsa da, nesnenin soyutlanmış anlamı üzerinde hiçbir gözlenebilir değişiklik yapmamış olabilir. Bu durumda okunabilirlik açısından şüphesiz söz konusu işlevin *const* üye işlev olması gerekir.

Peki, *const* üye işlev sınıfın elemanlarını değiştiremediğine göre bu nasıl mümkün olur? Bu durumda sınıfın elamanı *mutable* anahtar sözcüğü ile bildirilir. Sınıfın bir elemanının *mutable* anahtar sözcüğü ile bildirilmesinin derleyiciye verdiği bilgi şudur: Bu eleman *const* üye işlevler tarafından da değiştirilebilir. Aşağıdaki örneği inceleyin:

```
class X {
    mutable bool flag;
    //..
public:
    //...
```

```
    void foo() const;
};


void X::foo() const
{
    flag = false;
    //...
}
```

X sınıfının *bool* türden *flag* isimli elemanı *mutable* anahtar sözcüğü ile bildiriliyor. Sınıfın

foo isimli *const* üye işlevinin, sınıfın *flag* isimli elemanına atama yapması geçerlidir.

const Sınıf Nesneleri

const bir sınıf nesnesi değeri hiç değişmeyecek bir sınıf nesnesidir.

Bir sınıf nesnesi *const* olarak tanımlanabilir. *const* bir sınıf nesnesinin elemanları kullanılabilir ama herhangi bir biçimde değiştirilemez. Örneğin,

```
const Date birth_date(4, 3, 1964);
```

gibi bir bildirimle *birth_date* isimli sınıf nesnesi *const* olarak tanımlanıyor. Artık *birth_date* nesnesinin elemanları değiştirilemez. *Date* sınıfının *day*, *month* ve *year* elemanları *public* bölümde olsaydı bile,

```
birth_date.day = 10;
```

gibi bir atamayla elemanın değiştirilmesi mümkün olmazdı.

Bir sınıf nesnesinin elemanları, sınıfın üye işlevleri tarafından da değiştirilebilir. O halde *const* bir sınıf nesnesinin, sınıfın elemanlarını değiştirebilecek bir üye işlevi de çağıramaması gerekir! Örneğin:

```
const Date birth_date(4, 3, 1964);
birth_date.set(6, 1, 1966); //Geçersiz.
```

gibi bir işlemde *set* üye işlevi, *birth_date* nesnesinin elemanlarını değiştirir. Güvenlik ve okunabilirlik açılarından bu durumun da engellenmesi gerekir. Peki, ama derleyici *set* işlevinin sınıfın elemanlarını değiştirip

değiştirmeyeceğini nasıl anlar? İşlevin koduna bakarak bu bilgiyi alması mümkün olmayabilir. Çünkü program birkaç modül halinde yazılmış olabilir. İlgili işlev o modülde bulunmayabilir. Dilin kurallarına göre, *const* bir sınıf nesnesi ile ancak sınıfın *const* üye işlevlerini çağrılabilir. Çünkü *const* üye işlevlerin sınıfın elemanlarını değiştirmeyeceği başka bir kontrolle zaten güvence altına alınmıştır.

Örneğin:

```
const A a;
a.func();
```

Yukarıdaki kod parçasında derleyici *A* sınıfına ilişkin *func* işlevinin bildirimine bakarak onun *const* bir üye işlev olup olmadığını saptar. Eğer *func const* bir üye işlev ise çağrı geçerlidir, değilse çağrı geçersizdir. Durum bir hata iletisiyle bildirilir.

Derleyici açısından bakıldığında durum son derece açıktır:

```
a.func();
```

çağrısıyla *const a* nesnesinin adresi gizlice *func* işlevinin parametre değişkenine atanır. *a*

nesnesinin adresi

```
(const A *)
```

türündendir. Bu adres ancak gösterdiği yer *const* olan bir göstericiye atanabilir. Oysa *func*

işlevi *const* üye işlev değil ise, işlevin gizli parametre değişkeni

```
A *
```

türündendir. Dolayısıyla işlev çağrısı geçerli değildir.

Gösterdiği yer *const* olan bir sınıf göstericisi ya da *const* bir sınıf referansı da söz konusu olabilir. Böyle bir gösterici ya da referans yoluyla bir sınıf nesnenin elemanları değiştirilemez. Aşağıdaki örneği inceleyin:

```
void display_totaldays(const Date *pDate)
{
```

```
    cout << pDate->totaldays();    //totaldays işlevi const değilse
geçersiz!

}


int main()

{

    Date date
    display_totaldays(&date);



    return 0;

}
```

Burada *display_totaldays* işlevinin parametre değişkeni, gösterdiği yer *const* bir sınıf göstericisidir. Bu göstericinin gösterdiği sınıf nesnesinin elemanları değiştirilemez ve bu nesne için sınıfın *const* olmayan bir üye işlevi çağrılamaz. Aynı durum *const referanslar* için de söz konusudur. Örneğimizi aşağıdaki biçime dönüştürüyoruz:

```
void display_totaldays(const Date &r)

{

    cout << r.totaldays();     //totaldays işlevi const değilse geçersiz!

}


int main()

{

    Date date;
    display_totaldays(date);



    return 0;

}
```

const anahtar sözcüğünün, genel olarak okunabilirliği artırmak amacıyla kullanıldığını anımsayalım. Hizmet alacağı sınıfın arayüzünde *const* bir sınıf göstericisini ya da referansını gören programcı, bu gösterici ya da referans yoluyla sınıfın elemanlarının değiştirilemeyeceğini anlayarak daha fazla bilgi edinir. Bir işlevin adresini aldığı sınıf nesnesinin elemanlarını değiştirmeyecek ise, yani bir set işlevi değil ise, işlevin parametre değişkeni gösterdiği yer *const* olan bir gösterici ya da referans olmalıdır.

const Sınıf Nesneleri İçin Kurucu İşlevin Çağrılması

const bir sınıf nesnesi yaratılırken sınıfın kurucu işlevleri normal olarak çağrılır. Çünkü kurucu işlevlerin amacı bir sınıf nesnesine ilkdeğer vermektir. Nasıl doğal türlerinden *const* bir nesneye ilk değer

verilebiliyorsa, *const* bir sınıf nesnesine de kurucu işlev yardımıyla ilkdeğer verilebilir. Yani kurucu işlevin çağrılması nesnenin değerini değiştirilmesi değil nesneye ilk değer verilmesi anlamına gelir. Örneğin:

```
const Date d(29, 5, 1992);
```

deyimi ile sınıfın kurucu işlevi sınıfın nesnesinin elemanlarına işleve gönderilen argümanlar ile ilkdeğerini verir. Yani kurucu işlev, sınıfın elemanlarını atama yapmaz, onlara ilkdeğerini verir.

Sınıfın elemanlarını değiştirmeyen işlevler, *const üye işlev* olarak tanımlanmazlarsa, bu işlevlerin *const* sınıf nesneleri tarafından çağrılması engellenmiş olur. Yani bu işlevler *const* sınıf nesnelerine hizmet veremez.

Sınıfın const Elemanları

C++ dilinde *const* anahtar sözcüğünün bir kullanımı daha vardır. Sınıf bildirimi yapılırken sınıfın bir elemanı da *const* olarak bildirilebilir:

```
class A {
    int a;
    const int b;
    //...
public:
    void func();
};

void A::func()
{
    b = 10;     //Geçersiz!
}
```

Yukarıdaki bildirimde *A* sınıfının *b* isimli elemanı *const* olarak bildiriliyor. *A* sınıfının üye işlevleri içinde *b* elemanının değeri değiştirilemez.

Ancak bir sınıfın elemanının *const* olarak bildirilmesi durumunda, sınıfın kurucu işlevinin özel bir sözdizim kuralına göre tanımlanması gerekir. *M.I.L* sözdizimi diye isimlendireceğimiz bu kurala ileride ayrıntılı olarak değineceğiz.

const Yüklemesi

Bir sınıf aynı isimli ve aynı parametre yapısına sahip biri *const* biri *const* olmayan iki üye işleve sahip olabilir. Bu duruma “const yüklemesi” (*const overloading*) denir.

Üye işlevlerin amaç kod içine yazılmasında sınıf isimleri ve parametre türlerinin yanı sıra *const* anahtar sözcüğü de bir şekilde kodlanır. Başka bir deyişle *const* işlevin imzasının bir parçasıdır. Böyle bir durumda aynı ısimli üye işlev çağrıldığında gerçekte hangi üye işlevin çağrılmış olduğu, çağrı ifadesinde kullanılan nesnenin *const* olup olmadığına bakılarak saptanır. Eğer çağrı işlemi *const* bir sınıf nesnesi ile yapılmışsa *const* olan üye işlevin, *const* olmayan bir sınıf nesnesi ile yapılmışsa *const* olmayan üye işlevin çağrıldığı anlaşılır.

Aşağıdaki kodu derleyerek çalıştırın:

```
class A {
public:

    A(){}

    void func();

    void func() const;

};


#include <iostream>

using namespace std;


void A::func()

{

    cout << "A::func()" << endl;

}


void A::func() const

{

    cout << "A::func()const" << endl;

}


int main()

{

    A a;

    const A ca;
    a.func();

    ca.func();


    return 0;

}
```

Kopyalayan Kurucu İşlev

Bir sınıfın kopyalayan kurucu işlevi *(copy constructor)*, o sınıfa ait bir nesnenin ilkdeğer verilerek yaratılması durumunda çağrılan, özel bir kurucu işlevdir. Hatırlayacağınız gibi, ilkdeğer verme bir bildirim deyimidir, atama deyiminden farklı özellikler taşır.

```
int x = 10;
```

Yukarıdaki deyimde *int* türden *x* değişkeni, *10* ilkdeğerini alarak yaratılıyor. Bir sınıf nesnesi de,aynı türden bir başka sınıf nesnesinden ilkdeğerini alarak tanımlanabilir:

```
class A {
public:
    A();
};

int main()
{
    A a1;       //a1 nesnesi için varsayılan kurucu işlev çağrılıyor.
    A a2(a1);   //a2 nesnesi ilk değerini a1 nesnesinden alarak yaratılıyor.

    return 0;
}
```

C++"da bir sınıf nesnesinin yaratıldığı her durumda bir kurucu işlevin otomatik olarak çağrıldığını hatırlayalım. Bir sınıf nesnesi ilkdeğerini başka bir sınıf nesnesinden alarak yaratıldığı zaman da bir kurucu işlev çağrılır. Bu işlev "*kopyalayan kurucu işlev*" olarak isimlendirilir. Kopyalayan kurucu işlev, programcı tarafından tanımlanmaz ise derleyici tarafından otomatik olarak tanımlanır.

Derleyicinin otomatik olarak yazacağı kopyalayan kurucu işlev, sınıfın *public* bölümünde bildirildiği kabul edilen *inline* bir işlevdir ve parametrik yapısı aşağıdaki gibidir:

```
class A{
    //
public:
    A(const A&);    //Kopyalayan kurucu işlev
};
```

Derleyicinin yazdığı kopyalayan kurucu işlev, çağrılmış olduğu sınıf nesnesinin elemanlarına yani **this* nesnesinin elemanlarına, ilkdeğer verme işleminde atama işlecinin sağ tarafında bulunan sınıf nesnesinin ilgili elemanlarının değerlerini kopyalar. Böyle bir kopyalama ingilizcede *"memberwise copy"* olarak bilinir.

Kopyalayan Kurucu İşlev Hangi Durumlarda Çağrılır

Bir sınıf nesnesine üç ayrı biçimde ilkdeğer verilebilir:

1. Açık ilkdeğer verme biçiminde

```
A a1;
A a2(a1);
```

Yukarıdaki kod parçasında *a2* nesnesi, değerini daha önce tanımlanan *a1* nesnesinden alarak yaratılıyor. Tanımlanan *a2* nesnesi için derleyicinin yazdığı kopyalayan kurucu işlev çağrılır. *a1* nesnesi bu işleve referans yoluyla geçirilir.

Tek parametreli kurucu işlevlerin aşağıdaki biçimde de çağrılabileceğini biliyorsunuz:

```
A a2 = a1;
```

Bu durumda da *a2* nesnesi için kopyalayan kurucu işlev çağrılır. Bu işleve *a1* nesnesi referans yoluyla geçirilir.

2. Argümanlardan parametre değişkenlerine değerle aktarım

Bir işleve gönderilen argüman, ilgili işlevin parametre değişkeni olan nesneye ilkdeğerini(ilk değerini) verir. Yani işlev çağrıldığında, parametre değişkeni olan nesne, işleve argüman olarak gönderilen ifadeden ilkdeğerini(ilk değerini) alarak yaratılır:

```
void func(A p)
{
    //…
}
```

```
void foo()

{

    A a;
    func(a);

}
```

Yukarıdaki kod parçasında *func* işlevinin parametre değişkeni *A* sınıfı türündendir. *foo* işlevi içinde çağrılan *func* işlevine, yerel *a* nesnesinin değeri argüman olarak gönderiliyor. *func* işlevi çağrıldığında yaratılan *A* sınıfı türünden olan parametre değişkeni *p*, ilk değerini *a* nesnesinden alarak yaratılır. Bu durumda parametre değişkeni *p* için kopyalayan kurucu işlev çağrılır. Çağrılan kopyalayan kurucu işleve *a* nesnesi referans yoluyla geçirilir.

3. İşlevlerin geri dönüş değerlerini, bir geçici nesne oluşturarak dışarıya aktardığını biliyorsunuz. Geri dönüş değeri üreten bir işlevin kodunun çalıştırılması sırasında *return* deyimi yürütüldüğünde, işlevin geri dönüş değeri türünden bir geçici nesne ilkdeğer verme yoluyla yaratılır. Yaratılan geçici nesne ilk değerini *return* ifadesinden alır. Bir işlev bir sınıf türüne geri dönüyorsa, *return* ifadesi de bir sınıf nesnesi olmalıdır, değil mi? Bu durumda geri dönüş değerini içinde taşıyacak geçici nesne ilk değerini *return* ifadesi olan nesneden alarak yaratılır. Yani geri dönüş değerinin içinde taşıyacak geçici nesne için kopyalayan kurucu işlev çağrılır. *return* ifadesi olan sınıf nesnesi de kopyalayan kurucu işleve referans yoluyla geçirilir. Aşağıdaki kod parçasını inceleyin:

```
A func()

{

    A a;

    //…
    return a;

}
```

Yukarıdaki işlevde *return* deyimi yürütüldüğünde yaratılan geçici nesne ilk değerini yerel

a nesnesinden alır. Yani

```
A temp_object = a;
```

gibi bir işlem söz konusudur.

Bildiğiniz gibi, bir sınıfın farklı imzaya sahip birden fazla kurucu işlevi olabilir ama sınıfın sonlandırıcı işlevi tektir. Yukarıdaki üç durumda da kopyalayan kurucu işlevle yaratılan sınıf nesneleri için bu nesnelerin ömrü bittiğinde sınıfın sonlandırıcı işlevi çağrılır.

Programcının bir kopyalama kurucu işlevi yazmaması durumunda derleyici bu işlevi kendisi yazıyorsa ve derleyicinin yazdığı kopya kurucu işlevi tüm elemanları karşılıklı birbirine atıyorsa, neden programcı bir kopyalayan kurucu işlevi yazma gereği duysun?

Bazı durumlarda nesnelerin elemanlarının karşılıklı olarak birbirine atanması istenen bir durum değildir!

Bazı sınıflarda sınıf nesnelerinin bazı elemanları dışsal kaynakları kontrol eder. Bu elemanlarının değerlerinin birbirlerine atanması istenmeyen sonuçlara yol açabilir. Yazıları tutmak için tanımlanan ismi *String* olan bir sınıfı ele alalım. Sınıfın basitleştirilmiş tasarımı aşağıdaki gibi olsun:

```
//string.h
class String {

    char *m_p;
    int m_len;

public:

    String(const char *str);

    ~String();

    void make_upper();
    void print() const;

};


//string.cpp

//#include "string.h"
#include <iostream>
#include <cstring>
#include <cctype>


using namespace std;

String::String(const char *str)

{

    m_len = strlen(str);

    m_p = new char[m_len + 1];
    strcpy(m_p, str);

}


String::~String()

{
```

```
    delete[] m_p;

}


void String::make_upper()

{

    for (int k = 0; k < m_len; ++k)
        m_p[k] = toupper(m_p[k]);

}


void String::print() const

{

    cout << m_p;

}
```

Tanımlanan *String* sınıfını kullanan aşağıdaki *main* işlevini dikkatle inceleyin:

```
int main()

{

    String s1("Necati");

    {

        String s2 = s1;
        s2.print();

        //...

    }

    s1.make_upper();

    s1.print();
    return 0;

}
```

s1 nesnesi yaratıldığında çağrılan kurucu işlevin çalışmasıyla, *7* karakter uzunluğunda bir dinamik alan yaratılıyor. Yaratılan dinamik alanın başlangıç adresi *s1* nesnesinin elemanı olan *m_p* isimli gösterici elemanda tutuluyor.

Daha sonra yer alan blok içinde bu kez *s2* isimli nesne ilk değerini *s1* nesnesinden alarak yaratılıyor. Derleyicinin yazdığı kopyalayan kurucu işlev, *s1* nesnesinin elemanlarının değerlerini *s2* nesnesinin elemanlarına atayacağına göre, *s2* nesnesinin de *m_p* isimli elemanı olan gösterici aynı dinamik alanı gösterir, değil mi?

s2 nesnesinin tanımlandığı bloğun sonunda, *s2* nesnesinin ömrü tamamlanacağı için *s2* nesnesi için sınıfın sonlandırıcı işlevi çağrılır. Sonlandırıcı işlev daha önce elde edilen dinamik bloğu "free store"a geri vermek için tanımlanmıştır. *s2* nesnesinin *m_p* elemanının gösterdiği dinamik alan *free store*"a geri verilir. Ancak bu dinamik bloğun başlangıç adresi aynı zamanda *s1* nesnesinin de *m_p* elemanının değeridir. *s2* nesnesi ömrünü tamamlamasına karşın *s1* nesnesi halen hayatını sürdürmektedir. Ancak *s1* nesnesinin dinamik alanı artık serbest bırakılmıştır. Sınıfın *make_upper* ve *print* isimli üye işlevleri ayrılan dinamik alan üzerinde işlem yaptıkları için *s1.make_upper()* ve *s1.print()* işlev çağrıları çalışma zamanında gösterici hatasına neden olur.

Tabii bu arada *s1* nesnesi ömrünü tamamladığında *s1* nesnesi için de sonlandırıcı işlev çağrılır ve bu işlev de daha önce geri verilen dinamik alanı tekrar *free store*"a geri vermeye çalışır. Bu durum da bir başka çalışma zamanı hatasıdır*(undefined behavior)*. Tanımlanan *String* sınıfı için benzer hatalar farklı görüntülerle de karşımıza çıkabilirdi:

```
void func(String s)
{
    //…
}

int main()
{
    String name("Necati");
    func(name);

    name.print();  //Çalışma zamanı hatası

    //…
    return 0;
}
```

main işlevi içinde *String* sınıfı türünden *name* isimli bir nesne tanımlanıyor. Daha sonra *func* isimli işleve *name* nesnesi argüman olarak gönderiliyor. Çağrılan *func* işlevinin parametre değişkeni *String* sınıfı türündendir. Bu durumda işlevin çağrılmasıyla birlikte, parametre değişkeni olan *s* nesnesi için kopyalayan kurucu işlev çağrılır. *func* işlevinin ana bloğunun sonuna gelindiğinde bu kez *s* nesnesi için sonlandırıcı işlev çağrılır.

Nesnenin *m_p* elemanının gösterdiği alan *free store*"a geri verilir. Oysa işlev çağrısından sonra *name* nesnesinin ömrü halen devam etmektedir. *name* nesnesinin *print* üye işlevi çağrıldığında bu üye işlev artık geri verilen bir alana erişir. Yani bir gösterici hatasına neden olur.

Kopyalayan Kurucu İşlevin Yazılması

Programcı bir kopyalayan kurucu işlev yazarsa derleyici artık otomatik olarak bir işlev yazmaz. Çoğu zaman programcının yazdığı kopyalayan kurucu işlev, yaratılan nesneyi ilkdeğer(ilk değerini) veren nesnenin kaynağı ile doğrudan ilişkilendirmek yerine, yaratılan nesne için farklı bir kaynak yaratır. Böylece aynı

kaynağın birden fazla nesne tarafından paylaşılmasından doğan hatalar ortadan kaldırılır. Aşağıda, daha önce tanımlanmış *String* sınıfı için bir kopyalayan kurucu işlev bildiriliyor ve tanımlanıyor:

```
class String {

    //...
public:

    String(const String &);

    //...

};



String::String(const String &r)

{

    m_len = strlen(r.m_p) ;
    m_p = new char[m_len + 1];
    strcpy(m_p, r.m_p);

}
```

Yazılan kopyalayan kurucu işlevi inceleyelim:

İşlevin parametre değişkeni olan *r* referansı, yaratılan *String* nesnesine ilk değerini veren nesnenin yerine geçiyor. Önce yaratılan sınıf nesnesinin *m_len* isimli elemanına ilk değer veren nesnenin tuttuğu yazının uzunluğunun *1* fazlası atanıyor. Daha sonra *new[]* işleciyle *m_len + 1 byte* uzunluğunda bir blok dinamik olarak elde ediliyor. Dinamik bloğun başlangıç adresi yeni yaratılan nesnenin *m_p* isimli elemanında tutuluyor. Daha sonra standart *strcpy* işleviyle yeni bloğa, ilk değer veren nesnenin dinamik alanda tuttuğu yazı kopyalanıyor.

Yazılan kopyalayan kurucu işlevin kodunun çalışmasıyla, yaratılan nesne artık kendi dinamik alanını kullanıyor hale gelir. Bu nesne için sonlandırıcı işlev çağrıldığında da sonlandırıcı işlev içinde kullanılan *delete[]* işleci yalnızca nesne için ayrılmış olan bloğu geri verir. Kopyalayan kurucu işlev doğrudan sınıfın elemanlarının değerlerini birbirine aktarmamış, yaratılan nesne için yeni bir kaynak yaratmış ve ilk değer veren nesnenin kullandığı kaynağa ulaşıp, yeni elde edilen kaynağa kopyalamayı sağlamıştır. Böyle kopyalamaya *"derin kopyalama" (deep copy)* denir.

Kopyalayan Kurucu İşlevin Parametresi Referans Yerine Nesne Olabilir mi?

Yukarıda tanımlanan *String* sınıfına ilişkin kopyalayan kurucu işlevin aşağıdaki biçimde bildirildiğini düşünelim:

```
class String {
    //…
public:
    String(String);
    //…
};
```

String sınıfı türünden bir nesnenin bir işleve değerle gönderildiğini düşünelim:

```
void foo(String);
void func()
{
    String str("Necati");

    foo(str);
    //…
}
```

Bu durumda *foo* işlevinin parametre değişkeni olan nesnenin yaratılması için, kopyalayan kurucu işlev çağrılır. Ancak kopyalayan kurucu işlevin de parametresi *String* sınıfı türünden bir nesne olduğu için bu kez kopyalayan kurucu işlevin de parametresi olan *String* sınıfı türünden nesne için yine kopyalayan kurucu işlev çağrılır. Bu durumda özyinelemeli (*recursive)* olarak çağrılan kopyalayan kurucu işlev belirli bir süreden sonra yığın (*stack)* alanını tüketir.

Atama İşlecini Yükleyen İşlev

Bir sınıf nesnesine aynı türden bir sınıf nesnesi atandığı zaman da bir işlev otomatik olarak çağrılır. Çağrılan bu işlevi "atama işlevi" (*assignment operator function*) olarak isimlendireceğiz. Atama işlevi de özel bir işlevdir. Yani programcı bu işlevi tanımlamaz ise bu işlev de derleyici tarafından otomatik olarak yazılır. Derleyicinin otomatik olarak yazdığı atama işlevi, sınıfın statik olmayan bir üye işlevidir ve *inline* olarak bildirilir.

Örneğin *A* isimli bir sınıf için derleyicinin yazdığı atama işlevinin bildirimi aşağıdaki gibidir:

```
class A{
    //...
public:
    A &operator=(const A&);
    //...
};
```

Derleyicinin yazdığı atama işlevi atama işlecinin sağ tarafındaki sınıf nesnesinin elemanlarını atama işlecinin sol tarafındaki nesnenin elemanlarına karşılıklı olarak atar. İşlev atama işlecinin sol tarafındaki sınıf nesnesini referans yoluyla geri döndürür.

a1 ve *a2 A* sınıfı türünden nesneler olmak üzere

```
a1 = a2;
```

gibi bir atama derleyici tarafından *A* sınıfının atama işlevine yapılan bir çağrıya dönüştürülür:

```
a1.operator=(a2);
```

Çağrılan üye işleve *this* adresi olarak atama işlecinin solundaki nesnenin adresi yani *a1*

nesnesin adresi geçirilir.

Sınıf nesnelerinin karşılıklı elemanlarının birbirine atanmasının uygun olmadığı durumlarda derleyicinin yazdığı atama işlevi istenmez. Bu durumda programcı atamanın istediği gibi yapılmasını sağlamak için kendisi bir atama işlevi yazabilir. Programcının bir atama işleç işlevi yazması durumunda artık derleyici bu işlevi yazmaz. Bir sınıf için kopyalayan kurucu işlevin tanımlanması gerekiyorsa, çoğunlukla aynı sınıf için atama işlevinin de yazılması gerekir. Daha önce örnek olarak verilen *String* sınıfını düşünelim:

```
int main()
{
    String name1("Deniz");
```

```
    {
        String name2("Yusuf");
        name2 = name1;
    }
    name1.print();
    //...
}
```

Yukarıdaki kodu inceleyelim:

main işlevinin ana bloğunun başında *String* sınıfı türünden *name1* isimli bir nesne tanımlanıyor. Çağrılan kurucu işlev *6* karakterlik bir bloğu dinamik olarak elde ederek bu bloğun başlangıç adresini, elemanı olan *m_p* isimli göstericide saklar. Bu bloğa *"Deniz"* yazısını kopyalar. *main* işlevi içinde yer alan iç blokta bu kez *name2* isimli bir nesne yaratılıyor. Bu nesne de benzer biçimde *"Yusuf"* yazısını tutar.

```
name2 = name1
```

ataması ile derleyicinin yazdığı atama işlevi çağrılıyor: Bu işlev, atama işlecinin terimi olan nesnelerin elemanlarını karşılıklı olarak birbirine atadığı için *name2* nesnesinin *m_p* elemanına *name1* nesnesinin *m_p* elemanının değeri atanır. Bu durumda her iki nesnesinin *m_p* elemanı da aynı dinamik bloğu yani *"Yusuf"* yazısının bulunduğu dinamik bloğu gösterir. Bu durumda artık *"Deniz"* yazısının bulunduğu blokla ilişki kaybedilmiş olur. Ama asıl büyük hata içsel bloğun sonlanmasıyla oluşur.

İçsel bloğun sonunda *name2* nesnesi için sonlandırıcı işlev çağrılır. Çağrılan sonlandırıcı işlevin nesnenin *m_p* elemanının gösterdiği bloğu *free store*'a geri verdiğini biliyorsunuz. Oysa içsel bloğun kapanmasından sonra *name1* nesnesi halen hayattadır ama bu nesnenin *m_p* göstericisi artık geri verilmiş olan bir bloğu gösterir.

```
name1.print();
```

Çağrısıyla, geri verilen bir dinamik bloğa erişilir. Bu da bir gösterici hatasıdır. Bu kadar hatanın üzerine son hata da *name1* nesnenin ömrü sona erdiğinde oluşur. Bu kez *name1* nesnesi için çağrılan sonlandırıcı işlev, daha önce geri verilen dinamik bloğu ikinci kez geri vermeye çalışır. Bu da tanımsız davranıştır.

Atama işlevinin Yazılması

Atama işleminde, atama işlecinin sağ tarafında yer alan nesnenin kullandığı kaynağın, işlecin sol tarafında yer alan nesne tarafından paylaşılması istenmiyorsa atama işleci için bir işlev yazılmalıdır. Yukarıda daha önce bildirilen *String* sınıfı için atama işlevi aşağıdaki gibi tanımlanabilir:

```
String &String::operator=(const String &r)
```

```
{
    if (this == &r)
        return *this;


    delete[]m_p;
    m_len = r.m_len;

    m_p = new char[m_len + 1];
    strcpy(m_p, r.m_p);

}
```

Yazılan işlevi inceleyelim. İşlevin girişinde yer alan *if* deyimine daha sonra değineceğiz. Atama işlevinin kopyalayan kurucu işlevden önemli bir farkı vardır. Kopyalayan kurucu işlev hangi sınıf nesnesi için çağrılıyorsa o sınıf nesnesi işlev çağrısından önce yoktur. Nesne kurucu işlevin çağrılmasıyla hayata başlar. Ancak atama işlevi söz konusu olduğunda durum farklıdır. İşlev hangi nesne için çağrılmışsa bu nesne zaten daha önce yaratılmıştır. Daha önce yaratılmış bu nesnenin kullanımı için zaten bir blok elde edilmiştir.

String sınıfı için yazılan atama işlevi önce nesne için ayrılan dinamik bloğu geri vermelidir:

```
delete []m_p;
```

deyimiyle bu sağlanır:

Daha sonra atama işlecinin sol tarafındaki nesne için yani **this* nesnesi için

```
m_p = new char[m_len + 1];
```

deyimiyle yeni bir dinamik blok elde ediliyor ve

```
strcpy(m_p, r.m_p);
```

çağrısıyla da, atama işlecinin sağ tarafındaki nesnenin dinamik bloğunda tutulan yazı, yeni elde edilen dinamik bloğa kopyalanıyor. İşlevin

```
*this
```

değeriyle geri döndüğünü görüyorsunuz. İşlev hangi sınıf nesnesi için çağrılmışsa o sınıf nesnesini referans yoluyla geri döndürür. Bir başka deyişle

```
name1 = name2
```

atamasının dönüştürüldüğü işlev çağrısı, *name1* nesnesini referans yoluyla geri döndürüyor. Peki buna neden gerek duyulmuş olabilir?

Doğal türler söz konusu olduğunda, atama işlecinin ürettiği değerin nesneye atanan değer olduğunu biliyorsunuz. Aynı özelliğin bir sınıf türünden nesne içinde sağlanması atama işlecini yükleyen işlevin, **this* nesnesini döndürmesi ile sağlanabilir. Aşağıdaki kodu inceleyin:

```
int main()
{
    String name1("Deniz");
    String name2("Yusuf");
    String name3("Huseyin");
    String name4("İbrahim");

    name1 = name2 = name3 = name4;
    name1.print();
    name2.print();
    name3.print();
    name4.print();

    return 0;
}
```

Atama işlecinin öncelik yönünün soldan sağa olduğunu biliyorsunuz.

```
name1 = name2 = name3 = name4;
```

deyimi, derleyici tarafından aşağıdaki gibi bir işlev çağrısı zincirine dönüştürülür:

```
name1.operator=(name2.operator=(name3.operator=(name4)));
```

Bir nesnenin kendisine atanmasının bir sözdizim hatası oluşturmadığını biliyorsunuz.

String sınıfı türünden bir nesnenin kendisine atandığını düşünelim:

```
int main()

{

    String name("Necati");
    name = name;

    //...

}
```

Yazmış olduğumuz atama işlevi çağrıldığında ne olur? Atama işlecinin sol terimi olan nesne için çağrılan atama işlevi, ilk önce *m_p* göstericisinin gösterdiği dinamik bloğu geri verir. Ancak atama işlecinin sağ terimi olan nesne de aynı nesne olduğundan,

```
strcpy( m_p, r.m_p);
```

ifadesi gösterici hatasına neden olur. Çünkü *r.m_p* nesnesi artık geri verilen bir bloğu gösteriyor durumundadır.

Hata nasıl giderilebilir? Atama işlecinin sol terimi olan nesne ile sağ tarafındaki nesnenin aynı olması durumunda hiç bir işlem yapılmamalıdır:

```
if (this == &r)
    return *this;
```

Yukarıdaki deyimi inceleyelim: Atama işlevi içinde kullanılan *this* göstericisi işlecin sol terimi olan sınıf nesnenin adresidir, değil mi? *r* referansı da atama işlecinin sağ terimi olan nesnenin yerine geçmiştir. O zaman *&r* ifadesi, atama işlecinin sağ terimi olan nesnenin adresidir. Adresleri aynı olan iki nesne aynı nesnedir. Eğer nesne kendine atanıyorsa işlev hiç bir işlem yapmadan **this* nesnesini referans yoluyla döndürür.

Atama işleç işlevinin kodu içinde nesnenin kendisine atanıp atanmadığının sınaması yapılmalı mıdır? Şöyle düşünebilirsiniz: Programcı durup dururken neden bir sınıf nesnesini kendine atasın?

Bu hata daha çok göstericiler, referanslar ve işlev çağrılarına ilişkindir.

```
*ptr = x;
```

Yukarıdaki atama deyiminde *ptr* bir sınıf türünden gösterici *x* de aynı sınıf türünden bir nesne olsun. Bu atama işleminden önce *ptr x* nesnesini gösteriyorsa, nesne kendisine atanmış olur.

Dönüştüren Kurucu İşlev

Sınıfın tek argümanla çağrılabilen kurucu işlevlerine *dönüştüren kurucu işlev* denir. Böyle bir kurucu işlev ya tek bir parametreye sahiptir, ya da diğer parametreleri varsayılan değerler alır.

```
class Account  {
    int acc_no;
    double balance;

public:

    Account(double);   //Dönüştüren kurucu işlev

    //...

};
```

Yukarıda tanımlanan *Account* sınıfının

```
Account(double) ;
```

biçiminde bildirilen kurucu işlev, dönüştüren kurucu işlevdir. İşlev tek parametre değişkenine sahiptir.

```
class Complex {
    double r, i;

public:

    Complex(double r, double i = 0);  //Dönüştüren kurucu işlevi

    //...

};
```

Yine yukarıda tanımlanan *Complex* sınıfının

```
Complex(double r, double i = 0);
```

biçiminde bildirilen kurucu işlev, dönüştüren kurucu işlevdir. İşlev tek bir argümanla çağrılabilir.

Bir sınıfın dönüştüren kurucu işlevi, başka türden bir ifadenin sınıf türünden geçici bir nesneye otomatik olarak dönüştürülmesini sağlar.

Örneğin yukarıda tanımlanan *Account* sınıfı için aşağıdaki ifadelerden hepsi geçerlidir:

```
void func(Account p)
{
    //...
}

int main()
{
    Account a1 = 200;
    a1 = 300.;

    func(200.);


    return 0;
}

Account foo()
{
    double x;

    //...
    return x;
}
```

main işlevinde yer alan birinci deyimle, *Account* türünden *a1* nesnesi *200.* değeriyle ilkdeğerini alarak yaratılıyor. Bu durumda *200* değeri dönüştüren kurucu işlev yoluyla *Account* sınıfı türünden bir geçici nesneye dönüştürülür. Geçici nesne de kopyalayan kurucu işlevle *a1* nesnesine ilkdeğerini verir. Yani derleyici açısından böyle bir deyim

```
Account a1 = Account(200.);
```

ya da

```
Account a1 = (Account) 200;
```

deyimlerine eşdeğer kabul edilebilir. C++ tür dönüştürme işlecinin işlevsel biçiminin de geçerli olduğunu anımsayalım.

C++ derleyicilerinin çoğu eniyileme (*optimizasyon*) amacıyla, önce bir geçici nesne yaratıp geçici nesneyle kopyalayan kurucu işlevi çağırmak yerine bu durumda doğrudan *a1* nesnesi için sınıfın kurucu işlevi çağırır. Bu durumu ileride yeniden inceleyeceğiz.

Gelelim ikinci deyime:

```
a1 = 300.;
```

C'den bildiğimiz gibi bir yapı nesnesine ancak o yapı türünden bir başka nesne atanabilir. Yani C dilinde başka bir türden ifade hiçbir zaman bir yapı türüne otomatik olarak dönüştürülmez. Ancak C++ da bir sınıf nesnesine başka türden bir ifade atandığını gören derleyici, atamanın mümkün olup olmadığını anlamak için bir dönüştürme kurucu işlevinin var olup olmadığını sorgular. Böyle bir kurucu işlev varsa otomatik tür dönüşümü yapılır. Böyle bir işlevin olmaması durumunda dönüşüm mümkün olmadığı için, derleme zamanında hata oluşur. Yukarıdaki atama deyimi derleyici tarafından aşağıdaki gibi ele alınır:

```
a1.operator=(Account(300.));
```

Yani önce *300* değeri dönüştürme kurucu işlevine argüman olarak geçilerek *Account* sınıfı türünden bir geçici nesne yaratılır. Yaratılan geçici nesnesin değeri *a1* nesnesi için çağrılan atama işleç işlevine argüman olarak geçilir.

```
func(200)
```

çağrısı için de benzer durum söz konusudur. *func* işlevine gönderilen argüman olan *200.* ifadesi önce dönüştürme kurucu işleviyle *Account* sınıfı türünden bir geçici nesneye dönüştürülür. *func* işlevinin parametre değişkeni bu geçici nesne ile kopyalayan kurucu işlev aracılığıyla ilkdeğerini alır. Yani derleyici açısından kodun anlamı

```
func(Account(200));
```

gibi bir deyime eşdeğerdir.

Şimdi de *foo* işlevinin kodunu inceleyelim. *foo* işlevinin *return* ifadesi, işlevin geri dönüş değerini içinde taşıyacak geçici nesneye ilkdeğerini verir. *return* ifadesi olan *double* türden *x,* otomatik olarak *Account* sınıfı

türünden bir geçici nesnenin yaratılmasına neden olur. İşlevin geri dönüş değerini içinde taşıyacak geçici nesne kopyalayan kurucu işlev aracılığıyla ilkdeğerini bu geçici nesneden ilk değerini alır. Bu durumlarda yine derleyici eniyileme amacıyla daha kısa kodlar üretebilir, ama şimdilik bu konuya girmeyeceğiz.

Yukarıdaki tür dönüşümleri otomatik olarak derleyici tarafından yapılabileceği gibi programcı tarafından tür dönüştürme işleçleri kullanılarak da yapılabilirdi. Şimdi de *main* işlevini tür dönüştürme işleçlerini kullanacak biçimde değiştirelim:

```
int main()

{
    Account a1 = Account(200);     // Account a1 = (Account)200;
    a1 = Account(300.);            //  a1 = (Account)300.;

    func(Account(500.));           // (Account)500.;


    return 0;
}
```

Açıklama satırlarında tür dönüştürme işleçleri C biçimiyle kullanıldı.

İlerdeki derslerimizde C'de olmayan C++'ın yeni tür dönüştürme işleçlerini öğreneceğiz. Yukarıdaki tür dönüşümleri için C++'ın yeni tür dönüştürme işleci olan *static_cast* tür dönüştürme işlecinin de kullanılabileceğini şimdiden söyleyelim.

Otomatik ya da bilinçli dönüşümün gerçekleştirilebilmesi için dönüştürme kurucu işlevin parametresiyle, geçici sınıf nesnesine dönüştürülecek ifadenin türünün tamamen aynı olması gerekmez. Dönüştüren kurucu işlevinin çağrılmasından önce doğal veri türlerine ilişkin otomatik dönüşümler yapılabilir. Yukarıda bildirilen *func* işlevinin aşağıdaki gibi çağrıldığını düşünelim:

```
func(10);
```

func işlevine argüman olarak *int* türden bir ifade gönderiliyor. *func* işlevinin parametresi

Account sınıfı türündendir. *Account* sınıfının

```
Account(double);
```

işlevinin çağrılabilmesi için önce *int* türden *10* değeri otomatik tür dönüşümü ile *double* türüne, daha sonra da dönüştüren kurucu işlev ile *Account* sınıfı türünden geçici bir nesneye dönüştürülür. Yani derleyici için

```
func(10)
```

gibi bir çağrı

```
func(Account(double(10))
```

çağrısına eşdeğerdir.

Sınıf türünden olmayan bir ifadenin otomatik olarak bir sınıf türüne dönüştürülmesi her zaman istenmeyebilir. Bazı durumlarda böyle bir dönüşüm program yazarken oluşan bir kodlama hatasının derleyici tarafından bulunmasına engel olur. Aşağıdaki koda inceleyin:

```
int main()
{
    int a;
    Account ac(400.);
    func(a);
    //...

    return 0;
}
```

İşlevi yazan programcının *func* işlevine argüman olarak *Account* sınıf nesnesi *ac* yi göndermek yerine, yanlışlıkla *int* türden olan *a* nesnesinin değerini gönderdiğini düşünelim. Derleyici için

```
func(a)
```

çağrısı tamamen geçerlidir. *a* ifadesi önce *double* türüne, daha sonra da *double* türünden yukarıda açıkladığımız biçimde *Account* sınıfına dönüştürülerek *func* işlevine argüman olarak gönderilir. Yani derleyici tarafından işlev çağrı ifadesi

```
func(Account(double(a)));
```

biçiminde ele alınır.

Sınıf türünden olmayan bir ifadenin sınıf türüne dönüştürme kurucu işleviyle dönüştürülmesi istenmiyorsa bu durum derleyiciye *explicit* anahtar sözcüğüyle bildirilebilir.

explicit Anahtar Sözcüğü

Bir sınıfın kurucu işlevinin bildiriminde *explicit* anahtar sözcüğü kullanılabilir. *explicit* anahtar sözcüğü yalnızca işlevin bildiriminde yazılır, kurucu işlevinin sınıf dışında tanımlanması durumunda işlev tanımında yer almaz. *explicit* anahtar sözcüğü ile bildirilen bir kurucu işlev, otomatik tür dönüştürme amacıyla kullanılamayan bir dönüştürme kurucu işlevidir. Kurucu işlevin *explicit* anahtar sözcüğüyle bildirilmesi durumunda, halen açık bir biçimde yani tür dönüştürme işleçleri kullanılarak dönüşüm gerçekleştirilebilir ancak otomatik dönüşüme derleyicinin hata mekanizmasıyla engel olunur. Yukarıdaki örneğimizi aşağıdaki biçimiyle yeniden derlemeye çalışın:

```
class Account  {
    int acc_no;
    double balance;

public:

    explicit Account(double); //explicit Dönüştüren kurucu işlev

    //...

};


void func(Account p)

{

    //...

}


int main()

{

    Account a1 = 200; //Geçersiz. otomatik dönüşüm yapılamaz!
    a1  =  300.;  //Geçersiz.  otomatik  dönüşüm  yapılamaz!
    func(200.); //Geçersiz. otomatik dönüşüm yapılamaz!



    return 0;

}


Account foo()
```

```
{
    double x;
    //...
    return x;          //Geçersiz. otomatik dönüşüm yapılamaz!
}
```

Arkadaşlik Bildirimleri

Sınıfın üye işlevi olmayan herhangi bir işleve, o sınıfın *private* ve *protected* bölümlerine erişim hakkı verilebilir. Bir işlev bir sınıfın arkadaş işlevi olarak bildirilebilir. Bir işlevi bir sınıfın arkadaş işlevi olarak bildirmek için işlev bildiriminin önüne *friend* anahtar sözcüğü yazılır. Aşağıdaki sınıf bildirimini inceleyin:

```
class Account {
    int account_no;
    double balance;
public;
    //...
    friend double balance_dif(const Account &, const Account &);
    //...
};
```

Bu örnekte *balance_dif* global bir işlevdir. Ancak *Account* sınıfı içinde *friend* anahtar sözcüğüyle arkadaş işlev olarak bildiriliyor. Arkadaş işlevler yalnızca erişim bakımından ayrıcalıklı olan işlevlerdir. Bir arkadaş işlev içinde arkadaş olunan sınıfa ilişkin bir nesne, gösterici ya da referans tanımlanırsa, o nesne gösterici ya da referans yoluyla sınıfın her bölümüne erişilebilir. Yukarıdaki örnekte *balance_dif* işlevi *Account* sınıfının bir arkadaş işlevi olarak bildirilmiştir. Bu yüzden *balance_dif* işlevi içinde tanımlanan *Account* sınıfına ilişkin bir sınıf nesnesi ile sınıfın her bölümüne erişilebilir.

```
class A {
    int x, int y;
public:
    A(int, int);
    friend void func();
};
```

Yukarıdaki örneği inceleyelim. *A* sınıfının *x* ve *y* isminde iki *private* elemanı var. *func* isimli global işlev sınıfın arkadaş işlevi olarak bildiriliyor. *func* global bir işlev olmasına karşın *func* işlevi içinde tanımlanan *A* sınıfı türünden nesnelerin *private* elemanları için erişim sınırlaması ortadan kalkar:

```
void func()
{
    A a;
    a.x = 10;   // private bölüme erişiliyor ama geçerli
    a.y = 20;   // private bölüme erişiliyor ama geçerli
    //...
}
```

Eğer *func* normal bir işlev olsaydı, yani *A* sınıfının *arkadaş* işlevi olmasaydı *a.x* ve *a.y* erişimleri geçersiz olurdu. *friend* anahtar sözcüğünün yalnızca işlevin bildiriminde kullanılmalıdır. İşlevin tanımında *friend* anahtar sözcüğünün yazılması bir sözdizim hatasıdır.

Arkadaş işlevler sınıfın üye işlevleri değildir. Arkadaş işlevlerin diğer global işlevlerden tek farkı erişim ayrıcalığına sahip olmalarıdır. Arkadaş işlevlere *this* göstericisi geçirilmez. Bu nedenle sınıfın elemanlarına arkadaş işlevler içinde doğrudan erişilemez.

Uygulamalarda genellikle arkadaş işlevlerin parametre değişkeni arkadaş olunan sınıfa ilişkin bir gösterici ya da referans olur. İlk örnekteki *Account* sınıfına *balance_dif* işlevini inceleyelim:

```
double balance_dif(const Account &r1, const Account &r2)
{
    return r1.balance - r2.balance;
}
```

İşlevlerin parametre değişkenleri işlev sınırları içindedir. Yani arkadaş bir işlevin parametre değişkeni arkadaş olunan sınıfa ilişkin bir nesne, bir gösterici ya da bir referans ise bu nesne, gösterici ya da referans ile sınıfın her bölümüne erişilebilir. Yukarıdaki *balance_dif* işlevinin tanımında, işlevin parametre değişkeni olan *r1* ve *r2* referansları nokta işleci ile kullanılarak ismi *balance* olan *private* elemana erişiliyor. Şüphesiz *balance_dif* işlevi *Account* sınıfı içinde *friend* işlev olarak bildirildikten sonra, aşağıdaki biçimlerde de tanımlanabilirdi:

```
double balance_dif(const Account a1, const Account a2)
{
```

```
    return a1.balance - a2.balance;
}
```

ya da

```
double balance_dif(const Account *p1, const Account *p2)
{
    return p1->balance - p2->balance;
}
```

Bir işlev, birden fazla sınıfın arkadaş işlevi olabilir. Arkadaş işlev bildiriminin sınıfın hangi bölümünde yapıldığının hiçbir önemi yoktur. Yukarıdaki örnekte, *Account* sınıfında *balance_dif* işlevi *public* bölümde arkadaş olarak bildirildi. Ancak *private* ya da *protected* bölümde bildirilseydi de bir farklılık oluşmazdı.

Yalnızca global işlevler değil, başka sınıfların üye işlevleri de arkadaş işlev olabilir. Örneğin:

```
class  Myclass {
    //…
    void myfunc();
};

class Herclass{
    //…
    friend void Myclass::myfunc();
};
```

Yukarıdaki örnekte *Herclass* sınıfın, *Myclass* sınıfının *myfunc* isimli işlevini arkadaş olarak bildiriyor.

Arkadaş işlev bildirimi aynı zamanda işlevin bildirimi yerine de geçer. Bu durumda işlev bildiriminin bilinirlik alanı sınıf bildiriminin yapılış yeriyle belirlenir. Örneğin:

```
int main()
{
    class X {
```

```
        //…
        friend void f();
    };
    f();        // Geçerli
    // …
}
void func()
{
    //…
    f();        // Geçersiz! İşlev bildirimi bu blokta tanınamaz!
}
```

Burada *f* arkadaş işlevinin sınıf içindeki arkadaşlık bildirimi aynı zamanda işlevin prototip bildirimi olarak da kabul edilir. Ancak bu bildiriminin bilinirlik alanı *X* sınıfının bilinirlik alanı ile aynıdır. Yani bu bildirim yalnızca *main* işlevi içinde bilinir. Bu yüzden *func* işlevi içinde *f* işlevi çağrıldığında, daha önce bildirimi yapılmamış bir işlev çağrılmış olur.

Arkadaş İşlevler Ne Zaman Kullanılmalıdır

Arkadaş işlevler erişim bakımından ayrıcalıklı işlevler olduğundan sınıfın *private* elemanlarının korunmasını azaltır. Arkadaş işlevler sınıfın her bölümüne erişebildiğine göre, sınıfın *private* elemanlara yönelik bir değişiklik yapıldığında yalnızca üye işlevleri değil aynı zamanda arkadaş işlevleri de yeniden yazmak gerekir. Arkadaş işlevleri fazlaca kullanmak sınıfın *private* bölümünün önemini azaltmayı kabul etmek anlamına gelir.

Arkadaş işlevler karmaşık durumlarda tasarımı ve işlemleri kolaylaştırmak için kullanılmalıdır. Kodun daha kolay yazılmasını sağlamak için bazı işlevlere gereksiz bir şekilde arkadaşlık vermek iyi bir fikir değildir. Arkadaş işlevlerin kullanılmasını gerekli duruma getirebilecek en tipik örnek işleçleri yükleyen global işlevlerinin yazımıdır. Böyle işlevleri ileride ele alacağız.

Arkadaş Sınıflar

Bir başka sınıfın bir üye işlevine arkadaşlık verilebileceğini belirtmiştik. Bir sınıf bir başka sınıfın tüm üye işlevlerine arkadaşlık verebilir. *Arkadaş sınıf* bir sınıfın tüm üye işlevlerinin başka bir sınıfın arkadaş işlevi olması durumudur. Bir sınıfı arkadaş sınıf yapmak için, arkadaş olunan sınıf içinde,

```
friend class <sınıf ismi>;
```

bildirimini yapmak gerekir. Bu bildirimin sınıfın hangi bölümünde yapılmış olduğunun bir önemi yoktur. Bu durumda arkadaş olarak belirtilen sınıfın tüm üye işlevleri içeri ilgili sınıf türünden bir nesne, gösterici ya da referans yoluyla sınıfın her bölümüne erişilebilir.

Aşağıdaki örneği inceleyin:

```
class Node {

    int val;

    Node *next;

    friend class LList;

};


class LList {
public:

    void add(int val);
    void delete(size_t n);
    Node *get_head();

    // …
private:

    Node *phead;
    Node *pnode;
    size_t size;

};
```

Yukarıdaki örnekte *LList* sınıfı *Node* isimli sınıfın arkadaş sınıfıdır. *LList* sınıfının tüm üye işlevleri içinde *Node* sınıfına ilişkin nesne, gösterici ya da referans yoluyla, *Node* sınıfının her bölümüne erişilebilir. *Node* sınıfının elemanlarının *private* bölümde olduğuna ve herhangi bir işlev içinde *Node* sınıfı türünden nesne tanımlanabileceği halde, o nesne yardımıyla sınıfın *private* elemanlarına erişilemeyeceğine dikkat edin.

Arkadaş sınıflar da arkadaş işlevlerde olduğu gibi erişim kolaylığı sağlamak için kullanılabilir. Tabii arkadaş sınıfların da, *private* elemanlarının korunmasını azaltacağını söylemeliyiz.

Arkadaşlık Bildirimi Çift Yönlü Değildir

Bir sınıfın bir başka sınıfa arkadaşlık vermesi, arkadaşlık verilen sınıftan da bir arkadaşlık alındığı sonucunu doğurmaz. Aşağıdaki örneği inceleyin:

```
class A{
public:
```

```
    friend class B;
        void func();

};


class B{

    int b;

    //...

};


void A::func()

{

    B object;

    object.b = 1;     //Geçersiz!

}
```

Yukarıdaki örnekte *A* sınıfı *B* sınıfına arkadaşlık veriyor. Bu *B* sınıfının da *A* sınıfına arkadaşlık verdiği sonucunu doğurmaz. *A* sınıfının üye işlevi olan *func* işlevi içinde tanımlanan *B* sınıfı türünden *object* isimli nesnenin *private* elemanına ulaşma girişimi geçersizdir, derleme zamanında hata oluşturur.

Arkadaşımın Arkadaşı Benim de Arkadaşımdır

Bu cümle arkadaşlık bildirimleri söz konusu olduğunda doğru değildir. Arkadaşlık bildirimlerinde geçişme özelliği söz konusu değildir. Aşağıdaki örneği inceleyin:

```
class A{

    int a;
public:

    friend class B;

};
```

```
class B{

    friend class C;

    //...

};


class C{
public:

    void func();

};


void C::func()

{

    A object;

    object.a = 1;     //Geçersiz

}
```

Yukarıdaki örnekte *A* sınıfı *B* sınıfına, *B* sınıfı da *C* sınıfına arkadaşlık veriyor. Bu durumdan *A* sınıfının *C* sınıfına arkadaşlık verdiği gibi bir sonuç çıkmaz. *C* sınıfının *func* isimli üye işlevi içinde *A* sınıfının *private* kısmına erişme çabası derleme zamanında hata ile sonuçlanır.

Sınıfın Statik Elemanları ve İşlevleri

Öncelikle *static* anahtar sözcüğünün yerel ve global değişkenlerle kullanılması ile ilgili bazı noktaları hatırlatalım. *static* anahtar sözcüğü yerel değişkenlerle kullanıldığında tanımlanan değişkenin ömrü *(storage duration*) üzerinde belirleyici olur. Yani *static* yerel bir değişken tanımlandığı bloğun başında yaratılıp, blok sonlandığında bellekten boşaltılmaz; global değişkenlerde olduğu gibi programın yüklenmesiyle yaratılır; program sonlanana kadar bellekte kalır. *static* yerel değişkenler blok bilinirlik alanına (*block scope*) uyan statik ömürlü değişkenlerdir ve çoğu zaman global değişkenlere seçenek olarak kullanılırlar.

static anahtar sözcüğü ile tanımlanan global değişkenler, iç bağlantıya (*internal linkage*) aittir. Yani *static* global değişkenler başka bir modülde *extern* bildirimi yapılsa bile kullanılamaz. Bir işlev de bu anlamıyla "*static"* olarak tanımlanabilir. *static* işlevler başka modüllerden çağrılamaz.

C++ dilinde bir sınıfın elemanları ve işlevleri de *static* anahtar sözcüğüyle bildirilebilir. Şimdi *static* anahtar sözcüğünün sınıflarla ilgili kullanımını inceleyelim:

Sınıfın Statik Elemanları

Bazı durumlarda bir sınıf türünden tanımlanan tüm nesnelerin global bir nesneye erişmesi gerekir. Örnek olarak, bir sınıf türünden kaç nesnenin yaratıldığı değerini tutan bir global değişkene gereksinim duyulması durumunu düşünebilirsiniz. Ya da yine global bir gösterici değişkenin, tüm sınıf için ayrılmış olan dinamik bir bloğun başlangıç adresini tuttuğunu ve tüm sınıf nesnelerinin böylelikle bu dinamik alana eriştiğini düşünebilirsiniz. Bu bilgilerin sınıfın bir elemanı olarak tutulması durumunda, hem sınıf nesneleri gereksiz bir biçimde büyür, hem de bu değerlerin değiştirilmesi durumunda değişikliğin var olan tüm sınıf nesneler için ayrı

ayrı yapılması gerekir. Bu bilgilerin global değişkenlerde tutulması durumunda ise, dışarıyı ilgilendirmeyen yalnızca söz konusu sınıfa ilişkin bir kodlama detayı dışarıya sızdırılmış olur. Global değişkenin bir başka sakıncası da, bunlara yalnızca sınıfın ulaşmasını sağlayıp, sınıf dışında erişimi engellemenin mümkün olmayışıdır. Yani bir global değişken bir sınıf için *public* ya da *private* olamaz. Yine başka bir sakınca da, bu amaçla kullanılacak bir global değişkenin global isim alanını gereksiz bir şekilde kirletmesidir.

Sınıfın bir elemanı sınıf bildirimi içinde başına *static* anahtar sözcüğü getirilerek bildirilirse bir çeşit sınıfa özgü global değişken gibi ele alınır. Böyle bir elemana sınıfın *static* elemanı *denir.* Sınıfın *static* elemanı ile global değişkenler arasında amaç koda yazım biçimi bakımından hiçbir fark yoktur. Yalnızca, sınıfın *static* elemanları sınıf isimleri ile kombine edilerek amaç kod içinde yazılır. *static* elemanlar sınıf nesnesi içinde bir yer kaplamaz.

Dışarıda global değişkenler gibi saklanır. Yalnızca mantıksal açıdan sınıfla ilişkilendirilirler. Sınıfın *static* elemanları, global değişkenlerde olduğu gibi dışarıda ayrıca tanımlanmak zorundadır. Tanımlama işlemi, bildirim sınıfın hangi bölümünde yapılmış olursa olsun gereklidir ve geçerlidir. *static* elemanlarının tanımlanması aşağıdaki gibi yapılır:

```
<tür> <sınıf ismi> :: <değişken_ismi> [ = ilkdeğer];
```

Genel biçimden de görüldüğü gibi *static* elemanları sınıf ismi ile birlikte belirtilerek tanımlanır. Örneğin *X* sınıfının *int* türden *s* isimli bir *static* elemanı şöyle tanımlanır:

```
int X::s;
```

Sınıfın statik elemanları, yaratılan sınıf nesnesinin içinde yer almaz. Yani statik elemanlar sınıf nesnesinin *sizeof* değerini büyütmez. Sınıfın statik bir elemanından yalnızca bir tane bulunur. Statik elemanlar aslında global değişkendir. Yalnızca mantıksal bakımdan sınıf ile ilişkilendirilmiştir.

Aşağıdaki örneği inceleyin:

```
class Person {
private:

    char *name;
    int no;

public:

    Person(const char *, int);

    ~Person();

    void display() const;
public:

    static int count;

};
```

```
#include <iostream>
#include <cstring>


using namespace std;

int Person::count = 0;


Person::Person(const char *nm, int n)

{

    name = new char[strlen(nm) + 1];
    strcpy(name, nm);

    no = n;

    ++count;

}


void Person::display()const

{

    cout << name << "   " << no;

}


Person::~Person()

{

    delete [] name;

}


int main()

{

    Person person("Burak Gencer", 120);
    cout <<  sizeof(person) << endl;


    return 0;

}
```

Yukarıdaki örnekte *Person* sınıfı türünden *person* nesnesinin *sizeof* işleci ile boyutu elde ediliyor. Elde edilen boyut yalnızca *sizeof(char *) + sizeof(int)* kadar olur. Sınıfın *static* elemanı olan *count* aslında global bir değişkendir. Bir sınıf nesnesinin tanımlanmasına gereksinim duyulmadan statik ömürlü olacak biçimde yaratılır. Bu yüzden sınıf nesnesi içinde yer kaplamaz. Yukarıdaki programda eğer *DOS* altında ve yakın

modellerde çalışılıyorsa 4, *DOS* altında ve uzak modellerde çalışıyorsanız *6, UNIX* ya da *Win32* sistemlerinde çalışıyorsanız *8* değerini elde edersiniz. Örneğimizde sınıfın *static* elemanının ayrıca global bir biçimde sınıf ismi belirtilerek aşağıdaki gibi tanımlandığını da görüyorsunuz.

```
int Person::count = 0;
```

Gördüğünüz gibi statik elemanlara tanımlama sırasında ilkdeğer de verilebilir. Ancak ilkdeğer verilmeyen doğal türlerden statik elemanlarda, global ve *static* yerel değişkenlerde olduğu gibi içlerinde sıfır bulunur. Sınıfın statik bir elemanı sınıf dışında tanımlanmaz ise *static* elemanın kullanılması durumunda derleme zamanında hata oluşmaz. Hata bağlama aşamasında oluşur.

Sınıfın statik elemanlarına dışarıdan ilgili sınıf türünden nesne, gösterici ya da referans yoluyla erişilebilir. Tabii statik elemanının *public* bölümde bildirilmiş olması gerekir.

Aşağıdaki *main* işlevini inceleyin:

```
int main()
{
    Person x("Ali Serçe", 25);
    cout <<  x.count;                    // 1
    Person y("Ahmet Altıntartı", 32);
    cout <<  y.count;                    // 2

    return 0;
}
```

Burada *x.count* ile erişilen *count* değişkeni ile *y.count* ile erişilen *count* değişkeni aynı değişkenlerdir. Sınıfın statik bir elemanına hangi sınıf nesnesi ile erişildiğinin bir önemi yoktur. Bu yüzden statik elemanlara sınıf nesnesi olmadan da sınıf ismi ve çözünürlük işleci ile erişilebilir. Böyle bir erişimin geçerli olması için bildirimin *public* bölümde yapılmış olması gerekir. Aşağıdaki *main* işlevini inceleyin:

```
int main()
{
    Person x("Ali Serçe");
    cout<<  Person::count << endl;    // 1
```

```
    Person y("Ahmet Altıntartı");

    cout<<  Person::count << endl;     // 2


    return 0;

}
```

Örnekte erişim *Person::count* biçiminde hiç sınıf nesnesi kullanılmadan yapılıyor. *count* isimli eleman, sınıfın *public* bölümünde bildirildiği çin erişim geçerlidir. *static* elemanlara üye işlevler içinde normal elemanlar gibi doğrudan erişilebilir. Örneğin *Person* sınıfının kurucu işlevi içinde erişim bu biçimde sağlanıyor:

```
Person::Person(const char *nm, int n)

{

    name = new char[strlen(nm) + 1];
    strcpy(name, nm);

    no = n;

    ++count;

}
```

Sınıfın statik elemanı bir dizi ya da gösterici olabilir. Eğer dizi ise bildirim sırasında dizinin boyutu belirtilmeyebilir. Ancak dizinin tanımlaması sırasında dizinin boyutu belirtilmelidir. Sınıfa ilişkin statik bir diziye tanımlama sırasında küme ayraçları içinde ilkdeğer de verilebilir. Bu durumda dizi boyutu belirtilmeyebilir. Örneğin:

```
class Date {
private:

    int day, month, year;

    static const char *week_days[ ];
public:

    // …

};

const char *Date::week_days[] = {"Pazartesi", "Salı", "Çarşamba",
"Perşembe", "Cuma", "Cumartesi", "Pazar"};
```

Sınıfın statik elemanları yalnızca bir kez tanımlanmalıdır. Bunun için tanımlama işlemi bir

.cpp dosyası içinde yapılmalıdır. Eğer tanımlama *.h* dosyası içinde yapılır, bu dosya da proje içinde birden fazla yerde eklenirse bağlama zamanında hata oluşur. Sınıfın *statik* bir elemanını kullanmak için o sınıf türünden nesne tanımlamaya gerek yoktur. Yani sınıf türünden hiçbir nesne tanımlanmasa bile sınıfın *statik* elemanları yine de kullanılabilir.

statik Elemanlar Ne Zaman Kullanılmalı

Sınıfın *statik* elemanları aslında mantıksal olarak sınıfla ilişkilendirilmiş global değişkenlerdir. Global değişkenlere tüm işlevler içinde erişilebilirken, *statik* elemanlara yalnızca sınıfın üye işlevleri içinde doğrudan erişilebilir. Sınıfın *statik* elemanlarına ayrıca sınıf nesnesi, göstericisi ya da referansı ile ya da sınıf ismi ve çözünürlük işleci ile de dışarıdan erişilebilir. Bir global değişken yalnızca bir sınıf için anlamlı ise onun normal bir global değişken yerine *statik* elemanı olarak tanımlanması, algısal bakımdan bilinirlik alanını daraltır. Böylece kodu inceleyen kişi değişkenin kod ile ilgisini daha iyi anlamlandırır.

Date sınıfı için verilen örnekteki *01/01/1900*"den geçen gün sayısına ilişkin dönüşümleri yapan *totaldays* ve *revdays* isimli üye işlevleri anımsayın. Bu işlevler ayların kaç çektikleri bilgisini *mon_days* isimli bir global diziden elde ediyordu. *mon_days* isimli dizi global olmasına karşın yalnızca ayların kaç çektiklerinin belirlenmesi amacıyla kullanılmaktadır. Yani *mon_days* yalnızca *Date* sınıfının üye işlevlerinin kullandığı global bir dizidir. Eğer bu dizi sınıfın normal bir elemanı yapılsa, bütün *Date* türünden nesneler içinde gereksiz bir biçimde yer kaplardı, değil mi? İşte hem global olarak bırakmak hem de sınıfla ilişkilendirmek için *mon_days* dizisi statik eleman olarak tanımlandı. Ayrıca sınıfın *display_text* isimli işlevi, yeniden düzenlenerek, ay isimlerini de yazıyla yazdırabilir. Ay isimleri ve gün isimleri *char* türden statik gösterici dizilerine yerleştirilebilir. *Date* sınıfının biraz daha geliştirilmiş biçimini ileride vereceğiz.

Sınıfın statik elemanları da, sınıfın elemanlarıdır. Ancak bazı özellikler açısından, sınıfın statik elemanları diğer elemanlardan farklılık gösterir:

Bir sınıfın elemanı aynı sınıf türünden olamaz ama bir sınıfın aynı sınıf türünden statik bir elemanı olabilir. Aşağıdaki sınıf bildirimini inceleyin:

```
class A {

    static A sa;        //Geçerli

    A *mptr;            //Geçerli

    A ax;               //Geçersiz

};
```

Bir sınıfın statik bir elemanı, aynı sınıfın bir üye işlevinde varsayılan argüman olarak kullanılabilir. Ancak sınıfın statik olmayan bir elemanı bu biçimde kullanılmaz:

```
class A {

    int m_x;
```

```
    static int ms_y;
public:

    void func(int = m_x);              //Geçersiz

    void foo(int = ms_y);              //Geçerli

};
```

Sınıfın statik Üye İşlevleri

Sınıf bildirimi içinde bir işlev, başına *static* anahtar sözcüğü getirilerek bildirilebilir. Böyle işlevlerle sınıfın *statik* üye işlevleri denir. Sınıfın statik üye işlevleri global bir işlev gibidir. Ancak mantıksal bakımdan sınıfla ilişkilendirilmiştir. Bu işlevlere *this* göstericisi geçirilmez. Sınıfın *statik* üye işlevlerine *this* göstericisinin geçirilmemesi, o işlevler içinde sınıfın elemanlarına ve diğer üye işlevlere doğrudan erişilemeyeceği anlamına gelir.

Sınıfın statik üye işlevi her ne kadar global bir işlev gibiyse de, tanımlanması üye işlevlerde olduğu gibi sınıf ismi belirtilerek yapılır. *Date* sınıfında bazı değişiklikler yapıyoruz:

```
class Date {
private:

    int day, month, year;
    static int mon_days[12];
    static char *mon_tab[12];
    static char *day_tab[7];

public:

    enum Days {Sunday , Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday};

    Date();

    Date(int, int, int);
    void display() const;

    void display_text() const;
    long totaldays() const;
    void revdays(long);

    int weekday() const;

    static bool isleap(int);              //  static üye işlev

};
```

Önceki biçimde bir yılın artık yıl olup olmadığını bulan *isleap* isimli işlev normal bir global işlevdi. Ancak bu işlevin yalnızca tarih işlemleriyle anlamlı bir işlevi vardır. O halde mantıksal bakımdan *Date* sınıfı ile ilişkilendirilip algılama kuvvetlendirilmiştir. *isleap* işlevi dışarıda şöyle tanımlanabilir:

```
bool Date::isleap(int year)
{
    return year % 4  == 0 && year % 100 != 0 || year % 400 == 0;
}
```

static anahtar sözcüğünün yalnızca bildirim sırasında yazıldığına, tanımlama sırasında yazılmadığına dikkat edin. *isleap* içinde sınıfın *day*, *month* ve *year* elemanlarına doğrudan erişilemez. Sınıfın diğer üye işlevleri sınıf nesnesi olmadan doğrudan çağrılamaz.

Gördüğünüz gibi dışarıdaki sınıf ile mantıksal bakımdan ilişkili olan ama sınıfın elemanlarını ve üye işlevlerini kullanmayan global işlevler sınıfın statik işlevi yapılabilir. Sınıfın statik üye işlevi bir sınıf nesnesi, göstericisi ya da referansıyla çağrılabilir. Ya da eğer sınıfın *public* bölümündeyse çağrı doğrudan sınıf ismi belirtilerek *::* çözünürlük işleci ile de yapılabilir:

```
int main()
{
    Date x;
    if (x.isleap(1996))
        cout << "is a leap year" << endl;
    else
        cout << "is not a leap year" << endl;
    if (Date::isleap(2000))
        cout << "is a leap year" << endl;
    else
        cout << "is not a leap year" << endl;

    return 0;
}
```

Burada *x.isleap* ve *Date::isleap* biçiminde iki çağrı yapılıyor. *isleap* sınıfın *public* bölümünde olduğu için her iki çağrı biçimi de geçerlidir. Ancak yine de statik üye işlevlerin, sınıf nesnesi kullanmadan sınıf ismiyle çağrılması önerilir. Ne de olsa çağrıda kullanılan sınıf nesnesinin hiçbir özel anlamı yoktur.

Sınıfın statik işlevi, sınıfın bir üye işlevi içinde doğrudan çağrılabilir. Örneğin yukarıdaki programda *totaldays* işlevi içinde *isleap* işlevi doğrudan çağrılıyor.

Sınıfın *statik* bir üye işlevi içinde *statik* elemanları doğrudan kullanılabilir. Sınıfın statik elemanlarının sınıf nesnesi içinde yer kaplamadığını anımsayın. Bu durumda *statik* elemanlara erişmek için *this* göstericisi gerekmez. Aşağıda bu biçimde kullanıma ilişkin bir örnek veriliyor:

```
class Person {
private:

    char *name_;
    int no_;

    static int count_;
public:

    Person(const char *nm, int n);

    ~Person();

    void display() const;
    static int get_count();

};



#include <iostream>
#include <cstring>
using namespace std;


int Person::count_ = 0;

Person::Person(const char *nm, int n)

{

    name_ = new char[strlen(nm) + 1];
    strcpy(name_, nm);

    no_ = n;

    ++count_;

}


void Person::display() const

{

    cout << name_ << endl;
    cout << no_ << endl;

}


int Person::get_count()

{

    return count_;

}


Person::~Person()
```

```
{
    delete [] name_;
}


int main()
{
    Person x("Necati Ergin", 25);
    Person y("Haluk Yetis", 34);
    cout << Person::get_count();
    return 0;
}
```

Bu örnekte daha önce vermiş olduğumuz *Person* isimli sınıfta bazı değişiklikler yaptık. Örneğin statik *count_* elemanını sınıfın *private* bölümüne yerleştirdik. Bu elemanın değerinin elde edilmesi için *get_count* isimli *statik* bir üye işlev ekledik. *get_count* işlevinin yalnızca sınıfın statik elemanı olan *count_* ile ilişkili olduğuna dikkat edin.

Sınıfın *statik* üye işlevi ile aynı isimde ve parametre yapısına sahip global bir işlev bulunabilir. Çünkü sınıfın *statik* işlevleri sınıf ismi ile birleştirilerek amaç kod içine yazılır. Sınıfın kurucu ve sonlandırıcı işlevleri *statik* üye işlev olamaz. Bu işlevlerin yaratılan ve yok edilen nesne üzerinde doğrudan işlem yapmak zorunda olduğunu düşünürsek *static* üye işlev olmamaları gerektiği de açıktır.

Statik bir üye işlev *const* olarak da bildirilemez. Zira bir *const* üye işlev, *this* göstericisinin gösterdiği nesne *const* olan bir üye işlev anlamına gelir. *statik* üye işlevler *this* göstericisine sahip olmadıklarına göre, *const* da olamazlar.

Sınıfın *statik* üye işlevleri sonuçta sınıfa ilişkin bir üye işlev olduğu için, *statik* üye işlevleri içinde sınıfa ilişkin bir nesne tanımlandığında o nesne yoluyla dışarıdan sınıfın her bölümüne erişilebilir.

```
class Sample {
    int x;
public:
    static void func();
};


void Sample::func()
{
    Sample sam;
    sam.x = 20;     //Geçerli
    //...
}
```

Bazen de *static* üye işlevler özel amaçları gerçekleştirmek için belirli kod kalıplarında kullanılır. Örneğin bir sınıf türünden yalnızca dinamik nesnelerin yaratılmasına izin verilmesi istendiğini düşünelim. *DynamicOnly* bir sınıf olmak üzere

```
DynamicOnly d;
```

biçiminde bir nesne yaratılmasına engel olunsun. *DynamicOnly* sınıfına ilişkin tüm nesnelerin dinamik olarak, yani *new* işleciyle oluşturulması bir zorunluluk olsun. Bu nasıl sağlanabilir?

Sınıfın kurucu işlevleri sınıfın *private* bölümüne yerleştirilirse

```
DynamicOnly d;
```

biçiminde bir tanımlama geçersiz olur, değil mi? Ancak bu durumda *new* işleciyle de nesne yaratılamaz. Hatırlayacağınız gibi *new* işleciyle bir sınıf nesnesi yaratıldığında önce sınıf nesnesinin kaplayacağı yer kadar dinamik bir alan ayrılıyor, daha sonra bu alanın başlangıç adresi sınıfın kurucu işlevine *this* adresi olarak geçiriliyordu. Kurucu işlev sınıfın *private* bölümde olduğuna göre nesne yaratılması yine mümkün olmaz, değil mi?

Ancak sınıfın kurucu işlevi *private* bölümde ise, ve sınıfın bir üye işlevi, içinde sınıfın başka bir *private* bir üye işlevi çağrılabildiğine göre, *DynamicOnly* sınıfının bir üye işlevi içinde dinamik bir nesne yaratılabilir. Peki üye işlevi bir sınıf nesnesi ile çağırmak gerektiğine göre, böyle bir sınıf nesnesi nasıl elde edilecek? İşte bu noktada *static* bir üye işlev kullanılabilir: Dinamik bir sınıf nesnesinin adresi ile geri dönen statik bir üye işlev:

```
class DynamicOnly{

    DynamicOnly ();    //başlangıç işlevi private bölümde.
public:

    static DynamicOnly *create_object();

};


DynamicOnly * DynamicOnly::create_object()

{

    return new DynamicOnly;

    //Geçerli çünkü üye işlev içinde başka bir private işlev çağrılabilir.

}
```

```
#include <iostream>

using namespace std;


int main()

{

    // DynamicOnly d;  Geçersiz!

    // DynamicOnly *ptr = new DynamicOnly;        Geçersiz!
    DynamicOnly *ptr = DynamicOnly::create_object();

    //Geçerli çünkü create_object işlevi public bölümde
    delete ptr;    //Yaratılan dinamik nesne yok ediliyor.



    return 0;

}
```

Yukarıdaki örneğimizde *DynamicOnly* sınıfının kurucu işlevi *private* bölümde bildiriliyor. Böylece global bir işlev içinde *DynamicOnly* sınıfı türünden bir nesne yaratılması engellenmiş oluyor. Ancak bir üye işlev içinde nesne yaratılabilir. Sınıfın *public* bölümüne yerleştirilmiş olan *create_object* isimli *statik* üye işlev, *DynamicOnly* sınıfı türünden bir adresle geri dönüyor.

create_object işlevi *new* işleci ile yaratılmış bir nesnenin adresi ile geri dönüyor. İşlevimiz üye işlev statüsünde olduğu için, işlevimiz içinde *DynamicOnly* sınıfının *private* kurucu işlevinin çağrılması geçerlidir.

Şimdi de *main* işlevine bir göz atalım. *main* işlevi içinde, *statik* üye işlev olan

create_object işlevi

```
DynamicOnly::create_object()
```

biçiminde çağrılabilir. Böylece işlevin geri dönüş değeri olan dinamik nesnenin adresi ile yaratılan dinamik nesne istenildiği biçimde kullanılabilir.

Sınıfların const static Elemanları

Sınıfın bir statik elemanı *const* anahtar sözcüğü ile bildirilebilir. Bu durumda sınıfa özgü değeri değişmeyecek bir eleman yaratılmış olur. Sınıfın bir *const static* elemanı eğer bir tamsayı türünden ise bu elemana sınıf içinde ilkdeğer verilebilir:

```
class Myclass {

    const static int size = 100;
    int a[size];
```

```
    //...
};
```

Bir değişmez ifadesiyle ilkdeğerini almış *const statik* eleman bir değişmez ifadesi olarak kullanılabilir. Böyle elemanlar yalnızca bir sınıfı ilgilendiren simgesel değişmezler olarak görev yapabilir.

İŞLEÇ YÜKLEMESİ

C dilinde iki yapı nesnesi birbiriyle aritmetik işlemlere sokulamaz, karşılaştırma işleçleriyle yapı değişkenleri arasında karşılaştırma işlemi yapılamaz. C"de bir yapı nesnesi yalnızca üç işlecin terimi olabilir:

Nokta işleciyle bir yapı nesnesinin elemanına erişilebilir.

Adres işleciyle bir yapı nesnesinin adresi alınabilir. Bu durumda elde edilen adres yapı türünden bir adrestir.

Yapı nesnesi *sizeof* işlecinin terimi olabilir. Bu durmda üretilen değer söz konusu yapının bellekte kapladığı *byte* sayısıdır.

Aynı türden iki yapı değişkeni birbirine atanabilir.

Eğer yapı değişkenleri üzerinde çeşitli işlemlerin yapılması isteniyorsa işleme girecek yapı değişkenlerini argüman olarak alan özel işlevler yazmak gerekir. Örneğin *C*"de tarih işlemlerini gerçekleştirmek için, önce tarih bilgilerini bir yapı olarak belirlemek ardından da bu yapıya ilişkin bir grup işlev yazmak gerekir.

C++"da bir sınıf nesnesi bir işlecin terimi olabilir. Bu durumda derleyici, işlecin kullanıldığı ifadeyi bir işlev çağrısına dönüştürebilmek için, programcı tarafından tanımlanan uygun bir işlevin bulunup bulunmadığını araştırır. Derleyici böyle bir işlev bulursa, işlecin bulunduğu ifadeyi, bir işleve yapılan çağrıya dönüştürür. İşleç ifadesinin dönüştürüleceği uygun bir işlev yoksa, durum derleme zamanında hata olarak değerlendirilir.

Sınıfı tasarlayan programcı, tasarladığı sınıfa ilişkin nesnelerin belirli işleçlerin terimi olmasını istiyorsa, işlevler tanımlayarak bu amacını sağlayabilir. *C++*"ın var olan işleçlerine böylece tanımlanan sınıflara ilişkin ek anlamlar yüklenebilir. Bu araca "işleç yüklemesi" (*operator overloading*) denir.

İşleç yüklemesi, kullanıcı kodların işini kolaylaştırır, daha kolay ve daha iyi bir soyutlama yapmasını sağlar. İşleç yüklemesi ile, var olan işleçlerin verdiği güçlü çağrışımdan sınıf nesneleri için de faydalanılır. Sınıf nesneleri sanki doğal veri türlerinden nesnelermiş gibi işleçlerle birlikte kullanılabilir.

Bir sınıf nesnesi bir işlecin terimi olmuşsa aşağıdaki olasılıklar söz konusudur.

a) İfade derleyici tarafından bir sınıfın üye işlevine yapılan çağrıya dönüştürülür.
b) İfade global bir işleve yapılan çağrıya dönüştürülür.
c) İfade derleme zamanı hatası olarak değerlendirilir.

Bir işleç sınıfın bir üye işlevi tarafından yüklenebileceği global işleve de yüklenebilir.

İşleçleri Yükleyen Üye İşlevler

İşleçleri yükleyen üye işlevler sınıfın diğer üye işlevleri gibi bildirilip tanımlanır. Bu işlevlerin sınıf bildirimi içindeki bildirimi aşağıdaki gibi olmalıdır:

```
[geri dönüş değerinin türü] işleç <işleç simgesi> ([parametreler]);
```

Genel biçimde de görüldüğü gibi, bu işlevlerin bildirimi diğer üye işlevlerde olduğu gibi yapılır. İşleç yükleyen işlevler ile sınıfın diğer üye işlevleri arasındaki tek fark isimlendirmeye ilişkindir. İşleç yükleyen işlevin isimleri *operator* anahtar sözcüğü ile bir işleç simgesinden oluşur. Örneğin:

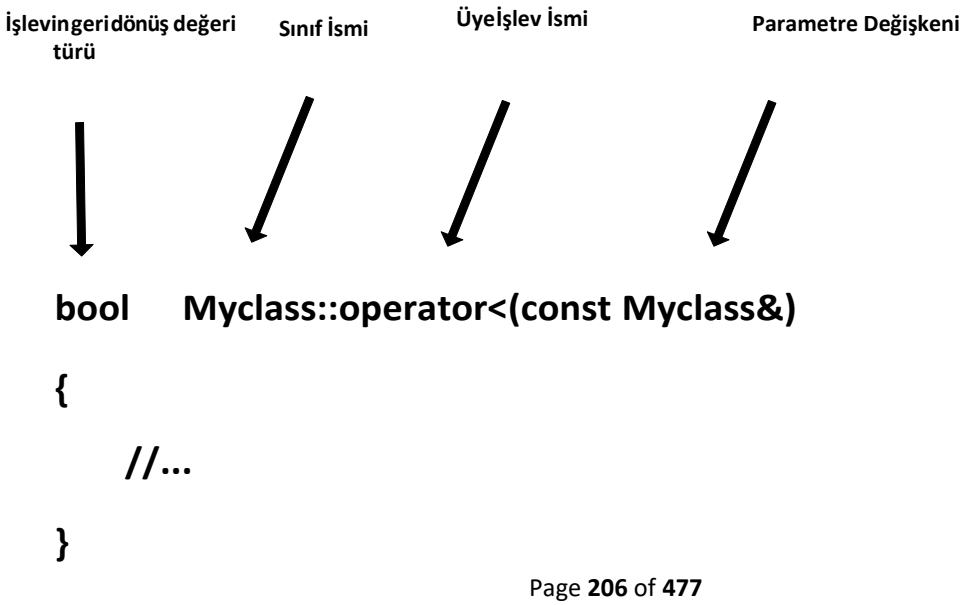
```
class Myclass{
private:

    int a;
public:

    bool operator<(int);

    //...

};
```

Yukarıdaki örnekte işlev ismi olarak *operator*< kullanılıyor. *operator, C++*'ın bir anahtar sözcüğüdür. Hem bildirimde hem de tanımlama sırasında kullanılması gerekir. *operator* ile > ayrı ayrı atomlar olduğu için aralarına istenildiği kadar boşluk karakteri koyulabileceği gibi bu iki atom bitişik de yazılabilir. İşleç simgesi ancak *C++*'ın işleçlerinden biri olabilir. *C++*'ın işleci olmayan bir simge için işleç yükleyen işlev yazılamaz. Örneğin aşağıdaki işlevin bildirimi geçerli değildir:

```
void operator $ (int);   //Geçersiz
```

Çünkü *C++*'da $ atomuyla belirtilen bir işleç yoktur.

İşleçleri yükleyen üye işlevlerin tanımlanması ve çağrılması diğer üye işlevler gibi yapılır. Örneğin yukarıda bildirilen *operator*> işlevi sınıfın dışında aşağıdaki gibi tanımlanabilir.



İşleçleri Yükleyen İşlevlerin Parametrik Yapısı

Bazı işleçler hariç olmak üzere işleçleri yükleyen işlevlerin geri dönüş değerleri herhangi bir türden olabilir. Ancak parametre değişkeni sayıları üzerinde bir koşul vardır. Bir sınıfın üyesi olan işleç yükleyici işlevler ya hiç parametre almaz ya da tek parametre alır. Tek terimli (*unary*) işleçlere ilişkin işlevlerin parametresiz olması, iki terimli (*binary*) işleçlere ilişkin işlevlerin ise tek parametre alması zorunludur. Aşağıdaki örneği inceleyin:

```
class Complex {
    double real, imag;

public:

    Complex(double, double);
    Complex operator/(double)const;
    bool operator!()const;

    //...

};
```

Complex isimli sınıfın *operator/* işlevinin tek bir parametre değişkeni var. İşlev *Complex* türüne geri dönüyor. Bölme işleci iki terimli olduğu için, üye işlev ile yüklendiğinde, işlevi tek parametresi olmalı. Örneğin bu işlevin sınıf içinde

```
Complex operator/(double, double)const;
```

biçiminde bildirilmesi geçersiz olurdu.

Complex sınıfının *operator!* işlevinin ise, parametre değişkeninin olmadığını bu işlevin *bool* türüne geri döndüğünü görüyorsunuz. Tek terimli mantıksal değil işleci üye işlev ile yüklendiğinde, işlevin parametre değişkeni olmamalıdır. Örneğin bu işlev

```
bool operator!(int)const;
```

biçiminde bildirilseydi, bildirim geçersiz olurdu.

Ancak bazı işleçler hem tek terimli hem de iki terimli olarak kullanılır. Örneğin * işleci çarpma işleci olarak kullanıldığında iki terimli iken, içerik işleci olarak kullanıldığında tek terimlidir. Bu işleç hem *binary* hem de *unary* işleç olarak yüklenebilir. Benzer durum +, -, & işleçleri için de geçerlidir. Bu işleçler de hem tek terimli hem de iki terimli işleç olarak yüklenebilir.

İşleçleri Yükleyen Üye İşlevlere Yapılan Çağrılar

İşleçleri yükleyen işlevler hem normal bir işlev gibi, hem de bir işlecin kullanılmasıyla kısa biçimde çağrılabilir. Zaten bu işlevlerin varlık nedeni kısa çağrının verdiği okuma ve yazma kolaylığından faydalanmaktır. *x* bir sınıf nesnesi, *y* herhangi bir türden nesne ya da değişmez olmak üzere,

```
x.operator <işleç simgesi> (y);
```

gibi bir çağrı biçimi ile,

```
x <işleç atomu> y
```

çağrı eşdeğerdir. Örneğin;

```
z = x + y;
```

ile

```
z.operator=(x.operator+(y));
```

aynı anlamdadır. Ya da örneğin,

```
x.operator!();
```

yerine kısaca

```
!x
```

yazılabilir. Şüphesiz bu kısa yazış biçimi daha kolay okunabilir. Kısa çağrı biçimi ile sanki sınıf nesneleri normal işleçlerle işleme sokuluyormuş gibi bir algı düzeyi elde edilmiş olur. Böylece sınıfı kullanan programcıların işini kolaylaşır.

object bir sınıf türünden bir nesne ve *right_operand* iki terimli bir işlecin sağ terimi olsun:

```
object + right_operand
```

biçimindeki bir ifade derleyici tarafından aşağıdaki gibi bir üye işlev çağrısına dönüştürülebilir:

```
object.operator+(right_operand);
```

Yukarıdaki örnekte toplama işlecinin solundaki sınıf nesnesi için, *object* nesnesinin ait olduğu sınıfın *operator+* üye işlevi çağrılıyor. Bir üye işlevin bir sınıf nesnesi ile çağrılması durumunda sınıf nesnesinin adresinin işleve gizlice *this* adresi olarak geçirildiğini biliyorsunuz. Toplama işleci için çağrılacak işlevin her iki nesne üzerinde işlem yapabilmesi için, iki nesneye de ulaşması gerekir, değil mi?

Soldaki sınıf nesnesinin adresi this adresi olarak işleve geçirildiğine göre, işlevin yalnızca sağdaki nesnenin değerini alması gerekir. İşte bu yüzden iki terimli bir işleç üye işlev olarak yüklendiğinde, bu işlevin tek bir parametresi olmalıdır.

```
!object
```

gibi bir ifade de üye işlev çağrısına dönüştürülebilir.

```
object.operator!();
```

Bu işlev zaten *object* nesnesinin adresini *this* adresi olarak alacağına göre ve *!* işlecinin başka da bir terimi olmadığına göre işlevin parametre değişkeninin olmaması gerekir.

Bir işleci yükleyen işlevi yazarken işlecin aldığı terim sayısı (*arity*) değiştirilemez. Örneğin

! işlecinin *Complex* isimli bir sınıf için üye işlev biçiminde yüklenmek istendiğini düşünelim. *!* işleci *C++* „ın tek terimli işleci olduğuna göre, bu işlecin görevini üslenecek işlevin parametre değişkeni olmamalıdır.

İşleçleri Yükleyen Global İşlevler

Bir sınıf nesnesi bir işlecin terimi olduğunda global bir işlevin çağrılması da sağlanabilir. Bu durumda çağrılan işlevlere ”işleç yükleyen gloabal işlev“ (*global operator function*) denir. *a* ve *b Myclass* isimli bir sınıfa ait nesneler olmak üzere *a + b* gibi bir ifade, uygun bir global işlevin tanımlanması durumunda

```
operator+(a, b)
```

gibi bir çağrıya dönüştürülebilir. Bu durumda böyle bir global işlevin iki parametre değişkeni olmalıdır. İşlevin bildirimi aşağıdaki gibi olabilir:

```
Myclass operator+ (const Myclass &, const Myclass &);
```

Bir işleci yükleyen gobal işlevin kaç parametre değişkeni olmalı? Global işlevler sınıf nesnelerinin adreslerini gizlice almadıklarına göre, işleç kaç terimli ise işlevin de o kadar parametresi olmalıdır. İki terimli bir işlecin sol teriminin bir sınıf nesnesi olmaması durumunda bir üye işlevin çağrılamayacağını biliyoruz.

```
Myclass m;
```

tanımlanmasından sonra *m + 5* gibi bir ifade için sınıfın *operator+* işlevi çağrılabilir. Ancak,

```
5 + m
```

gibi bir ifade için sınıfın *operator+* işlevinin çağrılması mümkün değildir. Ancak toplama işlecinin değişme özelliği olduğuna göre *Myclass* sınıfı için de bu durumun geçerli olması gerekir değil mi? *Myclass* sınıfı için global bir *operator+* işlevinin tanımlanmış olması durumunda yukarıdaki ifade aşağıdaki gibi bir çağrıya dönüştürülebilir:

```
operator+(5, m);
```

İşleçleri yükleyen işlevlere örnek verebilmek için aşağıda *Myint* isminde yeni bir sınıf tanımlıyoruz. *Myint* sınıfı türünden bir nesne *C++*'ın önceden tanımlanmış türü olan *int* türünün yerine kullanılabilecek. Sınıfı yalnızca işleç yükleyen işlevlere örnek vermek için tanımlıyoruz. Sınıfın bildirimini inceleyin:

```
class Myint{
    int m_val;
public:
    Myint(int = 0);
    //...
};
```

Myint sınıfı türünden bir nesnenin içinde tamsayı değeri tutması ve nesnenin *C++*'ın doğal veri türü olan *int* türünün sokulabileceği her türlü işleme sokulabilmesi hedefleniyor. Tabii istenirse sınıfa doğal veri türü olan *int* türünde olmayan bazı özellikler de eklenebilir. Sınıfın *private* bölümünde yer alan *m_val* isimli elemanı, sınıf nesnesinin temsil ettiği değeri tutmak için bildirildi.

Sınıfın kurucu işlevi varsayılan değer alıyor. Kurucu işleve bir argüman gönderilmediği zaman nesnenin *m_val* isimli elemanı *0* değerini alır. Yani nesne *0* değerini tutuyor olur. Diğer taraftan kurucu işlev tek parametre değişkenine sahip olduğu için "dönüştüren kurucu işlev" olarak da görev yapar. Yani *int* türden olan bir ifade ya da otomatik dönüşümle *int* türüne dönüştürülebilecek doğal türlerden bir ifade, otomatik olarak *Myint* sınıfı türünden geçici bir nesneye dönüştürülebilir. Otomatik dönüşümün istenmemesi durumunda kurucu işlev *explicit* anahtar sözcüğüyle tanımlanabilirdi.

Sınıfın kurucu işlevi aşağıda tanımlıyoruz:

```
Myint::Myint(int val): m_val(val) { }
```

Kurucu işlevin tamamında *MIL* sözdiziminin kullanıldığını görüyorsunuz.

Sınıf için kopyalayan kurucu işlevin, atama işlevinin, ve sonlandırıcı işlevin tanımlanmasınmasına gerek yok, değil mi? Derleyicinin yazacağı işlevler yeterli olur. *Myint* sınıfı türünden iki nesnenin elemanlarının karşılıklı olarak biribirine atanmasında bir sakınca olmaz.

Karşılaştırma İşleçlerinin Yüklenmesi

C++'ın tüm karşılaştırma işleçleri iki terimlidir. Karşılaştırma işleçlerini sınıfın bir üye işleviyle ya da global bir işlevle yüklemek mümkündür. Karşılaştırma işleçlerinin *bool* türden değer ürettiğini biliyorsunuz. Ortak arayüzü korumak ve aynı çağrışımdan faydalanmak için, bir sınıfa ilişkin karşılaştırma işlevlerinin de benzer biçimde *bool* türüne geri dönmesi tercih edilir. Üye işlevler olarak yazıldıklarında tek parametre değişkenine sahip olurlar. Global işlevler olarak yazılmaları durumunda ise, iki parametre değişkeninin olması gerekir.

Myint sınıfı için karşılaştırma işleçlerini üye işlevler olarak bildiriyoruz.

```
class Myint {
public:

    //...

    bool operator==(const Myint &r) const;
    bool operator!=(const Myint &r) const;
    bool operator<(const Myint &r) const;
    bool operator<=(const Myint &r) const;
    bool operator>(const Myint &r) const;
    bool operator>=(const Myint &r) const;

    //...
```

```
};
```

Tüm üye karşılaştırma işlevlerinin *bool* türüne geri döndüğünü görüyorsunuz. Karşılaştırma işleçlerinin yan etkisi yoktur. Yani karşılaştırma işleminin sonucunda işlecin sağ ya da sol terimi olan nesnelerin değeri değiştirilmez. İşlecin sol terimi olan nesnenin değerinin değiştirilmeyeceği için işlevleri *const* olarak bildirdik. Karşılaştırma işleçleri sağ terimleri olan nesneleri de değiştirmeyecekleri için işlevlerin parametre değişkenleri *Myint* sınıfı türünden bir *const* referans olarak seçildi. Bu referans ilgili işlecin sağ terimi olan sınıf nesnesinin yerine geçer. İşlevlerin *const* üye işlevler olmaları ve *const* referans parametreye sahip olmaları önemlidir. Böylelikle *const* sınıf nesneleri de bu işleçlerin terimi olabilir. Aşağıda işlev tanımları yer alıyor.

```
bool Myint::operator==(const Myint &r) const
{
    return m_val < r.m_val;
}



bool Myint::operator!=(const Myint &r) const
{
    return r < *this;
}

bool Myint::operator<(const Myint &r) const
{
    return !(*this < r);
}

bool Myint::operator<=(const Myint &r) const
{
    return !(*this > r);
}

bool Myint::operator>(const Myint &r) const
{
    return *this < r || r < *this;
}

bool Myint::operator>=(const Myint &r) const
```

```
{
    return !(*this != r);
}
```

Bir üye işlevin başka bir üye işlevi doğrudan çağrılabileceğini biliyorsunuz. İşleci yükleyen işlev içinde de başka bir üye işleç yükleyen işlev doğrudan çağrılabilir. *Myint* sınıfı önce < işlevini tanımlıyor. Diğer karşılaştırma işlevleri görevlerini bu işlevi çağırarak gerçekleştiriyor. Böylelikle ileride sınıfının içsel yapısında bir değişiklik olması durumunda, daha az üye işlevin kodunda değişiklik yapılması sağlanmış olur.

Myint sınıfının *int* türden parametresi olan bir dönüştüren kurucu işleve sahip olduğunu görüyorsunuz. İşleçleri yükleyen işlevlerin sağ terimleri doğal veri türlerinden olursa çağrılan işlevlere gönderilen argümanlar otomatik tür dönüşüm ile *Myint* sınıfına dönüştürülür. Üye işlevlerin parametre değişkenleri *Myint* sınıfı türünden *const* referans olduğuna göre, örneğin

```
int main()
{
    Myint i(25);

    if (i == 25)
        cout << "doğru" << endl;

    return 0;
}
```

gibi bir kod parçası geçerlidir. Çünkü

```
i == 25
```

ifadesi

```
i.operator==(25);
```

ifadesine eşdeğer olduğundan, derleyici bu ifadeyi de otomatik tür dönüşümüyle

```
i.operator==(Myint(25));
```

biçiminde ele alır.

Aritmetik İşleçlerin Yüklenmesi

İki terimli aritmetik işleçler, yani toplama, çıkarma, çarpma ve bölme işleçleri, üye işlevler ya da global işlevlere yüklenebilir. Bu işleçleri yükleyen işlevler çoğunlukla ilgili sınıf türüne geri döner. Üye işlevler olarak yazıldıklarında tek parametre değişkenine sahip olur. Global işlevlerin ise iki parametresi olur.

Myint sınıfının aritmetik işleçlerini sınıfın üye işlevleri olarak bildiriyoruz:

```
class Myint {
public:

    //...

    Myint operator+ (const Myint &r) const;
    Myint operator-  (const Myint &r) const;
    Myint operator*  (const Myint &r) const;
    Myint operator/  (const Myint &r) const;
    Myint operator% (const Myint &r) const;

};
```

*i*1 ve *i*2 *Myint* sınıfı türünden nesneler olmak üzere

```
i1 + i2
```

gibi bir toplama işleminden yine *Myint* sınıfı türünden bir değer elde edilmesi beklenir, değil mi? Bu durumda toplama işlevleri *Myint* türünden bir değer döndürmelidir. İşlevin neden *Myint* & ya da *Myint* * türüne geri dönemeyeceğini tartışın. İşlevimizin *Myint* sınıfı türünden bir referansa geri döndüğünü düşünelim. Geri dönüş değeri olan referans hangi nesnenin yerine geçebilir? Yerel bir nesneyi referans yoluyla geri döndürmenin bir programlama hatası olduğunu biliyorsunuz. Dinamik bir nesneyi, ya da statik yerel bir nesneyi geri döndürmek de işlevin kullanımını oldukça sınırlar.

Aritmetik işleçler terimleri olan nesneleri değiştirmez. Bu işleçler yükleyen üye işlevlerin *const* üye işlev olması gerekir. Bu işlevlerin hem kendileri *const üye* parametre değişkenleri *const* referans olmalıolan *const* üye işlevler olmalıdır. İşlevlerin tanımlarını inceleyin:

```
Myint Myint::operator +(const Myint &r) const
{
    return Myint(m_val + r.m_val);
}

Myint Myint::operator -(const Myint &r) const
{
    return Myint(m_val + r.m_val);
}

Myint Myint::operator *(const Myint &r) const
{
    return Myint(m_val * r.m_val);
}

Myint Myint::operator /(const Myint &r) const
{
    return Myint(m_val / r.m_val);
}

Myint Myint::operator %(const Myint &r) const
{
    return Myint(m_val % r.m_val);
}
```

İşlevlerin *return* ifadelerinin *Myint* türden bir geçici nesne olduğuna dikkat edin.

operator+ işlevi aşağıdaki gibi de yazılabilirdi:

```
Myint Myint::operator+(const Myint &r) const
{
    Myint result;
    result.m_val = m_val + r.m_val;
```

```
    return result;

}
```

Yukarıdaki işlev tanımında önce yerel *result* nesnesi için varsayılan kurucu işlev çağrılıyor. Daha sonra *result* nesnesine işlemin sonucu olan değer yerleştiriliyor. *result* nesnesinin değeri ile geri dönülüyor. Oysa yazılan ilk kodda doğrudan bir geçici nesnenin değeri ile geri dönülmüştü. Böylece hem işlem maaliyeti azaltılmış olur hem de derleyiciye daha fazla eniyileme işlemi yapma olanağı verilir.

İşlemli Atama İşleçlerinin Yüklenmesi

Şimdi de işlemli atama işleçlerini inceleyelim. Toplama işlevinin tanımlanması += işlevinin derleyici tarafından yazılacağı anlamına gelmez. Bir *Myint* sınıfı nesnesi += işlecinin terimi olursa, programcı tarafından tanımlanmış bir *operator*+= işlevi olmalıdır. İşlemli atama işleçleri de üye işlev ya da global işlevler olarak yüklenebilir. Ancak bu işlevler de sınıf nesnesini değiştirdikleri için üye işlev olarak yüklenmeleri çok daha doğal olur. İşlemli atama işlevlerinin sınıf içinde bildirimlerini yapalım:

```
class Myint {
public:

    //...

    Myint &operator+= (const Myint &r);
    Myint &operator-= (const Myint &r);
    Myint &operator*= (const Myint &r);
    Myint &operator/= (const Myint &r);
    Myint &operator%= (const Myint &r);

};
```

Doğal türler söz konusu olduğunda, atama işleçlerinin ürettiği değer nesneye atanan değerdir. Aynı arayüzü sağlayabilmek için bu işleçleri yükleyen işlevlerin geri dönüş değerinin *Myint* sınıfı türünden referans yapıldığını görüyorsunuz. Böylece bir ifade içinde birden fazla atama işlecinin kullanılması olanağı getiriliyor. İşlemli atama işlecinin sağ terimi olan nesne böyle bir işlemden etkilenmeyeceği için işlevin parametre değişkeni *const Myint* & türünden yapılmalı. İşlevlerin tanımını aşağıda yapıyoruz:

```
Myint &Myint::operator+= (const Myint &r)

{

    m_val += r.m_val;
    return *this;

}


Myint &Myint::operator-= (const Myint &r)
```

```
{

    m_val -= r.m_val;
    return *this;

}


Myint &Myint::operator*= (const Myint &r)

{

    m_val *= r.m_val
    return *this;

}


Myint &Myint::operator/= (const Myint &r)

{

    m_val /= r.m_val;
    return *this;

}


Myint &Myint::operator%= (const Myint &r)

{

    m_val %= r.m_val;
    return *this;

}
```

++ ve -- İşleçlerinin Yüklenmesi

++ ve –- işleçlerinin hem önek hem de sonek konumunda kullanıldıklarını biliyorsunuz. Bu işleçler global işlevler ile de yüklenebilir. Ancak bu işleçleri yükleyen işlevler çoğunlukla sınıf nesnesi üzerinde değişiklik yaptığı için üye işlevler olarak yazılmaları tercih edilir.

Önek ve sonek konumunda bulunabilen bu işlevlerin ayrı ayrı yüklenebilmesi için işlevlerin imzalarının farklı olması gerekir. İşte bu nedenle bu işleçleri yükleyen işlevler için şöyle bir kural getirilmiştir: Bu işleçler üye işlevler ile yüklendiklerinde, önek konumunda olanlar için işlevin parametre değişkeni olmaz. Ancak sonek konumundaki işleci yükleyen işlevin *int* türden bir parametre değişkeni olur. Bu parametre değişkeninin amacı bir değer taşımak değil, yalnızca işleve farklı bir imza kazandırmaktır. *Myint* sınıfının tanımı içinde yer alan ++ ve –- işlevlerinin bildirimlerini inceleyelim.

```
class Myint {
public:

    //...
```

```
    Myint &operator++ ();
    Myint &operator--  ();

    Myint &operator++  (int);  //sonek ++
    Myint &operator--  (int) ; //sonek --

};
```

Önek konumundaki işleçleri yükleyen işlevler *Myint* sınıfı türünden referansa dönerken, sonek konumunda olanları *Myint* türüne geri dönüyor. Acaba neden? Bildiğiniz gibi, bu işleçlerin terimi doğal türden nesneler olduklarında, işleçler yan etki olarak, terimi olan nesnelerin değerlerini *1* arttırır ya da azaltır. İşleç ister sonek ister önek konumunda olsun terimi olan nesnenin değeri yan etki sonucu değişir. Ancak işlecin ürettiği değer önek ya da sonek konumu için farklıdır. Önek konumundaki *++* işleci, terimi olan nesnenin değerinin 1 fazlasını üretirken sonek konumundaki ++ işleci terimi olan nesnenin kendi değerini üretir.

Sınıf nesneleri için yazılan ++ işlevlerinde de işlecin bu özelliğine çoğunlukla uyulur. Zaten daha önce de söylendiği gibi işleç yüklemesinin temel amaçlarından biri işleçlerin doğal türler için verdiği çağrışım ve soyutlamadan sınıf nesneleri için de faydalanmaktadır. *Myint* sınıfı için ++ işlevleri aşağıdaki gibi yazılabilir. Önce önek konumundaki ++ işleci için bir işlev tanımlayalım.

```
Myint &Myint::operator++()
{
    return *this += 1;
}
```

İşlevi birlikte inceleyelim. İşlevin *return* deyiminde daha önce yazılan += işlevi çağrılıyor.

```
*this += 1
```

Yukarıdaki işlev çağrısı ile *operator*+= işlevi **this* nesnesinin *m_val* elemanının değerini *1* arttırıyor. Daha önce yazılan += işlevi **this* nesnesini geri döndürdüğüne göre, ++ işlevi de **this* nesnesini geri döndürmüş oluyor. Böylece ++ işlecinin terimi olan sınıf nesnesinin artmış değerini kullanma olanağı getiriliyor.

```
Myint a, b(9);
```

gibi bir tanımlamadan sonra

```
a = ++b;
```

atama deyimi

```
a.operator=(b.operator++());
```

gibi bir çağrıya dönüştürülür değil mi?

Şimdi de sonek konumundaki ++ işlecini yükleyen işlevi tanımlayalım.

```
Myint Myint::operator++()
{
    Myint ret_val(*this);
    ++*this;

    return ret_val;
}
```

İşlevin *int* türden isimlendirilmiş bir parametre değişkenine sahip olduğunu görüyorsunuz. *C* dilinde sözdizimi hatası olan bu durum *C++* dilinde geçerlidir! *C++* da bir işlevin parametre değişkenine isim verme zorunluluğu yoktur. Daha önce de söylendiği gibi bu parametre değişkeninin tek amacı işlevin imzasını önek konumunda olan *operator*++ işlevinden farklı kılmaktır. Önek konumunda olan ++ işlevi terimi olan nesnenin kendi değerini ürettiği için, işlev de benzer biçimde nesnenin kendi değerini geri döndürmelidir. Bu yüzden nesnenin değeri, daha sonra geri dönüş değeri olarak kullanmak üzere *ret_val* isimli bir nesnede saklanıyor. Daha önce tanımlanan ++ işlevi ile *m_val* elemanının değeri 1 artırdıktan sonra nesnenin arttırılmadan önceki değerini tutan *ret_val* nesnesinin değeri ile geri dönülüyor.

Örnekler ++ işlevleri için verildi ancak –- işleçler için de tamamen benzer kodlar söz konusudur.

Yazılan işlevleri aşağıdaki main işleviyle sınayabiliriz:

```
#include <iostream>
using namespace std;

int main()
{
    Myint x1(10), x2(20);
    Myint y1(++x1), y2(x2++);
```

```
    cout << "y1 = " << y1 << endl;
    cout << "y2 = " << y2 << endl;
    cout << "x1 = " << x1 << endl;
    cout << "x1 = " << x1 << endl;


    return 0;
}
```

İşleçleri Yükleyen Global İşlevlerin Yazılması

İşleçlerin çoğu global işlev biçiminde de yüklenebilir. Yukarıdaki örnekte *Myint* sınıfı için toplama işleci üye işlev ile yüklenmişti. Şimdi aynı işleci global işlevle yükleyelim.

```
Myint operator+(const Myint &, const Myint &);
```

Böyle bir işlevin var olması durumunda

```
Myint a(10), b(20);
a + b
```

gibi bir ifade, daha önce söylendiği gibi

```
operator+(a, b)
```

gibi bir işlev çağrısına dönüştürülür.

İşleçleri yükleyen global işlevler sınıfın üye işlevler olmadığı için bu işlevler içinde normal olarak sınıfın *private* bölümüne erişilemez.

```
Myint operator+(const Myint &r1, const Myint &r2)
{
    return Myint(r1.m_val, r2.m_val);    //Geçersiz!
}
```

Yukarıdaki örnekte *operator+* işlevi içinde sınıfın *private m_val* elemanına erişilemez. İşlev sınıfın üyesi olmadığı için, *r1* ve *r2* nesnelerinin *private* elemanlarına erişmesi söz konusu değildir? Ancak uygun bir geri dönüş değeri üretmek için bu *private* elemanların değerlerine gereksinim duyuluyor.

Bu durumda işleci yükleyen global bir işlev, işini nasıl yapabilir?

İşlevin kendisinden beklenen işi yapabilmesi için farklı yöntemler kullanılabilir:

Sınıfın *public* arayüzünde bulunan bir *get* işlevi ile *m_val* elemanının değeri elde edilebilir. Ancak böyle bir *get* işlevinin bulunması, her sınıf için istenen bir durum değildir.

```
class Myint{
public:
     int get_val() const {return m_val;}
     //...
};
```

Global *operator+* işlevi *get_val public* üye işlevini çağırarak *m_val* değerini elde edebilir.

```
Myint operator+(const Myint &r1, const Myint &r2)
{
    return Myint(r1.get_val(), r2.get_val());
}
```

Myint sınıfı, bu işleve arkadaşlık bildirimi ile *private* elemanlarına özel erişim hakkı verebilir.

```
class Myint{
     //...
    friend Myint operator+(const Myint &, const Myint &);
};
```

Global işlev işini görebilmek için sınıfın bir *public* işlevini çağırabilir. Aşağıdaki kodu inceleyin:

```
Myint operator+(const Myint &r1, const Myint &r2)
{
    return Myint (r1) += r2;
}
```

İşlevin ana bloğu içinde yaratılan *Myint* sınıfı türünden geçici nesne, ilkdeğerini işlevin parametre değişkeni olan *r1* nesnesinden alıyor. Daha sonra geçici nesne için sınıfın *+=* işlevi çağrılıyor. Bu üye işleve *r*2 nesnesi argüman olarak gönderiliyor. İşleci yükleyen global işlev, üye += işlevinin geri dönüş değeri ileri geri dönüyor.

Aynı işlecin hem üye hem de global işlev olarak yüklenmesi, çoğu durumda algısal karmaşıklığa neden olur. Bu nedenle programcılar tarafından pek tercih edilmez. Ayrıca bazı durumlarda, işleve yapılan çağrı yüzünden çift anlamlılık hatası oluşabilir. Yani derleyicinin üye işlevin mi global işlevin mi çağrılmak istendiğini anlayamaması sonucu bir hata durumu oluşabilir. Şüphesiz bu hata durumu örneğin *operator+* işlevinin işlev çağrı işleciyle çağrılmasıyla engellenebilir:

```
class Myclass {
public:

    Myclass operator+(Myclass, Myclass);

};


Myclass operator+ (Myclass, Myclass);

void func()

{

    Myclass a, b, c;

    //...


    c = a + b;   //çift anlamlılık hatası - üye işlev mi global işlev mi?
    c = operator+ (a, b);   //çift anlamlılık hatası yok global işlev

    c = a.operator+(b);     //çift anlamlılık hatası yok üye işlev

    //…

}
```

İşleç Yükleyen İşlevlere İlişkin Kısıtlamalar

C++"ın bazı işleçleri yüklenemez. Bu işleçler şunlardır:

çözünürlük işleci (*::*)
sizeof işleci (*sizeof*) koşul
işleci *?:*

. işleci

.* işleci (C dilinde yer almayan bu işleci ileride ele alacağız)

Yukarıda listesi verilen işleçlerin sınıfın üye işlevleriyle ya da global işlevlerle yüklenmesi geçersizdir.

Aşağıdaki işleçler ise yalnızca sınıfın üye işlevleriyle yüklenebilir:

köşeli ayraç işleci
işlev çağrı işleci
atama işleci

ok işleci

Bu işleçlerin global işlevler biçiminde yüklenmeleri geçersizdir.

İşleçleri yükleyen işlevler varsayılan argüman alamaz. Bu kurala uymayan tek işlev, işlev çağrı işlecini yükleyen işlevdir. Bu işlevin yüklenmesini ileride ele alacağız.

İşleç Yükleyen İşlevlerin Çağrılma Sırası

İşleç yükleyen işlevler C++ dilinde işleçlerin belirlenmiş öncelik sırasına göre çağrılır. Aşağıdaki kod parçasını inceleyin:

```
Myint a(1), b(5), c(7), d;

d = ++a * b -c % 2;
```

Yukarıdaki ifade derleyici tarafından şöyle bir koda dönüştürülür:

```
d.operator=(a.operator++().operator*(b).operator-(c.operator%(2)));
```

İşleçlerin dil tarafından belilenmiş öncelik seviyeleri değiştirilemez.

Köşeli Ayraç İşlecini Yükleyen İşlev

Köşeli ayraç işleci sınıfın üye işleviyle yüklenebilir. İşlevin yalnızca bir parametre değişkeni olmalıdır.

a bir sınıf nesnesi olmak üzere

a[b] gibi bir ifade, köşeli ayraç işlevinin tanımlanmış olması durumunda

```
a.operator[](b)
```

gibi bir işlev çağrısına dönüştürülür.

Köşeli ayraç işlevi genellikle bir referans türüne geri döner. Böylece, işlev çağrı ifadesi doğrudan bir nesne olarak kullanılabilir.

int türden dinamik bir dizinin kullanımını kolaylaştırmak için, *Array* isimli bir sınıfın tanımlandığını düşünelim.

operator[] işlevinin uygun biçimde tanımlanması durumunda, *Array* sınıfı türünden a gibi bir nesne ile

```
a[5] = 20;
```

gibi bir atama yapılabilir. Konumuzla doğrudan ilgisi olmayan işlevleri çıkartarak, *Array*

sınıfını tanımlayalım.

```
class Array {

    int *m_p;
    size_t m_size;

public:

    Array(int size = 0, int val = 0);

    ~Array();

    int operator[](int) const;
    int &operator[](int);

    //…

};
```

```
Array::Array(int size, int val)

{

    m_size = size;
    if(m_size == 0){

        m_p = 0;
        return;

    }


    m_p = new int[m_size];
```

```
    for (int k = 0; k < m_size; ++k)
        m_p[k] = val;

}


Array::~Array()

{

    delete [] m_p;

}

int &Array::operator[](int index)

{

    return m_p[index];

}


int Array::operator[](int index)const

{

    return m_p[index];

}


#include <iostream>


int main()

{

    Array a(20);

    const Array ca(10, 1);

    a[3] = 5;

    //ca[2] = 10;  //Geçersiz
    Std::cout << ca[5] << std::endl;


    return 0;

}
```

Array sınıfının kurucu işlevi, istenen sayıda elemanın sığacağı büyüklükte bir blok elde ediyor. Sonlandırıcı işlev ise ayrılan dinamik bloğu *free store*"a geri veriyor.

Sınıfın *public* bölümünde iki ayrı *operator[]* işlevinin bildirildiğini görüyorsunuz. İşlevlerin imzalarındaki tek farklılık, birinin *const* üye işlev iken diğerinin *const* olmayan bir üye işlev olması. Böyle işlev yüklemesine *const* yüklemesi (*const overloading*) dendiğini anımsayın. Acaba neden iki ayrı işlev tanımlandı?

Önce *const* olmayan *operator[]* işlevini inceleyelim. İşlevimiz elde edilen dinamik dizinin istenen indisli elemanını, referans yoluyla döndürüyor. Böylece *main* işlevi içinde

```
a[3] = 5;
```

gibi bir atama ile, dinamik dizi içindeki *3* indisli elemana atama yapılmış olur. Bu atama deyimi derleyici tarafından aşağıdaki biçime dönüştürülür:

```
a.operator[](3) = 5;
```

const bir dizinin elemanlarına atama işleci ile atama yapılamaz, değil mi? Benzer biçimde *const* bir *Array* nesnesinin tuttuğu elemanlara da atama yapılamaması gerekir. İşte bu nedenle iki ayrı işlev tanımladık.

Array sınıfı türünden *const* bir nesne olan *a* nesnesi ile *operator[]* işlevi çağrıldığında, sınıfın *const* üye işlevi çağrılır. *const* üye işlevin geri dönüş değeri bir referans olmadığı için

```
ca[4] = 10;
```

gibi bir atama geçersizdir.

Sınıf Nesnelerinin Değerlerinin Yazdırılması ve Klavyeden Sınıf Nesnelerine Değer Alınması

C"den farklı olarak *C++* dilinde giriş çıkış işlemlerinin şablon bazlı sınıflar yardımıyla yapıldığını biliyorsunuz.

```
int i = 10;
double d = 3.5;
cout << i <<d;
```

Yukarıdaki kod parçasında yer alan *cout,* aslında *ostream* isimli bir sınıf türünden global bir nesnedir. *ostream* sınıfının bir seri *operator* << işlevi vardır. C++ dilinin doğal türlerinden parametre değişkenlerine sahip olan bu işlevler, yine *ostream* sınıfı türünden referansa geri döner.

```
cout << i << d
```

gibi bir ifade derleyici tarafından

```
cout.operator<<(i).operator<<(d)
```

biçimine dönüştürülür.

cout nesnesi için çağrılan ilk yükleyici işlev *i* değişkenini referans yoluyla alır. *i* değişkeninin değerini ekrana yazdırdıktan sonra *cout* nesnesini referans yoluyla geri döndürür. Geri döndürülen *cout* nesnesi için ikinci kez çağrılan *operator*<< işlevi, bu kez *d* nesnesini referans yoluyla argüman olarak alır. *d* nesnesinin değeri ekrana yazdırıldıktan sonra *cout* nesnesi yine referans yoluyla geri döndürülür.

Nasıl doğal türlerden ifadelerin değerleri *ostream* sınıfının üye *operator*<< işlevleriyle ekrana yazdırılabiliyorsa, programcı tarafından tanımlanan bir sınıf türünden nesnenin değeri de, global<< işlevi yardımıyla ekrana ya da bir dosyaya yazdırılabilir. Böylece çıkış işlemleri için doğal veri türleri ve programcı tarafından tanımlanan türler arasında ortak bir arayüz sağlanmış olur. *Myint* sınıfı türünden nesnelerin değerlerinin, aşağıdaki biçimde ekrana yazdırılmak istendiğini düşünelim.

```
Myint a(10), b(20);
cout << a << " " << b << endl;
```

ostream sınıfının kodlarına doğrudan ekleme yapılamaz. Ama global bir *operator*<< işlevi yazılabilir:

```
cout << a
```

gibi bir ifadenin global bir işlev çağrısına dönüştürülmesi için işlevin iki parametre değişkenine sahip olması gerekir. Birinci parametre *cout* nesnesini alırken ikinci parametre *Myint* sınıfı türünden *a* nesnesini alır. İşlevin geri dönüş değeri *ostream* sınıfı türünden bir referans yapılırsa, işlevin yine *cout* nesnesini geri döndürmesi sağlanabilir. Aşağıdaki işlevi inceleyin:

```
ostream &operator<<(ostream &os, const Myint &r)
{
```

```
    return os << r.m_val;
}
```

işlevimiz

```
operator<<(cout, a)
```

biçiminde çağrıldığına göre, işlevin parametre değişkeni olan referans *os, cout* nesnesinin yerine geçer. İkinci parametre olan *r* referansı da *a* nesnesinin yerine geçer. İşlevin tanımı içindeki

```
os << r.m_val
```

ifadesi, bu kez *ostream* sınıfının üye işlevini çağırarak *m_val* isimli elemanın değerini ekrana yazdırır. İşlevimiz

```
os.operator<<(r.m_val)
```

çağrısının geri dönüş değeriyle geri döner. Bu da *cout* nesnesinin kendisidir.

Yalnız ufak bir sorunumuz var. Global işlev içinde

```
r.m_val
```

erişiminin geçerli olması için *Myint* sınıfının, global işleve arkadaşlık vermesi gerekir, değil mi?

```
class Myint {
//...
    friend std::ostream &operator<<(std::ostream &os, const Myint &r);
};
```

C++"ın standart *ostream* sınıfı *std* isimalanı içinde tanımlandığından, arkadaşlık bildiriminde bu sınıfın ismi, *std::ostream* olarak yazılıyor.

Şimdi de *Myint* sınıfı türünden bir nesneye standart giriş biriminden (klavyeden) değer alan bir işlev tanımlayalım:

```
int i = 10;
double d = 3.5;
cin >> i >> d;
```

Yukarıdaki kod parçasında yer alan *cin, istream* isimli bir sınıf türünden global bir nesnedir. *istream* sınıfının bir seri *operator>>* işlevi vardır. *C++* dilinin doğal türlerinden parametre değişkenlerine sahip olan bu işlevler, yine *istream* sınıfı türünden referansa geri döner.

```
cin >> i >> d
```

gibi bir ifade derleyici tarafından

```
cin.operator>>(i).operator>>(d)
```

biçimine dönüştürülür.

Çağrılan ilk işlev, *i* değişkenini referans yoluyla alır. İşlev *i* değişkenini klavyeden aldığı değerle set ettikten sonra *cin* nesnesini referans yoluyla geri döndürür. Geri döndürülen *cin* nesnesi için ikinci kez çağrılan *operator>>* işlevi bu kez *d* nesnesini referans yoluyla argüman olarak alır. *d* nesnesini klavyeden alınan değerle set ettikten sonra *cin* nesnesi yine referans yoluyla geri döndürülür.

```
Myint a(10), b(20);
cin >> a >> b;
```

istream sınıfına doğrudan ekleme yapılamaz ama global bir *operator>>* işlevi yazılabilir.

```
cin >> a
```

gibi bir ifadenin global bir işlev çağrısına dönüştürülmesi için işlevin iki parametre değişkenine sahip olması gerekir. İşlevin birinci parametresi *cin* nesnesini referans yoluyla alırken ikinci parametre *Myint* sınıfı türünden *a* nesnesini yine referans yoluyla alabilir. İşlevin geri dönüş değeri *istream* sınıfı türünden bir referans yapılırsa, yeniden *cin* nesnesinin geri döndürülmesi sağlanabilir.

Aşağıdaki işlev tanımını inceleyin.

```
istream &operator<<(ostream &is, Myint &r)
{
    return is >> r.m_val;
}
```

işlevimiz

```
operator>>(cin, a)
```

biçiminde çağrıldığına göre, işlevin parametre değişkeni olan referans *is, cin* nesnesinin yerine geçiyor. İkinci parametre olan *r* referansı da *a* nesnesinin yerine geçiyor. İşlevin tanımı içindeki

```
is >> r.m_val
```

ifadesi ile bu kez *istream* sınıfının üye işlevi çağrılarak *m_val* isimli eleman klavyeden alınan değerle set ediliyor. İşlevimiz

```
is.operator>>(r.m_val)
```

çağrısının geri dönüş değeriyle geri dönüyor. Yani işlev *cin* nesnesini geri döndürüyor.

Numaralandırma Türleri İçin İşleçlerin Yüklenmesi

Bir işleç, bir numaralandırma türü için de yüklenebilir. Örneğin tipik bir kullanım, ++ ve –

- işleçlerinin yüklenmesiyle numaralandırma değerlerinin dolaşılmasını sağlamaktır. Aşağıdaki programı inceleyin:

```
#include <iostream>

enum Months {January, February, March, April, May, June, July, August,
September, October, November, December};
```

```
Months &operator++ (Months &m)

{

    if (m == December)
        return m = January;

    int temp = m;

    return m = Months(++temp);

}


std::ostream &operator<<(std::ostream &os, const Months &r)

{

    switch (r){

        case January    : return os << "January";
        case February   : return os << "February";
        case March      : return os << "March";
        case April      : return os << "April";
        case May        : return os << "May";
        case June       : return os << "June";
        case July       : return os << "July";
        case August     : return os << "August";

        case September  : return os << "September";
        case October    : return os << "October";
        case November   : return os << "November";
        case December   : return os << "December";

    }

}

int main()

{

    Months m = January;
    for (;;) {

        std::cout << m << std::endl;

        ++m;

        if (m == January)
            break;

    }

    return 0;

}
```

İşleç yükleyen işlevlerin kullanımına örnek vermek amacıyla, aşağıda *Date* isimli bir sınıfın kodlarını veriyoruz. *Date* isimli sınıfı türünden bir nesne bir tarih bilgisi tutmaktadır.

```
//////////////////////  date.h  ////////////////////////////
#include <iosfwd>

#include <stdexcept>


class Date{

    int m_d, m_m, m_y;
    int m_totaldays;

    static bool is_valid (int, int , int);
    static bool is_leap (int y){

        return y % 4 == 0 && 100 != 0 || y % 400 == 0;

    }

    static int ms_daytabs[][13];
    static int ms_yeartabs[2];
    static const char *ms_days[];
    static const char *ms_mons[];

    const static int msc_yearbase = 1700;
    const static int msc_factor = 0;

    void set_totaldays();
    void set(int, int, int);

    Date &revdate(int totaldays);
public:

    Date(int, int, int);
    Date();

    int get_year_day() const;
    int get_week_day() const;

    int get_mday() const {return m_d;}
    int get_mon() const {return m_m;}
    int get_year() const {return m_y;}
    Date &operator+=(int n);

    Date &operator-=(int n);
    Date &operator++();

    Date operator++(int n); //postfix
    Date &operator--();

    Date operator--(int n); //postfix

    //friend functions

    friend std::ostream &operator<<(std::ostream &, const Date &);
    friend std::istream &operator>>(std::istream &, Date &);
    friend bool operator<(const Date &, const Date &);
```

```
    friend int operator-(const Date &, const Date &);
};


class BadDate:public std::out_of_range{
public:
    BadDate(const char *pmsg):std::out_of_range(pmsg){}
};
//global functions
bool operator>(const Date &, const Date &);
bool operator>=(const Date &, const Date &);
bool operator<=(const Date &, const Date &);
bool operator==(const Date &, const Date &);
bool operator!=(const Date &, const Date &);
Date operator+(const Date &, int);
Date operator+(int, const Date &);
Date operator-(const Date &, int);


////////////////////////////////////////////////////////////////////////////

//date.cpp file
#include <iostream>
#include <iomanip>
#include <ctime>


using namespace std;


int Date::ms_daytabs[2][13] = {
{0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
{0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
};


int Date::ms_yeartabs[] = {365, 366};
const char *Date::ms_mons[] = {"", "Ocak", "Subat", "Mart", "Nisan",
"Mayıs",
"Haziran", "Temmuz", "Agustos", "Eylul", "Ekim", "Kasim", "Aralik"};


const char *Date::ms_days[] = {"Pazartesi", "Sali", "Carsamba", "Persembe",
"Cuma", "Cumartesi", "Pazar"};
```

```
////////////////////////////////////////////////////////////////////////////
void Date::set(int d, int m, int y)

{
    if (!is_valid(d, m, y))

        throw BadDate("gecersiz tarih\n");
    m_d = d;

    m_m = m;
    m_y = y;

    set_totaldays();

}
////////////////////////////////////////////////////////////////////////////
void Date::set_totaldays()

{
    m_totaldays = 0;



    for (int k = msc_yearbase; k < m_y; ++k)
        m_totaldays += ms_yeartabs[is_leap(k)];

    m_totaldays += get_year_day();

}
////////////////////////////////////////////////////////////////////////////
Date &Date::revdate(int totaldays)

{
    m_totaldays = totaldays;
    m_y = msc_yearbase;



    while (totaldays > ms_yeartabs[is_leap(m_y)])
        totaldays -= ms_yeartabs[is_leap(m_y++)];

    m_m = 1;



    while (totaldays > ms_daytabs[is_leap(m_y)][m_m])
        totaldays -= ms_daytabs[is_leap(m_y)][m_m++];


    m_d = totaldays;

    return *this;

}
```

```
////////////////////////////////////////////////////////////////////////////
Date::Date(int d, int m, int y)

{

    set(d, m, y);

}

////////////////////////////////////////////////////////////////////////////
Date::Date()

{

    time_t timer = time(0);

    tm *tp = localtime(&timer);

    set(tp->tm_mday, tp->tm_mon + 1, tp->tm_year + 1900);

}

////////////////////////////////////////////////////////////////////////////
int Date::get_year_day()const

{

    int yearday = m_d;



    for (int k = 1; k < m_m; ++k)

        yearday += ms_daytabs[is_leap(m_y)][k];
    return yearday;

}

////////////////////////////////////////////////////////////////////////////
int Date::get_week_day()const

{

    return (m_totaldays + msc_factor) % 7;

}

////////////////////////////////////////////////////////////////////////////
Date &Date::operator+=(int n)

{

    return revdate(m_totaldays + n);

}

////////////////////////////////////////////////////////////////////////////
Date &Date::operator-=(int n)

{

    return revdate(m_totaldays - n);
```

```
}

////////////////////////////////////////////////////////////////////////////
Date &Date::operator++()

{

    return *this += 1;

}

////////////////////////////////////////////////////////////////////////////
Date Date::operator++(int)

{

    Date retval(*this);

    ++*this;
    return retval;

}

////////////////////////////////////////////////////////////////////////////
Date &Date::operator--()

{

    return *this -= 1;

}

////////////////////////////////////////////////////////////////////////////
Date Date::operator--(int)

{

    Date retval(*this);

    --*this;
    return retval;

}

////////////////////////////////////////////////////////////////////////////
bool Date::is_valid(int d, int m, int y)

{

    if (y < msc_yearbase)
        return false;

    if(m < 1 || m > 12)
        return false;



    return m > 0 && d <= ms_daytabs[is_leap(y)][m];

}

////////////////////////////////////////////////////////////////////////////
ostream &operator<<(ostream &os, const Date &r)
```

```
{
    return os << setw(2) << r.m_d << " " << Date::ms_mons[r.m_m] << " " <<
r.m_y
    << " " << Date::ms_days[r.get_week_day()];
}
////////////////////////////////////////////////////////////////////////////
istream &operator>>(istream &is, Date &r)
{
    int d, m, y;
    is >> d >> m >> y;
    r.set(d, m, y);

    return is;
}
////////////////////////////////////////////////////////////////////////////
bool operator<(const Date &d1, const Date &d2)
{
    return d1.m_totaldays < d2.m_totaldays;
}
////////////////////////////////////////////////////////////////////////////
bool operator>(const Date &d1, const Date &d2)
{
    return d2 < d1;
}
////////////////////////////////////////////////////////////////////////////
bool operator>=(const Date &d1, const Date &d2)
{
    return !(d1 < d2);
}
////////////////////////////////////////////////////////////////////////////
bool operator<=(const Date &d1, const Date &d2)
{
    return !(d2 < d1);
}
////////////////////////////////////////////////////////////////////////////
int operator-(const Date &d1, const Date &d2)
{
```

```
    return d1.m_totaldays - d2.m_totaldays;
}
////////////////////////////////////////////////////////////////////////////


Date operator+(const Date &d, int n)
{
    return Date(d) += n;
}
////////////////////////////////////////////////////////////////////////////
Date operator+(int n, const Date &d)
{
    return d + n;
}
////////////////////////////////////////////////////////////////////////////
Date operator-(const Date &d, int n)
{
    return Date(d) -= n;
}
////////////////////////////////////////////////////////////////////////////
int main()
{
    Date today;
    Date next_year_today(today + 365);


    while (today <= next_year_today)
        cout << today++ << endl;


    return 0;
}
```

İSİM ALANLARI

Global olarak bildirilmiş olan varlıklar için seçilmiş isimlerin birbirinden farklı olması gerekir.

Global bölgede programcı tarafından bildirilen türler, nesneler, işlevler, şablonlar, global varlıklardır. Örneğin bir işlevin ismiyle, global bir nesnenin ismi aynı olamaz. Üstelik bu durum yalnızca tek bir kaynak dosya için değil, projeyi oluşturan diğer kaynak dosyalar için de geçerlidir. Aynı kaynak dosyada aynı ismin global düzeyde

kullanılması derleme zamanında hataya neden olur. Aynı ismin global düzeyde farklı dosyalarda dış bağlantıya ait olacak biçimde kullanılması, bağlama (*linking*) zamanında hataya neden olur.

Bu durumun programcı için uygulamadaki önemi şudur: Projede başka kütüphaneler ya da modüller kullanılıyorsa, kullanılan modüllerdeki global varlıkların isimleri, programcının yazmakta olduğu kaynak dosyadaki global isimler ile aynı olmamalıdır. Yani global düzeydeki isimler çakışmamalıdır.

İsim çakışmasını engellemek her zaman kolay değildir. Çünkü kullanılan modüller ya da kütüphaneler farklı kişiler ya da firmalar tarafından üretilmiş olabilir. Özellikle büyük programlar söz konusu olduğunda global isimlerin çakışma riski çok fazladır. Farklı kişilerin geliştirdiği kütüphanelerin birleştirilmesi durumunda isimlerin çakışması nasıl engellenebilir? Global isimlere yönelik bu çakışma problemi "global isim alanının kirlenmesi problemi" olarak *(global namespace pollution)* bilinir.

C++ dilinin standartlaştırılma dönemi öncesinde, programcılar global isim alanının kirlenmesini, global varlıklara çok uzun isimler vererek engellemeye çalışıyorlardı. Bu isimler de çoğunlukla önceden belirlenen önek sözcüklerle oluşturuluyordu. Bir örnek verelim: Dinamik bir diziyi soyutlayan bir sınıfın tanımlandığını düşünelim. Sınıfın ismi *Array* olarak seçilirse isim çakışması olasılığı artar. Kullanılan modüller içinde bu isim bir başka sınıfa verilmiş olabilir. Olası bir isim çakışmasını engellemek için sınıfa daha uzun bir isim verilebilir:

```
class C_ve_Sistem_Programcilari_Array {
    //…
};

void display(const C_ve_Sistem_Programcilari_Array &);
```

Bunun iyi bir çözüm olduğu söylenemez. C++ dilinde yazılmış bir program, bütün program içinde görülebilen çok sayıda sınıf, işlev ya da şablon içerebilir. Sürekli uzun isimler yazmak, hem programcı için zahmetli bir iştir, hem de böyle isimler programın okunabilirliğini azaltır.

“İsim alanları” (*Namespaces)* global isim alanı kirlenmesi problemini büyük ölçüde çözen, C++ dilinin önemli bir aracıdır.

Bilgisayarımızın hard diskinin sadece bir kök dizin içerdiğini düşünelim. Bu dizin içinde aynı isimli iki dosya olamaz, değil mi? Kök dizin altında çeşitli alt dizinler oluşturabiliriz. Bu alt dizinler içinde isimleri aynı olan dosyalar bulunabilir. Örneğin ismi *necati.txt* olan bir dosya, hem kök dizinde hem de kök dizinin altındaki *"notlar", "mektuplar", "odevler"* isimli dizinlerde bulunabilir.

Kaynak dosyanın global düzeyi, yani global isim alanı, yukarıdaki örnekteki kök dizine benzetilebilir. Nasıl kök dizin içinde iki dosyanın ismi aynı olamıyorsa kaynak dosyada da global iki varlığın ismi aynı olamaz. İşte kök dizin içinde farklı isimli dizinler yaratılması gibi, kaynak dosyanın global alanında farklı isim bölgeleri yaratılabilir. Bu durumda bu farklı bölgeler içinde aynı isimler kullanılabilir.

Bir programda belirli isimleri, global alandan gizlemek için isim alanları tanımlanabilir:

```
namespace CDernek{

    class CDString{ /* ...   */};
    void push_back(CDString &);

}
```

CDernek programcı tarafından bildirilen bir isim alanıdır. Programcı tarafından bildirilen her isim alanı, ayrı bir bilinirlik alanı belirler. Programcı tarafından bildirilen bir isim alanı, başka isim alanı bildirimlerini, işlevlerin nesnelerin şablonların, türlerin, bildirimlerini ya da tanımlarını içerebilir.

Global isim alanında olduğu gibi , programcı tarafından bildirilen isim alanındaki her isim de tek bir varlığa ilişkin olmalıdır. Yani bildirilen bir isim alanında aynı isim birden fazla kez bulunamaz. Ama farklı isim alanı bildirimleri farklı bilinirlik alanları belirlediği için, farklı isim alanlarında aynı isimler kullanılabilir.

İsim alanında kullanılan her isme *"isim alanı elemanı"* (*namespace member*) denir.

İsim Alanı Tanımı

Bir isim alanı tanımı *namespace* anahtar sözcüğü ile başlar. *namespace* anahtar sözcüğünü isim alanının ismi izler. İsim alanı ismi isimlendirme kurallarına uygun olarak seçilmelidir. İsim alanı ismini izleyen blok içinde isim alanı elemanlarının bildirimleri ve/veya tanımlamaları yer alır. İsim alanı tanımı sonunda sonlandırıcı atom yer almaz:

```
namespace CDernek {
    class CDString {

    //…

    };

    //…

}
```

İsim alanı ismi global düzeyde başka bir isimle çakışmamalıdır. İsim alanı ismi de, tanımın yapıldığı isim alanında tek olmalıdır. Yani global isim alanı kirlenmesi sorunu tam olarak ortadan kaldırılmış olmaz. Ancak isim alanlarının kullanılmasıyla sorun en aza indirilir. Bir bildirimin ya da bir tanımın, bir isim alanı içinde yapılması bildirimin ya da tanımın anlamını değiştirmez. İsim alanı içinde yapılmış bir bildirimi ya da tanımı diğer bildirim ya da tanımlardan ayıran tek nokta, bildirilen ya da tanımlanan isimlerin isim alanı içinde gizlenmiş olmasıdır. Bu isimlere dışarıdan ancak *nitelendirilmiş isimler* olarak yani çözünürlük işleciyle erişilebilir.

Bir isim alanı tanımı ya global düzeyde ya da başka bir isim alanı tanımının içinde yapılmalıdır. Yerel düzeyde, yani bir işlevin tanımı içinde bir isim alanı tanımlanamaz.

İsim Alanı Bir Bilinirlik Alanıdır

Programcı tarafından bildirilen her isim alanı, yeni bir bilinirlik alanı oluşturur. Bir isim alanı içinde başka isim alanı bildirimleri, işlev bildirimleri ve tanımlamaları tür bildirimleri nesne bildirim ve tanımlamaları yer alabilir. Aynı bilinirlik alanında aynı isim birden fazla kez kullanılamaz. İsim alanı da yeni bir bilinirlik alanı oluşturduğuna göre bir isim alanı içinde de aynı isim bulunamaz. Farklı isim alanları farklı bilinirlik alanı belirttiklerine göre, farklı isim alanları içinde aynı isim kullanılabilir. Örneğin yukarıda bildirilen *CDernek* isim alanına ek olarak *Project* isimli bir isim alanını daha tanımlanmış olsun:

```
namespace CDernek {
    class CDString {

        //…

    };

    //…

}


namespace Project {
    class CDString {

        //…

    };

}
```

Her iki isim alanında da, tanımlanan sınıfın ismi *CDString* olarak seçilmiştir. Bilinirlik alanları farklı olduğuna göre iki sınıfın isminin aynı olması geçerlidir.

C++'da C'de olmayan iki bilinirlik alanı kuralının daha yer aldığını hatırlatalım. Bu bilinirlik alanlarından birincisinin sınıf bilinirlik alanı olduğunu daha önceki konularda değinilmişti. İsim alanı bilinirlik alanı da C dilinde olmayan C++ dilinin eklediği yeni bir bilinirlik alanıdır.

Çözünürlük İşleci ve Nitelenmiş İsim

Programcı tarafından bildirilen bir isim alanının elemanlarına, isim alanı dışında erişmek için iki terimli araek konumunda çözünürlük işleci kullanılır. Bu biçimde yazılmış isme "nitelenmiş isim" (*qualified name*) denir. Bir isim alanı içinde bildirilmiş isme, isim alanı dışında ancak nitelenmiş isimle erişilebilir:

```
namespace Cdernek {
    class CDString {

        //…

    }

};
```

```
void foo(CDString &r);             // Geçersiz
void func(Cdernek::CDString &r)        // Geçerli
```

Yukarıdaki örnekte, *CDernek* isim alanı içinde tanımlanan *String* sınıfı, *CDernek* isim alanı dışında nitelenmiş isim olarak kullanılıyor. Global düzeyde bildirilen *func* işlevinin parametre değişkeni

```
CDernek::CDString &
```

türündendir. Bir isim alanı içinde bildirilen isim, söz konusu isim alanı içinde gizlenir. Bu ismi kullanmak için derleyiciye, söz konusu ismi hangi isim alanında araması gerektiği bildirilmelidir. Derleyiciye bu bilgi verilmez ise, derleyici bu ismi önce bulunulan bilinirlik alanı içinde arar. Orada bulamazsa aramayı bilinirlik alanını kapsayan diğer bilinirlik alanlarında sürdürür. Yukarıdaki örneği değiştirelim:

```
class CDString {
    //…
};

namespace CDernek {
    class CDString {
        //…
    };
}

void foo(CDString &);
void func(CDernek::CDString &)
```

Yukarıda bildirilen *foo* işlevinin paramesi, global isim alanında bildirilen *CDString* sınıfı türünden iken, *func* işlevinin parametresi *CDernek* isim alanında bildirilen *CDString* türündendir.

İsim alanında bildirilen bir ismin, nitelenmeden doğrudan kullanılmasını sağlayan araçlar da vardır. Bu araçlar "using bildirimi", "using namespace komutu" ve "argumana bağlı isim arama"dır. Bu araçlardan birinin kullanımıyla, isim alanının bir elemanına çözünürlük işleci olmaksızın doğrudan erişilebilir.

Global İsim Alanı

Tüm blokların dışında kalan bölge C++"da *"global isim alanı"* (*global namespace*) olarak isimlendirilir. Global isim alanında bildirilen isimler diğer doğrudan kullanılabilir. Aslında çözünürlük işlecinin tek terimli biçimi,

global isim alanındaki bir isme erişmeyi sağlar. Normal olarak global isim alanındaki bir isme doğrudan erişilebileceği için, bu işleci kullanmak zorunlu değildir. Ancak bazı durumlarda global isim alanındaki isim, iç bilinirlik alanlarındaki bir isim tarafından maskelenir. Bu durumda global isim alanındaki isme erişebilmek için, tek terimli çözünürlük *işlecini* kullanmak zorunludur. Aşağıdaki örneği inceleyin:

```
#include <iostream>

const int max = 1000;      // global isim alanındaki max

namespace CDernek {

    const int max = 500;   //Cdernek isim alanındaki max

};


int main()

{

    int max = 100;                          //blok bilinirlik alanındaki max


    std::cout << max << std::endl          //blok bilinirlik alanındaki max
    std::cout << ::max << std::endl;       //global isim alanındaki max
    std::cout << Cdernek:: max << std::endl; //Cdernek isim alanındaki max
    return 0;

}
```

İsim Alanları Eklemeli Bir Yapıya Sahiptir

Daha önce bildirilen bir isim alanına, aynı kaynak dosya içinde ya da başka bir kaynak dosya içinde ekleme yapılabilir. Yani isim alanı elemanları, isim alanı içinde bir defada bildirilmek zorunda değildir. Aşağıdaki örneği inceleyin:

```
namespace CDernek {
    class CDString {

        //…

    };

}


namespace CDernek {
    class CDVector {

        //…
```

```
    };

}
```

Aynı dosya içinde yapılan yukarıdaki tanımlamalar geçerlidir. Derleyici bir isim alanı tanımı görünce, önce *namespace* anahtar sözcüğünü izleyen ismin, daha önceden tanımlanmış olan bir alana ilişkin olup olmadığını araştırır. Eğer bu isimli bir isim alanı daha önce tanımlanmışsa, ikinci bildirimdeki isimleri otomatik olarak ilk isim alanı tanımıyla birleştirir. Yani daha önceden tanımlanmış olan bir isim alanına, kaynak kod içinde başka bir noktada ekleme yapılabilir. Yukarıdaki örnekte hem *CDString* sınıfı hem de CD*Vector* sınıfı *CDernek* isim alanındadır. Hata oluşturan bir durum söz konusu değildir.

İsim alanlarının eklemeli bir yapıya sahip olması özellikle kütüphane oluştururken programcının işine yarar. Bir modüle ilişkin bildirimler, yani modülün ara yüzü bir başlık dosyası içinde tanımlanan bir isim alanı içinde yer alırken, modülün kodlaması (*implementasyonu*) başka bir kaynak dosyada, yine aynı isim alanına eklenebilir:

```
//cdstring.h dosyası
namespace CDernek {

    class CDString {

        //
    public:

        String operator+(const CDString &);

        //…

    };

}


//cdstring.cpp dosyası
#include <string.h>


namespace CDernek {

    CDString CDString:: operator+(const CDString &)

    {

    }

}
```

İsim alanlarının eklemeli bir yapıya sahip olması özelliğinden faydalanılarak, bir isim alanı içinde sunulan bir kütüphanenin arayüzü birden fazla başlık dosyasına bölünebilir. Buna en iyi örnek *std* isim alanıdır.

std İsim Alanı

C++"ın standart kütüphanesinin bütün bileşenlerinin bildirimleri ve tanımlamaları *std* ismi verilen bir isim alanı içinde yapılmıştır. *iostream*, *vector*, *string* gibi standart başlık dosyalarının herhangi birinde bildirilen sınıflar, işlevler, nesneler, şablonlar *std* isim alanı içinde bildirilmiştir.

Standart kütüphane içinde bildirilen bir işleve ya da sınıfa erişmek için, normal olarak sınıfın ya da işlevin ismi, yer aldığı isim alanının ismiyle nitelenmelidir:

```
std::cout
std::string
std::cin
```

Bu isimleri nitelemeden kullanabilmek için *using* bildirimi ya da *using namespace* komutu kullanılacağı gibi argumana bağlı arama da (*Koenig lookup)* yapılabilir. *std* isim alanı bildirimleri içinde değişiklik yapmak, yani bu isim alanına eklemeler yapmak ya da bu isim alanından bazı bildirimleri silmek tanımsız davranıştır.

İçsel İsim Alanları

Bir isim alanı içinde başka bir isim alanı bildirilebilir. Böylelikle içsel *(nested)* isim alanları oluşturulabilir. İçsel isim alanları kütüphanedeki kodların düzenlenmesini iyileştirmek için kullanılabilir.

```
namespace CCernek {
    namespace CStringLib {

        //...

    }

    namespace CMathLib{

        //...

    }

}
```

İçsel bir isim alanındaki elemanların isimleri, o isim alanı içinde gizlenir. İçsel isim alanındaki isimlere

```
kapsayan isim alanı ismi :: içsel isimalanı ismi :: isimalanı elemanı ismi
```

biçiminde erişilir.

İçsel bir isim alanı, başka bir isim alanında yer alan yeni bir bilinirlik alanı yaratır. Derleyicinin isimleri çözümlemesi sırasında, içsel isim alanları içsel bloklar gibi ele alınır. İçsel isim alanında bir isme rastlandığında, bu ismin bildirimi, önce içsel isim alanında, daha sonra sırasıyla içten dışa doğru kapsayan isim alanlarında aranır. İsim kapsayan isim alanlarında da bildirilmemişse son olarak isim global isim alanında aranır.

İsim Alanı Elemanlarının Tanımlamaları

Bir isim alanı elemanının tanımının yine aynı isim alanı içinde yapılabilir.

Tanım isim alanı dışında da yapılabilir. Bu durumda tanımlanacak isim alanı elemanı, tanımlama sırasında nitelenmiş ismiyle kullanılmalıdır:

```
namespace CDernek {
    namespace CDStringLib {

        class CDString {
            int size;

        public:

            int get_size() const {return size;}

        };

        CDString operator+(const CDString &, const CDString &);

    }

}
```

Yukarıdaki örnekte *CDernek* isimli isim alanı içinde *CDStringLib* isimli başka bir içsel isim alanı tanımlanıyor. *CDStringLib* isim alanı içinde tanımlanan *CDString* isimli sınıfın *get_size* isimli üye işlevi yine aynı isim alanı içinde tanımlanıyor. *CDString* sınıfının içinde *operator+* işlevi bildiriliyor. Ancak bu işlevin tanımı *CDStringLib* isimli isim alanı içinde yapılmıyor. Bu işlevin global isim alanında aşağıdaki biçimde tanımlandığını düşünelim:

```
Cdernek::CDStringLib::CDString

Cdernek::CDStringLib::operator+(const CDString &r1, const CDString &r2)

{

    CDString result;

        //...
    return result;

}
```

İşlevin geri dönüş değerinin türünün yazıldığı yer, *Cdernek::CDStringLib* isim alanı ile belirlenen bilinirlik alanı dışındadır. Bu yüzden işlevin geri dönüş değerinin türü nitelendirilmiş isim olarak yazılmalıdır. Bu yüzden global isim alanında bu tür

```
Cdernek::CDStringLib::CDString
```

biçiminde yazılır. İşlevin tanımında işlevin ismi de nitelenmiş isim olarak yazılır:

```
Cdernek::StringLib::operator+
```

Ancak işlevin parametre ayracı içi ile işlevin ana bloğu *Cdernek::StringLib* isim alanının bilinirlik alanındadır. Bu yüzden işlevin parametre değişkenleri olan *r1* ve *r2* ile işlev içindeki yerel *result* değişkeninin tür bilgileri doğrudan

```
CDString
```

olarak yazılabilir. Zira isim alanı elemanları, isim alanı bilinirlik alanı içinde nitelenmemiş isimleriyle doğrudan kullanılabilir.

Bir isim alanı elemanın tanımı, isim alanı dışında her yerde yapılamaz. Tanım isim alanının dışında yapılacaksa, ancak bildirimin yapıldığı isim alanını kapsayan isim alanlarından birinde yapılabilir. Yukarıdaki örnekte *operator+* işlevinin tanımı ya *Cdernek* isim alanı içinde ya da global isim alanı içinde yapılmalıdır. İşlevin tanımı *CDernek* isim alanı içinde aşağıdaki biçimde yapılabilirdi:

```
namespace Cdernek {
    //...
    namespace CDStringLib {
        class CDString {

            int size;

            //...
        public:

            int get_size() const {return size;}

            //...

        };

        CDString operator+(const CDString &, const CDString &);

    }

    CDStringLib::CDString

    CDStringLib::operator+(const CDString &r1, const CDString &r2)

    {

        //...
```

```
    }
}
```

Bu kez işlevin tanımı *CDStringLib* isim alanının dışında yapılmasına karşın, *Cdernek* isim alanı içinde yapılıyor. İşlevin geri dönüş değerinin türünün ve işlev isminin

```
String_Lib::String
String_Lib::operator+
```

biçiminde yazıldığına dikkat edin.

using Bildirimi

Bir isim alanı elemanına ulaşmak için her defasında bu elemanın ismini nitelenmiş ismi yazmak programcı açısından çok zahmetlidir. *using* bildirimiyle isim alanı elemanına nitelenmemiş ismi ile doğrudan erişilebilir.

using bir anahtar sözcüktür. Bu anahtar sözcüğü isim alanı elemanının nitelendirilmiş ismi izler. Bildirim sonlandırıcı atomla sonlanır.

```
using CDernek::CDString;
```

Yukarıdaki bildirimden sonra, *CDernek* isim alanı içinde tanımlanan *CDString* sınıfı

```
CDernek::CDString
```

biçiminin yanısıra, doğrudan

```
CDString
```

biçiminde kullanılabilir. Yani *using* bildiriminden sonra *CDString* ismi kullanıldığında derleyici bu ismi

```
CDernek::CDString
```

olarak ele alır.

using bildirimi ile bildirilen isim, bildirimin yapıldığı bilinirlik alanına eklenir.

using bildirimi de diğer bildirimlerin ortak özelliklerini taşır. Bu bildirimin de bir bilinirlik alanı vardır. *using* bildirimi yerel olarak yapılabileceği gibi, global isim alanı içinde ya da başka bir isim alanı içinde de yapılabilir. Diğer tüm bildirimlerde olduğu gibi *using* bildirimiyle bilinirlik alanına eklenen isim

1. Eklendiği bilinirlik alanında tek olmalıdır.
2. Daha geniş bilinirlik alanındaki aynı ismi maskeler.
3. Daha dar bilinirlik alanındaki aynı isim tarafından maskelenir.

Aşağıdaki örneği dikkatle inceleyin:

```
namespace Mynamespace {

    int a = 1, b = 2, c = 3;

    //diger bildirimler

}


int b = 0;

int main()

{

    using Mynamespace::a;
    a = 10;



    using Mynamespace::b;
    b = 20;



    int c;

    //using Mynamespace::c;        //Geçersiz!

}


void func()

{

    int y = a;

}
```

main işlevi içinde yapılan `using`

`Mynamespace::b`

bildiriminden sonra yapılan `b =`

`10`

atamasıyla erişilen *b, Mynamespace* isim alanı içinde tanımlanan *b* olur. Yani *main* işlevinin blok bilinirlik alanına, *using* bildirimiyle eklenen *b* ismi, global değişken olan *b* değişkenini maskeler. *main* işlevi içinde *c* isimli bir yerel değişkenin tanımlanmasından sonra yapılan

```
using Mynamespace::c
```

bildiriminin hata oluşturduğunu görüyorsunuz. Aynı bilinirlik alanında aynı ismin kullanılamayacağını hatırlayın. *using* bildiriminden önce, blok bilinirlik alanında zaten *c* isimli bir yerel değişken vardır. Bu durumda bilinirlik alanına ikinci bir *c* isminin sokulmaya çalışılması geçersizdir. Son olarak *main* işlevinden sonra tanımlanan *func* işlevine bir göz atalım. İşlev içinde tanımlanan bir yerel değişken olan *y* isimli değişkene *a* değişkeninin değeri atanmış. Ancak *a* isimli bir nesne bu noktada görünür değil. *using* bildiriminin de bir bilinirlik alanı olduğunu hatırlayalım.

```
using Mynamespace::a;
```

bildirimi yerel düzeyde, yani *main* işlevinin içinde yapıldığından yalnızca *main* işlevi içinde nitelenmeden kullanılabilir. *func* işlevinin ana bloğu içinde *a* ismi nitelenmiş isim olarak yani *Mynamespace::a* biçiminde kullanılmalıdır.

Ancak *using* bildirimi yerel düzeyde değil de global düzeyde yapılsaydı, bu erişim de geçerli olurdu. Aşağıdaki kodu inceleyin:

```
namespace Mynamespace{

    int a = 1, b = 2, mk = c;

    //diğer bildirimler

}


using Mynamespace::mi;

int main()

{

    a = 10;


    return 0;

}
```

```
void func()
{
    int y = a;
}
```

using bildirimi ile, isim alanı elemanının kullanılması daha kolay hale gelir. *using* bildirimi ile, bir defada yalnızca isim alanının bir elemanı ilgili bilinirlik alanına katılabilir. *using namespace* komutuyla bir defada birden fazla ismin nitelenmeden kullanılması sağlanabilir.

using namespace Komutu

Bir isim alanı içinde tanıtılan isimlerin her birine nitelenmemiş isimle erişebilmek amacıyla, her isim için ayrı bir *using* bildirimi yapılabilir. Ancak çok sayıda isim için *using* bildiriminin teker teker yapılması zaman alıcıdır. Böyle bildirimler programın okunabilirliğini de bozar. Bir isim alanındaki isimlerin hepsinin tek bir komutla nitelenmemiş isim olarak kullanılabilmeleri mümkün kılınmıştır.

Bir *using namespace* komutu, *using* ve *namespace* anahtar sözcükleriyle başlar. Bu sözcükleri daha önce tanımlanmış bir isim alanının ismi izler:

```
using namespace CDernek;
using namespace std;
```

Daha önce tanımlanmış bir isim alanına ilişkin olmayan bir ismin kullanılması geçersizdir. *using namespace* komutuna konu isim alanındaki tüm isimler, nitelenmemiş isimler olarak kullanılabilir hale gelir.

using namespace komutu, isim alanı elamanlarının, sanki bu elemanlar isim alanının tanımının bulunduğu yerde, ancak isim alanının dışında tanımlanmış gibi görülmesini sağlar. Aşağıdaki örneği inceleyin:

```
namespace Mynamespace{
    int a = 1, b = 2, c = 3;
    //diger bildirimler
}

void func();
int b = 0;

int main()
```

```
{
    using namespace Mynamespace;
    a = 10;

    //b = 20;      //Geçersiz
    Mynamespace::b = 100;

    ::b = 178;

    int c;

    c = 30;    //yerel c


    return 0;
}


void func()
{
    //int y = a;
}
```

main işlevinin ana bloğunun başında *using namespace* komutu yer alıyor. Komutun görülür olduğu kaynak kod aralığında, *Mynamespace* isim alanı içindeki tüm isimler adeta isim alanının dışında, yani global bölgede tanımlanmış gibi görülebilir duruma gelir.

```
a = 20;
```

ataması geçerlidir. Bu atamayla *Mynamespace* isim alanı içinde tanımlanan *a* değişkenine değer atanır.

```
b = 20;
```

ataması derleme zamanında çift anlamlılık hatası oluşturur. Zira bu noktada hem global *b* değişkeni hem de *using namespace* komutunun etkisiyle, *Mynamespace* isim alanı içindeki *b* değişkeni görülür durumdadır. Hata, yalnızca *b* nesnesinin nitelenmemiş isim olarak kullanılması durumunda oluşur. *main* işlevinin ana bloğu içinde *b* nesnesi hiç kullanılmaz ise ya da *b* nesnesi nitelenmişle kullanılırsa, çift anlamlılık hatası oluşmaz:

```
//...
Mynamespace::b = 100;
::b = 178;
```

main işlevi içinde *c* değişkeninin tanımlanması geçerlidir. Zira isim alanının içinde tanımlanan *c* ile, *main* bloğu içinde tanımlanan *c* değişkenlerinin bilinirlik alanları farklıdır.

```
c = 30
```

ataması yerel *c* değişkenine yapılmış olur.

using namespace komutunun da bir bilinirlik alanı vardır. Yukarıdaki örnekte komut *main* işlevi içinde verildiğinden komutun bilinirlik alanı yalnızca *main* işlevini kapsar. *main* işlevini izleyen *foo* işlevi içinde, *a* isminin nitelenmeden kullanılması geçersizdir.

using namespace komutunun kullanımı son derece kolaydır. Tek bir komutun kullanılmasıyla, isim alanı içindeki tüm isimler doğrudan kullanılabilir hale gelir. Bu kolay bir çözüm gibi görünse de, gereksiz ya da aşırı kullanımı da başka sorunlara yol açabilir. Birden fazla kütüphane kullanılıyorsa kütüphanelerdeki isimleri kısa biçimleriyle kullanmak için her bir isim alanı için *using namespace* komutu kullanıldığında, global isim kirlenmesi sorunu yine ortaya çıkar.

```
//header1.h
namespace HDR1 {

    void foo();

    //...

}
```

```
//header2.h
namespace HDR2 {

    void foo();

    //...

}
```

```
//app.cpp

#include "header1.h"
#include "header2.h"


using namespace HDR1;
using namespace HDR2;
```

```
int main()

{

    foo(); //çift anlamlılık hatası

    //...

}
```

Bir Kütüphanenin İsim Alanı İçine Alınması

İsim alanları C++ dilinin standartlaştırılması sürecinde dile katılan son araçlardan biridir. Standartlar öncesi geliştirilen derleyiciler isim alanlarını desteklemedikleri için eskiden yazılmış bir çok kütüphane bir isim alanı içinde bildirilmemiştir. İsim alanı içine alınmamış kütüphanelerin kullanımı isim çakışması riskini artırır.

Bir isim alanı içinde yapılmamış bildirimlerin, daha sonradan bir isim alanı içine alınmak istendiğini düşünelim. Örneğin daha önce yazılan *cdate.h* isimli başlık dosyasındaki bildirimlerin *CDernek* isim alanına katılmak istendiğini düşünelim:

```
//cdate.h

namespace CDernek {
    class CDate {

        //...

    };

    //diğer bildirimler

}
```

Ancak bu durumda *CDate* sınıfını kullanan kodların bulunduğu, örneğin *app1.cpp* isimli bir dosyada *cdate.h* isimli başlık dosyası yeni biçimiyle kaynak dosyaya *include* önişlemci komutuyla eklendiğinde artık *cdate.h* içinde bildirilen isimler doğrudan kullanılamaz. *cdate.h* içinde yapılan her bildirim artık *CDernek* isim alanı elemanı durumuna geldiğine göre, bu elemanlar ancak nitelenmiş isimleriyle kullanılabilir. *app1.cpp* dosyasında fazlaca bir değişiklik yapmamak için bu dosyanın başına bir *using namespace* komutu yerleştirilebilir:

```
//app.cpp

#include "cdate.h"


using namespace CDernek;

//...
```

İsim Alanı Eşismi

İsim alanı isimleri de global bölgede tek olmak zorundadır. Bu yüzden isim alanı isimleri genellikle uzun seçilir. Nitelenmiş isim kullanılırken, isim alanı elemanını niteleyen ismin sürekli olarak yazılması istenmeyebilir. Daha kısa isimler kullanabilmek için, isim alanı ismi yerine kullanılabilecek başka -daha kısa- isimler yaratılabilir. Bu isimalanı eşismi bildirimi ile sağlanır.

İsim alanı eşismi bildirimi *(namespace alias) namespace* anahtar sözcüğü ile başlar. Anahtar sözcüğü, isim alanı isminin yerine geçecek sözcük izler. Daha sonra atama işleci yer alır. Atama işlecinin sağ tarafına daha önce tanımlanan bir isim alanının ismi yazılır. Bildirimi her zaman olduğu gibi sonlandırıcı atom tamamlar.

```
namespace CSD = C_ve_Sistem_Programcilari_Dernegi;
```

Bu bildirimden sonra artık *C_ve_Sistem_Programcilari_Dernegi* isim alanı isminin kullanılabileeği yerlerde *CSD* ismi kullanılabilir.

Bir isim alanı eşismi, içsel bir isim alanı için de kullanılabilir:

```
namespace CSDSTR = C_ve_Sistem_Programcıları_Dernegi::CDStringLib;
```

Biri isim alanının birden fazla eşismi olabilir.

İsim alanı eşismi, projelerde sürüm denetiminde de kullanılır. Başarılı yazılım projelerinin ömrü ürün piyasaya verildikten sonra da sürer. Bazı ürünlerin sürekli yeni sürümleri üretilir. Bir programın yeni ve eski sürümleri söz konusu olduğunda, iki sürümün de ortak olarak kullandığı kod bileşenleri vardır. Ancak bazı kod bileşenleri ise her sürüm için farklıdır. İsim alanı eşisimleri farklı sürümler için dinamik isim alanlarının yaratılmasında kullanılabilir:

```
namespace Ver_1_0 {
    class File {

        //…

    };

    class Usb {

        //…

    };

}


namespace Ver_2_0 {
    class File {

        //…
```

```
    };

    class Usb {

        //…

    };

}


int main()

{

    namespace current = Ver_2_0;
    using current::File;

    using current::Usb;
    File f;

    Usb u;

    //…
    return 0;

}
```

Yukarıdaki kodu inceleyelim: Ayrı sürümlerde kullanılmak üzere *Ver_1_0* ve *Ver_2_0* isimli iki ayrı isim alanı tanımlanıyor. *main* işlevi içinde yapılan isim alanı eşismi bildirimiyle, *current* isminin *Ver_2_0* isim alanının yerine geçmesi sağlanıyor. Daha sonra yapılan *using* bildirimleriyle *current* isim alanının *File* ve *Usb* sınıf isimlerinin nitelenmeden kullanılması sağlanıyor. *main* işlevi içinde yer alan

```
namespace current = Ver_1_0;
```

bildiriminin

```
namespace current = Ver_2_0;
```

bildirimi ile değiştirilmesiyle, farklı *File* ve *Usb* sınıflarının kullanılması sağlanabilir.

İsimsiz İsim Alanı

Her kaynak dosya içinde, isimsiz bir isim alanının bulunduğu varsayılır. Bu isim alanına programcı isterse ekleme yapabilir:

```
namespace {

    void func();
    int g;


    class A{

        //…

    };

    //…

}
```

Bu isim alanındaki isimlere nitelenmemiş isimlerle, kendi tanımlandıkları kaynak dosya içinde doğrudan erişilebilir. Yukarıdaki örnekte isim alanı bildiriminin görülebildiği bir kaynak kod noktasında

```
int main()

{

    A a;

    //..
    return 0;

}
```

Yukarıdaki *main* işlevi içinde *A* ismi nitelenmeden (*unqualified*) kullanılıyor. *A* sınıfı isimsiz isim alanı içinde tanımlanmıştır. İsimsiz isim alanının her kaynak dosya için tek olacağı güvence altına alınmıştır. Yani böyle bir isim alanı içinde bildirilen isimler doğrudan iç bağlantıya (*internal linkage*) sahiptir.

İsimsiz isim alanı ne işe yarar? C dilinde global varlıkların iç bağlantıya sahip olması için *static* anahtar sözcüğüyle bildirilmeleri gerektiğini hatırlayın. Oysa C++ da iç bağlantıya sahip yani yalnızca kendi modülünde bilinen global varlıklar için, *static* anahtar sözcüğünün kullanılması “eskimiş” (*deprecated)* olarak kabul edilmiştir. Önerilen araç, isimsiz isim alanı oluşturulması ve iç bağlantıya sahip global varlıkların bu isim alanı içinde bildirilmesidir.

Global düzeyde, *static* anahtar sözcüğü ile işlevler ve değişkenler için kullanılabilir. Ancak isimsiz isim alanı içinde programcı tarafından yaratılan türlerin de tanımı yapılabilir.

Koenig İsim Araması

Bir işlev çağrısıyla karşılaşan derleyici, hangi işlevin çağrıldığını anlamak için işlev çağrısının yapıldığı noktada görülebilir işlevleri araştırır.

```
int main()
```

```
{
    func(object)
}
```

Yukarıdaki kod içinde bir *using* bildirimi ya da *using namespace* komutu kullanılmadığına göre, normal olarak *func* işlevinin bir isim alanı içinde aranmaması gerekir. Standartlarla dile eklenen *Koenig* isim araması (*Andrew Koenig isminden*) bu kuralı değiştirir. İşleve gönderilen arguman olan *object* eğer bir isim alanı içinde bildirilen sınıfa ilişkin bir nesne ise, bu sınıfın bildirimini kapsayan isim alanı içinde de *func* isimli uygun bir işlevin varlığı araştırılır. Bu kurala "*Koenig* isim araması" ya da "argumana bağlı isim arama" denir.

Aşağıdaki kodu derleyerek çalıştırın:

```
#include <iostream>

using namespace std;

namespace NMSPC {
    class A {
       //...
    };
    void foo(A a)
    {
        cout << "NMSPC::foo(A)" << endl;
    }
}

int main()
{
    NMSPC::A den;
    foo(den);

    return 0;
}
```

Koenig isim araması otomatik olarak uygulanır. Yani *Koenig* isim aramasını etkin kılmak için bir bildirim ya da komut kullanılmak gerekmez. Bu özelliği devre dışı bırakmak da mümkün değildir. *Koenig* isim aramasının yapılması, bazı durumlarda çift anlamlılık hatasının oluşmasına da yol açabilir:

```
namespace NMSPC {
    class A {

        //...

    };

    void foo(A);

}


void foo(NMSPC::A);


int main()

{

    NMSPC::A a;

    foo(a);    //Çift anlamlılık hatası


    return 0;

}
```

Yukarıdaki kod parçasında *main* işlevi içinde çağrılan *foo* işlevi, hangi işlevdir? *NMSPC* isim alanı içinde bildirilen mi, global isim alanı içinde bildirilen mi? *Koenig* isim araması söz konusu olmasaydı, şüphesiz çağrılan işlev global isim alanında bildirilen *foo* işlevi olurdu. Ancak *Koenig* isim araması *NMSPC* isim alanı içindeki *foo* işlevini de görülebilir kıldığı için, derleme zamanında çift anlamlılık hatası oluşur.

İsim Alanlarında Yapılan Arkadaşlık Bildirimleri

Arkadaşlık bildirimi aynı zamanda arkadaşlık verilen işlev için bir prototip bildirimidir. Bir isim alanı içinde bir işleve arkadaşlık verildiği zaman arkadaşlık bildirimi isim alanı içinde yapılırsa, işlev de isim alanında bildirilmiş olur. Böylece arkadaşlık verilen işlev de isim alanının bir elemanı olur.

```
namespace CDernek{
    class Sample {

        int x;

        friend void foo();

    };

    //...

}
```

```
void foo()
{
    CDernek::Sample sam;
    sam.x = 10;      //Geçersiz
}
```

Yukarıda *Cdernek* isim alanı içinde tanımlanan *Sample* isimli sınıf içinde yapılan

```
friend void foo();
```

bildirimi aynı zamanda bu işlevin bildirimi olduğundan arkadaşlık verilen işlev

```
void Cdernek::foo()
```

işlevidir. Eğer *Cdernek* isim alanı içinde global isim alanında bildirilmiş bir işleve arkadaşlık verilmek istenseydi arkadaşlık bildirimi

```
friend void ::foo();
```

biçiminde yapılmalı ve isim alanı tanımından önce global *foo* işlevinin bildirimi yapılmalıydı.

Global isim alanında tanımlanan *foo* isimli işlev içinde *Sample* sınıfının *private* elemanına erişilmeye çalışıldığını görüyorsunuz. *CDernek::Sample* sınıfı arkadaşlığı global *func* işlevine vermemiştir. Bu nedenle

```
sam.x
```

erişimi geçerli değildir.

İsim Alanları Ne Zaman Kullanılmalı

İsim çakışması olasılığının büyük projeler söz konusu olduğunda daha da artacağı açıktır. Yüzlerce kaynak dosyanın olduğu bir projede hem isim çakışması riski hem de isim çakışmasının neden olacağı ek maliyet yüksektir. Büyük projelerde isim alanlarının kullanılması kaçınılmazdır. Ancak tek bir programcının geliştirdiği çok fazla dosyaya sahip olmayan projelerde isim alanlarının kullanılması fazla bir getiri sağlamaz. Ne de olsa

isim çakışması olması durumunda programcı kolayca bazı isimleri değiştirerek çakışma durumunu ortadan kaldırabilir.

İsim Alanlarının Maliyeti

İsim alanı kullanımı programın çalışma zamanı söz konusu olduğunda, çalışan programın hızı ya da kullandığı bellek açısından ek bir yük getirmez. İsim alanlarına ilişkin kontroller programın derleme zamanında yapılır.

TÜRETME

Türetme (*inheritance*) nesne yönelimli programlama tekniğinin en önemli araçlarındandır. Türetme yoluyla önceden tanımlanmış olan bir sınıfın üzerinde değişiklik yapmadan,sınıfın işlevleri genişletilebilir. Önceden tanımlanmış olan sınıflar türetme işlemi ile gereksinimlere göre biçimlendirebilir. Türetme yoluyla bir sınıf başka bir sınıfın var olan özelliklerini alarak, o sınıf türünden bir nesneymiş gibi kullanılabilir. Türetme, nesne yönelimli programlamanın olmazsa olmaz araçlarındandır.

Dış dünyadaki nesneler arasında bazı ilişkiler kurmamız, o nesneleri soyutlayarak algılamamızı kolaylaştırır. Bu ilişkilerden birincisi İngilizcede *"has a"* ilişkisi diye bilinen ilişkidir. Bir nesne başka bir nesneye sahiptir. *"Bilgisayarın monitörü var"*, *"Arabanın 4 tekerleği var"* gibi cümlelerde, bilgisayarın ya da arabanın nelere sahip olduğunu görmek, nesneleri temel bileşenlerine ayırmak, bu nesneleri daha iyi soyutlamamıza yardımcı olur. Şüphesiz araba ya da bilgisayar belki yüzlerce farklı parçaya sahiptir. Ancak zaten soyutlama(*abstraction)*, bazı ayrıntıları görmezden gelip temel nitelikler üstünde odaklanarak, genel kavrama derecesini artırmaya dayanır.

Soyutlamada ve algılamada kullandığımız ikinci teknik ise ingilizcede *"is a"* ilişkisi diye bilinir. *"Aslan bir hayvandır."* Burada verilen ana bilgi şudur. *"Her aslan bir hayvandır."* Her aslan hayvanların genel özelliklerini taşır. Ancak aslan hayvanların genel özellikleri dışında bazı başka özelliklere de sahiptir. Aslan yırtıcı bir hayvandır. Aslan etobur bir hayvandır. Aslan hayvan kavramının daha özelleştirilmiş bir üyesidir. Şüphesiz bunun tersi doğru değildir. Yani her hayvan aslanların genel özelliklerini taşımaz. Her hayvan yırtıcı değildir. Her hayvan etobur değildir. Ancak öznesi hayvan olan bir cümle içinde özneyi değiştirip aslan yaparsak, cümle yanlış olmaz. Örneğin, *"Hayvanlar su içer."* cümlesi doğru olduğu gibi *"Aslanlar su içer."* cümlesi de doğrudur.

"is a" ilişkisi bir *ana tür-yan tür* ilişkisidir. Bu ilişki algılamayı güçlendirmek için kademelendirilebilir. Böylece bir tür ağacı oluşturulabilir. Bazı hayvanlar et yiyerek beslenir. Bu hayvalara etobur hayvanlar diyelim. Her etobur hayvan bir hayvandır. Aslan etobur bir hayvandır. Bazı etobur hayvanların nesli tükenmektedir. Bazı etobur hayvanlar evcilleştirilebilir. Kedi evcilleştirilebilen bir etobur hayvandır.

Nesne yönelimli programlama, dış dünyanın daha iyi modellenmesine ve programın problem düzleminde soyutlanarak gerçekleştirilmesine dayanır. Gerçek dünyayı nesnelere ve bu nesnelerin ilişkileri yoluyla soyutluyorsak, program yazarken de soyutlamayı bu biçimde yapmamız işimizi kolaylaştırır.

C++ dilinde "*has a"* ilişkisi bileşik nesne oluşturma yoluyla (*composition*) sağlanır. "Bileşik nesneler" isimli bölümde bu konuyu incelemiştik. *"is a"* ilişkisinin kurulmasına yardımcı olan araç ise türetmedir.

Bir sınıfın kendisini değiştirmeden işlevleri genişletilmek istenirse, yani o sınıfa çeşitli üye işlevler ve elemanlar eklenmek istenirse bu farklı biçimlerde yapılabilir. Örneğin eski sınıfın farklı bir isimde yeni bir kopyası oluşturulabilir, eklemeler yeni sınıf üzerinde yapılabilir. Örneğin, *A* isimli bir sınıfın *B* isimli bir kopyası oluşturulup bütün eklemeler *B* sınıfı üzerinde yapılabilir. Böylece hem *A* sınıfı bozulmamış olur hem de genişletilmiş bir *B* sınıfı elde edilir. Ancak bu yöntemin önemli bir dezavantajı vardır. Bu yöntemde *A* sınıfının

bütün üye işlevlerin *B* sınıfı için yeniden yazılması gerekir. Yani bu yöntem aynı işlevlerin farklı sınıf isimleriyle yeniden tanımlanmasını gerektireceğinden kodun büyümesine yol açar. İkinci önemli bir dezavantajı da sınıfın kodlarında değişiklik yapılması durumunda olur. A sınıfında bir değişiklik yapılması durumunda bu değişiklik yeni oluşturulan *B* sınıfına yansımaz. Belki de en önemlisi, böyle bir değişikliğin yapılabilmesi için sınıfın kaynak kodlarına gereksinim duyulmasıdır. Elde sınıfın kodlama dosyası değil de yalnızca sınıfın arayüzü yani başlık dosyası varsa böyle bir değişiklik yapmak mümkün olmaz.

Bir başka yöntem olarak, bileşik nesne oluşturma yolu seçilebilir. Örneğin önce aşağıdaki gibi bir *A* sınıfı tanımlanır:

```
class A {

    //Var olan işlevler ve elemanlar

};
```

Sonra *B* sınıfı içinde *A* sınıfını bir eleman olarak kullanılır:

```
class B {
private:

    A a;
public:

    //Yeni eklenen işlevler ve elemanlar

};
```

Artık *B* sınıfı türünden tanımlanmış olan *b* nesnesi ile hem *A* sınıfının hem de *B* sınıfının üye işlevleri çağrılabilir. Çünkü *b.a* ifadesi ile *A* sınıfına erişilerek bu ifadeyle *A* sınıfının üye işlevleri çağrılabilir. Bu tasarım biçimi işlev yinelemesine yol açmamakla birlikte erişim bakımından sorunludur.

Nesne yönelimli programlama tekniğinde ağırlıklı kullanılan araç türetmedir. *C++* dilinde önceki sınıfı bozmadan işlev genişletmek amacıyla "türetme" aracı kullanılmaktadır.

Türetme burada açıklanan iki yöntemden çok daha etkin ve geniş bir araçtır.

Türetme İşleminin Genel Biçimi

Bir sınıf başka bir sınıftan türetilir. Kendisinden türetme yapılan sınıfa taban sınıf (*base class*), türetilmiş olan sınıfa da *türemiş sınıf (derived class)* denir.

Türetme işleminin genel biçimi şöyledir:

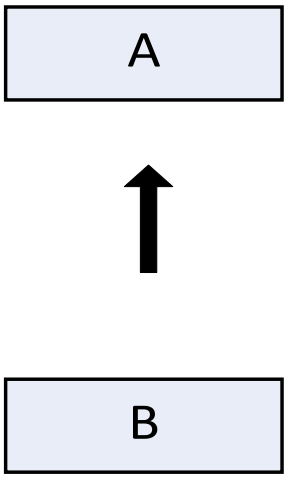
```
<class>    <türemiş sınıf ismi :>  [türetme biçimi]   <taban sınıf ismi>
<struct>                           <public>
                                   <protected>
                                   <private>
{
        sınıf elemanlarının bildirimi
};
```

class ya da *struct* anahtar sözcüğünden sonra türemiş sınıf isminin belirtildiğine, daha sonra *":"* atomu ile türetme biçimi ve taban sınıf isminin yazıldığına dikkat edin. Genel biçimden de anlaşılacağı gibi, türetme biçimi hiç belirtilmeyebilir. Bu durumda sınıflar için *private*, yapılar için *public* türetmesinin yapıldığı kabul edilir. Yani türetme biçimi olarak bir şey yazılmamışsa sınıflar için *private*, yapılar için *public* yazılmış gibi işlem görür.

Aşağıda, örnek bir türetme işleminin sözdizimini görüyorsunuz.

```
class A {
    //…
};


class B : public A {
    // …
};
```

Burada *B* türemiş sınıf *A* ise taban sınıftır. Türetme biçimi olarak *public* türetmesi seçilmiştir. Açıklamayı ve anlatımı kolaylaştırmak amacıyla sınıfların türetilmesi şekilsel olarak da gösterilebilir. Böyle bir şekilde sınıflar daire, elips ya da dikdörtgenlerle gösterilir. Taban sınıf türemiş sınıfın daha yukarısında gösterilir ve türemiş sınıftan taban sınıfa bir ok çizilir. Okun yanına isteğe bağlı olarak türetme biçimi yazılabilir. Örneğin *B* sınıfı *A* sınıfından türetilmişse, bu durumu şekilsel olarak aşağıdaki gibi gösterilebilir:



EMBED Visio.Drawing.11

Yukarıdaki şekil yanlış çizilmedi. Çizilen okun yönü pek çok kişiye olduğu gibi size de ters gelmiş olabilir. Okun yönü yukarıdaki gibi taban sınıfı gösterecek biçimde olmalıdır.

Bundan sonra yalnızca *"türetme*" dediğimizde "*public türetmesi*" anlaşılmalıdır. *"is a"* ilişkisi *public* türetmesiyle sağlanır. Diğer türetme biçimlerinin amacı *"is a"* ilişkisini kurmak değildir. *protected* türetmesi ve *private* türetmesi bazı özel amaçlar için kullanılır. *public* türetmesinde türemiş sınıf taban sınıfın *public* bölümünü olduğu gibi devir alarak kendi müşterilerine (*clients*) sunar. Yani taban sınıfın *public* bölümü türemiş sınıfın *public* bölümünün bir parçasıdır.

Böyle bir türetme işleminde taban sınıfın *protected* bölümü türemiş sınıfın *protected* bölümüymüş gibi kullanılabilir. Taban sınıfın *private* bölümü tam olarak korunmuştur. Taban sınıfın *private* bölümüne türemiş sınıf tarafından hiçbir biçimde erişilemez.

Türemiş Sınıf Nesnelerinin Bellekte Yerleşimi

Bir türemiş sınıf nesnesi kendi içinde taban sınıfın elemanlarını da içerir. Örneğin:

```
class A {
private:

    int a, b;
public:

    // …

};


class B : public A {
private:

    int c, d;
public:
```

```
    // …
};
```

gibi bir türetme işleminden sonra *B* sınıfı türünden bir nesne tanımlanırsa, bu nesne hem *A* sınıfının hem de *B* sınıfının elemanlarını içerir. Yani, *sizeof(B)*, dört tane *sizeof(int)* toplamı kadardır (*DOS* altında *8*, *UNIX* ve *Win32* sistemlerinde *16*). Türemiş sınıf elemanları ile taban sınıf elemanları blok olarak bitişik bir biçimde yerleştirilir. Ancak taban sınıf elemanlarının mı yoksa türemiş sınıf elemanlarının mı daha düşük adrese yerleştirileceği standart olarak belirlenmemiştir. Derleyiciden derleyiciye değişebilir.

Taban sınıf ile türemiş sınıf elemanlarının türemiş sınıf içindeki yerleşimi standart bir biçimde belirlenmemiş olsa da, popüler derleyicilerin tümünde taban sınıf elemanları daha düşük sayısal adreste olacak biçimde yerleşim kullanılır. Yerleşim biçiminin standart bir biçimde belirlenmemiş olması herhangi bir taşınabilirlik problemine yol açmaz.

Türemiş sınıf nesnesinin taban sınıfın elemanlarını da içerdiğini görmek için aşağıdaki örneği yazarak çalıştırın:

```
class Base {
public:
    int b1, b2;
};

class Der : public Base {
public:
    int d1, d2;
};

#include <iostream>
using namespace std;

int main()
{
    Der der_object;
    cout <<  "sizeof(der_object) = " << sizeof(der_object)  << endl;

    return 0;
}
```

Yukarıdaki programda *Der* sınıfı türünden *der_object* isimli bir nesne tanımlanıyor. Bu nesne *Base* taban sınıfının elemanlarını da içerir. Nesnenin uzunluğu *DOS* altında *8* byte, *Win32* ve *UNIX* sistemlerinde ise *16* byte olur. Yani *2 * sizeof(int) + 2 * sizeof(int)* .

Türemiş Sınıflarda Erişim Kuralı

Türemiş sınıf yalnızca kendi elemanlarına ve işlevlerine değil, aynı zamanda taban sınıfın elemanlarına ve işlevlerine de erişebilir. Öncelikle vereceğimiz örnekler için bir sınıf tanımlaması yapalım:

```
#include <iostream>


class Base {
private:

    int b;
protected:

    void base_pro_func();
public:

    void set_base(int);

    void display_base()const;

};


class Der : public Base {
private:

    int d;
protected:

    void der_pro_func();
public:

    void set_der(int);

    void display_der()const;

};


using namespace std;
```

```
void Base::set_base(int x)
{
    b = x;
}

void Base::base_pro_func()
{
    cout << "base_pro_func()" << endl;
}

void Base::display_base() const
{
    cout << b << endl;
}

void Der::set_der(int x)
{
    d = x;
}

void Der::der_pro_func()
{
    cout <<"der_pro_func()" << endl;
}

void Der::display_der() const
{
    cout << d << endl;
}
```

public Türetmesinden Çıkan Sonuçlar

1. Sınıf bilinirlik alanı dışında, türemiş sınıf nesnesi ya da göstericisi yoluyla dışarıdan *nokta* ya da *ok* işlecini kullanarak taban sınıfın *public* bölümüne erişilebilir. Ancak taban sınıfın *protected* ve *private* bölümlerine erişilemez.

Aşağıdaki örneği inceleyelim:

```
int main()
{
    Der d;

    x.set_der(20);      // Geçerli.
    x.set_base(10);     // Geçerli.
    //x.der_pro_func();     Geçersiz. Türemiş sınıfın protected bölümü.
    //x.base_pro_func();  Geçersiz. Taban sınıfın protected bölümünü.
    //x.d = 20;         Geçersiz. Türemiş sınıfın private bölümü.
    //x.b = 10;         Geçersiz. Taban sınıfın private bölümü.

    return 0;
}
```

2. Türemiş sınıf bilinirlik alanı içinde taban sınıfın *public* ve *protected* bölümlerine doğrudan erişilebilir. Ancak *private* bölümüne doğrudan erişilemez. Yani türemiş sınıf üye işlevleri, taban sınıfın *public* ve *protected* bölümlerindeki değişkenleri doğrudan kullanabilir, taban sınıfın üye işlevlerini de doğrudan çağırabilir.

Taban sınıfın *private* bölümü tam olarak korunmuştur. Bu bölüme herhangi bir biçimde erişilemez.

Örneğin *Der* türemiş sınıfının *der_pro_func* üye işlevi içinde taban sınıfın *protected* ve

public bölümlerine erişilebilir.

```
void Der::der_pro_func()
{
    display_base();
    base_func();

    d  = 100;
    //b  = 200;    Geçersiz!
}
```

Türemiş Sınıf Üye İşlevlerinin Taban Sınıf Üye İşlevlerini Çağırması

Bir türemiş sınıf üye işlevi, taban sınıfın *public* ve *protected* bölümümüne erişerek taban sınıfın üye işlevlerini çağırabilir. Türemiş sınıf nesnesiyle taban sınıf üye işlevi çağrıldığında, ya da türemiş sınıf üye işlevi içinde taban sınıfın üye işlevi doğrudan çağrıldığında, taban sınıf üye işlevine *this* göstericisi olarak türemiş sınıf nesnesi içinde yer alan taban sınıf alt nesnesinin adresi geçirilir. Yani taban sınıfın üye işlevine aslında fiziksel olarak bir taban sınıf nesnesinin adresi geçirilmektedir. Aşağıdaki örneği inceleyin:

```
class Base {
public:

    int b;

    void base_func();

};


#include <iostream>

using namespace std;

void Base::base_func()

{

    cout << "base_func() icinde this adresi = " << this << endl;

}


class Der:public Base {
public:

    int d;

    void der_func();

};


void Der::der_func()

{

    cout << "der_func() icinde this adresi = " << this << endl;

}


int main()

{
```

```
    Der der_object;

    cout << "&der_object   = " << &der_object << endl;
    cout << "&der_object.d = " << &der_object.d << endl;
    cout << "&der_object.b = " << &der_object.b << endl;
    der_object.der_func();

    der_object.base_func();
    return 0;

}
```

Örnek bir ekran çıktısı:

```
&der_object  =  0012FF78
&der_object.d = 0012FF7C
&der_object.b = 0012FF78

der_func() icinde this adresi = 0012FF78
base_func() icinde this adresi = 0012FF78
```

Taban Sınıf Kurucu ve Sonlandırıcı İşlevlerinin Çağrılması

Türemiş sınıf nesnesi kendi içinde taban sınıfa ilişkin bir alt nesneyi de içerir. Taban sınıf elemanlarının taban sınıfın *private* bölümünde olduğunu düşünelim. Bu durumda türemiş sınıfın üye işlevleri oradaki veri elemanlara erişip onlara ilkdeğer verebilir mi? Türemiş sınıfın kurucu işlevi içinde yalnızca türemiş sınıf elemanlarına ilkdeğer verilebilir. Peki türemiş sınıfın *private* bölümde yer alan taban sınıf elemanlarına nasıl ilkdeğer verilebilir? Türemiş sınıf nesnesine ait taban sınıf elemanlarına ilkdeğer verilmesi, türemiş sınıfın kurucu işlevinin ana bloğunun başında taban sınıfın varsayılan kurucu işlevine yapılan gizli çağrıyla olur. Yani türemiş sınıf kurucu işlevi kendi içinde taban sınıf varsayılan kurucu işlevini çağırır. Aşağıdaki programı derleyerek çalıştırın:

```
class Base {
    int b;

public:

    Base();

    Base(int);

    void display_base() const;

};



#include <iostream>


using namespace std;


Base::Base()

{
```

```
    cout << "Base::Base()" << endl;
    b = 0;

}


Base::Base(int val)

{

    cout << "Base::Base(int)" << endl;
    b = val;

}


void Base::display_base() const

{

    cout << "b = " << b << endl;

}


class Der: public Base {
    int d;

public:

    Der();

};



Der::Der()

{

    cout << "Der::Der()" << endl;
    d = 0;

}


int main()

{

    Der der_object;
    der_object.display_base();



    return 0;

}
```

main işlevi içinde türemiş sınıf olan *Der* sınıfı türünden *der_object* simli bir nesne tanımlanıyor:

```
Der der_object;
```

Bu nesne için *Der* sınıfının varsayılan kurucu işlevinin çağrılacağını biliyorsunuz. *Der* sınıfının varsayılan kurucu işlevinin ana bloğunun hemen başında da, derleyici tarafından yerleştirilen gizli bir çağrı kodu ile taban sınıf olan *Base* sınıfının varsayılan kurucu işlevi çağrılır.

```
Der::Der()

{

    //Derleyici tarafından buraya Base::Base() çağrı kodu ekleniyor!
    cout << "Der::Der()" << endl;

    d = 0;

}
```

Türemiş sınıfın bütün kurucu işlevlerinde bu gizli çağrı işlemi yapılır.

Taban sınıfın varsayılan kurucu işlevi yerine, taban sınıfın parametreli başka bir kurucu işlevin çağrılması istenebilir. Bu durumda türemiş sınıfın kurucu işlevi *MIL* (*Member Initialization List*) sözdizimi ile tanımlanmalıdır. MIL sözdiziminde taban sınıfın hangi kurucu işlevinin çağrılacağı aşağıdaki gibi belirlenir.

```
B::B(parametre değişkenleri) : taban sınıf ismi(parametre değişkenleri)

{

    //...

}
```

Görüldüğü gibi tanımlama, bir sınıfın başka bir sınıf türünden elemanlara sahip olması durumunda olduğu gibi *MIL* sözdizimi kullanılarak yapılır. Ancak bu kez '*:*' atomunu elemanının ismi değil taban sınıf ismi izler. Taban sınıf isminden sonra ayraç içinde bir parametre değişkeni listesi yazılabilir. Bu durumda derleyici parametre listesini inceleyerek parametre listesine uygun olan taban sınıf kurucu işlevini çağırır. Örneğin yukarıdaki türetme işleminde *Der* sınıfının kurucu işlevi şöyle yazılmış olabilirdi:

```
Der ::Der() : Base(1)

{

    cout <<  "Der ::Der()" << endl ;

}
```

Bu durumda *Base* taban sınıfının *int* türden parametre değişkeni olan kurucu işlevi çağrılır.

Türemiş sınıf kurucu işlevine geçilen argümanlar taban sınıf kurucu işlevine aktarılabilir. Örneğin türetilen sınıf olan *De*r sınıfının

```
Der(int, int);
```

biçiminde bir kurucu işlevi olsun.

```
Der::Der(int x, int y) : Base(x), d(y) { }
```

Burada *Der* sınıfının kurucu işlevinin *x* parametre değişkeni, taban sınıfın kurucu işlevi için kullanılıyor.
Böylece örneğin :

```
Der der_object(10, 20);
```

biçiminde bir nesne tanımlaması ile *10* değeri taban sınıfın kurucu işlevi için *20* değeri ise türemiş sınıfın kurucu işlevi için kullanılır.

Türemiş sınıfın kurucu işlevinin tanımlamasında eğer *MIL* sözdizimi kullanılmamışsa, taban sınıfı için varsayılan kurucu işlevi çağrılır. Örneğin *Y* sınıfı *X* sınıfından türetilmiş olsun.

```
Y::Y()
{
    //...
}
```

Bu durumda kurucu işlevin ana bloğunun başında *X* sınıfı için varsayılan kurucu işlev çağrılır. Yani bu biçimde bir tanımlama aşağıdaki gibi tanımlamayla eşdeğer kabul edilebilir:

```
Y::Y() : X()
{
    //...
}
```

Eğer taban sınıf olan *X* sınıfının herhangi bir kurucu işlevi varsa, ama varsayılan kurucu işlevi yoksa bu durum derleme aşamasında hata oluşturur.

Kurucu İşlevin Sınıfın private Bölümünde Olması

Taban sınıf kurucu işlevlerinin türemiş sınıfın kurucu işlevi içinden bu biçimde otomatik olarak çağrılabilmesi için, taban sınıf kurucu işlevinin kendi sınıfının *protected* ya da *public* bölümünde bildirilmiş olması gerekir. Bir sınıfın kurucu işlevi sınıfın *protected* bölümüne koyulursa global bir işlev içinde o sınıf türünden nesne tanımlanamaz. Çünkü sınıfın *protected* kurucu işlevine erişilemez. Ancak o sınıftan türetme yapıldığında türemiş sınıf kurucu işlevi taban sınıfın *protected* kurucu işlevini çağırabilir.

```
class Base {
protected:

    Base();
public:

    //...

};


class Der: public Base {

    //...
public:

    Der();

};


int main()

{

    Der der_object;    //Geçerli
    Base base_object;  //Geçersiz!


    return 0;

}
```

Taban Sınıfın Sonlandırıcı İşlevinin Çağrılması

Bir sınıf nesnesinin ömrü sona erdiğinde o sınıf nesnesi için sonlandırıcı işlevin çağrıldığını hatırlıyorsunuz. Türemiş sınıf türünden bir nesne kendi içinde taban sınıf alt nesnesini içerdiğine göre, türemiş sınıf nesnesinin ömrü tamamlandığında, içerilen taban sınıf nesnesinin de ömrü tamamlanmış olur. Peki bu durumda, içerilen taban sınıf nesnesinin sonlandırıcı işlevi nasıl çağrılır?

İçerilen taban sınıf nesnesi için sonlandırıcı işlevin çağrılması derleyicinin eklediği bir kod ile olur. Derleyici türemiş sınıf sonlandırıcı işlevinin sonuna taban sınıfın sonlandırıcı işlevine yapılan çağrı kodunu ekler. Aşağıdaki örneği derleyerek çalıştırın.

```
#include <iostream>


class Base {
public:

    Base() {std::cout << "Base::Base()" << std::endl;}

    ~Base() {std::cout << "Base::~Base()" << std::endl;}

};


class Der: public Base {
public:

    Der() {std::cout << "Der::Der()" << std::endl;}

    ~Der();

};


Der::~Der()

{

    std::cout << "Der::~Der()" << std::endl;

}


void func()

{

    Der der_object;

}


int main()

{

    func();
```

```
    return 0;
}
```

Programın ekran çıktısı

```
Base::Base()
Der::Der()
Der::~Der()
Base::~Base()
```

şeklinde olur.

```
Der::~Der()
{
    cout << "Der::~Der()" << endl;
    //Derleyici buraya Base ::~Base çağrısını ekliyor!
}
```

Bu durumun anlamı şudur. Bir türemiş sınıf nesnesi yok edilirken önce türemiş sınıfın eklediği kısım yok edilir. Daha sonra taban sınıf kısmı yok edilir. Kurucu ve sonlandırıcı işlevlere ilişkin daha önce açıklanan kuralı anımsayalım. Kurucu işlevler ile sonlandırıcı işlevler ters sırada çağrılır. Yani kurucu işlevi en önce çağrılan nesnenin sonlandırıcı işlevi en sonra çağrılır. Taban sınıfın kurucu ve sonlandırıcı işlevlerinin çağrılması durumunda da bu kural geçerlidir.

Türetme İle Bileşik Nesne OluşturmanınKarşılaştırılması

Mantıksal ve işlevsel karşılıkları farklı olsa da, türetme ile bileşik nesne oluşturmanın ortak noktaları vardır. Her iki araçta da bir sınıf nesnesi bir başka sınıf nesnesini fiziksel olarak içerir.

```
class Member {
    //...
};
```

```
class Owner {
    Member m;
```

```
    //...
};
```

Yukarıdaki örnekte *Owner* sınıfı *Member* sınıfı türünden bir elemana sahiptir. *Owner* sınıfı türünden bir nesne yaratıldığında bu nesnenin içinde *Member* sınıfı türünden bir içsel nesne (*embedded object*) de yer alır. *Member* sınıfı türünden içsel nesne *m*"nin varsayılan kurucu işlevi, *Owner* sınıfının kurucu işlevlerinin başına derleyici tarafından eklenen edilen bir kodla çağrılır. Yine *Member* sınıfı türünden içsel nesne *m*"nin sonlandırıcı işlevi, *Owner* sınıfının sonlandırıcı işlevinin eklenen edilen kodla çağrılır. İçerilen nesne *m*"nin varsayılan kurucu işlevinin değil de, parametreli kurucu işlevlerinden birisinin çağrılması *MIL* sözdizimiyle mümkün kılınmıştır. Şimdi de türetmeye bakalım:

```
class Base{
    //...
};

class Der: public Base {
    //...
};
```

Der sınıfı türünden bir nesne yaratıldığında bu nesnenin içinde *Base* türünden bir alt nesne (*subobject*) yer alır. Bu alt nesne türemiş sınıf olan *Der* sınıfının taban sınıf olan *Base* sınıfından devir aldığı kısımdır. Derleyici ürettiği kodda *Der* sınıfının kurucu işlevlerinin başına taban sınıfın varsayılan kurucu işlevine yapılan bir çağrı kodunu ekler. Taban sınıf nesnesinin sonlandırıcı işlevinin çağrılması da yine, türemiş sınıfın sonlandırıcı işlevinin koduna derleyici tarafından otomatik olarak eklenen, taban sınıfın sonlandırıcı işleve yapılan çağrı koduyla sağlanır. Türemiş sınıf nesnesinin yaratılması sırasında taban sınıf nesnesi için parametreli bir kurucu işlevinin çağrılması yine *MIL* sözdizimiyle mümkün kılınmıştır.

Fiziksel karşılıkları birbirine bu kadar yakında olsa iki aracın mantıksal karşılıkları farklı olduğu gibi modelledikleri durumlar da tamamen farklıdır. Bileşik nesneler "*has a*" ilişkisini modellerken türetme "*is a*" ilişkisini modeller. Bazı durumlarda iki sınıf arasında bir sahiplik ilişkisi mi yoksa bir genellik - özellik ilişkisi mi olduğu kolayca anlaşılamayabilir. Yani bir bileşik nesne mi oluşturulmalıdır yoksa bir sınıf mı türetilmelidir? Bir proje içinde tanımlanacak sınıfların neler olacağını saptamak, bu sınıflar arasındaki ilişkilerin türünü belirlemek için bazı teknikler kullanılır. Bu tekniklere ileride değineceğiz.

Sınıfın protected Bölümü

Sınıfın *protected* bölümü yalnızca türetme yapıldığında anlam kazanan bir bölümdür. Bu bölüm tıpkı *private* bölümü gibi dışarıya kapalıdır. Ancak bir türetme yapıldığında türemiş sınıfın üye işlevleri tarafından taban

sınıfın *protected* bölümüne erişilebilir. Yani *private* bölümü ile *protected* bölümü arasındaki fark şudur. Türemiş sınıfın işlevleri taban sınıfın *private* bölümüne erişemezken taban sınıfın *protected* bölümüne erişebilir.

Bu durumda bir sınıfın

- *public* bölümü tüm kodların her zaman erişebileceği bir bölümüdür.
- *private* bölümü yalnızca sınıfın kendi üye işlevlerinin ve arkadaş işlevlerin erişebileceği bir bölümdür.
- *protected* bölümü sınıfın kendi üye işlevlerinin ve arkadaş işlevlerinin yanı sıra türemiş sınıf üye işlevlerinin de erişebileceği bir bölümdür. Türemiş sınıf üye işlevlerinin erişebilmesi onu *private* bölümden ayırır.

Peki hangi elemanlar ve üye işlevler sınıfın *protected* bölümüne yerleştirilmelidir? İşte buna karar verebilmek için önce sorulması gereken soru şudur: "Tasarlanan sınıftan türetme yapılacağını düşünülüyor mu?". Eğer düşünülüyorsa türeten kod parçası hangi elemanlarından ve işlevlerinden faydalanabilir? İşte türetmeyi yapan kod parçasının doğrudan kullanabileceği, onun işlerini kolaylaştıracağı düşünülen üye işlevler ve veri elemanları sınıfın *protected* bölümüne yerleştirilmelidir. Böylelikle bu elemanlar ve üye işlevler dışarının algısından uzaklaştırılarak yalnızca türetme işlemini yapacak kodların kullanımına sunulur. Örneğin tarih işlemlerini yapan *Date* sınıfına bakalım. Bu sınıftan bir sınıf türetip bu sınıfa yeni işlevler eklemek istenirse *day*, *month*, *year* elemanları doğrudan kullanıllması için bu elemanlar sınıfın *protected* bölümüne yerleştirilebilir. Bir sınıfın *protected* arayüzü kendisinden türetme yapan sınıfları ilgilendiren bir arayüzdür.

```
class Date {
public:

    Date();

    Date(int day, int month, int year);

    // …
protected:

    int day, month year;

};


class SpecialDate : public Date{

    // Yeni üye işlevler ve elemanlar

};
```

Bir sınıfın *protected* bölümü koruma bakımından *private* bölümden gevşek *public*

bölümden sıkı olduğuna dikkat edin. Şimdi yerleşim bakımından bir özet yapalım.

Bölüm	**Neler Yerleştirilmeli?**
public	Dışarıdan tüm kodların çağırabileceği üye işlevler, tüm kodların kullanabileceği elemanlar ve bildirimler
protected	Dışarıdan erişilmesi istenmeyen, yalnızca sınıftan türetme yapacak kodların erişmesi istenilen üye işlevler, elemanlar ve bildirimler
private	Yalnızca sınıfın kendi kodlarının erişmesi istenilen üye işlevler, elemanlar ve bildirimler.

Sınıf Hiyerarşisi

Türetme işlemi birden fazla kademeyi içerebilir.

Bir taban sınıftan farklı türetme işlemleriyle doğrudan yeni sınıflar türetilebilir. Örneğin "*Araç*" isimli bir sınıftan, "*hava aracı*", "*kara aracı*", "*deniz aracı*" sınıfları türetilebilir.

Bir taban sınıftan türetilmiş bir sınıfın kendisi bu kez taban sınıf olarak kullanılarak da yeni bir türetme yapılabilir. Örneğin "*Kara Aracı*" sınıfından "*otomobil*" sınıfı türetilip bu kez "*otomobil*" sınıfından da, "*Yarış Otomobilir*" sınıfı türetilebilir.

Bir türetme zinciri içinde mantıksal olarak ilişkisi kurulan sınıfların tümüne "*sınıf hiyerarşisi*" denir. Türetmeden elde edilmek istenen ana faydalardan biri, türetme hiyerarşi içinde yer alan sınıfların tümü üzerinde belirli işlemleri yapacak ortak kodların, hiyerarşinin tepesinde yer alan sınıfa dayalı olarak yazılmasıdır. Bu konuyu ileride ayrıntılı olarak ele alacağız.

Türetme ve Bileşik Nesnelerin Birlikte Kullanılması

Başka bir sınıftan türetilen bir sınıf, yine bir başka sınıf türünden elemana sahip olabilir. Bu durumda kurucu işlevler hangi sıraya göre çağrılır?

Türemiş bir sınıfın başka sınıf türünden bir nesneye sahip olması durumunda, önce taban sınıfın kurucu işlevi çağrılır. Taban sınıfın (ya da sınıfların) kurucu işlevleri çağrıldıktan sonra bu kez eleman olarak içerilen sınıf nesneleri için kurucu işlevleri çağrılır. Aşağıdaki programı derleyerek çalıştırın:

```
#include <iostream>


class Member {
public:

    Member() {std::cout << "Member::Member()" << std::endl;}

    ~Member() {std::cout << "Member::~Member()" << std::endl;}

};


class Base {
public:
```

```
    Base() {std::cout << "Base::Base()" << std::endl;}
    ~Base() {std::cout << "Base::~Base()" << std::endl;}
};

class DerComp: public Base {
    Member mem;
public:
    DerComp() {std::cout << "DerComp::DerComp()" << std::endl;}
    ~DerComp() {std::cout << "DerComp::~DerComp()" << std::endl;}
};

void func()
{
    DerComp d;
}

int main()
{
    func();

    return 0;
}
```

Programın ekran çıktısı :

```
Base::Base()
Member::Member()
DerComp::DerComp()
DerComp::~DerComp()
Member::~Member()
Base::~Base()
```

Türemiş Sınıflarda Bilinirlik Alanı

Bilinirlik alanı *(scope)* bir ismin derleyici tarafından tanınabildiği program aralığıdır. C ve C++ da bir blok içinde aynı isimli birden fazla değişken biliniyorsa o blok içinde aynı isimli değişken kullanıldığında dar bilinirlik alanına sahip olana erişilir. Yani dar bilinirlik alanındaki isim kendisini kapsayan daha geniş bilinirlik alanında bulunan ismi maskeler.

Onun görülmesini engeller. Aynı kural C++‟ da türetme işlemlerini kapsayacak biçimde geçerlidir. Önce şu soruyu soralım: Türetme durumunda taban sınıf türemiş sınıf tarafından erişilebildiğine göre taban sınıftaki bir isim mi daha dar bilinirlik alanına sahiptir, türemiş sınıftaki bir isim mi? Türemiş sınıftaki isim değil mi? Çünkü taban sınıftaki isimler hem kendi sınıfı içinde hem de türemiş sınıf içinde kullanıldığına göre daha geniş bir bilinirlik alanına sahiptir. Bu durumda hem taban sınıf içinde hem de türemiş sınıf içinde aynı isimli bir değişken varsa türemiş sınıf üye işlevleri içinde o değişken kullanıldığında, dar bilinirlik alanına sahip olan yani türemiş sınıfta tanımlanmış olan anlaşılır. Aşağıdaki kodu inceleyelim:

```
class Base {
protected:

    int b;
public:

    Base():b(0){}

    void display()const;

};


#include <iostream>

void Base::display()const

{

    std::cout << "b = " << b << std::endl;

}


class Der: public Base {
    int b;

public:

    Der():b(1){}

    void set(int);

    void display() const;
    void func() const;

};


void Der::set(int val)
```

```
{
    b = val;        //Der::b = val;
}

void Der::display()const
{
    std::cout << "b = " << b << std::endl;
}

void Der::func()const
{
    display();
}
```

Burada *Der* sınıfının *set* üye işlevi içinde kullanılan *b* ismi, daha dar bilinirlik alanına sahip olan *Der* sınıfının *b* isimli elemanına ilişkindir. *Der* sınıfının *b* isimli elemanı *Base* sınıfının *b* isimli elemanını maskeler yani görünmesini engeller. Tabii eğer *Der* sınıfının *b* isimli bir elemanı olmasaydı burada kullanılan *b, Base* sınıfına ilişkin olurdu.

Ayrıca *Base* sınıfının *display* üye işlevi içinde kullanılan *b* elemanı *Base* sınıfına ilişkin olandır. Zaten taban sınıf türemiş sınıfa erişemediğine göre hiçbir biçimde bunun *Base* sınıfına ilişkin olması söz konusu değildir.

Türemiş sınıf içinde aynı isimli taban sınıf elemanlarına ve işlevlerine çözünürlük işleci ile erişilebilir. Örneğin bu kez *Der* sınıfının iki parametreli *set* isimli bir üye işlevi daha olduğunu düşünelim:

```
void Der::set(int x, int y)
{
    b = x;
    Base::b = y;
}
```

İşlevde doğrudan kullanılan *b*, bilinirlik alanı kurallarına göre *Der* sınıfına ilişkin olandır. Ancak çözünürlük işleci ile, yani *Base::b* ifadesi ile kullanılan *b*, *Base* sınıfına ilişkin olandır. İki terimli çözünürlük işlecini taban ve türemiş sınıflarda aynı isimli elemanlar ya da işlevler varken taban sınıfın elemanlarına erişmek amacıyla bu biçimde kullanılabilir.

Bilinirlik alanı kuralları işlev isimleri için de geçerlidir. Türemiş ve taban sınıfta aynı isimli işlevler varsa dar bilinirlik alanı kuralına göre türemiş sınıftaki anlaşılır. Örneğin *Der* sınıfının *func* işlevi içinde çağrılan *display* işlevi *Der* sınıfına ilşkin olandır.

Çözünürlük işleci, aynı amaçla üye işlevler için de kullanılabilir. Örneğin türemiş sınıf olan *Der* sınıfının *func* isimli üye işlevi içinde bu kez taban sınıfın *display* isimli işlevi çağrılmak istenseydi bu aşağıdaki gibi yapılabilirdi:

```
void Der::func()

{

    Base::display();

}
```

Dar bilinirlik alanı kuralı dışarıdan sınıf elemanlarına erişirken de geçerlidir. Yukarıda verdiğimiz türetme işlemi için aşağıdaki kod yazılmış olsun:

```
int main()

{

    Der der_object, *der_ptr;


    der_object.display();          //1
    der_object.Base::display();    //2
    der_ptr = &der_object;

    der_ptr->display();            //3
    der_ptr->Base::display();      //4
    Base base_object;
    base_object.display();         //5
    return 0;

}
```

Yukarıdaki *main* işlevinde yorum satırlarıyla işaretlenen işlev çağrılarına bakalım:

//1 *Der* sınıfının *display* işlevi çağrılır.

//2 *Base* sınıfının *display* işlevi çağrılır.

//3 *Der* sınıfının *display* işlevi çağrılır.

//4 *Base* sınıfının *display* işlevi çağrılır.

//5 Base sınıfının *display* işlevi çağrılır. Taban sınıf türemiş sınıfa erişemez!

Bilinirlik alanı ile erişilebilirlik aynı şeyler değildir. C++"da her zaman önce isim araması daha sonra erişim denetimi yapılır. Bu kuralın etkisini aşağıdaki gibi bir türetme işleminde değerlendirelim:

```
class Base{
public:

    int x;

};



class Der: public Base {
    int x;

};



void func()

{

    Der der_object;

    der_object.x = 0;      //Geçersiz!

}
```

C++'a yeni başlayanlar *func* işlevi içindeki

```
der_object.x = 0;
```

atamasının geçerli olması gerektiğini düşünürler. "Hem türemiş sınıfta hem de taban sınıfta *x* isimli bir eleman olduğuna göre, önce türemiş sınıfta bulunan *x* isimli elemana erişilmeye çalışılır. Türemiş sınıfta bulunan *x* sınıfın *private* bölümünde bulunduğu için bu erişim gerçekleşmez. Ancak taban sınıfın *public* bölümünde bulunan *x*"e erişilir.

Ancak durum böyle değildir. Önce isim araması yapılır. Türemiş sınıf olan *Der* sınıfının bilinirlik alanında *x* ismi bulunduğunda isim arama sona erdirilir. Aranan isim bulunmuş ve *x* isminin türemiş sınıf bilinirlik alanındaki *x* olduğu anlaşılmıştır. Artık ne nedenle olursa olsun yeniden bir isim araması yapılmaz. Şimdi bu *x*'e erişimin geçerli olup olmadığı sorgulanır. Türemiş sınıfın *x* isimli elemanı türemiş sınıfın *private* bölümünde olduğu için erişim geçersizdir.

İşlev yüklemesi (*function overloading*) aynı bilinirlik alanındaki aynı isimli işlevler için geçerli bir araçtır. Farklı bilinirlik alanındaki aynı isimli işlevler yüklenemez. Yine, dar bilinirlik alanındaki ismin daha geniş bilinirlik alanındaki ismi gizlemesi söz konusudur. Bu kez aşağıdaki kod parçasını inceleyin:

```
class Base{
public:

    void foo();

    void foo(int, int);

};
```

```
class Der:public Base {
public:

    void  foo(int);
    void foo(int, int);

};


int main()

{

    Der der_object;

    //der_object.foo();           //Geçersiz

    der_object.foo(10);           //void Der::foo(int)
    der_object.foo(5, 8);         //void Der::foo(int, int)
    der_object.Base::foo();       //void Base::foo()
    der_object.Base::foo(5, 8);   //void Base::foo(int, int)



    return 0;

}
```

main işlevi içinde yapılan işlev çağrılarını teker teker inceleyelim:

```
der_object.foo();
```

çağrısı geçersizdir. Çünkü türemiş sınıf bilinirlik alanında *foo* ismi bulunduğunda isim arama sona erer. Yalnızca *Der* sınfı içinde bildirilmiş olan *foo* işlevleri, çağrılan işlev için aday olur. *Der* sınıfı bilinirlik alanı içinde parametresi olmayan *foo* isimli bir üye işlev bildirilmediğine göre, çağrı derleme zamanında hata olarak değerlendirilir. Şüphesiz türemiş sınıf içinde *foo* isimli hiç bir işlev tanımlanmamış olsaydı bu kez taban sınıfın parametre değişkeni olmayan *foo* isimli işlevi çağrılırdı:

```
der_object.foo(10);
```

çağrısı ile türemiş sınıfın *int* parametreli *foo* işlevi çağrılır. *Der* sınıfının bu çağrı için iki aday işlevi vardır. Ancak seçilecek işlev *int* parametreli olandır:

```
der_object.Base::foo();       //void Base::foo()
der_object.Base::foo(5, 8);   //void Base::foo(int, int)
```

çağrılarında işlev ismi sınıf ismi ile nitelendirildiğinden isim araması türemiş sınıfın bilinirlik alanından başlamaz. Arama doğrudan *Base* sınıfının bilinirlik alanından başlar. Bu durumda *Base* sınıfının aynı ismli iki işlev içinden işlev yükleme kurallarına göre uygun olan işlevler çağrılır.

```
der_object.foo(5, 8);
```

ile bu kez çağrılan *Der* sınıfının iki parametreli *foo* işlevi iken

```
der_object.foo(10);
```

ile *Der* sınıfının tek parametreli *foo* işlevi çağrılır.

Benzer durum türemiş sınıfın üye işlevi içinde taban sınıfın bir elemanına erişirken ya da taban sınıfın üye işlevi çağrılırken de geçerlidir. Aşağıdaki programı inceleyin:

```
class Base{
    int b;
public:
    void foo();
    void func(int);
};

class Der:public Base {
    int b;
public:
    void foo();
    void func();
};

void Der::foo()
{
    b = 10;         //Der::b = 10;
    //Base::b = 50; Taban sınıfın private bölümüne erişilemez. Geçersiz
    foo();          //İşlev kendi kendini çağırır.
    Base::foo();    //Base::foo() çağrılır
```

```
    //func(10);     //Geçersiz!

    func();         //Der::func() çağrılır.
    Base::func(10);//Base::func(int) çağrılır.

}
```

Türemiş Sınıf İçinde Yapılan using Bildirimi

Türemiş ve taban sınıfların farklı bilinirlik alanı oluşturduklarını biliyorsunuz. Türemiş sınıf bilinirlik alanı içinde bildirilen bir isim, taban sınıf içinde de kullanılmışsa, taban sınıftaki isim maskelenir. Maskelenme istenmiyorsa türemiş sınıf içinde *using* bildirimi yapılabilir. Sınıf içi *using* bildirimiyle taban sınıflara ilişkin maskelenen isimler görünür kılınır. Böylece taban sınıf ve türemiş sınıf işlevleri farklı bilinirlik alanında olsalar bile yüklenebilir.

Aşağıdaki kodu inceleyin:

```
#include <iostream>


class Base {
public:

    void foo(){std::cout << "Base::foo()" << std::endl;}

};

class Der: public Base {
public:

    using Base::foo;

    void foo(int){std::cout << "Der::foo(int)" << std::endl;}

};


int main()

{

    Der der_object;


    der_object.foo();
    der_object.foo(25);


    return 0;

}
```

Der sınıfının tanımı içinde yapılan *using* bildirimiyle *Base* sınıfının *foo* isimli işlevi *Der* sınıfı içinde görülebilir hale getiriliyor. Böylece

```
der_object.foo();
```

çağrısı söz konusu olduğunda

```
void Base::foo();
```

işlevi de çağrıya aday işlev olarak belirlenir. *using* bildirimiyle taban ve türemiş sınıfın *foo*

işlevleri ayrı bilinirlik alanlarına ait olmasalar da yüklenebilir. Şimdi de
aşağıdaki koda bakalım:

```
#include <iostream>


class Base {
public:

    void foo(){std::cout << "Base::foo()" << std::endl;}

};


class Der: public Base {
public:

    using Base::foo;

    void foo(){std::cout << "Der::foo()" << std::endl;}

};


int main()

{

    Der der_object;
    der_object.foo();


    return 0;

}
```

main işlevi içinde çağrılan, *Der* sınıfının *foo* işlevidir. *Der* sınıfının *foo* işlevinin gizli parametre değişkeni *Der &* türünden iken, *Base* sınıfının *foo* işlevinin gizli parametre değişkeni *Base &* türündendir.

Sınıflarda İsim Arama

İsim arama *(name lookup)* bilinirlik alanının derleyici bakış açısıyla ele alınma biçimidir. Bir değişken kullanıldığında derleyici o değişkeni sırasıyla dardan genişe doğru çeşitli bilinirlik alanlarında arar. Eğer ismi bir bilinirlik alanında bulursa aramayı keser.

Değişkeninin o bilinirlik alanındaki isim olduğuna karar verir. Derleyici eğer aradığı ismi hiçbir bilinirlik alanında bulamazsa bu durum hata olarak değerlendirilir. Örneğin *C*'de derleyici bir blok içinde bir değişken ile karşılaştığında sırasıyla onu şu bloklarda arar.

1. Değişkenin kullanıldığı blok içinde
2. İşlev içinde onu kapsayan bloklarda
3. Global isim alanında

Şimdi aynı isimli hem bir yerel hem de global değişken tanımlanmış olduğunu düşünelim. Derleyici onu yerel blokta bulduğu için arama işlemini sürdürmez. İsim arama sırasında aranan isim bulunduğu zaman, isim ister erişilir olsun isterse olmasın işlem sonlandırılır. Erişim geçerliliği yukarıda da belirtildiği gibi isim araması bitirildikten sonra yapılır.

Gördüğünüz gibi aslında isim arama işlemi dar bilinirlik alanı kuralının derleyici bakış açısıyla yorumlanmış biçimidir. *C++*'da sınıflar söz konusu olduğunda bilinirlik alanına ilşkin karmaşık durumlar söz konusu olabildiği için anlatımda isim arama kavramından da faydalanılır. Bir isim ya nitelenmemiş olarak *(unqualified)* ya da çözünürlük işleci ile nitelenmiş olarak *(qualified)* kullanılır. Sınıflarda isimlerin aranmasını nitelenmemiş isimler açısından inceleyeceğiz.

Sınıf Bildirimi İçindeki İsimlerin Aranması

Bir sınıf bildirimi içinde bir eleman, üye işlev ismi, *typedef* ismi, *enum* değişmezleri gibi isimlerin aranma sırasını belirler. Örneğin:

```
class A {
public:

    // …

    void putstr(PCSTR);

    // …

};
```

Burada *putstr* üye işlevin parametre değişkeninin türü olarak kullanılan *PCSTR* ismi sınıf bildirimi içindedir. Bu isim nerede aranır?

Sınıf bildirimi içinde kullanılan bir isim sırasıyla şu bilinirlik alanlarında aranır:

1. İsim kendi sınıf bildiriminin başından kullanım yerine kadar olan bölgede aranır.

Aşağıdaki örneği inceleyin:

```
class A {
public:

    Days get_day(Str);
private:

    typedef char *Str;

    enum Days {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday};

public:

    Str func(Days);

    //...

};
```

Yukarıdaki sınıf tanımında *get_day* üye işlevinin bildiriminde *Days* ve *Str* isimlerinin kullanılması geçersizdir. Çünkü derleyici bu noktaya geldiğinde bu isimlerle henüz karşılaşmamıştır.

Ancak func isimli üye işlevin geri dönüş değerinin türü olarak *Str* ismi kullanılabilir. Çünkü bu türün bildirimi sınıf içinde daha önce yapılmıştır.

2. İsim sınıfın taban sınıf bildirimlerinin her yerinde aranır. Bir dizi türetme söz konusu olabilir. Bu durumda aynı özellik tüm taban sınıflar için geçerli olur. Örneğin aşağıda türemiş sınıf içindeki *SIZE* isminin kullanılması geçerlidir.

```
class A

{

protected:

    enum {SIZE = 50 };

};
```

```
class B : public A {
private:

    int a[SIZE];

};
```

Tabii burada *SIZE* taban sınıfın *public* ya da *protected* bölümünde bildirilmiş olmasaydı, kullanım bilinirlik alanı bakımından geçerli fakat erişim bakımından geçersiz olurdu.

3. Sınıf başka bir sınıfın içinde bildirilmişse, isim kapsayan sınıf bildiriminden ismin bulunduğu sınıfın bildirimine kadar olan bölgede aranır.

Örneğin aşağıda *B* sınıfı içindeki *Str* isminin kullanılması geçerlidir. Ayrıca bu durumda erişim geçerliliği için ismin kapsayan sınıfın *public* bölümünde bildirilmiş olması gerekir.

```
class A {
public:

    typedef char *Str;


    class B {
    private:

        Str pStr;

    };

};
```

4. Sınıf başka bir sınıf içinde bildirilmişse, isim kapsayan sınıf bildiriminin taban sınıflarının her yerinde aranır. Bu durumda erişim geçerliliği için kapsayan sınıfın *public* türetme biçimiyle türetilmiş olması ve kullanılan isimlerin de sınıfın *public* bölümünde bildirilmiş olması gerekir. Aşağıdaki örneği inceleyelim:

```
class A {
public:

    enum { SIZE = 50 };

};


class B : public A {
    class C {
    private:
```

```
        int x[SIZE]; // geçerli erişim
    // ...
    };
};
```

Burada *C* sınıfı içinde *SIZE* isminin kullanılması geçerlidir.

5. İsim, sınıfın içinde bulunduğu isim alanının başından kapsayan sınıf bildirimine kadar olan bölgede aranır. Aşağıdaki örnekte *SIZE* ismine erişim geçerlidir.

```
namespace X {
    enum {SIZE = 50 };
    class B {
        class C {
            private:
            int x[SIZE];    // geçerli erişim
            // ...
        };
    };
}
```

6. İsim dosya başından sınıfın içinde bulunduğu isim aralığına kadar olan global isim aralığında aranır. Sınıf bildirimi içinde bir isim *çözünürlük* işleci ile sınıf ismi ya da isim aralığı ismi belirtilerek kullanılmış olabilir. Bu durumu ileride ele alacağız.

Sınıfın Üye İşlevi İçinde Kullanılan İsimlerin Aranması

Sınıfın üye işlevi içindeki isimlerin aranma sırası tanımlamanın sınıf içinde (yani *inline* olarak) yapılıp yapılmadığına göre değişiklik göstermez. Üye işlev ister sınıfın içinde tanımlanmış olsun isterse dışarıda tanımlanmış olsun arama işlemi aşağıda açıklanan sırada yapılır:

1. İsim, üye işlev kendi yerel blokları ve parametre bildirim ayraçları içinde aranır.
2. İsim üye işlevin ilişkin olduğu sınıf bildiriminin her yerinde aranır. Örneğin:

```
class A {
public:
```

```
    void set(int x)
    {
        a = x;
    }
    int get() const
    {
        return a;
    }
private:
    int a;
};
```

Burada *set* ve *get* üye işlevlerinin içinde *a* ismi daha sonra bildirildiği halde kullanılıyor. Sınıfın her yeri arandığı için kullanım geçerlidir. Görüldüğü gibi arama işleminde üye işlevin sınıf içinde tanımlanıp tanımlanmadığının bir önemi yoktur.

3. İsim üye işlevin ilişkin olduğu sınıfın taban sınıf bildirimlerinin her yerinde aranır. Arama işlemi ilk taban sınıftan başlanarak yukarıya doğru sırasıyla yapılır. Ayrıca, erişim geçerliliği için ismin taban sınıfın *public* ya da *protected* bölümlerinde bulunması gerekir. Örneğin:

```
class A {
public:
    int x;
};

class B : public A {
public:
    int x;
};
class C : public B {
public:
    void func();
};
```

```
void C::func()

{

    x = 100;

}
```

Burada *x* ismi sırasıyla *B* ve *A* taban sınıflarında aranır. İsim bulunduğu anda arama işlemine son verilir. Dolayısıyla buradaki örnekte bulunan *x, B* sınıfına ilişkin olan *x* ismidir.

4. Üye işlevin ilişkin olduğu sınıfı kapsayan bir sınıf varsa, isim kapsayan sınıf bildiriminin her yerinde aranır. Erişim geçerliliği için ismin sınıfın *public* bölümünde bulunması gerekir. Örneğin:

```
class A {
public:

    class B {
    public:

        void func()

        {

            int a[SIZE];

            for (int i = 0; i < SIZE; ++i)
                a[i] = 0;

            // …

        }

    };

    enum { SIZE = 100 };

};
```

Burada *B* sınıfı içindeki *func* işlevinde *SIZE* ismi kullanılabilir. Tabii böyle durumlarda kapsayan sınıfa ilişkin bir eleman kullanılamaz. Çünkü kapsama işlemi türetme işlemi değildir. Bu yüzden kullanılacak ismin de bir *enum* ismi, *typedef* ismi gibi bildirimlerden oluşması gerekir.

5. Sınıfı kapsayan sınıf varsa, isim kapsayan sınıfın taban sınıf bildirimlerinin her yerinde aranır. Bu durumda erişim geçerliliği için ismin kendi sınıfının *public* bölümünde bildirilmiş olması gerekir. Türetme biçiminin bir önemi yoktur. Örneğin:

```
class A {
public:
```

```
    typedef char *Str;
};


class B : public A {
public:
    class C {
    public:
        void func(){Str pStr = "Deneme";
            // ...
        }
    };
};
```

Burada *Str* ismi *C* sınıfı içinden kullanılabilir.

6. İsim sınıfın ya da kapsayan sınıfın içinde bulunduğu isim alanının (*namespace*) başından kapsayan sınıf bildirimine kadar olan bölgede aranır. Örneğin:

```
namespace X {
    typedef char *Str;
    class A {
    public:
        class C {
        public:
            void func()
            {
                Str pStr = "Deneme";
                // ...
            }
        };
    };
}
```

Burada *Str* isminin kullanımı geçerlidir.

7. İsim, sınıfın ya da kapsayan sınıfın içinde bulunduğu isim alanını kapsayan isim alanında ve sonra da global isim alanında (*global namespace*) aranır. Örneğin aşağıdaki isimlerin hepsi doğru kullanılmıştır:

```
enum {SIZE = 10 };


namespace X {

    typedef int *Pint;
    namespace Y {

        typedef char *Str;
        class A {

            void func()

            {

                Str pStr = "Deneme";
                Pint pInt = 0;

                int a[SIZE];

                // ...

            }

        };

    }

}
```

Türetme İlişkisinde Arkadaşlık Bildirimleri

Arkadaşlık türetme yoluyla devir alınmaz. Taban sınıfın arkadaşı türemiş sınıfın da arkadaşı değildir. Aşağıdaki örneği inceleyin:

```
class Base {
    int b;

    friend void func();
public:

    Base(int val):b(val){}

};


class Der:public Base {
    int d;

public:
```

```
    Der(int val1, int val2):Base(val1), d(val2){}

};



void func()

{

    Der der_object(10, 20);

    //der_object.d = 34;       //Geçersiz!
    der_object.Base::b = 13;

}
```

Base sınıfı içinde global *func* işlevine arkadaşlık veriliyor. Global *func* işlevi içinde tanımlanan *Der* sınıfı türünden *der_object* nesnesinin *private* elemanı olan *d* elemanına erişim derleme zamanında hataya neden olur. Ancak *func* işlevi içinde *der_object* nesnesinin taban sınıftan devir aldığı *b* isimli *private* elemana erişebilir.

Türemiş Sınıf Nesne Adresinin Taban Sınıf Göstericisine Atanabilmesi

C++ dilinde katı bir tür denetiminin esas alındığını biliyorsunuz. Bir göstericiye ancak aynı türden bir adres atanabilir. Bir referans da ancak aynı türden bir nesne ile ilkdeğerini alabilir. *public* türetmesi bu kuralın bir istisnasıdır. *public* türetmesiyle amaçlanan *"is a"* ilişkisini modellemektir. Türetilmiş bir sınıf türünden nesne aynı zamanda taban sınıf türünden bir nesne olduğuna göre, taban sınıf türünden nesne gereken her yerde türemiş sınıf türünden bir nesne kullanılabilir.

1. Türemiş sınıf nesnesi taban sınıf nesnesine doğrudan atanabilir.
2. Türemiş sınıf türünden bir nesnenin adresi taban sınıf türünden bir gösterici değişkene atanabilir.
3. Taban sınıf türünden bir referans türemiş sınıf türünden bir nesne ile ilkdeğerini alabilir.

Aşağıdaki kod parçasını inceleyelim:

```
class Base {
public:

    //...

};



class Der:public Base {
public:

    //...

};



int main()
```

```
{

    Base base_object;

    Der der_object;

    Base &base_ref = der_object;
    Base *base_ptr = &base_object;
    base_object = der_object;



    return 0;

}
```

"is a" ilişkisinin anlamı dilin kuralları ve araçları tarafından da desteklenir. Türemiş sınıf nesnesinin taban sınıf nesnesi gereken yerde kullanılabilmesinin anlamı şudur. Bir sınıf hiyerarşisi içinde bir işlevin parametresi en tepedeki taban sınıf türünden bir gösterici ya da bir referans ise, bu işlev hiyerarşi içinde herhangi bir sınıf türünden nesnenin adresi ile (ya da kendisi ile) çağrılabilir. Böylece ortak özellikleri olan sınıfların hepsi üzerinde genel işlem yapan işlevler yazılabilir.

```
void process(Base &);
```

Yukarıda bildirimi verilen işlevin parametre değişkeni *Base* türünden bir referanstır. Bu işlev *Base* türünden bir nesne ile çağrılabildiği gibi *Base* sınıfından türeyen herhangi bir sınıf türünden nesne ile de çağrılabilir.

Türemiş sınıf nesnesinin taban sınıf nesnesiymiş gibi kullanılabilir. Ama bunun tersi doğru değildir. Bir türemiş sınıf nesnesine taban sınıf nesnesinin atanması ya da türemiş sınıf gösterici değişkene taban sınıf türünden bir adresin doğrudan atanması geçersizdir.

Türetmede Kopyalayan Kurucu İşlevin Durumu

Türemiş sınıf nesnesi kendi içinde bir taban alt nesnesini de içerir. Bir türemiş sınıf nesnesi ilkdeğerini bir başka türemiş sınıf nesnesinden alarak yaratılırsa, türemiş sınıf nesnesi içindeki taban sınıf nesnesinin durumu ne olur?

Eğer türemiş sınıf nesnesi için bir kopyalayan kurucu işlev yazılmamışsa, derleyici yazacağı kopyalayan kurucu işlevin başına, taban sınıfın kopyalayan kurucu işlevin çağrısı kodunu ekler. Yani ilkdeğer alan türemiş sınıf nesnesinin taban sınıf kısmı için de taban sınıfın kopyalayan kurucu işlevi çağrılır.

Burada dikkat edilmesi gereken durum şudur: Eğer türemiş sınıfın kurucu işlevi programcı tarafından yazılırsa, taban sınıf nesnesine ilkdeğer verilmesinden de programcı sorumlu olur. Eğer taban sınıf kısmına ilkdeğer verilmez ise, türemiş sınıf nesnesinin taban sınıf kısmı için varsayılan kurucu işlev çağrılır. Aşağıdaki örneği inceleyin:

```
#include <iostream>


class Base {
public:

    Base(const Base &r){std::cout << "Base(const Base &)" << std::endl;}
    Base(){std::cout << "Base()" << std::endl;}

};


class Der: public Base {
public:

    Der() {std::cout << "Der()" << std::endl;}

    Der(const Der &) {std::cout << "Der(const Der &)" << std::endl;}

};


int main()

{

    Der der1;

    Der der2(der1);


    return 0;

}


Der der2(der1);
```

tanımlama deyimiyle *der2* nesnesi ilkdeğerini *der1* nesnesinden alarak yaratılıyor. Bu durumda yaratılan *der2* nesnesi için kopyalayan kurucu işlev çağrılır. *der2* nesnesi için çağrılan kurucu işlevin başına derleyici, taban sınıfın varsayılan kurucu işlevine yapılan çağrı kodunu ekler. Yani taban sınıf için kopyalayan kurucu işlev değil varsayılan kurucu işlev çağrılır. Şimdi *Der* sınıfının kopyalayan kurucu işlevin tanımını programdan çıkartarak programı derleyin ve yeniden çalıştırın. Bu kez *der2* sınıf nesnesinin taban sınıf kısmı için kopyalayan kurucu işlevin çağrıldığını göreceksiniz. Türemiş sınıfın kopyalayan kurucu işlevini yazan programcı, taban sınıf nesnesinin kopyalayan kurucu işlevin çağrılmasını *MIL* sözdizmiyle sağlayabilir:

```
Der::Der(const Der &r) :Base(r) {cout << "Der(const Der &)" << endl;}
```

Türemiş sınıf nesnesinin taban sınıf kısmı için hangi kurucu işlev çağrılır? Taban sınıfın kopyalayan kurucu işlevin parametre değişkeni *Base* türünden referans olduğuna göre, bu referans ilkdeğerini türemiş sınıfın kopyalayan kurucu işlevin parametre değişkeni olan *r* referansından alabilir değil mi? Sonuca taban sınıf türünden bir referansa türemiş sınıf türünden bir nesne ile ilkdeğer veriliyor.

Türetmede Atama İşlevinin Durumu

Benzer durum atama işlevini yükleyen işlev için de geçerlidir. C++" da aynı sınıf türünden iki nesnenin birbirine atanması durumunda, atama işlecinin sol tarafındaki sınıf nesnesi için atama işlecini yükleyen işlevin çağrıldığını biliyorsunuz. Sınıf için atama işlevi yüklenmemişse, atama işlecini yükleyen işlevi derleyici yazar. Derleyicinin yazdığı atama işlevi sınıfın statik olmayan *public inline* üye işlevidir. Bu işlev sınıf elemanlarını karşılıklı olarak birbirine atayarak referans yoluyla atamanın yapıldığı nesneye geri döner. Örneğin *Myclass* isimli bir sınıf için, derleyici tarafından otomatik olarak yazılan atama işlevinin bildirimi aşağıdaki gibi olur:

```
class Myclass {
public:

    Myclass &operator=(const Myclass &);

};
```

Türemiş sınıf için bir atama işlevi yazılmazsa türemiş sınıf için işlevi derleyici yazar. Derleyicinin yazdığı atama işlevi sınıf nesnesinin elemanlarını karşılıklı olarak birbirine atamakla kalmaz, nesnelerin taban sınıf alt nesnelerini de birbirine atar. Bu atama için yine taban sınıfın atama işlevi çağrılır. Burada yine dikkat edilmesi gereken nokta şudur: Türemiş sınıf için atama işlecini yükleyen bir işlev yazılırsa, artık derleyici bu işleve herhangi bir kod eklemez. Yani taban sınıf nesnelerinin birbirine atanması da, türemiş sınıfın atama işleç işlevinin sorumluluğundadır. Aşağıdaki kodu inceleyin:

```
class Der : public Base {
public:

    Der &operator=(const Der &);

};


Der &Der::operator=(const Der &r)

{

    *(Base *)this = r; //taban sınıf kısmı için yapılan atama
    return *this;

}
```

Der sınıf için yazılan atama işlevi içindeki

```
*(Base *)this = r;
```

atamasını inceleyelim. İşlev içinde kullanılan *this* adresi atama işlecinin sol tarafına gelen türemiş sınıf nesnesinin adresidir. Bu adres, tür dönüştürme işleci ile önce taban sınıf türünden bir adrese dönüştürülüyor. Böyle bir dönüşümle elde edilen adresin türemiş sınıf nesnesinin içindeki taban sınıf nesnesinin adresi olduğunu biliyorsunuz. Bu adresin içerik işlecinin terimi olmasıyla elde edilen nesne, türemiş sınıf nesnesinin taban sınıf alt nesnesidir. Bu durumda taban sınıfının atama işlevi çağrılır. Bu işleve argüman olarak türemiş sınıf nesnesi gönderilmiş olur. Taban sınıf nesnesinin atama işlevinin parametresi taban sınıf türünden refarans olduğundan bu işleve türemiş sınıf nesnesinin gönderilebileceğini biliyorsunuz. Derleyicinin bu deyim için aşağıdaki gibi bir kod ürettiğini düşünebilirsiniz:

```
((Base *)this)->operator = (r);
```

Aynı işi gerçekleştirmek için aşağıdaki gibi bir çağrı da yapılabilirdi:

```
Base::operator=(r);
```

SANAL İŞLEVLER VE ÇOKBİÇİMLİLİK

Sanal işlevler nesne yönelimli programların geliştirilemesini ve büyütülmesini büyük ölçüde kolaylaştıran, *C++* dilinin çok önemli bir aracıdır. Sanal işlevler, "nesne yönelimli programlama tekniği"nin olmazsa olmaz özelliklerinden biri olan, "çalışma zamanına ilişkin çok biçimliliği" sağlayan temel araçtır.

Bu araç ile bir taban sınıftan doğrudan ya da dolaylı biçimde türetilmiş tüm alt sınıflara ilişkin nesneler, yani bir sınıf hiyerarşisi içinde kullanılabilecek tüm nesneler, taban sınıf nesneleriymiş gibi düşünülerek genel programlar yazılabilir. Programın geliştirilmesi sırasında henüz var olmayan sınıflar bu araç ile daha sonra programın genel yapısına eklenebilir.

Bir sınıf hiyerarşisi içinde bir sınıf nesnesi için bir işlev yardımıyla bir işin yaptırılması durumunu inceleyelim. Örneğin bir sınıf nesnesinin elemanlarının değerlerinin ekrana yazdırılması gerektiğini düşünelim. Bu iş sınıfların *display* isimli üye işlevler tarafından yapılsın:

```
class A {

//…
public:

    void display() const;  // A sınıfının elemanlarını yazdıracak.

};


class B:public A {

//…
public:

    void display() const;  //  B sınıfının elemanlarını yazdıracak.

};


class C:public A {

//…
public:

    void display() const;  // C sınıfının elemanlarını yazdıracak.

};
```

Yukarıdaki örnekte *A* sınıfından türetilmiş *B* ve *C* sınıflarında *A* sınıfında bulunan *display()* işlevi yeniden tanımlanıyor. Şimdi türemiş bir sınıf nesnesi yoluyla yani nokta işleci kullanılarak *display* işlevi çağrılırsa, dar bilinirlik alanındaki isim geniş bilinirlik alanındaki aynı ismi maskeleyeceği için, türemiş sınıfın *display* işlevi çağrılır.

```
int main()
```

```
{

    A a;
    B b;
    C c;

    a.display();  //A  sınıfının  display  işlevi  çağrılır.
    b.display():  //B  sınıfının  display  işlevi  çağrılır.
    c.display(); //C sınıfının display işlevi çağrılır;



    return 0;

}
```

Çağrı sınıf nesnesi yoluyla değil de sınıf türünden gösterici ile yapılmış olsaydı da sonuç aynı olurdu:

```
int main()

{

    A a, *pa = &a;
    C c, *cp = &c;
    B b; *bp = &b;

    pa->display();  //A  sınıfının  display  işlevi  çağrılır;
    pb->display();  //B  sınıfının  display  işlevi  çağrılır;
    pc->display(): //C sınıfının display işlevi çağrılır;



    return 0;

}
```

Taban sınıf ve türemiş sınıflarda aynı isimli işlevler varsa, çağrı çözünürlük işleci ile yapılmamışsa, eğer taban sınıf nesnesi ya da göstericisine ilişkin bir çağrı söz konusu ise doğal olarak taban sınıfın işlevi çağrılır.

Öyle bir işlev olsun ki, bu işlev sınıf hiyerarşisi içinde yer alabilecek bir nesnenin, hangi sınıf türünden olursa olsun elemanlarının değerlerini yazdırsın. Böyle bir işlev nasıl tasarlanabilir? Taban sınıf türünden bir göstericiye, bu sınıftan türetilmiş herhangi bir sınıf nesnesinin adresi atanabileceğine göre böyle bir işlevin parametre değişkeni taban sınıf

türünden bir gösterici olabilir. Global olarak tanımlanan *display* isimli işlevin bildirimi şöyle olabilir:

```
void display(const A *ptr);
```

şimdi *display* işlevi ister taban sınıf olan *A* sınıfı türünden bir nesnenin adresiyle ister bu sınıfta türemiş sınıflar olan *B* ve *C* sınıfından nesnelerin adresleri ile çağrılabilir.

Ancak böyle bir işlev çağrıldığı zaman işini doğru yapabilmesi için hangi sınıf türünden bir nesnenin adresiyle çağrıldığını şüphesiz bilmek zorundadır. İşlev bu durumu nasıl bilebilir?

Bunu sağlamanın bir yolu şu olabilir:

Taban sınıf bildiriminde, taban sınıfa bir eleman daha eklenenerek ve bu elemanının sınıf türünden nesnenin tür bilgisini saklaması sağlanabilir.

```
enum {CLASSA, CLASSB, CLASSC};

class A {
    //...
    int type;
public:
    //
};
```

A sınıfı ve bu sınıflardan türetilecek sınıfların kodu, bir sınıf nesnesi yaratıldığında sınıf nesnesinin *type* isimli elemanına *CLASSA, CLASSB, CLASSC* vs. değerleri atanacak biçimde yazılabilir. Bu durumda global *display* işlevi, aşağıdaki gibi bir *switch* deyimini kullanarak işini görebilir.

```
void display(const A *ptr)
{
    switch (ptr->type) {
        case CLASSA: ((A *)ptr)->display(); break;
        case CLASSB: ((B *)ptr)->display(); break;
        case CLASSC: ((C *)ptr)->display();
    }
    //..
```

```
}
```

Ancak yukarıdaki kod parçasının çok önemli bir dezavantajı programın geliştirilmesi sırasında ortaya çıkar. Taban sınıf olan *A* sınıfından ya da türetilmiş sınıflar olan *B* ve *C* sınıflarından yeni bir sınıf türetildiğinde yukarıdaki *display* işlevinin bu yeni sınıf türünden bir nesnenin de değerlerini yazdırması istendiğinde, *display* işlevinin kaynak kodunu değiştirmek gerekir. Bu da şüphesiz *switch* deyiminin içine yeni bir *case* daha eklemekle mümkün olur. Ayrıca yukarıdaki gibi bir kodu okumak zorlaşır. İşte sanal işlevler ve çalışma zamanına yönelik çok biçimlilik bu konuda yeni ufuklar açacak bir araçtır.

Sanal İşlevler

Sınıfın bir üye işlevi sanal *(virtual function)* yapılabilir. Bir üye işlevi sanal yapabilmek için işlev bildiriminin önüne *virtual* anahtar sözcüğü getirilir. *virtual* anahtar sözcüğü yalnızca işlevin bildiriminde kullanılır. İşlev tanımlanırken kullanılmaz. Bir global işlev ya da bir sınıfın statik işlevi *virtual* anahtar sözcüğü ile bildirilemez. Bir üye işlev sanal yapılırsa, o sınıfın türemiş sınıflarında bulunan aynı isimli, aynı bildirime sahip tüm işlevler da sanal olur. Yani *virtual* anahtar sözcüğü yazılmasa da yazılmış gibi işlem görür. Aynı bildirime sahip olması demek geri dönüş değerlerinin parametre yapılarının ve işlev isimlerinin aynı olması demektir. Türemiş sınıfın taban sınıfın sanal işleviyle tamamen aynı parametrik yapıya sahip ve aynı isimli bir işlev tanımlamasına, türemiş sınıfın taban sınıfın sanal işlevi "ezmesi" (*override*) denir.

```
class Base {
    //…
public:
    virtual int vfunc(int, int);
    //…
};

class Der: public Base {
    //…
public:
    public:
    virtual int vfunc(int, int);
    //…
};
```

Yukarıdaki örnekte, *Base* sınıfından türetilmiş *Der* sınıfı, *Base* sınıfının sanal *vfunc* işlevini eziyor. *Der* sınıfı içinde bildirimi yapılan *vfunc* işlevinin bildiriminde *virtual* anahtar sözcüğünün kullanılması zorunlu değildir.

Türemiş sınıf nesnesinin adresi taban sınıf göstericisine atanır bu gösterici yoluyla da sanal bir işlev çağrılırsa, adresi alınan nesne hangi sınıfa aitse o sınıfın sanal işlev çağrılır. Hangi işlevin çağrılacağı derleme zamanında değil ancak programın çalışma zamanında belli olur. Bu yüzden bu duruma "geç bağlama" (*late binding)* ya da

"dinamik bağlama" (*dynamic binding)* denir. Taban sınıfın türemiş sınıfa erişmesi ancak bu sanal işlev kullanımıyla mümkün olur. Bir dizi türetme yapıldığında türemiş sınıflardan birine ilişkin nesnenin adresi taban sınıflardan birine ilişkin bir göstericiye ya da referansa atanabilir.

Bu gösterici ya da referans yoluyla sanal işlev çağrılabilir.

Bir sanal işlev sınıf ismi belirtilerek çözünürlük işleci ile çağrılırsa sanallık özelliği kalmaz.

Sanal İşlevlerin Çağrılma Biçimleri

Sanal işlevler çeşitli biçimlerde çağrılabilir. Bu biçimleri gösterebilmek için önce aşağıdaki sınıfları tanımlayalım:

```
class Base {

//...
public:

    virtual void vfunc();

};


class Der: public Base {
public:

    virtual void vfunc();

};
```

1. Türemiş sınıf nesnesinin adresinin açık bir biçimde taban sınıf göstericisine atanması yoluyla:

```
int main()

{

    Der der_object;

    Base *base_ptr;

    base_ptr = &der_object;
    base_ptr->vfunc();
    (*base_ptr).vfunc();


    return 0;

}
```

Yukarıdaki örnekte

```
base_ptr->vfunc();
(*base_ptr).vfunc();
```

çağrılarıyla *Der* sınıfının *vfunc* işlevi çağrılır.

2. Bir işlevin parametre değişkeni taban sınıf türünden bir göstericidir. İşlev de bir türemiş sınıf nesnesinin adresiyle çağrılır. Parametre değişkeni olan gösterici yoluyla sanal işlev çağrılabilir.

```
void func(Base *ptr)
{
    ptr->vfunc();
}

int main()
{
    Der der_object;
    func(&der_object);
    return 0;
}
```

Yukarıdaki örnekte

```
func(&der_object);
```

deyimiyle çağrılan *func* işlevi içinde çağrılan *vfunc* işlevi *Der* sınıfının *vfunc* işlevidir.

3. Taban sınıf türünden bir referans türemiş sınıf türünden bir nesneyle ilkdeğer verilerek tanımlanır. Bu referans yoluyla sanal işlev çağrılabilir. Bu durumda türemiş sınıfa ilişkin sanal işlev çağrılır.

```
int main()
{
```

```
    Der der_object(10, 20);

    Base &base_ref = der_object;
    base_ref.vfunc();

    return 0;

}
```

Yukarıdaki örnekte

```
base_ref.vfunc();
```

deyimiyle çağrılan *vfunc* işlevi *Der* sınıfının *vfunc* işlevidir.

4. İşlevin parametre değişkeni taban sınıf türünden bir referans olur. İşlev de türemiş sınıf nesnesinin kendisiyle çağrılır. İşlev içinde bu referans yoluyla türemiş sınıfa ilişkin sanal işlev çağrılır.

```
void foo(Base &r)

{

    r.vfunc();

}


int main()

{

    der der_object(10, 20);
    foo(der_object);
    return 0;

}
```

Yukarıdaki örnekte

```
foo(der_object);
```

deyimiyle çağrılan *foo* işlevi içinde çağrılan *vfunc* işlevi *Der* sınıfının *vfunc* işlevidir.

5. Türemiş sınıf türünden bir nesne ya da gösterici ile taban sınıfa ilişkin sanal olmayan bir üye işlev çağrılmış olsun. Çağrılan bu üye işlev içinde sınıfın sanal bir işlevine çağrı yapıldığında, üye işlev hangi sınıfa ilişkin bir nesne için çağrılmışsa o sınıfa ilişkin sanal işlev çağrılır. Aşağıdaki örneği inceleyin:

```
#include <iostream>
using namespace std;


class Base {

//...
public:

    virtual void vfunc() {cout << "Base::vfunc()" << endl;}
    void nfunc();

};


class Der: public Base {
public:

    void vfunc(){cout << "Der::vfunc()" << endl;}

};


void Base::nfunc()

{

    cout << "Base::nfunc()" << endl;
    vfunc();

}

int main()

{

    Der der_object;
    der_object.nfunc();
    return 0;

}
```

Base sınıfından *Der* isimli bir sınıf türetiliyor. *Der* sınıfı *Base* sınıfının *vfunc* isimli sanal işlevini eziyor. Taban sınıfın sanal olmayan *nfunc* isimli bir işlevi olduğunu görüyorsunuz. *nfunc* işlevi içinde sanal *vfunc* işlevi çağrılıyor.

main işlevi içinde *Der* sınıfı türünden *der_object* isimli bir nesnenin yaratıldığını görüyorsunuz. Bu nesne ile taban sınıfın *nfunc* işlevi çağrılıyor. Bu durumda *nfunc* işlevi içinde çağrılan *Der* sınıfının *vfunc* işlevi olur.

Taban sınıf sanal işleve sahip olduğu halde türemiş sınıf sanal işleve sahip olmayabilir. Yani türemiş sınıfın taban sınıf olarak aldığı sınıftaki sanal işlevi ezmesi (*override*) zorunlu değildir. Bu durumda taban sınıf göstericisine bir türemiş sınıf nesnesinin adresi atanarak sanal işlevi çağrılırsa, taban sınıfın üye işlevi çağrılır.

Sanal işleve sahip olmayan türemiş sınıfa ilişkin bir sınıf nesnesinin adresi taban göstericisine atanır, bu gösterici yoluyla sanal işlev çağrılırsa türemiş sınıfın sanal işleve sahip ilk taban sınıfının sanal işlevi çağrılır.

```
#include <iostream>

using namespace std;


class A{
public:

    virtual void foo(){cout << "A::foo" << endl;}

};



class B: public A{
public:

     void foo(){cout << "B::foo" << endl;}

};



class C: public B{

    //...

};



class D: public C{

    //...

};



int main()

{

    D d;

    A *aptr = &d;
    aptr->foo();



    return 0;
```

```
}
```

Yukarıdaki örnekte *D* sınıfı türünden *d* nesnesinin adresi *A* sınıfı türünden *aptr* gösterici değişkenine atanıyor. *aptr* göstericisi ile sanal *foo* işlevi çağrıldığında, *B* sınıfının *foo* işlevi çağrılır. Çünkü *D* sınıfı ve *D* sınıfının doğrudan taban sınıfı olan *C* sınıfı sanal işlev olan *foo* işlevini ezmiştir. Hiyerarşi içinde işlevi ilk ezen sınıf *B* sınıfı olduğu için çağrılan işlev de *B* sınıfının işlevidir.

private Sanal İşlevler

Sanal işlev çağrılabilmesi için türetme biçiminin *public* olması gerekir. Bir sanal işlev çağrıldığında gerçekte çağrılacak olan türemiş sınıfın sanal işlevi türemiş sınıfın herhangi bir bölümünde olabilir. Ancak çağrı örneğin taban sınıf türünden bir referans ya da gösterici ile yapılıyorsa, çağrılan işlev taban sınıfın sanal *public* bölümde olmak zorundadır. Örneğin:

```
Base *base_ptr
Der der_object;

baseptr = &derObject;
baseptr->vfunc();
```

Burada *vfunc* sanal bir işlev olsun, gerçekte çağrılacak olan işlevin hangi sınıfın *vfunc* işlevi olduğu programın çalışma zamanında belirlenir. Ancak *vfunc* işlevinin taban sınıfın hangi bölümünde bildirildiği derleme zamanında araştırılır. *Der* sınıfının *vfunc* sanal işlevi *Der* sınıfının herhangi bir bölümünde bildirilmiş olabilir, ancak çağrının geçerli olabilmesi için *Base* sınıfının *vfunc* sanal işlevinin *Base* sınıfının *public* bölümünde bildirilmiş olması gerekir.

Sanal İşlev Çağırmanın Nedenleri

Sanal işlev çağırmanın başlıca iki faydalı nedeni vardır:

1. Bir sınıfın işlevini değiştirmek
2. Türden bağımsız işlemler yapılmasına olanak sağlamak

Örneğin *A* gibi bir sınıf varsa, bu sınıf belirli işlemleri yapıyorsa, bu sınıfa hiç dokunmadan sınıfın yaptığı işlemler üzerinde değişiklik yapılması sağlanabilir.

Sınıf hiyerarşisi içinde sınıfa ilişkin bir iş yapan işlevin, nesnesinin hangi sınıftan olduğunu saptayarak iş yapmasına dayanan tasarımlar kötü tekniktir. Bu tür kodlar yerine sanal işlev mekanizmasını kullanmak daha iyi tekniktir.

Sınıf hiyerarşisine zaman içinde yeni sınıflar katılabilir. Sınıf hiyerarşisine yeni bir sınıf eklendiğinde yalnızca eklenen sınıfın kodları yazılır, genel işlem yapan işlevlerin kodlarında bir değişiklik yapılması gerekmez. Sanallık mekanizmasının yeni sınıfı da kapsaması için daha önce yazılmış kaynak kodlarda da bir değişiklik yapılması gerekmez. Böyle bir kolaylık olmasaydı başkasının tasarladığı ve yazdığı sınıf hiyerarşilerine yeni türetmelerle

ekleme yapıldığı zaman, faydalanılan sınıfların yalnızca başlık dosyalarına değil kaynak kodlarına da sahip olmak gerekirdi.

Sanal işlevler kullanılarak bir sınıf hiyerarşisi içinde yer alan sınıfların ortak özelliklerine dayanarak kod yazılabilir.

Sanallık Devreden Çıkartılması

Sanal bir işlevin çağrılması hakkındaki kararın programın çalışma zamanında değil de *(late binding)* derleme zamanınında verilmesi *(early binding)* isteniyorsa, sanal işlev sınıf ismi ve çözünürlük işleci yardımıyla çağrılabilir:

```
#include <iostream>

class Base {

    int b;
public:

    Base(int val = 0): b(val) {}

    virtual void display()const {std::cout << b << std::endl;}

};


class Der: public Base {
    int d;

public:

    Der(int val1 = 0, int val2 = 0): Base(val1), d(val2) {}
    void display()const {std::cout << d << std::endl;}

};


int main()

{

    Base *base_ptr;

    Der derObject(10, 20);


    base_ptr = &derObject;

    base_ptr->Base::display(); //Base sınıfının display işlevi çağrılır!


    return 0;

}
```

Sanal İşlev Kullanılmasına İlişkin Örnekler

1. İngilizce yazılar üzerinde işlem yapan bir *Cstring* sınıfı olsun. Bu sınıfın yazıları karşılaştıran, büyük harf ya da küçük harfe dönüştüren üye işlevleri olsun. Yazıların karşılaştırılması ve harf dönüşümünün yapılması dile bağlı bir durumdur. *Cstring* sınıfının *compare* isimli karşılaştırma işlevi karşılaştırma işlemini yaparken iki karakteri karşılaştıran sanal *cmpchr* işlevini çağırıyor olsun. *Cstring* sınıfından bir sınıf türetilir. *cmpchr* sanal işlevi türetilmiş sınıf için yeniden yazılırsa, artık *compare* işlevi türemiş sınıfın *cmpchr* işlevini çağırır. Böylece işlemler başka bir dilde de doğru bir biçimde yapılabilir.

2. Bir dizinin *Array* isimli bir sınıf ile temsil edildiğini düşünelim. Bu sınıfın sıraya dizme işlemini yapan *sort* isimli sanal bir işlevi olsun. Bazı üye işlevler de bu işlevi çağırarak sıralama işlemini gerçekleştiriyor olsunlar. Sıralama algoritmaları çok çeşitli olabilir. *Array* sınıfından bir türetme yapılarak, türetilen sınıfın sanal işlevi başka bir sıralama algoritmasını kullanacak biçimde yeniden yazılabilir. Bu durumda sıralama işlemi istenilen algoritmayla yapılabilir.

3. *MFC* sınıf sisteminde her türlü pencere işlemleri *CWnd* sınıfı tarafından yapılmaktadır. Diyalog penceresi de özel bir tür penceredir. Diyalog penceresi işlemleri *CWnd* sınıfından türetilen *CDialog* sınıfı ile yapılmaktadır. Her diyalog penceresi diğerinden farklı özelliklere sahip olabilir. O işlemler de *CDialog* sınıfından türetilen sınıfla temsil edilir.

Diyalog penceresini görünür hale getirmek için *CDialog* sınıfının *DoModal* işlevi çağrılır. *CDialog* sınıfının *OnOk* ve *OnCance*l sanal işlevleri de vardır. *CWnd* sınıfından türetilen bir sınıfa ilişkin bir nesne tanımlandığında *CWnd* sınıfının kurucu işlev ile yaratılan nesnenin adresi *MFC* sistemi tarafından global bir biçimde saklanır. Ne zaman bir diyalog penceresinde *OK* ya da *Cancel* tuşlarına basılırsa *MFC* saklamış olduğu adresle *OnOk* ya da *OnCancel* işlevlerini çağırır. Eğer bu işlevler yeniden yazılırsa yeniden yazılan çağrılır. Tabii orjinal *OnOk* ve *OnCancel* işlevleri kritik bazı işlemleri de yapmaktadır. Bu durumda bu işlevlerin doğrudan çağrılmaları da gerekebilir.

```
void MyDialog::onOk()
{
    CDialog::OnOk();
    //...
}
```

Sanal İşlev Mekanizmasının Oluşturulması

Bir türemiş sınıf nesnesinin adresi taban sınıf göstericisiyle işlevden işleve dolaştırılmış olabilir. En sonunda sanal işlev çağrılmış olsa bile nesnenin ait olduğu sınıfa ilişkin sanal işlev çağrılır. Peki derleyici bu olayı derleme sırasında belirleyebilir mi? Bu olayın derleme sırasında saptanması mümkün değildir. İşlevlerin kodunun yürütülmesi programın çalışma zamanında gerçekleşir. Gerçekte hangi sanal işlevin çağrılacağını belirlemek ancak programın çalışma zamanı sırasında kurulacak bir mekanizmayla mümkün olabilir. Bu mekanizmanın çalışma zamanında kurulmasına ingilizce "*late binding*" ya da "*dynamic binding*" denir. Hangi işlevin

çağrılacağına programın çalışma zamanında değil de derleme zamanında karar verilmesine ise ingilizcede *"early binding*" ya da *"static binding"* denir.

Aşağıdaki kodu inceleyin:

```
#include <iostream>
#include <cstdlib>
#include <ctime>


class Calisan {
public:

    virtual void kendini_tanit() {std::cout << "Ben calisanim" <<
std::endl;}

};


class Mudur:public Calisan {
public:

    virtual void kendini_tanit(){std::cout << "Ben mudurum" << std::endl;}

};


class GenelMudur:public Mudur{
public:

    virtual void kendini_tanit(){std::cout << "Ben genel mudurum" <<
std::endl;}

};


class Sekreter: public Calisan {
public:

    virtual void kendini_tanit(){std::cout << "Ben sekreterim" <<
std::endl;}

};


class Sofor: public Calisan {
public:

    virtual void kendini_tanit(){std::cout << "Ben soforum" << std::endl;}

};
```

```
class Isci: public Calisan {
public:

    virtual void kendini_tanit(){std::cout << "Ben isciyim" << std::endl;}

};



Calisan *calisan_yarat()

{

    Calisan *pd;

    switch (rand() % 5) {

    case 0: pd = new Mudur; break;

    case 1: pd = new GenelMudur; break;
    case 2: pd = new Sekreter; break;
    case 3: pd = new Sofor; break;

    case 4: pd = new Isci;

    }

    return pd;

}


int main()

{

    srand(time(0));



    for (int k = 0; k < 20; ++k) {
        Calisan *pd = calisan_yarat();
        pd->kendini_tanit();

        delete pd;

    }

    return 0;

}
```

Yukarıdaki örnekte *Calisan* sınıfından *Mudur* sınıfı, *Mudur* sınıfından *GenelMudur* sınıfı türetiliyor. *Calisan* sınıfından ayrıca *Sekreter*, *Sofor* ve *Isci* sınıfları türetiliyor. Türetilen tüm sınıflar *Calisan* sınıfının sanal *kendini_tanit* isimli işlevini eziyor. Global *calisan_yarat* işlevi içinde, türetme hiyerarşisi içinde yer alan sınıflardan birine ait rastgele bir dinamik nesnenin yaratıldığını görüyorsunuz. *rand* işlevinin ürettiği rastgele değere göre yaratılan dinamik nesnenin türü *Mudur*, *GenelMudur*, *Sekreter*, *Sofor*, *Isci* olabilir. Dinamik nesnenin hangi türden olacağı programın çalışma zamanında belli olur. *calisan_yarat* işlevi yaratilan dinamik sınıf nesnesinin adresini döndürüyor.

main işlevi içinde bir döngü içinde önce *calisan_yarat* işlevi çağrılarak elde edilen adres *Calisan* sınıfı türünden bir gösterici değişkende tutuluyor. Sonra bu gösterici değişkenle sanal işlev *kendini_tanit* çağrılıyor. Çağrılan *kendini_tanit* işlevi hangi sınıfın sanal işlevidir?

Peki hangi işlevin çağrılacağının çalışma zamanında belirlenmesi nasıl mümkün oluyor? Sanal işlev çağrı mekanizmasının derleyiciler tarafından ne şekilde sağlanacağı standart olarak belirlenmemiştir. Yani derleyiciler bu mekanizmayı istedikleri gibi oluşturabilir.

Ancak derleyicilerin çoğu bu mekanizmayı kurabilmek için her sınıfa ilişkin bir sanal işlev tablosu yaratır. Bu tablolar kısaca *vtable* (*virtual table*) olarak isimlendirilir. Bu sanal işlev tablolarında ilgili sınıfın sanal işlevlerin adresleri bulunur. Sanal işleve sahip bir sınıfa ilişkin bir nesne tanımlandığında o nesne için bir gizli işlev göstericisi kadar daha ek yer ayrılır. Bu gizli göstericide sınıfın sanal işlev tablosunun adresi tutulur. Bu gizli göstericinin nesnenin neresinde tutulduğu standart olarak belirlenmemiştir. Ancak popüler derleyicilerin hemen hemen hepsinde nesnenin en düşük anlamlı adresinde tutulur.

Base Sınıfının Sanal İşlev Tablosu	
Sıra No	*Adres*
1	&Base::func1()
2	&Base::func2()

Der1 Sınıfının Sanal İşlev Tablosu	
Sıra No	*Adres*
1	&Der1::func1()
2	&Der2::func2()

Der2 Sınıfının Sanal İşlev Tablosu	
Sıra No	*Adres*
1	&Der2::func1()
2	&Der2::func2()

Bu durumda bir sanal işlev çağrıldığında aşağı seviyeli şu işlemler yapılır:

1. Sanal işlev göstericisi (*vpointer*) alınır. Sanal işlev tablosunun yeri (*vtable*) bulunur.
2. Sanal işlev tablosunda ilgili sanal işlevin adresi bulunur.
3. Adresi bulunan sanal işlev çağrılır.

Sanal İşlevlerin Maliyeti

Sanal işlev mekanizması şüphesiz ek bir bellek kullanımına ve daha fazla işlemci zamanına neden olur. Ek bir bellek kullanımı söz konusudur. Çünkü her çokbiçimli sınıf için bir sanal işlev tablosu ve her çokbiçimli sınıf nesnesi için sanal işlev tablo göstericisi bellekte fazladan bir yer kaplar. Sanal işlevin çağrılabilmesi için önce sanal işlev tablosunun adresinin alınarak tabloya erişilmesi ve tablodan da çağrılacak işlevin adresinin alınması gerekir. Yani fazladan iki "içerik alma" (*indirection)* işlemi yapılır.

Ancak ortalama bir uygulama programında sanal işlevlerin getirdiği ek bellek ve zamansal maliyet yok sayılabilir. Sanal işlevlerin kazandırdığı faydanın yanında getirdiği ek maliyetler gerçekten önemsenmeyecek boyuttadır.

Sanal Sonlandırıcı İşlevler

Bir sınıf hiyerarşisi içinde dinamik olarak yaratılan sınıf nesneleri için çalışma zamanına ilişkin çokbiçimliliğin uygulanmasında bazı istenmeyen durumlar ortaya çıkabilir.

Daha önceki bölümlerden hatırlayacağınız gibi *new* işleciyle sınıf türünden bir nesne yaratıldığı zaman ilgili sınıfın kurucu işlevi çağrılır. *delete* işleciyle dinamik olarak yaratılan nesneye ilişkin blok *free store*"a geri verildiğinde ilgili sınıf nesnesi için sonlandırıcı işlevi çağrılır. Aşağıdaki kodu inceleyin:

```
class Base {

    char *base_ptr;
public:

    Base();

    ~Base();

};



#include <iostream>
using namespace std;



Base::Base()

{

    cout << "Base::Base()" << endl;
    base_ptr = new char[1000];

    cout << "Base::Base()icinde 1000 byte'lık blok elde edildi!" << endl;
```

```
}


Base::~Base()

{

    cout << "Base::~Base()" << endl;
    delete[] base_ptr;

    cout << "Base::~Base icinde 1000 byte'lik blok geri verildi!" << endl;

}


class Der: public Base {
    char *der_ptr;

public:

    Der();

    ~Der();

};



Der::Der()

{

    cout << "Der::Der()" << endl;
    der_ptr = new char[2000];

    cout << "Der::Der()icinde 2000 byte'lık blok elde edildi!" << endl;

}


Der::~Der()

{

    cout << "Der::~Der()" << endl;
    delete[] der_ptr;

    cout << "Der::~Der icinde 2000 byte'lik blok geri verildi!" << endl;

}


int main()

{

    Base *ptr = new Der;
    delete ptr;
```

```
    return 0;
}
```

Çalıştırılan programın ekran çıktısı:

```
Base::Base()
Base::Base()icinde 1000 byte'lık dinamik blok elde edildi!
Der::Der()
Der::Der()icinde 1000 byte'lık dinamik blok elde edildi!
Base::~Base()
Base::~Base icinde 1000 byte'lik blok geri verildi!
```

Sınıfın sonlandırıcı işlev (*virtual destructor*) da sınıfın bir üye işlevi olduğuna göre, *delete*

işlecinin terimi olan adres hangi sınıf türünden ise o sınıfın sonlandırıcı işlevi çağrılır. Yukarıdaki programda dinamik olarak yaratılan türemiş sınıf (*Der*) türünden adres, taban sınıf türünden *ptr* göstericisine atanıyor. Ancak *delete* işlecinin terimi *ptr* olduğundan doğal olarak taban sınıfa (*Base*) ilişkin sonlandırıcı işlev çağrılır. Bazı durumlarda dinamik yaratılan sınıf nesnesi için, türemiş sınıfın değil taban sınıfın sonlandırıcı işlevinin çağrılması programın tamamen yanlış ya da zararlı bir şekilde çalışmasına neden olabilir. Örneğin türemiş sınıf nesnesi taban sınıf alt nesnesinin dışında ayrı bir kaynak kullanıyor olabilir. Türemiş sınıf nesnesine bu kaynak, türemiş sınıfın kurucu işlevinin çağrılmasıyla bağlanır. Türemiş sınıf nesnesine ilişkin sonlandırıcı işlevi, nesneye bağlanan kaynağı geri veriyor ya da serbest bırakıyor olabilir. Türemiş sınıfın sonlandırıcı işlevi çağrılmadığı zaman bağlanan kaynağın geri verilme işlemi de yapılamaz.

Sonlandırıcı işlevler de *virtual* anahtar sözcüğü ile bildirilebilir. Bu durumda sonlandırıcı işlev, tıpkı diğer sanal işlevler gibi çok biçimli davranış gösterir.

Yukarıdaki kod parçasında taban sınıfın sonlandırıcı işlevinin bildirimine *virtual* anahtar sözcüğünü ekleyelim:

```
class Base {
public:
    virtual ~base();
    //…
};
```

Artık *Base* sınıfı türünden bir gösterici *delete* işlecinin terimi yapıldığında, gösterici değişkene hangi sınıf türünden bir adres atanmışsa o sınıfa ilişkin sonlandırıcı işlev çağrılır. Yani yukarıdaki değişiklikten sonra, çalıştırılan programda çağrılan *Base* sınıfının değil *Der* sınıfının sonlandırıcı işlevi olur.

Türemiş sınıf sonlandırıcı işlevinin kodunun yürütülmesinin sonunda türemiş sınıf nesnesinin taban sınıf kısmı için taban sınıfın sonlandırıcı işlevinin de çağrılacağını unutmayalım. Yoksa bu kez de taban sınıf nesnesine bağlanmış kaynaklar serbest bırakılamazdı. Yukarıdaki değişiklikten sonra program çalıştırıldığında ekrana

```
Der sınıfının sonlandırıcı işlevi çağrıldı!..
Base sınıfının sonlandırıcı işlevi çağrıldı!..
```

yazar.

Genel olarak şu kural verilebilir: Bir sınıfın eğer en az bir sanal işlevi varsa, sınıfın sonlandırıcı işlevi de sanal yapılmalıdır. Çoğu zaman sanal sonlandırıcı işlev bir koda sahip olmaz ve boş olarak tanımlanır. Bu durumda işlev sınıfın *inline* işlevi olarak yazılır.

```
class Myclass {
public:

    virtual void func();
    virtual ~Myclass() {}

};
```

Saf Sanal İşlevler

Bir sınıf hiyerarşisi oluşturulmasının çoğu zaman amacı, mantıksal olarak ilişki halindeki sınıfların ortak olarak paylaşılabilecek kodlarını bir taban sınıf içinde toplamaktır. Böylece genel işlem yapan kodlar en tepedeki taban sınıfa göre yazılabilir. Daha önce yazılan kodlar, daha sonra yazılacak kodları yani türetilmiş sınıflara ilişkin kodları çağırabilir.

Ancak bazen taban sınıflar, kendilerinden türetilecek sınıflara yalnızca bir arayüz sağlamak amacıyla oluşturulur. Ama taban sınıfın sağladığı arayüze ilişkin türemiş sınıflara sunacağı bir kodun *(implementasyon)* varlığı söz konusu olmayabilir. Başka bir deyişle taban sınıf, kendisinden türetilecek sınıflara kendi arayüzünü vermekte ancak bu arayüze ilişkin kodları türemiş sınıfın kendisinin tanımlamasını istemektedir.

Örnek olarak *Shape* isimli bir sınıf tanımlandığını ve bu sınıftan *Triangle*, *Square*, *Circle*

sınıflarının türetildiğini düşünelim. *Shape* sınıfının *draw* isimli bir sanal üye işlevi olsun:

```
class Shape {
    public:

    //...

    virtual void draw () const;

    //...
```

```
};
```

Bir işlevin sanal olması ne anlama gelir? Sanal bir işlev türemiş sınıf tarafından ezilebilir (*override* edilebilir). Ancak bu bir zorunluluk değildir. Türemiş sınıf taban sınıfın tanımladığı sanal işlevi ezmez ise türemiş sınıfa ilişkin bir nesne ile sanal işlev çağrıldığında, taban sınıfın üye işlevi çağrılır. Tasarım açısından bakıldığında bu durumun anlamı şudur: Sanal bir işlev, hem arayüzü hem de default kodu (*default implementation*) devir alınan bir işlevidir. Sanal bir işlevin türemiş bir sınıfa verdiği ileti şudur: *"Beni istediğin gibi tanımlayabilirsin. Tanımlarsan, senin tanımladığın çağrılır, ancak tanımlamaz isen çağrılan ben olacağım!"*

Yine *Shape* sınıfına dönelim. *Shape* sınıfından türeyecek sınıflar *draw* işlevini ezmezlerse taban sınıfın tanımladığı *draw* işlevi çağrılır. Peki *Shape* sınıfının *draw* işlevi ne yapabilir? Bir karenin ya da bir üçgenin çizilmesi mantıksal olarak bir anlam taşır. Ama bir *"Şekil"* in çizilmesi ne demektir? *Shape* sınıfının *draw* işlevi için sağlayabileceği bir kod parçası yoktur. *Shape* sınıfı *draw* işlevini yalnızca türemiş sınıflar için bir arayüz oluşturması amacıyla bildirmiştir.

Evet bazı durumlarda bir taban sınıf sanal işlevinin bildirilmesinin tek nedeni kendisinden türetilecek sınıflara uygun bir arayüz sunmaktır. Bu nedenle taban sınıf sanal işlevinin tanımının yapılması mantıklı değildir.

C++'da bu durum *saf sanal* (*pure virtual*) işlevlerle gerçekleştirilir.

Bir işlevin saf sanal olarak bildirilmesi işlevin bildiriminde işleve atama işlecinin kullanılarak *0* değerinin atanmasıyla gösterilir. Bu gerçek bir atama işlemi değildir. Yalnızca bildirilen işlevin saf sanal olduğunu gösteren bir sözdizim kuralıdır.

```
class Shape {
public:

    virtual void draw() = 0;   //saf sanal işlev

    //...

};
```

Yukarıda tanımlanan *Shape* sınıfının *draw* isimli işlevi saf sanal yapılmış. Saf sanal bir işlev, kendisinden türetilecek sınıflara arayüz sağlayan, kendi kodu olmayabilen bir işlevdir.

Saf sanal işlevin tanımlanmaması durumunda bağlama zamanında bir hata oluşmaz. Peki saf sanal işlevin tanımlanması geçerli bir durum ise, bir taban sınıf nesnesi ile taban sınıfın saf sanal işlev çağrıldığında ne olur?

En az bir saf sanal işlev içeren sınıfa "soyut sınıf" (*abstract class*) denir. Bir soyut sınıf türünden nesne yaratılamaz. Bir soyut sınıf türünden nesne yaratılması durumunda derleme zamanında hata oluşur. Aşağıdaki kodu inceleyin:

```
class Shape {
public:

    virtual void draw() = 0;   //saf sanal işlev

};
```

```
int main()

{

    Shape myshape;          //Geçersiz!

    Shape *shape_ptr = new Shape; //Geçersiz!


    return 0;

}
```

Yukarıdaki *main* işlevi içinde yazılan her iki deyim de geçersizdir. Her iki deyimde de soyut *Shape* sınıfı türünden bir nesne yaratılmak istenmiştir.

Türemiş sınıf taban sınıfın saf sanal işlevini ezmelidir. Ezmezse, çağrılacak bir taban sınıf işlevi yoktur. Türemiş sınıf bir soyut sınıfın herhangi bir saf sanal işlevini ezmezse, bu durumda türemiş sınıf da soyut olur. Yani türemiş sınıf türünden de bir nesne tanımlanamaz. Saf sanal bir işlevin kendisinden türetilecek sınıflara verdiği ileti şudur: *"Beni tanımlaman gerekiyor. Çünkü ben yalnızca bir arayüzüm. Beni tanımlamaz isen sen de benim gibi bir soyut sınıf olursun. Senin türünden de nesne yaratma olanağı olmaz."* Soyut bir sınıf türünden nesne yaratılmaz, ancak şüphesiz soyut bir sınıf türünden bir gösterici değişken ya da referans tanımlanabilir. Böylece sınıf hiyerarşisi üzerinde ortak ve genel işlem yapacak kodlar, yine soyut taban sınıf türünden referanslar ya da gösterici değişkenler kullanılarak yazılabilir.

Aşağıdaki *Shape* sınıfının bildiriminde yer alan *draw* ve *write_shape_name* isimli saf sanal işlevler, türetilmiş sınıflar tarafından yeniden tanımlanıyor:

```
class Shape {
public:

    virtual void draw() const = 0;

    virtual void write_shape_name() const  = 0;

};
```

Shape sınıfından doğrudan ya da dolaylı olarak türetilmiş sınıflardan herhangi birine ait bir nesne üzerinde işlemler yapmak üzere tanımlanan *process* isimli global bir işlevin yazıldığını düşünelim:

```
void process(Shape &r)

{

    r.write_shape_name();

    r. draw();

}
```

process işlevine hangi sınıf türünden nesne gönderilirse o nesnenin *write_shape_name* ve *draw* işlevleri çağrılır. *Shape* sınıfının ve *process* işlevinin yazılmasından yıllarca sonra bile, *Shape* sınıfından yeni bir sınıf türetilip o türemiş sınıf türünden bir nesne tanımlansa, o nesne ile *process* işlevi çağrıldığında mantıksal olarak istenen iş yapılır. Yani *process* işlevi içinde yeni türetilmiş sınıfa ilişkin *write_shape_name* ve *draw* işlevleri çağrılır.

Shape sınıfının ara yüzü saf sanal işlevleriyle, türetilecek sınıfların tanımlamasının zorunlu olduğu üye işlevlerini gösterir.

Bir sınıf soyut değil ise, yani bir sınıf türünden nesne tanımlanabiliyor ise bu sınıfa "somut sınıf" (*concrete class*) denir.

Bir saf sanal işlevin tanımlanmaması, derleme ya da bağlama aşamasında bir hata oluşturmaz. Ancak saf sanal işlevler istenirse tanımlanabilir. Programcı türetme hiyerarşisi için ortak olan bir kod parçasını isterse taban sınıfın saf sanal işlevi içinde toplayabilir. Bu durum az da olsa bazı kod kalıplarında karşımıza çıkar.

Saf Sanal Sonlandırıcı İşlevler

Sonlandırıcı işlevler de saf sanal olarak bildirilebilir. Ancak saf sanal sonlandırıcı işlevlerin diğer saf sanal işlevlere göre önemli bir farkı vardır. Bu işlevler tanımlanmak zorundadır. Eğer tanımlanmazlar ise bu sınıftan türemiş bir somut sınıf türünden bir nesne yaratıldığında bağlama zamanında hata oluşur. Türemiş sınıf nesnesinin sonlandırıcı işlevinin kodu sonunda taban sınıfın sonlandırıcı işlevinin çağrıldığını hatırlayalım. Taban sınıfın sonlandırıcı işlevi saf sanal da olsa, işlevin tanımı yoksa, işlev çağrısı bağlayıcı tarafından hata olarak işaretlenir. Aşağıdaki kod parçasında saf sanal sonlandırıcı işlevin de normal sonlandırıcı işlevler gibi çağrıldığı gösteriliyor:

```
#include <iostream>


class Shape {
public:

    virtual ~Shape() = 0;

};


Shape::~Shape()

{

    std::cout << "Shape::~Shape()" << std::endl;

}


class Square:public Shape {
public:

    ~Square() {std::cout << "Square::~Square()" << std::endl;}

};
```

```
int main()

{

    {

        Square s;

    }

    Shape *ptr = new Square;
    delete ptr;



    return 0;

}
```

Programın ekran çıktısı:

```
Square::~Square()
Shape::~Shape()
Square::~Square()
Shape::~Shape()
```

Taban sınıfın saf sanal bir işlevinin türemiş sınıf tarafından ezilmemesi durumunda türemiş sınıfın da soyut sınıf olur. Ancak bu kural saf sanal sonlandırıcı işlev için geçerli değildir. Yani taban sınıfın saf sanal sonlandırıcı işlevini türemiş sınıf ezmese de soyut sınıf olarak ele alınmaz. Halen türemiş sınıf türünden bir nesne tanımlanabilir. Soyut bir sınıfın tek saf sanal işlevinin saf sanal sonlandırıcı işlev olması durumu özel bir durumdur. Zira bu durumda türemiş sınıf bu işlevi yeniden tanımlamasa da soyut bir sınıf olarak ele alınmaz. Yani taban sınıftan türetilen bütün sınıflar somut sınıflar olarak ele alınır.

Kurucu İşlevler İçinde Yapılan Sanal İşlev Çağrıları

Bir sınıfın kurucu işlevi içinde sanal bir işlev çağrılırsa sanallık mekanizması devreye girmez. Aşağıdaki kodu inceleyin:

```
#include <iostream>

class Base {

    //...
public:

    Base();

    virtual void vfunc();

};
```

```
using namespace std;

Base::Base()

{

    cout << "Base::Base()" << endl;
    vfunc();

}


void Base::vfunc()

{

    cout << "Base::vfunc()" << endl;

}


class Der:public Base {
public:

    Der();

    void vfunc();

};


Der::Der()

{

    cout << "Der::Der()" << endl;

}


void Der::vfunc()

{

    cout << "Der::vfunc()" << endl;

}


int main()

{

    Der der_object;


    return 0;

}
```

Yukarıdaki örnekte *Base* sınıfından türetilen *Der* sınıfı *Base* sınıfının *vfunc* işlevini eziyor. *main* işlevi içinde *Der* sınıfı türünden bir nesne tanımlandığını görüyorsunuz. *Der* sınıfının kurucu işlevinin başında taban sınıf olan *Base* sınıfının kurucu işlevi çağrılır. *Base* sınıfının kurucu işlevi içinde sanal *vfunc* işlevi çağrılıyor.

```
vfunc();
```

çağrısı aslında

```
this->vfunc();
```

çağrısıdır.

Normal olarak *this* göstericisi içinde *Der* sınıfı türünden bir adres olduğuna göre çağrılan *vfunc* işlevini türemiş sınıfın *vfunc* işlevi olması gerekir. Ancak çağrılan *Der* sınıfının değil *Base* sınıfının *vfunc* işlevidir. Bu C++ dilinin bir kuralıdır. Peki kurucu işlev içinde neden sanallık mekanizması devreye girmiyor? Türemiş sınıf nesnesi yaratıldığında önce taban sınıf kısmı oluşturulur. Bu da taban sınıfın kurucu işlev tarafından yapılır. Taban sınıfın kurucu işlevi içinde çağrılan sanal işlev türemiş sınıfın sanal işlevi olsaydı, bu durumda bu işlev ilkdeğerini almamış elemanlar üzerinde işlem yapmak durumunda kalabilirdi.

Türemiş sınıf nesnesinin bazı kaynaklar kullandığını düşünelim. Bu kaynaklar türemiş sınıf nesnesine türemiş sınıf nesnesinin kurucu işlevi tarafından bağlanır. *vfunc* işlevi içinde bu kaynakların kullanıldığını düşünelim. Eğer sanallık mekanizması devreye girseydi, türemiş sınıfın *vfunc* işlevi bu kaynaklar henüz nesneye bağlanmadan bu kaynakları kullanmaya çalışırdı. Bu da çalışma zamanı hatalarına neden olurdu.

Sonlandırıcı İşlevler İçinde Yapılan Sanal İşlevÇağrıları

Benzer durum sonlandırıcı işlevler için de geçerlidir. Bir sonlandırıcı işlev içinde sanal bir işlev çağrıldığında sanallık mekanizması devreye girmez. Aşağıdaki örneği inceleyin:

```
class Base {
public:

    ~Base();

    virtual void vfunc();

//...

};


#include <iostream>

using namespace std;


Base::~Base()
```

```
{
    std::cout << "Base::~Base()" << std::endl;
    vfunc();
}

void Base::vfunc()
{
    std::cout << "Base::vfunc()" << std::endl;
}

class Der:public Base {
public:
    ~Der();
    void vfunc();
    //...
};

Der::~Der()
{
    std::cout << "Der::~Der()" << std::endl;
}

void Der::vfunc()
{
    std::cout << "Der::vfunc()" << std::endl;
}

int main()
{
    Der der_object;

    return 0;
}
```

main işlevi içinde yaratılan *der_object* nesnesinin ömrü sona erdiğinde bu nesne için sonlandırıcı işlev çağrılır. *Der* sınıfının sonlandırıcı işlevinin kodunun sonunda derleyici tarafından eklenen kod ile taban sınıf olan *Base* sınıfının sonlandırıcı işlevi çağrılır. *Base* sınıfının sonlandırıcı işlevi içinde yapılan *vfunc* çağrısında sanallık mekanizması devreye girmez. Taban sınıfın sonlandırıcı işlevi çağrıldığında artık türemiş sınıf nesnesi kısmı yok edilmiştir. Eğer sanallık mekanizması devreye girseydi, yani türemiş sınıfın *vfunc* işlevi çağrılsaydı, bu işlev artık var olmayan kaynaklar üzerinde işlem yapmaya çalışabilirdi. Bu da çalışma zamanı hatalarının oluşmasına neden olurdu.

Taban Sınıf Nesnesiyle Yapılan Sanal İşlev Çağrısı

Sanal işlev mekanizmasının devreye girmesi için sanal işlev çağrısının bir gösterici ya da bir referans yoluyla yapılması gerekir. Bir türemiş sınıf nesnesi bir taban sınıf nesnesine atandıktan sonra taban sınıf nesnesiyle bir sanal işlev çağrılırsa sanallık mekanizması devreye girmez. Aşağıdaki örneği inceleyin:

```
#include <iostream>


class Base {
public:

    virtual void vfunc()const {std::cout << "Base::vfunc()" << std::endl;}

};


class Der: public Base {
public:

    void vfunc()const {std::cout << "Der::vfunc()" << std::endl;}

};


void func1(Base b)

{

    b.vfunc();

}


void func2(Base *ptr)

{

    ptr->vfunc();

}


int main()
```

```
{

    Der der;
    func1(der);
    func2(&der);



    return 0;

}
```

main işlevi içinde yapılan

```
func1(der);
```

çağrısında sanallık mekanizması devreye girmez. *func1* işlevi değerle çağrılıyor. Argüman olan *der* nesnesi işlevin parametre değişkeni olan *Base* türünden *b* nesnesine atanırken nesne dilimlenir (*object slicing*). İşlevin parametre değişkeni olan *b* nesnesi ile yalnızca *Base* sınıfının *vfunc* işlevi çağrılabilir. Oysa

```
func2(&der);
```

çağrısında sanallık mekanizması devreye girer, çünkü *func1* işlevi adresle çağrılmıştır. Programın çalışma zamanında **ptr* nesnesi hangi türden ise, o sınıfın *vfunc* işlevi çağrılır. **Kurucu İşlevler Sanal Olabilir mi**

Bir sınıfın kurucu işlevi sanal olamaz. *virtual* anahtar sözcüğünün bir kurucu işlevin bildiriminde kullanılması bir sözdizim hatasıdır. Ancak bazı uygulamalarda "sanal bir kurucu işleve" gereksinim duyulur. Aşağıdaki gibi bir işlevin varlığını düşünelim:

```
void func(Base *base_ptr)

{

Base *d_object  =  //new (base_ptr'nin içinde hangi sınıf türünden  //adres
varsa o sınıf türünden bir dinamik nesne yaratılacak)

    //..

}
```

Yukarıdaki *func* işlevinin taban sınıf türünden bir göstericiye ya da referansa sahip olduğunu düşünelim. Taban sınıf türünden bir göstericiye herhangi bir türemiş sınıf türünden nesnenin adresinin atanabileceğini biliyorsunuz. *func* işlevi sınıf hiyerarşisi içinde hangi sınıf türünden bir nesnenin adresi ile çağrılırsa işlev içinde o sınıf türünden dinamik bir nesne yaratılmak istendiğini düşünelim. Bu nasıl

yapılabilir? Adeta sanal bir kurucu işleve gereksinim duyuluyor. Bu durumu sağlamak için kullanılan idiyoma "sanal kurucu işlev idiyomu" (*virtual constructor idiom*) denir. Aşağıdaki kod parçasını inceleyin:

```
#include <iostream>


class Base {
public:

    virtual Base *create_similar() const = 0;
    virtual Base *clone() const = 0;

    virtual ~Base() {}

    //...

};


class Der1: public Base {
public:

    Der1() {std::cout << "Der1::Der1()" << std::endl;}
    Base *create_similar()const {return new Der1;}
    Base *clone() const {return new Der1(*this);}

    //...

};


class Der2: public Base {
public:

    Der2() {std::cout << "Der2::Der2()" << std::endl;}
    Base *create_similar() const {return new Der2;}
    Base *clone()const {return new Der2(*this);}

    //...

};


class Der3: public Base {
public:

    Der3() {std::cout << "Der3::Der3()" << std::endl;}
    Base *create_similar()const {return new Der3;}
    Base *clone()const {return new Der3(*this);}

};


void func(Base *ptr)
```

```
{

    Base *ptr1 = ptr->create_similar();
    Base *ptr2 = ptr->clone();

    delete ptr1;
    delete ptr2;

}

using namespace std;


int main()

{

    cout << "Der1, Der2 ve Der3 nesneleri yaratiliyor." << endl;
    Der1 der1;

    Der2 der2;

    Der3 der3;



    cout << "Her bir nesne sirasiyla func işlevine gonderiliyor" << endl;
    func(&der1);

    func(&der2);
    func(&der3);



    return 0;

}
```

Taban sınıf olan *Base* sınıfı içinde isimleri *create_similar* ve *clone* olan iki saf sanal işlev bildiriliyor. *Base* sınıfından türeyen her sınıf *create_similar* ve *clone* sanal işlevlerini ezer. *Base* sınıfından türeyen her sınıf - örneğimizde *Der1*, *Der2* ve *Der3* sınıfları- ezdikleri *create_similar* işlevlerini kendi sınıfları türünden dinamik bir nesnenin adresiyle geri döndürür. *clone* işlevinin tek farkı ise adresi döndürülen dinamik nesnenin varsayılan kurucu işlev ile değil kopyalayan kurucu işlevle ilkdeğerini almasıdır. Yani *clone* işlevi hangi sınıf nesnesi için çağrılmışa o nesnenin değerine eşit değerde dinamik bir nesnenin adresini döndürür. Böylece taban sınıf türünden bir gösterici ya da referans ile sanal *create_similar* ya da *clone* işlevleri çağrıldığında, sanal işlev mekanizması devreye girer. Taban sınıf göstericisi içinde hangi nesnenin adresi varsa o sınıfın sanal işlevinin çağrılmasıyla o sınıf türünden bir dinamik nesne yaratılmış olur.

Sanal İşlevler ve Varsayılan Argümanlar

Sanal işlevler de varsayılan argüman alabilir. Taban sınıfın sanal işlevini ezen türemiş sınıf işlevi farklı bir varsayılan argümanla bildirilebilir:

```
#include <iostream>


class Base {
public:

    virtual void func(int val = 20) {std::cout << "val = " << val <<
std::endl;}

};


class Der : public Base{
public:

    virtual void func(int val = 100) {std::cout << "val = " << val <<
std::endl;}

};


int main()

{

    Der der;

    Base *bptr = &der;
    bptr->func();


    return 0;

}
```

Yukarıdaki örnekte *bptr* göstericisine *Der* sınıfı türünden *der* isimli nesnenin adresi atanıyor. Daha sonra *bptr* göstericisi ile sanal *func* işlevi argüman gönderilmeden çağrılıyor. İşleve varsayılan argüman olarak hangi değer geçilir? *20* mi *100* mü? Sanal işlevin bağlanması programın çalışma zamanında olsa da, varsayılan argüman degerlendirilmesi programın derleme zamanında yapılır. Yukarıdaki program çalıştırıldığında ekrana *20* değeri yazdırılır.

Ezilen Bir Sanal İşlevin Parametrik Yapısı

Taban sınıfın bir sanal işlevinin türemiş sınıf tarafından ezilmesi için türemiş sınıfın tamamen aynı parametrik yapıya sahip bir işlev bildirmesi gerekir. Türemiş sınıfın bildirdiği işlevin imzasında bir farklılık olursa, türemiş sınıf taban sınıfın sanal işlevini ezmiş olmaz. Yeni bir işlev bildirmiş olur. Türemiş sınıfın bildirdiği işlevin imzası taban sınıfın sanal işlevi ile aynı fakat geri dönüş değeri türü farklı ise bu durum derleme zamanında hata olarak değerlendirilir:

```
class Base {
public:

    virtual int func1(int);
    virtual int func2(int);
    virtual int func3(int);

};


class Der: public Base{
public:

    int func1(int);
    int  func2(double);

    double func3(int);     //Geçersiz

};
```

Yukarıdaki bildirimleri inceleyelim:

Base sınıfından türetilen *Der* sınıfı sanal *func1* işlevini eziyor. Ancak sanal *func2* işlevini ezmiyor. Çünkü bildirilen *func2* işlevinin imzası taban sınıfın bildirdiği sanal *func2* işlevinin imzasından farklıdır. Türemiş sınıf içinde yapılan *func3* işlevi bildirimi ise derleme zamanında hataya neden olur.

Türemiş sınıfın taban sınıfın sanal işlevini ezebilmesi için, tamamen aynı parametrik yapıya sahip bir işlev bildirmesi gerekir.

Ancak bu durumun bir istisnası vardır (*variant return type*): Taban sınıfın sanal işlevi bir sınıf türünden adrese ya da referansa geri dönüyorsa, türemiş sınıfın işlevinin geri dönüş değeri, taban sınıfın geri döndürdüğü sınıftan türeyen bir sınıfa ilişkin bir türden olabilir. Aşağıdaki örneği inceleyin:

```
class B {

    //...

};


class D : public B {

    //...

};
```

```
class Base {
public:

    virtual B *vfunc1();
    virtual B &vfunc2();

};


class Der: public Base {
public:

    D *vfunc1();

    D &vfunc2();

};
```

Yukarıdaki örnekte *Base* sınıfı içinde iki ayrı sanal işlev bildiriliyor. Sanal *vfunc1* işlevinin geri dönüş değeri *B** türündendir. Oysa *Base* sınıfından türetilmiş *Der* sınıfı bu işlevi ezerken işlevin geri dönüş değerinin türünü *D** olarak bildiriyor. *D* sınıfı *B* sınıfından türetildiği için bu geçerli bir durumdur.

Benzer durum referans ile geri dönen sanal işlevler için de geçerlidir. *Base* sınıfının sanal *vfunc2* işlevi *B&* türüne geri dönerken bunu ezen *Der* sınıfının *vfunc2* işlevi *D&* türüne geri dönüyor.

Bu kural neden getirilmiş olabilir?

Der sınıfı türünden bir nesne ile *vfunc1* ya da *vfunc2* işlevinin çağrıldığını düşünelim:

```
Der der_object;

D *ptr = der_object.vfunc1();
D &r = der_object.vfunc2();
```

Eğer türemiş sınıfın *vfunc1* ve *vfunc2* işlevleri *A** ve *A&* türlerine geri dönselerdi yukarıdaki işlev çağrıları geçerli olmazdı.

Çokbiçimlilik

Nesne yönelimli programlama tekniğinde sanal işlevlerin kullanılarak işlev çağrılarının bağlanmasının programın çalışma zamanına kaydırılması genel olarak çokbiçimlilik (*polymorphism*) ya da çalışma zamanı çok biçimliliği (*runtime polymorphism*) olarak bilinir. Nesne yönelimli programlama tekniği gerçek gücünü çalışma zamanı çokbiçimliliğinden alır.

Bu teknikle genelleştirilmiş ve soyutlanmış bazı işlemler farklı sınıf nesneleri için farklı biçimlerde yapılır. Hangi işlevin çağrıldığının programın çalışma zamanında belirlenmesi program yazımında daha büyük esneklik sağladığı gibi çok daha iyi soyutlama olanağı verir.

Çokbiçimlilik farklı türden sınıf nesnelerinin gerçek türleri bilinmeden, bu nesnelerin ortak özelliklerine dayanılarak işlenmesidir.

İŞLEV ŞABLONLARI

C ve C++ dillerinde işlevler türlere göre yazılır. Oysa bir çok işlev belirli bir algoritmayı kodlamak için yazılır. İşlev belirli bir türe göre yazılmasına karşın algoritma türden bağımsızdır. Örneğin "iki değerden büyük olanını bulma" algoritması değerlerin hangi türden olacağına göre değişmez. İki nesnenin değerini takas etmek için yapılması gerekenler bu nesnelerin türlerine bağlı değildir. Ya da bir diziyi sıralamak için yapılması gerekenler dizinin türüne göre değişmez. Ancak bu işleri yapacak işlevler işlem yapacakları türlere göre yazılır.

Bir çok programda, aynı algoritma söz konusu olmasına karşın, farklı türler için işlevler programcı tarafından birden fazla kez yazılır. Programcının aynı algoritmayı farklı türler için yeniden kodlamasının bazı sakıncaları olabilir:

1. Olası yazım hataları sonucunda işlevlerden bazıları yanlış tanımlanmış olabilir.
2. Genel algoritmada bir değişiklik yapıldığı zaman türe bağlı her bir işlev için bu değişiklikler ayrı ayrı yapılmalıdır. Yani değişikliği tek bir yerde yaparak gerçekleştirmek mümkün değildir.

Bir dizinin en büyük elemanının değeri ile geri dönecek bir işlev tanımlamak isteyelim. İşlev *int* türden bir dizi için aşağıdaki biçimde tanımlanabilir:

```
int get_max(const int *ptr, size_t size)
{
    int max = *ptr;

    for (size_t k = 1; k < size; ++k)
        if (ptr[k] > max)
            max = ptr[k];

    return max;
}
```

İşlev *double* türden bir dizi için yazılmak istendiğinde

```
double get_max(const double *ptr, size_t size)
{
    double max = *ptr;
```

```
    for (size_t k = 1; k < size; ++k)
        if (ptr[k] > max)

            max = ptr[k];
    return max;

}
```

Bu iş nasıl daha kolay bir hale getirilebilir? İşlevlerin yazımı önişlemci programa yaptırılabilir.
Makrolar konusunu hatırlayalım:

```
#include <string>

#define  generate_get_max(T)      T get_max(const T *ptr, int size) { \

                          T max = *ptr; for (int k = 1; k < size; ++k)\
                         if (ptr[k] > max)  max = ptr[k]; return max;}

generate_get_max(int)
generate _get_max(double)

generate _get_max(std::string)
```

Yukarıdaki kod parçasında ismi *generate_get_max* olan bir makro tanımlanıyor. Argüman olarak farklı tür isimlerinin kullanılmasıyla açılan makrolarla, farklı türlere göre iş gören işlevler tanımlanmış olur.

Ancak bu amaç için makrolar tanımlanmasının bazı sakıncaları da vardır. Makrolar birden fazla satıra yayıldığında satır sonlarında '\' karakteri kullanmak gerekir. Ayrıca bazı durumlarda güvenlik nedeniyle fazladan ayraçlar kullanılır. Makro işlevler gerçek işlevlere göre farklı bir sözdizime sahiptir.

Makrolar önişlemci program tarafından açılır. Makro-İşlev kullanılması durumunda programı yazan, hangi makronun açılıp açılmaması gerektiğini düşünmek zorundadır. Aynı tür için bir makro-işlevin iki kez açılmamasını güvence altına almak programcının sorumluluğundadır.

Makro-işlevlerde hata oluşması durumunda kaynak kodda hatanın oluştuğu yerin derleyici tarafından belirlenmesi daha zor olur. Derleyiciler gerçek işlevler için bu konuda çok daha iyi destek verebilir.

Başka bir çözüm, her tür için doğru şekilde çalışacak, türden bağımsız bir işlevin yazımı olabilir. C dilinin standart işlevlerden *qsort* işlevinin herhangi türden bir diziyi sıralayabildiğini anımsayın. Türden bağımsız işlevler *void* türden gösterici parametreleri ile yazılabilir. Ancak bu yöntem, katı bir tür denetiminin hedeflendiği C++ dili için pek uygun değildir. Ayrıca türden bağımsız yazılan işlevler, türe bağlı olarak yazılan işlevlere göre daha karışık olma eğilimindedir. Türden bağımsız işlevlerin sınıf nesneleri için çağrılmasında bazı sorunlar oluşabilir.

Bir başka çözüm de bütün türlerin tek bir taban sınıftan türetilmesi olabilir. İstenen algoritma en tepedeki taban sınıfa göre yazılırsa, bu işleve argüman olarak bütün türler gönderebilir. Bu durumda yine katı tür denetimi yapılması engellenmiş olur. Ayrıca ilgili işlevden faydalanabilmek için mutlaka bir türetme işlemi yapmak gerekir.

Şablonlar C++‟ a görece olarak geç eklenmiş bir araçtır. Şablonlar diğer programlama dillerinde ya yoktur ya da kısıtlı ölçüde destek verecek şekilde vardır. C++ diline son dönemde yapılan eklemelerin büyük bir bölümü şablonlar konusunda yapılmıştır.

Programcı aynı algoritmaya göre farklı türler için iş gören çok sayıda işlev tanımlamak yerine yalnızca tanımlayacağı işlevlere ilişkin bir şablonu derleyiciye bildirir. Derleyici bu şablondan faydalanarak gerektiği zaman istenilen türden bir işlevin tanımını kendi yapar. Yani derleyici işlevin kodunu yazar.

İşlev şablonu tanımlanma sözdiziminin genel biçimi aşağıdaki gibidir:

```
template<class T>
<geri dönüş değeri türü> <işlev ismi>([parametre değişkenleri])
{
}
```

Örnek:

```
template <class T>
void func(T &a)
{
    //...
}
```

template bir anahtar sözcüktür. *template* anahtar sözcüğünden sonra açısal ayraç < > gelmelidir. Açısal ayraç içinde yer alan *class* anahtar sözcüğünün normal sınıf işlemleriyle bir ilgisi yoktur. *class* anahtar sözcüğü yerine *typename* anahtar sözcüğü de kullanılabilir. *class* ya da *typename* anahtar sözcüğünden sonra isimlendirme kurallarına uygun herhangi bir isim *(identifier)* seçilebilir. Seçilen bu isim, yani yukarıdaki örnekteki *T* ismi bir tür belirtir. Böyle işlevler birer şablondur. Yani kendi başlarına *çalışan* kodda yer kaplamaz. Bir işlev şablonuna ilişkin bir işlev çağrıldığında derleyici önce çağrı ifadesi olarak kullanılan ifadenin türünü saptar. Bu türe uygun olarak şablonda belirtilen işlevin kodunu programcı için yazar:

```
#include <iostream>

template <class T>
void print(T a)
{
    std::cout << a << std::endl;
}

int main()
{
```

```
    print(3.5);
    print(30L);
    print('a');

    print("Ahmet");


    return 0;
}
```

Yukarıdaki *print* işlev şablonunun tanımında *typename* anahtar sözcüğü de kullanılabilirdi:

```
template <typename T>
void print(T a)
{
    std::cout << a << std::endl;
}
```

main işlevi içinde *print* isimli işlev şablonunun dört ayrı türden argüman ile çağrıldığını görüyorsunuz. Derleyici bu durumda ilgili her işlev çağrısını gördüğünde şablondan ayrı ayrı işlevler üretir. Yukarıdaki kaynak kodun derlenmesi sırasında derleyici aşağıdaki işlevleri tanımlar. Bu duruma şablondan işlev üretilmesi (*"function generation ", "function instantiation")* denir.

```
print(3.5);
```

çağrı ifadesi için derleyicinin yazdığı işlev:

```
void print(double a)
{
    std::cout << a << std::endl;
}
```

```
print(30L);
```

çağrı ifadesi için derleyicinin yazdığı işlev:

```
void print(long a)
{
    std::cout << a << std::endl;
}

print('a');
```

çağrı ifadesi için derleyicinin yazdığı işlev:

```
void print(char a)
{
    std::cout << a << std::endl;
}

print("Ahmet");
```

çağrı ifadesi için derleyicinin yazdığı işlev:

```
void print(const char *a)
{
    std::cout << a << std::endl;
}
```

İşlev şablonlarında tür belirten isme "şablon parametresi" *(template parameter)* denir. İşlev şablonunda kullanılan parametre değişkenlerine "işlev çağrı parametreleri*" (call parameters)* denir.

Aynı şablon parametresinin açısal ayraç içinde birden fazla yer alması geçersizdir:

```
template<class Type, class Type>      //Geçersiz!
Type max(Type a, Type b)
{
    //...
}
```

Bir şablon parametresi, işlev şablonunun her yerinde tür belirten sözcük olarak kullanılabilir:

```
#include <iostream>

template <class T>

T max_of_three(T a, T b, T c)

{

    T max = a;
    if (b > max)

        max = b;
    if (c > max)

        max = c;
    return max;

}


int main()

{

    std::cout << max_of_three(12, 51, 67) << std::endl;
    std::cout << max_of_three(17.2, 15.3, 6.7) << std::endl;
    return 0;

}
```

Şablon parametresi olarak kullanılan isim işlev şablonu dışında görünür değildir, işlev şablonu dışında bilinmez. Aynı isim başka işlev şablonlarında, ya da normal işlevlerde bir çakışmaya neden olmadan kullanılabilir.

Birden Fazla Şablon Parametresi Olabilir

Bir işlev şablonu birden fazla şablon parametresine sahip olabilir.

Bu durumda işlev şablonuna ilişkin işleve yapılan çağrıda kullanılan argümanların türlerine göre, derleyici tarafından farklı parametrik yapıya sahip işlevler üretilebilir:

```
#include <iostream>


template<class A, class B>
void display(A a, B b)

{

    std::cout << a << "  " << b << std::endl;

}
```

```
#include <string>

using namespace std;


int main()

{

    string s("Necati");
    int x = 10;

    double d = 3.25;



    display(x, d);          // void display(int, double);
    display(d, s);          // void display(double, string);
    display('A', s);        // void display(char, string);



    return 0;

}
```

İşlev Şablonu Bildirimleri

Bir işlev şablonuna ilişkin bildirim de yapılabilir:

```
template <class T>
T min(T, T);
```

İşlev bildirimlerinde parametre değişkeni isimlerinin yazılmasının zorunlu olmadığını hatırlayalım. Aynı bildirimi aşağıdaki gibi de yapabilirdik:

```
template <class Type>

Type min(Type a,  Type b);
```

Normal işlevlerin bildiriminde olduğu gibi, işlev şablonu bildirimlerinde de parametre ayracı içinde kullanılan isimlerin bilinirlik alanı bu ayracın içiyle sınırlıdır. Yani bu isimlerin işlevlerin tanımlamasında kullanılan isimlerle aynı olması gerekmez. Yukarıda bildirimi yapılan işlev şablonunun tanımı aşağıdaki gibi yapılabilir:

```
template <class Type>

Type min(Type t1,  Type t2)

{
```

```
    return t1 > t2 ? t1 : t2;

}
```

Derleyicinin Şablon Parametresinin Türünü Çıkarması

Bir işlev şablonuna ilişkin işleve çağrı yapıldığında, derleyici işleve gönderilen argümanların türlerine bakarak, şablon parametresi olan tür yerine hangi gerçek türün koyulması gerektiğini anlar. Bu işleme ingilizcede *"template argument deduction"* denir. Bu aşamada derleyici tarafından aşağıdaki işlemler sırayla yapılır:

1. İşlev çağrısında kullanılan her bir argümana karşılık, bir çağrı parametresinin karşılık gelip gelmediği kontrol edilir.
2. Eğer uygun bir argüman bulunmuş ise, buna karşılık gelen şablon parametresinintür bilgisi belirlenmiş olur.
3. Eğer aynı şablon parametresi işlev şablonu tanımı içinde birden fazla kullanılmış ise, bu durumda, işlev çağrı ifadesindeki argümanların türü de benzer şekilde aynı olmalıdır.

Bu açıklamadan sonra aşağıdaki işlevleri inceleyelim :

```
template<class T>
void func(T a, T b)

{

    //…

}

int main()

{

    char ch = 'K';
    func(ch, 'A');

    //…
    return 0;

}
```

Bu örnekte derleyici

```
func(ch, 'A‟)
```

gibi bir çağrı için birinci aşamada işlev çağrı ifadesine bakarak *ch* ifadesinin türünün *char* türü olduğu anlar. İşlev şablonu tanımında şablon parametresi olarak kullanılan *T*'nin *char* türü olduğuna karar verir. *func* işlevine

yapılan çağrı ifadesindeki ikinci argüman olan *'A'* da *char* türden bir değişmez olduguna göre, ikinci şablon çağrı parametresinin de *char* türden olması gerekir.

Gerçekten ikinci çağrı parametresi de *T* türünden olduguna göre aşağıdaki işlevin üretilmesi doğal sonuçtur:

```
void func(char a, char b)
{
    //…
}
```

Ancak yukarıdaki *func* işlev şablonunun aşağıdaki biçimde çağrıldığını varsayalım:

```
void foo()
{
    double d= 3.15;
    float f = 10.7F;

    func(d, f);    //Geçersiz!
}


func(d, f)
```

çağrısı geçersizdir. Derleyici işlev çağrı ifadesinde yer alan birinci argümanın türüne bakarak *T* türünün *double* türü olduğu sonucunu çıkarır. Bu durumda ikinci şablon argümanı da *T* türünden olduğuna göre, işlev çağrı ifadesindeki ikinci argüman da *double* türünden olmalıdır. Oysa işleve gönderilen ikinci argüman *float* türünden olduğundan, işleve yapılan çağrı geçersizdir.

Başka bir örnek verelim:

```
template<class T>
T get_max(const T *ptr, size_t size)
{
    T max_elem = ptr[0];

    for(size_t k = 1; k < size; ++k)
        if (ptr[k] > max_elem)
```

```
            max_elem = ptr[k];


    return max_elem;
}

int main()
{
    int a[10] = {1, 2, 3, 7, 8, 4, 5, 6, 9, 1};
    double b[3] = {1.0, 2.0, 3.0};


    cout << "max int    = " << get_max(a, 10) << endl;
    cout << "max double = " << get_max(b, 10u) << endl;


    return 0;
}
```

Yukarıdaki işlev çağrıları geçerlidir, derleme zamanında hata oluşturmaz. Derleyici başarılı bir şekilde yani çift anlamlılık hatası olmadan işlev şablon parametresi olan türü, işlev çağrı ifadesine bakarak saptayabilir.

İşlev şablonlarına ilişkin işlevler çağrıldığında, işleve gönderilen arguman sayısı ile, işlev şablonunda belirtilmiş parametre sayısının da uyumlu olması gerekir. Uygun parametre yapısı bulunmaz ise derleme zamanında hata oluşur. Aşağıdaki kodu inceleyin:

```
template<class T>
T sum_square(T a, T b)
{
    return a * a + b * b;
}

#include <iostream>

using namespace std;
int main()
{
```

```
    cout << sum_square(5, 8) << endl;         //Geçerli
    cout << sum_square(2.7, 3.5) << endl;     //Geçerli
    cout << sum_square(10) << endl;           //Geçersiz
    cout << sum_square(4, 7.9) << endl;       //Geçersiz!


    return 0;

}
```

```
sum_square(10)
```

çağrısı geçersizdir. Çünkü işlev şablonunun iki çağrı parametresi varken işleve tek bir argüman gönderiliyor.

```
sum_square(4, 7.9)
```

çağrısı da geçersizdir. İşleve iki farklı türden argümanlar gönderiliyor. Oysa işlev çağrı parametrelerinin türü aynıdır.

Aşağıda örnek olarak aynı türden iki nesnenin değerini takas eden *swap* isimli bir işlev şablonu yazılıyor:

```
template<class T>
void swap(T &r, T &b)

{

    T temp = a;
    a = b;

    b = temp;

}
```

```
#include <iostream>


int main()

{

    int x = 10;
    int y = 20;
    swap(x, y);

    std::cout << "x = " << x << "\ny = " << y << std::endl;
    double d = 3.75;
```

```
    double e = 4.50;
    swap(d, e);

    std::cout << "d = " << d << "\ne = " << e << std::endl;


    return 0;

}
```

swap işlevi şablonunda referans yerine gösterici değişkenler de kullanılabilirdi:

```
template<class T>

void swap(T *p1, T *p2)

{

    T temp = *p1;

    *p1 = *p2;

    *p2 = temp;

}


#include <iostream>

using  std::cout;


int main()

{

    int x = 10;
    int y = 20;
    swap(&x, &y);

    cout << "x = " << x << "\ny = " << y << std::endl;
    double d = 3.75,  e = 4.50;

    swap(&d, &e);

    cout << "d = " << d << "\ne = " << e << std::endl;


    return 0;

}
```

İşlev şablonundan üretilecek işlevlerin parametre değişkeni ya da değişkenlerinin tür ya da türleri derleyici tarafından akıllıca saptanır. Örneğin:

```
template <typename T>
void func(T p)

{

    T *ptr = &p;

    //…

}


int main()

{

    int a[10];

    func(a);


    return 0;

}
```

main işlevi içinde *func* işlev şablonuna ilişkin işlev, *int* türden *a* dizisinin başlangıç adresi ile çağrılıyor. İşlev şablonuna gönderilen argümanın türü *(int *)* olduğuna göre, derleyici *T* türünü *(int *)* türü olarak belirleyerek işlevin kodunu yazar. Bu durumda işlev şablonunun içinde tanımlanan *ptr* isimli nesnenin türü de (*int **) olur. Şimdi de aşağıdaki işlev şablonunu inceleyelim:

```
template <typename T>

void func(T f)

{

    f();

}


#include <iostream>


void display_message()

{

    Std::cout << "merhaba!" << std::endl;

}


int main()

{
```

```
    func(display_message);


    return 0;

}
```

main işlevi içinde *func* işlevi *display_message* işlevinin ismi ile çağrılıyor. *func* işlevi bir şablona ilişkindir. Bir işlev isminin, işleme sokulduğunda işlevin adresine dönüştürüldüğünü biliyorsunuz. Bu durumda derleyici şablon parametresi olan *T* türünü

```
void (*)(void)
```

türü olarak belirler. İşlevi bu türe göre yazar. Derleyici tarafından aşağıdaki gibi bir işlev yazılır:

```
void func(void (*f)())
{
    f();
}
```

Derleyicinin yazdığı *func* işlevinin çağrılması ile, işlevin ana bloğu içinde *f* değişkeninin gösterdiği işlev, yani *display_message* işlevi çağrılır.

Şablon Argümanları Bir Sınıf Türüne Bağlanabilir

İşlev şablonuna ilişkin bir işlev yapılan çağrı ifadesinde kullanılan argüman ya da argümanlar bir sınıf türünden de olabilir.

```
class Complex{
    double m_r, m_i;
public:
    Complex(double r = 0, double i = 0.): m_r(r), m_i(i){}
    friend Complex operator+(const Complex &r1, const Complex &r2);
    friend Complex operator*(const Complex &r1, const Complex &r2);
    //…
};
```

```
template<class T>
T sum_square(T a, T b)
{
    return a * a + b * b;
}



int main()
{
    Complex c1(4., 7);
    Complex c2(3, 8);
    Complex c3 = sum_square(c1, c2);

    return 0;
}
```

Yukarıdaki kod parçasında eğer *Complex* sınıfı için *operator+* ve *operator** işlevleri bildirilmemiş olsa derleme zamanında hata oluşurdu. Derleyici yazılmış olan şablonda *T* template parametresi yerine *Complex* türünü yerleştirdiğinde,

```
a * a + b * b
```

işlemini yapabilmek için ilgili işleç işlevlerini arar ama bulamazdı.

Tür ve Referans Çağrı Parametreleri

İşlev şablonu çağrı parametresinin bir gösterici ya da referans olması, bazı dönüşümlerin yapılıp yapılmayacağını belirler. Aşağıdaki işlev şablonları bildirilmiş olsun:

```
template<typename T>
void f(T);


template<typename T>
void g(T&);
```

```
double a[20];
const int ci = 10;
```

Şimdi bu şablonlara ilişkin yapılacak bazı çağrıları ele alalım:

```
f(a);
```

İşlev şablonu çağrı parametresi referans olmadığı için, bu durumda dizi ismi otomatik olarak dizinin ilk elemanın adresine dönüştürülür. Yukarıdaki işlev çağrısı sonucunda derleyici *f* işlev şablonu için *T* türünü *double** türü olarak belirler.

```
g(a);
```

işlev şablonu çağrı parametresi referans olduğundan bu durumda *T* türü, *20* elemanlı

double türden bir dizi türüdür.

```
f(ci);
```

işlev şablonu çağrı parametresi referans olmadığından *T* türü *int* türü olarak belirlenir.

```
g(ci);
```

işlev şablonu çağrı parametresi referans olduğundan *T* türü *const int* türü olarak belirlenir.

```
f(5);
```

Bu çağrı sonucunda *T* türü *int* türü olarak belirlenir. Çağrı geçerlidir.

```
g(7);
```

T türü *int* türü olarak belirlenir. Çağrı derleme zamanında hata oluşturur. Ancak bir *const*

referansa, bir değişmez ile ilkdeğer verilebileceğini anımsayın. Aşağıdaki
örneği derleyerek çalıştırın:

```
#include <iostream>


template<class T>
void display(T &a)

{

    for (int k = 0; k < sizeof(a) / sizeof(a[0]); ++k)
        std::cout << a[k] << " ";

    std::cout << std::endl;

}


int main()

{

    int a1[5] = {1, 2, 3, 4, 5};

    display(a1);


    int a2[3] = {10, 20, 30};

    display(a2);


    int a3[10] = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};

    display(a3);


    return 0;

}
```

Nasıl oluyor da bir dizinin elemanlarını yazan işlev dizinin eleman sayısını almadan bu işi başarıyor? *display* işlevinin çağrı parametresinin türü *T&* olarak seçildiğinden,

```
display(a1)
```

çağrısıyla, derleyici işlev şablonu çağrı parametresinin *5* elemanlı *int* türden bir dizinin yerine geçen referans olduğu bilgisini çıkartır. Derleyici aslında

```
display(a1)
display(a2)
display(a3)
```

çağrıları için üç ayrı işlev yazmıştır.

Dizgelerin Argüman Olarak Kullanılması

Bir dizge bir işlev şablonuna ilişkin işleve argüman olarak gönderilirse, şablon parametresinin türü ne olarak belirlenir? Çağrı parametresi bir referans değilse şablon parametresi *const char ** türü olarak belirlenir. Ancak işlev çağrı parametresi bir referans ise argüman türü belirli bir uzunlukta *char* türden dizi türü olarak ele alınır. Dizinin uzunluğu dizgeyle belirlenen yazının uzunluğundan bir fazla olarak alınır:

Aşağıdaki örneği inceleyin:

```
template <class T>
void func(T a, T b);


template <class T>
void foo(T &a, T &b);


int main()
{
    func("veli", "murat");     //Geçerli
    foo("ali", "can");         //Geçerli
    foo("veli", "murat");      //Geçersiz


    return 0;
}


func("veli", "murat");
```

çağrısında *func* işlev şablon parametresi *T* türünü *char ** türü olarak alır.

```
foo("ali", "can");  //Geçerli
```

Çağrısında ise şablon parametresi *char [4]* türüdür. Her iki argümanın türü de aynı olduğu için çağrı derleyici tarafından geçerli olarak değerlendirilir.

```
foo("veli", "murat");
```

Derleyicinin yaptığı çıkarımda ise her iki argümanın türü aynı değildir. İşleve gönderilen birinci argümana göre *T* türü *char[5]* türü olarak belirlenirken, ikinci argümana ilişkin tür *char[6]* türüdür. Oysa işlev şablonu çağrı parametreleri aynı türdendir. İşlev çağrısı geçersizdir.

Bir İşlev Şablonu İçinde Başka Bir İşlev Şablonu çağrılabilir

Şüphesiz bir işlev şablonu içinde başka bir işlev şablonuna ilişkin işlev çağrılabilir. Bir diziyi küçükten büyüğe doğru sıralayan bir işlev şablonu tanımlayalım:

```
#include <iostream>

template<class T>

void mswap(T &a, T &b)

{

    T temp = a;
    a = b;

    b = temp;

}


template <class T>

void sort(T *p, size_t size)

{

    for (size_t i = 0; i < size - 1; ++i)

        for (size_t j = 0; j < size - 1 - i; ++j)
            if (p[j] > p[j + 1])

                mswap(p[j], p[j + 1]);

}


int main()

{

    int a[10] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

    double b[5] = {5.4, 2.3, 3.5, 8.7, 1.9};
```

```
    sort(a, 10);

    sort(b, 5);


    return 0;
}
```

Yukarıdaki işlev şablonlarını inceleyelim. *mswap* isimli şablon herhangi türden iki nesnenin değerlerinin takas edilmesini sağlayan işlevler üretmek amacıyla tanımlanıyor. *sort* işlev şablonu ise dizileri *bubble sort* algoritmasıyla sıraya sokmak için tanımlanıyor. *sort* işlev şablonu içinde *mswap* işlev şablonuna çağrı yapıldığını görüyorsunuz.

Bir İşlev Şablonuyla Aynı İsimli Normal Bir İşlev Bir Arada Bulunabilir

Bir işlev şablonuyla aynı isimli normal bir işlev bir arada bulunabilir.

Bu durumda çift anlamlılık hatası (*ambiguity*) oluşmaz. Normal işlevin işlev şablonuna karşı bir önceliği vardır:

```
#include <iostream>

template<class T>

T sum_square(T a, T b)

{

    std::cout<< "template function " << std::endl;
    return a * a + b * b;

}


int sum_square(int a, int b)

{

    std::cout << "non template function" << std::endl;
    return a * a + b * b;

}


int main()

{

    int x = sum_square(4, 7);


    return 0;
```

```
}
```

Normal bir işlevle bir işlev şablonunun aynı isimde olabilmesi, işlev şablonlarının özelleştirilmesine yardımcı olur. Örneğin:

```
template <class T>
void foo(T a)

{

    //...

}
```

Yukarıda verilen *foo* işlev şablonunun yazılmış olan şablon kodunun *Date* isimli bir sınıf için uygun düşmediğini düşünelim. Bu durumda aynı isimli *Date* parametreli normal bir işlev yazılabilir:

```
void foo(Date date)

{

    //...

}
```

Bu durumda *foo* işlevi *Date* türünden bir argüman ile çağrıldığında işlev şablonu tarafından işlev üretilmez. Özel olarak yazılan *foo* işlevi çağrılmış olur.

Belirlenmiş İşlev Şablonu Argümanları

Derleyici bir işlev şablonuna ilişkin işlev çağrısıyla karşılaştığında, işlev şablonu parametresinin hangi tür olması gerektiğini çıkarmaya çalışır (*template argument deduction*). Eğer derleyici bu çabasında başarılı olamaz ise derleme zamanında hata oluşur. Aşağıdaki örneği inceleyelim:

```
template<class T>
void func(T a, T b);


int main()

{

    unsigned int uix = 10u;
    int y = -3;

    func(uix, y);  //Geçersiz
```

```
    //..
    return 0;

}
```

Yukarıdaki örnekte

```
func(uix, y);
```

çağrısı geçersizdir. Çünkü derleyici *func* işlevine yapılan çağrı ifadesindeki birinci argümana bakarak *T* türünün *unsigned int* olduğu sonucuna varır. Daha sonra kinci argümanın *unsigned int* türden olmadığını görünce bu durumda hata iletisi verir.

İstenirse şablon parametresinin türünün ne olduğu derleyicinin çıkarımına bırakılmayabilir. Özel bir sözdizim kuralı ile işlev çağrısıyla derleyiciye doğrudan bildirilebilir. Bu duruma "belirlenmiş işlev şablonu argümanları" (*explicit template arguments*) denir.

Bu amaçla işlev çağrısında, işlev ismini izleyen bir açısal ayraç içinde, şablon parametresine ilişkin tür bilgisi derleyiciye açıkça bildirilir. Açısal ayracı işlev çağrı işleci izler:

```
func<unsigned int>(uix, y);
```

Yukarıdaki çağrıyla, *func* işlev şablon parametresinin *(T), unsigned int* türü olarak kabul edilmesi gerektiği derleyiciye açıkça bildiriliyor. Bu durumda derleyici bir çıkarım yapmaya çalışmadan, şablondan işlev üretirken *T* türü yerine *unsigned int* türünü kullanır. Yani derleyici aşağıdaki gibi bir işlev yazar:

```
void func(unsigned int a, unsigned int b)

{

    //..

}
```

Belirlenmiş şablon argümanı kullanımının bazı faydaları vardır:

1. Bazı durumlarda işlev şablonu kullanımı gereksiz yere, fazla sayıdaişlevin tanımlanmasına yol açar. Örneğin:

```
template<class T>
void func(T a);
```

bildirilen *func* işlev şablonu aşağıdaki deyimlerle çağrılırsa derleyici 6 ayrı farklı işlev tanımlar:

```
func('A');
func(10);
func(10U);
func(25L);
func(12.f);

func(3.8);
```

Oysa programcı yukarıdaki tüm çağrıların parametresi *double* türden olan bir işleve yapılmış olmasını isteyebilir. Ne de olsa diğer doğal türlerden argümanlar derleyici tarafından *double* türüne dönüştürülebilir. Böylece programcı gereksiz işlev tanımlarından kaçınmış olur:

```
func<double>('A');
func<double>(10);
func<double>(10U);
func<double>(25L);
func<double>(12.f);
func(3.8);
```

2. Bazı durumlarda da, işlev şablonu parametresinin derleyici tarafından çıkarımı zaten olanaksızdır:

```
template <class T>
T func()

{

    //...

}
```

Yukarıdaki işlev şablonunda şablon parametresi olan *T,* işlevin geri dönüş değerinin türü olarak kullanılıyor. Peki derleyicinin bu şablona ilişkin bir işlev çağrısını gördüğünde *T* türünün ne olduğunu çıkarma şansı var mıdır? Bir de aşağıdaki şablonu inceleyelim:

```
template <class T>
void foo()

{

    T a;

    //...
```

```
}
```

Yukarıdaki işlev şablonunda ise *T* türü, işlev ana bloğu içinde tanımlanan *a* isimli nesnenin türü olarak kullanılıyor. Bu durumda da, işlev çağrısına bakarak derleyicinin *T* türünün ne olduğunu anlama şansı olmaz. Ancak her iki örnekte de işlev çağrısında, *T* türünün ne olduğu açıkça belirtilirse, derleyicinin çıkarım yapmadan *T* türünün hangi tür olduğunu anlaması sağlanabilir.

Öyle bir işlev şablonu yazılmak istensin ki, bu şablon iki şablon parametresine sahip olsun. İşlev şablonu bu parametrelere gönderilecek değerlerin toplamı değeriyle geri dönsün. Ancak geri dönüş değeri de, toplam değerini içinde tutabilecek genişlikte bir türden olsun:

```
template<class T, class U>

???? sum (T, U);
```

işlevin geri dönüş değeri türü yerine ne yazılmalıdır? *T* mi *U* mu? Bu örneğe bakılırsa her iki türün de geri dönüş değeri türü olması mümkün değildir:

```
char ch;
unsigned uix;

//...
sum(ch, ui)
sum(uix, ch)
```

Eğer işlev şablonunun tanımı

```
template<class T, class U>
T sum (T, U);
```

biçiminde ise

```
sum(ch, ui)
```

işlevinin çağrılması durumunda derleyicinin yazacağı işlev

```
char sum(char, unsigned int);
```

biçiminde olur ki bu durumda bilgi kaybı kaçınılmazdır. Eğer şablon işlevin tanımı

```
template<class T, class U>
U sum (T, U);
```

biçiminde yapılmışsa

```
sum(uix, ch)
```

işlevinin çağrılması durumunda derleyicinin yazacağı işlev

```
char sum(unsigned int, char);
```

şeklinde olur ki, yine bu durumda bilgi kaybı kaçınılmaz olur.

Sorunu çözmek için işlev şablonu tanımına üçüncü bir şablon parametresi eklenebilir:

```
template<class T1, class T2, class T3>
T1 sum(T2 a, T3 b);
```

```
unsigned int ux = sum('a' + 768U);
```

İşlev çağrısı böyle yapılırsa, *T1* türünün ne olduğu derleyici tarafından işlev çağrı ifadesinden çıkarılamaz. Oysa söz konusu işlev belirlenmiş argümanla çağrılırsa bu durumda hata oluşmaz:

```
unsigned int ux = sum<unsigned int>('a' + 768U);
```

Yukarıdaki çağrı ile derleyiciye *T1* türünün hangi tür olarak ele alınması gerektiği açık bir şekilde bildiriliyor. Derleyici bu çağrıya yönelik olarak aşağıdaki parametrik yapıya sahip bir işlev yazar:

```
unsigned int sum(char a, unsigned int b);
```

Birden fazla şablon parametresinin bulunması durumunda, şablon parametrelerinin hepsinin belirlenmesi zorunlu değildir. Soldan başlayarak istenilen sayıda şablon parametresi, işlev çağrısında açıkça belirlenebilir. Açık olarak belirlenmemiş şablon parametrelerinin hangi türden alınacakları derleyicinin çıkarımına bırakılmış olur.

Aşağıdaki örneği inceleyin:

```
template <class T, class M, class U>
T func(M a, U b)

{

    T temp;

    //...

    return temp;

}


int main()

{

    func<int>('A', 12);

    func<int, double>('A', 12);
    func<int, double, double>('A', 12);

    //...
    return 0;

}
```

main işlevi içinde yapılan birinci çağrıda *func* işlev şablonunun birinci parametresi olan *T* türü açık bir şekilde belirtiliyor. Bu durumda *T* türü *int* türü olarak seçilir. *M* türü ve *U* türü derleyicinin çıkarımına bırakılıyor. Derleyici işlev çağrısına bakarak *M* türünü *char* türü, *U* türünü ise *int* türü olarak belirler, işlevi bu türlere göre yazar.

main işlevi içinde yapılan ikinci çağrıda *func* işlev şablonunun birinci ve ikinci argümanları olan *T* ve *M* türü açık bir şekilde belirtiliyor. Bu durumda *T* türü *int M* türü *double* türü olarak seçilir. *U* türü ise derleyicinin çıkarımına bırakılıyor. Derleyici işlev çağrısına bakarak *U* türünü *int* türü olarak belirler, işlevi bu türlere göre yazar.

Üçüncü çağrıda ise tüm şablon argümanları açıkça belirleniyor. Hiçbir argüman derleyicinin çıkarımına bırakılmıyor. Derleyici *T* türünü *int* türü olarak, *M* ve *U* türlerini *double* türü olarak belirler. İşlevi bu türlere göre yazar.

İşlev şablonları ve İşlev Yüklemesi

Şablon işlevlerin kullanımı işlev yüklemesi ile ilişkili olsa da, şablon işlevler ve işlev yüklemesi ayrı araçlardır:

1. Aynı isme sahip ancak parametre değişkeni sayısı farklı işlevler tanımlanarak bu işlevler çağrılabilir. Ancak işlev şablonu söz konusu olduğunda verilen şablona ilişkin derleyicinin yazacağı tüm işlevlerin parametre değişkeni sayısı aynıdır.

2. *İşlev yüklemesi* mekanizmasına konu olan, aynı isimli işlevlerin kaynak kodları farklı olabilir. Yani bu işlevler isimleri aynı olmasına karşın farklı işler yapabilir. Ama bir şablondan üretilen işlevlerin kaynak kodları aynı, yalnızca parametre değişkenleri türleri farklıdır.

İşlev şablonları da Yüklenebilir

Parametrik yapıları farklı, isimleri aynı olan işlev şablonları tanımlanabilir.

Nasıl aynı isimli işlevler bulunabiliyorsa, aynı isimli işlev şablonları da bir arada bulunabilir. Tabii bu işlev şablonlarının, parametrik yapıları, şablon parametre ya da çağrı parametrelerine bağlı olarak verilen genel imzaları birbirinden farklı olmak zorundadır. Aşağıdaki kod parçasını derleyerek çalıştırın:

```
inline const int &getmin(const int &a, const int &b)
{
    return a < b ? a : b;
}

template <typename T>
inline T const& getmin(const T& a, const T& b)
{
    return a < b ? a : b;
}

template <typename T>
inline const T &getmin(const T &a, const T &b, const T &c)
{
    return getmin(getmin(a,b), c);
}
```

```
#include <iostream>

using namespace std;


int main()

{

    cout << getmin(10, 20, 30) << endl;
    cout << getmin(7.0, 42.0) << endl;
    cout << getmin('a', 'b') << endl;
    cout << getmin(7, 42) << endl;

    cout << getmin<>(7, 42) << endl;

    cout << getmin<double>(7, 42) << endl;
    cout << getmin('a', 42.7) << endl;



    return 0;

}
```

main işlevi içinde yapılan tüm çağrıları tek tek ele alalım:

```
getmin(10, 20, 30)
```

çağrısı için

```
template <typename T>

inline const T &getmin(const T &a, const T &b, const T &c)
```

şablonundan *T* türünün *int* türü olması çıkarımıyla bir işlev üretilir.

```
getmin(7.0, 42.0);
```

çağrısı için

```
template <typename T>

inline T const& getmin(T const& a, T const& b)
```

şablonundan *T* türünün *double* türü olması çıkarımıyla bir işlev üretilir.

```
getmin('a', 'b')
```

çağrısı için

```
template <typename T>
inline T const& getmin(const T& a, const T& b)
```

şablonundan *T* türünün *double* türü olması çıkarımıyla bir işlev üretilir.

```
getmin(7, 42)
```

çağrısı bir işlev şablonuna ilişkin değildir. Çünkü

```
inline const int &getmin(const int &a, const int &b);
```

biçiminde bir işlev bilidirmi yapılmıştır. Gerçek işlevin işlev şablonuna önceliği vardır.

```
getmin<>(7, 42)
```

çağrısı gerçek işleve değil işlev şablonuna ilişkindir. Çünkü işlev çağrısında işlev ismini içi boş bir açısal ayraç izliyor.

```
getmin<double>(7, 42)
```

çağrısı ise iki çağrı parametreli işlev şablonuna ilişkindir. Ancak şablon parametresinin türü derleyicinin çıkarımına bırakılmadan *double* türü olarak bildiriliyor.

Son olarak

```
getmin('a', 42.7)
```

çağrısı yine gerçek işleve ilişkindir. İşlev çağrısında kullanılan argümanların türleri farklıdır. Ancak iki çağrı parametreli işlev şablonunun çağrı parametreleri aynı şablon parametresi türündendir. Bu durumda işlev şablonundan işlev üretilmesi mümkün değildir.

İşlev Şablonunun Belirlenmiş Bir Tür İçinÖzelleştirilmesi

Yazılan bir işlev şablonu tüm veri türleri için uygun düşmeyebilir. Belirli bir tür için, üretilecek işlevin, şablonda verilen algoritmaya göre değil de verilecek başka bir tanıma göre yazılması istenebilir. Buna işlev şablonunun bilinçli özelleştirilmesi *(explicit specialization)* denir. Aşağıdaki örneği inceleyelim:

```
template<class Type>
Type max(Type a, Type b)

{

    return a > b ? a: b;

}


#include <iostream>



int main()

{

    const char *p1 = "Mehmet";
    const char *p2 = "Ahmet";

    std::cout << max(p1, p2) << std::endl;



    return 0;

}
```

max işlev şablonu kendisine gönderilen iki nesneden büyük olanın değeri ile geri dönen işlevlerin üretiminde kullanılabilir.

Büyüklük ilişkisi *char*, *int*, *double* gibi doğal türler için açıktır. Yukarıdaki işlev şablonun kullanılmasında bir hata söz konusu olmaz. Ancak iki dizgenin daha büyüğünün bulunması için bu şablonun kullanılması doğru olmaz. Yukarıdaki genel algoritma karşılaştırmayı “>” işleci ile yapmaktadır. İki dizge “>” işleci ile karşılaştırıldığına karşılaştırılan yalnizca dizgelerin yerleştirildiği bellek bloklarının adresidir. Şüphesiz istenilen dizgelerin sözlükteki konumlarına ilişkin (*lexicographical)* bir karşılaştırmadır. C++ dili böyle durumlarda işlev şablonu için türe bağlı özel işlev tanımları yapılabilmesine izin verir. Bu mekanizmaya ″İşlev şablonunun belirlenmiş bir tür için özelleştirilmesi″ (*Template Explicit Specialization*) denir. Aşağıdaki kodu inceleyin:

```
template<class Type>
Type tmax(Type a, Type b)
{
    return a > b ? a: b;
}



template<>
const char *max<const char *>(const char *p1, const char *p2)
{
    return (strcmp(p1, p2) > 0 ? p1 : p2);
}

#include <iostream>


int main()
{
    int i = max(50, 10);
    std::cout << "i = " << i << std::endl;
    const char *p1 = "Necati";
    const char *p2 = "Ergin";
    std::cout << "max = " << max(p1, p2) << std::endl;


    return 0;
}
```

Sözdizimi inceleyelim. *template* anahtar sözcüğünü bu kez içi boş bir açısal ayraç izliyor. Bu açısal ayraçtan sonra işlevin geri dönüş değerinin türü yazılıyor. Ancak bu kez işlev bilinen bir tür bilgisi ile, yani şablon işlev argümanına bağlı kalmadan yazılıyor. Daha sonra işlev ismi geliyor. İşlev isminden sonra gelen açısal ayraç içine bu kez yine tür bilgisi yazılıyor.

Yukarıdaki sözdizime göre belirli bir türe yönelik özelleştirme yapılabilmesi için, ana işlev şablonu tanımının derleyici tarafından görülebilir olması gerekir. Yani ancak daha önce tanımlanmış olan bir işlev şablonu özelleştirilebilir.

İşlev şablonlarında Sözdizim Kontrolleri

Derleyicilerin çoğu, işlev şablonlarında sözdizim kontrolünü iki ayrı aşamada gerçekleştirir. Birinci kontrol henüz işlev şablonu çağrılmadan yapılır. Bu aşamada bazı temel kontroller yapılır. Hiçbir biçimde ve olasılıkta geçerli olamayacak durumlar daha bu aşamada derleme zamanı hatası olarak belirlenir. Gerçek sözdizim kontrolü işlev şablonuna ilişkin bir çağrıyla karşılaşıldığında, yani derleyicinin gerçek işlevi yazması sırasında yapılır. Aşağıdaki örneği ele alalım:

```
template<typename T>
void func(T p)

{

    p.foo();

}


int main()

{

    func(10);   //Geçersiz


    return 0;

}
```

Yukarıdaki kod parçasında derleyici işlev şablonunun kodunun üzerinden geçtiğinde herhangi bir hata vermez. Çünkü uygun bir işlev çağrısıyla *T* türü bir sınıf türü olarak belirlenebilir. O sınıfın da *func* isimli bir *public* üye işlevi olabilir. Ancak işlev uygun olmayan bir türden argümanla çağrıldığında, derleyici işlev çağrısını bir hata olarak belirler.

Derleyicinin sözdizim kontrollerini hangi aşamalarda yapacağı standart olarak belirlenmemiştir. Bazı derleyiciler işlev şablonuna ilişkin bir çağrı ile karşılaşmadıkça neredeyse hiç bir sözdizim geçerlilik kontrolü yapmaz.

İşlev şablonlarının Yeri

İşlev şablonları kod dosyasına, yani *.cpp* dosyasına, yerleştirilemez. Derleyicinin her kaynak dosyayı derlemesi sırasında , işlev şablonu tanımlamalarını yeniden görmesi gerekir. Projenin birden fazla modülden oluşması durumunda, bağımsız derlenen her modülde, eğer bu işlev şablonu çağrılmış ise tanımlamasının derleyici tarafından her modülde görülmesi gerekir. Farklı modüllerde aynı işlev şablonu kullanılıyorsa bu işlev şablonu tanımlamalarının tamamen özdeş olması gerekir.

İşlev şablonlarının başlık dosyalarında tanımlanması, şablonun kullanıldığı her modüle bu başlık dosyasının eklenmesi en iyi çözümdür.

İşlev şablonu tanımlamaları, *static* anahtar sözcüğü ile başlatılmamışsa dış bağlantıya sahiptir. Bu durumda örneğin, aynı şablon işlev aynı açılımla hem *a.cpp* hem de *b.cpp* dosyasından kullanılmış olsun. Derleyici aynı

açılımı hem *a.obj* hem de *b.obj* içine yazar. Ancak bağlayıcı bunlardan yalnızca bir tanesini çalışan koda yerleştirir. Yani farklı modüllerden aynı işlev şablonu kullanılmış olsa bile, çalışan dosyada bir tane bulunur.

İşlev şablonlarında Varsayılan Argümanlar

Normal işlevlerde olduğu gibi, işlev şablonları da varsayılan argümanlar alabilir. Varsayılan argüman olarak kullanılan ifade şablon parametresine bağlı olabileceği gibi şablon parametrelerinden bağımsız da olabilir. Aşağıdaki şablonu inceleyelim:

```
template <typename T>
void functemp (T &r, T val = 0)
{
    r = val;
}
```

Yukarıda tanımlanan *functemp* isimli şablondan üretilecek bir işleve tek bir argüman geçildiğinde, işlevin ikinci parametresi olan *val* değişkenine otomatik olarak *0* değeri kopyalanır. Şimdi de aşağıdaki şablonu inceleyin:

```
template<typename T>
void ftemp (T* ptr, const T & val = T())
{
    *ptr = &val;
    //…
}
```

Tanımlanan *ftemp* isimli şablondan üretilecek bir işleve, tek bir argüman geçildiğinde işlevin ikinci parametresi olan *val* referansı varsayılan kurucu işlevle yaratılacak bir geçici nesneye bağlanır. Eğer *T* türü doğal veri türlerinden biri olarak belirlenirse bu referans değeri *0* olan bir geçici nesneye bağlanır. Daha önce değindiğimiz "sözde başlangıç işlevi*" (pseudo constructor)* kuralını anımsayın.

Varsayılan argüman olarak kullanılan ifadeye dayanılarak şablon parametresinin hangi türe ilişkin olduğu bilgisi çıkarılamaz. Aşağıdaki örneği inceleyin:

```
template <typename T>
void func (T x = 42);

int main()
```

```
{
    func<int>();     // Geçerli. T int türü olarak alınır.
    func();         // Geçersiz. T türünün ne olduğu bilgisi çıkarılamaz.

    return 0;
}
```

typename Anahtar Sözcüğü

Bir işlev şablonunun parametresinin bildiriminde *class* veya *typename* anahtar sözcüklerinin kullanılabileceğini biliyorsunuz. Ancak *typename* anahtar sözcüğünün ikinci bir kullanımı vardır. Burada *class* anahtar sözcüğü *typename* anahtar sözcüğünün yerine kullanılamaz. Bir işlev şablonu içinde şablon parametresine bağlı olarak bir tür bilgisinin kullanıldığını düşünelim:

```
template <class T>
void func(T val)
{
    T::Dollar sum;     //Geçersiz
    //…
}
```

Yukarıdaki işlev şablonu ana bloğu içinde *T::Dollar* bir tür bilgisi olarak kullanılıyor. Burada *typename* anahtar sözcüğünün kullanılması zorunludur.

```
typename T::Dollar sum;
```

deyimiyle tanımlanan *sum T* türü hangi sınıfa ilişkin ise, o sınıf türünün içinde bildirilmiş olan *Dollar* türünden bir nesnedir.

şablonların Tür Belirtilen parametreleri ve Tür Belirtmeyen Parametreleri

Şablonların, şablon parametreleri tür belirten (*type parameter*) ya da tür belirtmeyen (*non type*) şeklinde olabilir. Parametre yazılırken *class* ya da *typename* anahtar sözcüğü kullanılırsa parametrenin aslında herhangi bir tür bilgisi ifade eden bir anlama geldiği varsayılır. Fakat *class* ya da *typename* yerine *int*, *long* gibi doğal türlere ilişkin bir tür kullanılırsa bu durumda şablon parametresi tür belirtmez. O türden bir "değişmez ifadesi" belirtir. Bu durumda şablon işleve çağrı yapılırken, bu parametreler için bir değişmez ifadesi kullanılması gerekir. Örneğin:

```
#include <iostream>
#include <algorithm>
#include <cstdlib>
#include <ctime>



template <class T, size_t size, class R>
void display(R fptr)

{

    T a[size];



    for (int k = 0; k < size; ++k)
        a[k] = fptr();

    sort(a, a + size);

    for (int k = 0; k < size; ++k)
        std::cout << a[k] << " ";

    std::cout << "\n*******************************" << std::endl;

}


double drand()

{

    return static_cast<double>(rand()) / RAND_MAX;

}


int main()

{

    srand(time(0));
    display<int, 100>(rand);
    display<double, 10>(drand);
```

```
    return 0;
}
```

Yukarıda tanımlanan display isimli işlevin üç şablon parametresi var. Bunlardan ikincisi türe bağlı olmayan bir parametre. Bu parametreye karşılık gelen değer işlev çağrısında belirlenmelidir.

İşlevin amacı belirli türden *N* tane rastgele değeri küçükten büyüğe doğru ekrana yazdırmak. Şablonun türe bağlı birinci parametresi yazdırılacak rastgele değerlerin türünü belirliyor. Şablonun ikinci yani tür dışı parametresi rastgele değerlerin sayısını belirliyor. Şablonun türe bağlı olan üçüncü parametresi rastgele sayı üretiminde kullanılacak işlevin türünü belirliyor. Aşağıdaki çağrıyı inceleyelim:

```
display<double, 10>(drand);
```

Derleyicinin bu şablondan üreteceği işlevin çağrılmasıyla *drand* işlevi kullanılarak *10* tane *double* türden değer elde edilir. Bu değerler sıralı bir biçimde ekrana yazdırılır. İşlev şablonu içinde tür dışı parametre olan *size* işlev tanımı içindeki dizinin boyutu olarak kullanılıyor. Çağrılan *sort* işlevi ise *STL*'in standart bir işlev şablonudur.

Standart C++ Şablon Kütüphanesi Nedir

STL (Standard Template Library) ilk kez 90‟ lı yılların başlarında kullanılmaya başlamıştır. Daha sonra alınan bir kararla *STL*'in C++'ın standart kütüphanesi yapılmasına karar verilmiştir. 1998 standartlarıyla çeşitli eklentilerle *STL,* C++‟ın standart kütüphanesi yapılmıştır.

STL işlev ve sınıf şablonlarından oluşur. Bu işlev ve sınıf şablonlarının tanımları çeşitli başlık dosyalarının içine yerleştirilmiştir. Örneğin, *<algorithm>* isimli başlık dosyasında çok sayıda standart işlev şablonu vardır. *STL* deki tüm işlev ve sınıf şablonları *std* isim alanı içinde tanımlanmıştır. Bu nedenle programcının bunları kullanırken, *std* isim alanını dikkate alması gerekir. Kütüphanedeki global işlev şablonları *algoritma* olarak isimlendirilir. Sınıf şablonları ise farklı veri yapılarının kodlanması amacıyla tanımlamıştır. Algoritmalar *iterator* kavramı kullanılarak sınıf şablonlarıyla bütünleştirilmiştir. “İteratörler”, nesne tutan sınıflar üzerinde bazı genelleştirilmiş işlemlerin yapılmasını sağlayan göstericiler ya da akıllı göstericilerdir (*smart pointers*).

Örnek STL İşlev şablonları

STL'de çok sayıda işlev şablonu bulunur. Bu işlev şablonları, yaptıkları işlere göre çeşitli gruplara ayrılmıştır. İyi bir *STL* kullanımı için özel bir eğitim gerekir.

Bu işlev şablonlarının kullanılabilmesi için yapılması gereken o şablonun bulunduğu başlık dosyasını kaynak koda eklemektir. *STL* işlevlerinin büyük çoğunluğu istenilen bir veri yapısı üzerinde (dinamik dizi, bağlı liste, ikili ağaç) işlem yapmaktadır. Diziler de bir veri yapısı olarak ele alındıklarından, bir çok *STL* işlev şablonu diziler üzerinde de işlem yapabilir. Bu işlevler daha sonra ayrıntılı olarak ele alacağımız *iterator* mantığına dayandırılmıştır. Algoritmaların çoğu bir veri yapısında bulunan ilk elemanının ve sondan bir sonraki elemanının konum bilgisini alacak biçimde düzenlenmiştir. Algoritmaların çoğu normal diziler için de kullanılabilir. Normal bir dizi söz konusu olduğunda, dizinin ilk elemanının ve son elemandan bir sonraki elemanın adresi algoritmaya

geçilerek, algoritmanın dizinin tüm elemanları üzerinde iş yapması sağlanabilir. Aşağıda *STL*'de bulunan bazı işlev şablonları örnek olarak veriliyor. *STL* bölümünde tüm işlev şablonlarını tek tek ele alacağız.

sort İşlev Şablonu

Elemanları bellekte ardışıl olarak yer alan bir veri yapısında tutulan elemanları sıralamak için kullanılır. Şablonun bildirimi aşağıdaki gibidir:

```
template<class RandomIt>
void sort(RandomIt first, RandomIt last);
```

Herhangi türden bir dizi ya da dinamik bir dizi söz konusu olduğunda işlevin *first* ve *last* isimli çağrı parametrelerine, sıralanacak dizinin ilk elemanının ve son elemandan bir sonraki nesnenin adresi geçilmelidir.

find İşlev Şablonu

find işlev şablonu ile bir veri yapısında bir değer sıralı bir biçimde aranır, aranan değer bulunursa bulunduğu yerin konum bilgisi ile bulunamaz ise son elemandan bir sonraki elemanın konum bilgisiyle geri dönülür:

```
template<class InputIt, class ValueType>
InputIt find(InputIt first, InputIt last, const ValueType& val);
```

copy İşlev Şablonu

copy işlev şablonu bir veri yapısındaki elemanları bir başka veri yapısına kopyalamak için kullanılır. İşlev, kopyalanacak veri yapısındaki ilk ve sondan bir sonraki elemanın konum bilgilerini ve kopyalamanın yapılacağı hedef veri yapısındaki ilk elemanın konum bilgisini alır. İşlev hedef veri yapısındaki sondan bir sonraki elemanın konum bilgisiyle geri döner:

```
template <class InputIt, class OutputIt>
OutputIterator
copy (InputIt sourceBeg, InputIt sourceEnd, OutputItdestBeg);
```

random_shuffle İşlev Şablonu

Bellekte bitişik olarak yer alan bir veri yapısında tutulan elemanları karıştırmak için kullanılır. Karıştırma işlemi rastgele elemanların yer değiştirmesiyle yapılan bir işlemdir.

```
template<class T>
void random_shuffle(RandomIt first, RandomIt last);
```

for_each İşlev Şablonu

Bir veri yapısındaki tüm elemanların belirli bir işlemden geçirilmesi amacıyla kullanılır. Şablonun bildirimi aşağıdaki gibidir:

```
template <class InputIt,  class UnaryProc>
UnaryProc
for_each (InputIt beg, InputIt end, UnaryProc op)
```

Veri yapısının *(first last)* aralığındaki tüm elemanlar *op* çağrı parametresiyle belirlenen işleve argüman olarak gönderilir.

Aşağıda örnek olarak verilen işlev şablonlarına ilişkin örnek bir kod veriliyor:

```
template <typename T>
void display(T begin, T end)
{
    for (T iter = begin; iter != end; ++iter)
        std::cout << *iter << " ";
    std::cout << endl;
}

void func(int &r)
{
    r %= 5;
}

#include <iostream>
#include <algorithm>
#include <cstdlib>
```

```
int main()

{

    const int size = 20;
    int a[size];

    for (int k = 0; k < size; ++k)
        a[k] = rand() % 100;

    cout << "a dizisi yazdırılıyor" << endl;
    display(a, a + size);

    sort(a, a + size);

    cout << "a dizisi sıralandı" << endl;
    cout << "a dizisi yazdırılıyor" << endl;
    display(a, a + size);

    random_shuffle(a, a + size);

    cout << "a dizisi karıştırıldı\na dizisi yazdırılıyor" << endl;
    display(a, a + size);

    cout << "dizide aranacak değeri girin : ";
    int val;

    cin >> val;

    int *ptr = find(a, a + size, val);
    if (ptr == a + size)

        cout << "dizide " << val << " degeri bulunamadı!" << endl;
    else {

        cout << "bulundu!" << endl;

        *ptr *= *ptr;

        cout << "a dizisi yazdırılıyor" << endl;
        display(a, a + size);

    }

    int b[size];

    copy(a, a + size, b);

    cout << "b dizisi yazdırılıyor" << endl;
    display(b, b + size);

    for_each(b, b + size, func);

    cout << "for_each işlevinden sonra b dizisi yazdırılıyor" << endl;
    display(b, b + size);



    return 0;

}
```

ÇALIŞMA ZAMANI HATALARININ YAKALANMASI VE İŞLENMESİ

Çalışan programlarda herşey beklendiği gibi gelişmeyebilir. Oluşturulan programların çalışmasında beklenmedik durumlar ortaya çıkabilir. Örneğin programın çalışması anında bir dosyanın açılması gerekiyor iken çeşitli nedenlerden bu dosya açılamayabilir. *Heap* alanından (C++ dilindeki ismiyle *free store*'dan) yapılmaya çalışılan bir dinamik yer ayırma işlemi, yeteri kadar boş alan olmadığı için başarısız olabilir. Programı kullanan kişinin girdiği veri ya da veriler, programın beklediği doğrultuda, kabul edilebilir sınırlar içinde olmayabilir. Program içinde beklenmedik bir verinin işlenmesi sonucu sıfıra bölme işlemi yapılıyor olabilir. Bir dizinin bir elemanına bir atama yapılmak istenirken, ayrılmış olan bir bellek alanının dışına yanlışlıkla erişmek söz konusu olabilir. Buna benzer beklenmedik durumlara C++ programcılığında *"exception"* denir. Bir işlev içinde böyle bir hata oluştuğunda ne yapılmalıdır? Eğer ne yapılması gerektiği açık bir şekilde biliniyorsa, bir hatanın oluşması durumunda yapılması gerekenler yapılabilir. Ancak temel sorun şudur: Dışarıya hizmet vermek amacıyla tanımlanan bir işlev, ne durumda çağrılmış olduğunu bilemez, yani çoğu zaman tanımlanan işlevin içinde bir hata durumu oluştuğu saptanabilse de, ne yapılması gerektiği konusunda yeterli bilgi yoktur. Hata oluştuğu zaman ne yapılması gerektiğine çoğu zaman, içinde hata oluşan işlevi çağıran kodlar tarafından karar verebilir.

O zaman içinde hata oluşan işlevin sorumluluğu hata durumunu kendisini çağıran işlevlere bildirmektir. İçinde hata oluşan kod, kendisini çağıracak işlevleri bu durumdan haberdar etmelidir. Böylece bu işlevi çağırmış olan işlevler, hata durumu hakkında bilgi sahibi olduklarında, gerekirse bir önlem alma şansına sahip olur.

Peki bir işlev içinde bir çalışma zamanında beklenmedik bir durumun oluşması durumunda ne yapılabilir? Programcı bu durumda, programı kullananın bir zarar görmemesi için ne gibi önlemler alabilir? İşte *exception handling* (Çalışma zamanı hatalarının yakalanması ve işlenmesi) C++ dilinin doğal araçlarından biri olan ve bu konuda kullanılan bir mekanizmadır. C++ dilinin hataların yakalanması ve işlenmesi mekanizmasını yakından tanımaya başlamadan önce, bu konuda kullanılan geleneksel araçları inceleyelim ve nesne yönelimli programlama tekniğinde bu araçların neden yetersiz kaldığını görelim.

C++ dilinin ilk dönemlerinde dilin kendi yapısı içinde çalışma zamanı hatalarının yakalanması ve işlenmesi mekanizması yoktu. Bu amaç için C dilinde de kullanılan geleneksel yöntemler kullanılıyordu. Bu yöntemler 3 ana grup altında toplanabilir:

1. İşlevlerin, yaptıkları iş konusunda başarılı ya da başarısız olduklarını gösteren bir geri dönüş değeri üretmeleri (Bir hata değerinin geri döndürülmesi).

2. Başarısızlık durumunda global bir bayrak değişkene ilgili hatayı belirleyen ayırt edici bir değer atanması (hata değeri) ve diğer işlevlerin bu global değişkenin değerini sınyarak önlem almaları.

3. Programın çalışmasının sona erdirilmesi.

Nesne yönelimli programlama söz konusu olduğunda her üç yöntemin de çok önemli yetersizlikleri ve sakıncaları vardır. Bu yetersizlik ve sakıncalardan bir kısmı özellikle çok büyük projeler söz konusu oldugunda, kabul edilebilir olmaktan çıkar. Şimdi bu sakıncaların neler olabileceği konusuna değinelim:

Bir Hata Değerinin Geri Döndürülmesi

Bu teknik küçük programlarda belirli bir yere kadar kullanılabilir. İşlevler bir hata durumunda, daha önceden belirlenmiş olan hata değerlerine geri döner. Çağrılan işlevlerin geri dönüş değerleri sınanarak bir hata oluşup oluşmadığı, ve ne tür bir hatanın oluştuğu saptanır. Ancak bu tekniğin önemli dezavantajları vardır. Geri dönüş değeri olarak verilen hata değerleri bir standarda bağlı değildir. Bir kütüphanede örneğin *0* değeri başarısızlık belirtirken başka bir kütüphanede *0* değeri başarı durumunu gösterebilir. C dilinin standart işlevleri için bile bu konuda bir birlik olduğunu söyleyemeyiz. Dosya açan standart C işlevi olan *fopen* işlevinin *0* (NULL) değeri dosyanın başarılı bir şekilde açılamadığını gösterirken, dosyayı kapayan *fclose* işlevinin *0* olan geri dönüş değeri dosyanın başarılı bir şekilde kapatıldığını göstermektedir. C dilinde uygulama geliştirenler genellikle işlevin başarı ya da başarısızlıklarını gösteren geri dönüş değerlerini simgesel değişmezler biçiminde tanımlayarak başlık dosyası içine yerleştirir.

Ancak şüphesiz bu simgesel değişmezler de standart bir hale getirilmemiştir.

Ayrı ekipler tarafından farklı zamanlarda geliştirilmiş kütüphaneler, aynı proje içinde kullanıldığında, hata durumları söz konusu olabilir. İşlevlerin geri dönüş değerlerinin farklı biçimlerde yorumlanması karışıklıklar yaratır.

Ayrıca her bir geri dönüş değerinin sınanması ve yorumlanması, zahmetli ve zaman alıcı bir işlemdir. Çoğu zaman bunun için ek bir koda gereksinim duyulur. İşlev her çağrıldığında geri dönüş değeri mutlaka işlenmelidir. Geri dönüş değeri sınanmadan kullanılırsa, bazı durumlarda çalışan program bile çökebilir. Örneğin *malloc* işlevinin geri dönüş değerinin, sınanmadan kullandığını düşünelim. *malloc* işlevi başarısız olduğunda *NULL* adresine geri döner. Böylece bir gösterici hatasına neden olunur.

Çağrılan bir işlev içinde bir hata durumu oluşursa, çağrılan işlevin yürütülmesi, normal olarak üreteceği geri dönüş değeri ifadesine gelmeden önce, hata oluştuğu bilgisini veren bir başka *return* deyimiyle sonlandırılır. Başka bir kod da işlevin geri dönüş değerini sınayan ve işlev herhangi bir hata değeri geri döndürmüşse, programın devam edip etmemesi hakkında karar veren ek kod parçasıdır. Bu da bir çok durumda yazılan kodun boyutunu büyük ölçüde artırır ayrıca programın okunabilirliğini bozar, programın geliştirilmesi ve test edilmesi konusunda sakıncalar yaratır.

Bazı durumlarda çağrılan işlev içinde bir başarısızlık ya da hata oluştuğunda, bu durumu dışarıya iletecek özel bir hata değeri kullanılamaz. Örneğin bir adrese geri dönen bir işlev, başarısızlık ya da bir hata durumunda *NULL* adresine geri dönebilir. Ama hesap edilmek istenen bir tamsayı değerine geri dönen bir işlev, bir hata durumu oluştuğunda hangi değere geri dönebilir? Çünkü ayırt edici ve hatayı gösterebilecek bir değer söz konusu değildir. C dilinden tanıdığımız *atoi* işlevini ele alalım:

```
int atoi(const char *str);
```

atoi işlevi adresini aldığı yazıdan, karakterler olarak kodlanmış *int* türden bir değeri çekerek geri dönüş değeri olarak iletir. Peki ya yazı bir *int* değeri gösteren geçerli bir yazı değilse *atoi* işlevi hangi değer ile geri dönebilir? Derleyicilerin çoğu bu durumda *atoi* işlevi *0* değeriyle geri döndürür. Bu da şüpheli bir durum yaratır. Peki *atoi* işlevine gönderilen yazının içinde gerçekten *"0"* varsa ve bu değerin elde edilmesi gerekiyorsa?

Önemli bir nokta da C++ dilinde bazı işlevler için geri dönüş değeri diye bir kavramın söz konusu olmamasıdır. Örneğin kurucu işlevler (*constructors*), tür dönüştürme işlevler için geri dönüş değeri diye bir kavram söz konusu değildir. Bu işlevler içinde oluşabilecek hata durumu yukarıda anlatılan geleneksel teknikle yakalanamaz.

Global Bir Bayrak Değişkene Değer Atanması

Çalışma zamanı hatalarının bildirilmesinde alternatif bir teknik de global bir değişkene atama yapılmasıdır. Global bir değişkenin değeri yapılan son işlemin başarısı hakkında bir fikir verir. C dilinde hata durumunda geri dönüş değeri mekanizması standart bir hale getirilmemiştir ama global bir değişkene atama yapılması mekanizması standart bir hale getirilmiştir. C dilinin standart bir başlık dosyası olan *<errno.h>* dosyası global bir değişken olan *errno* değişkenine değer atanması konusundaki mekanizmayı tanımlar.

Standart işlevler içinde bir hata durumu oluştuğunda *errno* değişkenine ayırt edici bir değer atanır. Bu yöntemin de önemli sakıncaları vardır: Çok işlemli bir çalışmada (*multi- thread*) bir akış içindeki işlev çağrısı yüzünden oluşan bir hata durumu, ilgili global değişkenin değeri daha sınama amacıyla alınamadan başka bir akış tarafından tekrar değiştirilebilir. Ayrıca böyle bir hata kodunun global bir değişkenden sürekli olarak alınarak okunması çok disiplinli bir programlama tekniği gerektirmektedir.

Global bir değişkene yapılan atama ya da işlevin geri dönüş değerinin sınanması işlev içinde oluşan bir hatayı işlevi çağırana iletebilir ancak bu hatanın ortadan kaldırılması konusunda bir şey yapılmaz. Örneğin dosya açan bir işlev dosyanın açılamadığını dışarıya bildirebilir ancak başka bir işlevin açılamamış olan bu dosyaya bir veri aktarması ve bu dosyayı kapatması konusunda bir önlem alamaz. Dahası *errno* gibi bir değişkenin değeri alındıktan sonra, sınamayı yapan kişi tarafından tekrar sıfırlanması gerekir ki, bundan sonra saptanacak hata bildirimlerinde bir yanlışlık oluşmasın. Eğer programcı global değişkeni tekrar normal değerine getirmeyi unutursa başka bir işlevin başarısı sınandığında, işlev aslında başarılı olmuş iken programcı tarafından işlevin başarısız olduğu gibi yanlış bir yargıya varılabilir. Böylece gereksiz bir şekilde, olmayan bir hata işlenmeye çalışılır ki bu da şüphesiz beklenmeyen sonuçlar yaratabilir.

Programın Sonlandırılması

Çalışma zamanında oluşan hataların işlenmesi konusundaki en kökten araçlardan biri ciddi bir hata ile karşılaşıldığında programın akışının sonlandırılmasıdır. Bu durum şüphesiz ilk iki yöntemdeki bazı yüklerden programcıyı kurtarır. Program bir hata oluştuğunda sonlandırılacağı için çağrılan bir işlevin geri dönüş değerinin bir hata durumunun saptanması amacıyla sınanmasına, ya da global bir bayrak değişkeninin değerinin her işlev çağrısından sonra sınanmasına gerek kalmaz. Standart C dilinde bir programın sonlandırılması iki ayrı işlevle gerçekleştirilebilir. *exit* ve *abort* işlevleri. Bu işlevleri kısaca hatırlayalım:

exit İşlevi

Bir standart C işlevi olan *exit, cstdlib* başlık dosyası içinde bildirilmiştir.

```
void exit(int status);
```

exit işlevi, çağrıldığı zaman programı sonlandırır. *exit* işlevi, *main* işlevi başarılı bir şekilde sona erdiğinde çağrılabileceği gibi, bir hata durumununun oluşması durumunda da çağrılabilir. *exit* işlevi, kontrolü işletim

sistemine devretmeden önce bütün dosyaların tampon alanlarını tazeler (*flush eder*), tüm açık dosyaları kapatır. *exit* işlevine gönderilen *0* değeri, programın başarıyla sonlandırıldığını anlatırken sıfır dışı bir değer ise programın başarısızlık nedeniyle sonlandırıldığı iletisini verir. *exit* işlevine çağrı olarak geçmek için, okunabilirlik açısından c*stdlib* başlık dosyası içinde aşağıdaki simgesel deeğişmezler tanımlanmıştır:

```
#define     EXIT_SUCCESS        0

#define     EXIT_FAILURE        1
```

C++'ta *exit* işlevi çağrılırsa program sonlandırılmadan önce tüm global sınıf nesneleri için sonlandırıcı işlevler çağrılır. Ancak yaratılmış olan yerel sınıf nesneleri ve dinamik sınıf nesneleri için sonlandırıcı işlevler çağrılmaz.

exit işlevi çağrıldığında programın sonlandırılmasından önce, programcının belirleyeceği bazı işlevlerin çağrılması sağlanabilir. Bu amaçla standart *atexit* işlevi kullanılır:

atexit işlevinin de bildirimi *stdlib.h* başlık dosyası içindedir:

```
int atexit(void (*)(void);
```

Yukarıdaki bildirimden de görüldüğü gibi *atexit* işlevinin parametre değişkeni geri dönüş değeri ve parametresi olmayan bir işlevi gösterecek göstericidir. *atexit* işlevine adresi gönderilen işlev, *exit* işleviyle program sonlandırılmadan önce çağrılmak üzere kaydedilir. İşlevin geri dönüş değeri işlevin başarı durumunu gösterir. İşlevin *0* değerine geri dönmesi kayıt işleminin başarılı olduğunu anlatırken *0* dışı bir değere geri dönmesi başarısızlık durumunu bildirir. *atexit* işleviyle birden fazla işlev kayıt edilebilir. Bu durumda kayıt edilen işlevler, kayıt edilmelerine ters sırada çağrılır. Aşağıdaki programı derleyerek çalıştırınız:

```
#include <iostream>
#include <cstdlib>

using namespace std;

void func1()

{

    cout << "func1()" << endl;

}


void func2()

{
```

```
    cout << "func2()" << endl;
}

void func3()
{
    cout << "func3()" << endl;
}

int main()
{
    atexit(func3);
    atexit(func2);
    atexit(func1);

    exit(EXIT_FAILURE);
}
```

abort İşlevi

abort işlevi de *cstdlib* başlık dosyasında bildirilmiştir:

```
void abort(void);
```

abort işlevi çağrıldığında program sonlanır. *abort* çağrıldığı yere başarı durumu gösteren bir değer iletmez. Bu işlev programın anormal bir şekilde sonlandırılmasına işaret eder. İşlev *stderr* akımına *"abnormal program termination"* yazısını yazarak programı sonlandırır. *abort* işlevi ile program sonlandırıldığında dosyaların tampon alanları tazelenmez, ne yerel ne de global sınıf nesneleri için sonlandırıcı işlevler çağrılır.

Kritik uygulamalarda bir hata oluşması durumunda birdenbire programın sonlandırılması kabul edilebilecek bir yaklaşım değildir. Örneğin bir hastahanedeki yaşam destek ünitesini yöneten bir program içinde beklenmedik bir durum oluştuğunda programın sonlandırılması durumu kabul edilebilir mi? Ticari bir çok uygulamada da durum böyledir. Bir faturalama programı, bankacılığa yönelik bir uygulama, bir hata oluştu diye birdenbire sonlandırılamaz. Gerçek hayata ilişkin programlarda bu durumlar için yeterli ve gerekli önlemlerin alınması gereklidir.

İşletim sistemi gibi bir uygulamada bile, çalışma zamanında ciddi bir hatanın oluşması durumunda programın (işletim sisteminin) sonlandırılması sorunlu durumlar yaratabilir. Çalışan bir işlev bir hata durumu saptadığında çoğu zaman oluşan hatanın ciddiyeti konusunda bir fikir sahibi olamaz. Örneğin dinamik blok elde etmeye çalışan bir işlev, söz konusu işlemin başarısız olması durumunda, dinamik yer ayırma işleminin ne amaçla yapıldığını bilmez. Örneğin dinamik yer elde etme işlemi programcının bir *debugger* kullanması anında ya da bir tablolama programını kullanması durumunda istenmişse, pekala bir uyarı iletisi verilerek kullanıcının açık olan

bütün programları kapatarak işini öyle sürdürmesi istenebilir. Ancak dinamik yer ayırma işleminin başarısızlık nedeni tamamıyla donanımsal da olabilir. Bu durumda daha ciddi bir önlem alınması gerekir. Yani hatanın ne nedenle olup olmadığı programın sonlandırılacağı mı yoksa bir önlem alınarak sürdürüleceği mi konusunda bir seçim şansı verilmelidir.

Bazı durumlarda *exit* ya da *abort* işlevleriyle programın akışı sonlandırılabilse de nesne yönelimli programlama ortamında, programın akışı bu işlevler ile sonlandırılmamalıdır, zira bu işlevler C++ dilinin nesne tabanlı programlama modeli hakkında bilgi sahibi değillerdir.

C++ dilinde bir sınıf nesnesi, kurucu işlev ya da herhangi bir üye işlevi kanalıyla bir takım kaynakları tutabiliyor olabilir. Bu kaynaklar, dinamik olarak elde edilen blokların başlangıç adresleri, dosyalara ilişkin *handle* değerleri, iletişim amacıyla portlara ilişkin set edilmiş değerler, giriş çıkış birimlerine ilişkin aygıtlar, kilit kontrolleri vs. olabilir. Birçok durumda bu kaynakların, nesnenin yaptığı iş bitince serbest bırakılması ya da geri verilmesi gerekir. C++ dilinde genel olarak bu kaynaklar nesneye ilişkin sonlandırıcı işlev yardımıyla geri verilirler ya da serbest bırakılır. Yerel sınıf nesneleri, bilinirlik alanları sona erdiğinde otomatik olarak serbest bırakılır. Ancak ne *exit* ne de *abort* işlevi yerel nesnelerin sonlandırıcı işlevlerinin çağrılmasını sağlar. Bu nedenle programların bu işlevler ile ani olarak sonlandırılması, geri dönüşü olmayan zararların oluşmasına neden olabilir. Bir veri tabanı sistemi tamamıyla bozulabilir, dosyalar kaybedilebilir, değerli veriler kaybolabilir. Nesne yönelimli programlamada *exit* ve *abort* işlevleri kullanılmamalıdır!

Görüldüğü gibi, C dilinin hata saptamasına ve işlenmesine ilişkin geleneksel yöntemlerden hiçbiri C++ dili için yeterli değildir. C++ dilinin kullanılmasındaki ana temalardan biri büyük projelerin C diline göre daha iyi ve daha güvenli bir şekilde yazılmasına olanak sağlamasıdır.

C++ dilinin dinamik gelişme süreci içinde C++ dilinin yaratıcıları çalışma zamanına ilişkin hataların yakalanmasına ve işlenmesine ilişkin yeterli bir mekanizmanın eksikliğinin farkında olduklarından, yukarıdaki sakıncaları içermeyen bir yöntemin arayışına girdiler. Önerilen sistem, beklenmeyen bir durum oluştuğunda programın akışının otomatik olarak, hatayı işleyecek bir kod parçasına akışın sağlanmasıydı. Söz konusu yöntem basit ve etkili olmalıydı, bir takım değerlerin sürekli olarak test edilmesi ilkesine dayanmamalıydı. Fakat en önemlisi, bir hata oluştuğunun saptanması durumunda, hatayı işleyen kodun bu durumdan otomatik olarak haberdar olmasının sağlanmasıydı. Ayrıca bir hata yerel olarak kendi kod grubu içinde işlenemediğinde, o kapsamdaki yerel sınıf nesnelerinin tutmakta olduğu kaynaklar, daha genel bir hata işleyicisinin devreye girmesinden önce, sisteme iade edilmeliydi. Uzun süren incelemeler ve araştırmalar sonucunda çalışma zamanında oluşan hataların yakalanmasına ve işlenmesine ilişkin bir mekanizmaya yönelik tasarı 1989 yılında kabul edilerek C++ diline eklendi. C++ öncesinde başka programlama dilleri bu yönde mekanizmalar oluşturmuştu. Örneğin daha 1960'lı yıllarda *PL/1* dili dilin kendi yapısı içinde var olan bir hata işleme mekanizmasına sahipti. Yine 1980'li yılların başlarında *Ada* dili böyle bir mekanizmayı kendi bünyesi içine katmıştı. Fakat bu mekanizmalar C++ dilinin nesne yönelimli programlama yaklaşımına uygun düşmüyorlardı. Bu yüzden C++ diline eklenen mekanizma kendi türü için ilkti ve kendisinden sonra geliştirilen bir çok programlama dili tarafından kullanıldı.

Çalışma zamanında hataların yakalanmasına ve işlenmesine ilişkin bir mekanizmanın programcılara sağlanması derleyiciyi yazanlara çeşitli zorluklar getirir. Örneğin ilk C++ derleyicisi olan *cfront UNIX* işletim sistemi altında çalışıyordu. Bir çok *UNIX* derleyicisi gibi *cfront* da önce *C++* kaynak kodunu *C* koduna dönüştürüyor daha sonra elde edilen *C* kodunu derliyordu. *cfront* derleyicisinin *4.0* sürümünün hata işleme mekanizmasına sahip olması planlanmıştı. Ancak bütün gerekleri sağlayan bir hata işleme mekanizmasının geliştirilmesi o kadar karmaşık bir prosesti ki, *4.0* sürümünü geliştiren ekip, çalışmaya başlamasından 1 yıl sonra projeyi sonlandırma kararı aldı.

cfront 4.0 sürümü hiç olmadı. Ancak daha sonra hata işleme mekanizması C++ dilinin doğal bir aracı durumuna getirildi. Daha sonraki derleyiciler daha sonra bu aracı desteklemeye başladı.

Çalışma Zamanı Hatalarını Saptama ve İşleme Mekanizmasını İmplemente Etmek Neden Zordur?

Mekanizmayı bir derleyicinin implemente etmesindeki birinci zorluk oluşan belirli hatayı işleyecek uygun bir kodun *(handler)* bulunmasıdır.

İkinci zorluk, hatayı saptayan kod parçasının hatayı işleyecek koda göndereceği hata nesnesinin çok biçimli özelliklere sahip olabilmesidir. Yani türemiş sınıf türünden bir hata nesnesini işleyecek olan uygun bir işleyici kod bulunamadığı zaman taban sınıf türünden nesneyi alabilecek bir hata işleme kodunun devreye girebilmesidir. Ancak ilk zamanlarda, çok biçimliliği destekleyen bir nesnenin çalışma zamanındaki türünü saptayacak bir mekanizma *(Dynamic typing)* C++ diline eklenmemişti. Hata işleme aracının yeterli bir duruma getirilebilmesi için önce dinamik tür kontrolü mekanizmasının sıfırdan geliştirilmesi gerekiyordu.

Başka bir komplikasyon da hatanın işlenmesi durumunda, hata işleyici işleve programın akışını yönlendirmeden önce yerel sınıf nesnelerinin sonlandırıcı işlevlerinin çağrılmasının sağlanmasıydı. Bu mekanizmaya İngilizcede *yığın dengelenmesi (stack unwinding)* denir.

Çalışma Zamanı Hata İşleme Mekanizmasının Uygulanması

Exception Handling esnek ve karmaşık bir araçtır. C dilinde kullanılan geleneksel hata işleme yöntemlerinin sakıncalarını içermez ve çok çeşitli çalışma zamanı hataları için

kullanılabilir. Ancak bir kaynak kod içinde, C++ dilinin diğer araçları gibi bu araç da kötü kullanılmış olabilir. Bu mekanizmayı etkili bir şekilde kullanabilmek için, mekanizmayı iyi bir şekilde tanımak ve geri planda makina düzeyinde neler olduğunu anlamak gerekir.

Çalışma Zamanı Hata İşleme Mekanizmasının Bileşenleri

CZHI mekanizmasında, çalışma zamanı içinde bir hata oluştuğunda programın akışı bu hatayı işleyecek uygun bir kod parçasına (*handler*) yönlendirilir. Bu yönlendirmeyle birlikte hatayı işleyen koda bir nesne değerle (*call by value*) ya da adresle (*call by reference*) geçilir. Söz konusu nesneye bundan sonraki bölümlerde "hata nesnesi" (*exception object*) diyeceğiz. Hata nesnesi doğal veri türlerinden olabileceği gibi, bir sınıf türünden de olabilir.

Mekanizmanın dört bileşeni vardır.

- *try* bloğu
- *try* bloğu ile ilişkilendirilen *catch* blokları (hata işleyen kodlar -*handler*)
- *throw* ifadesi
- hata nesnesinin kendisi Mekanizma kabaca

şöyle özetlenebilir:

Çalışma zamanı içinde bir hata oluştuğunda bu durumda *throw* ifadesiyle bir hata nesnesi gönderilir. Gönderilen bu hata nesnesini hata işleyen kodlardan uygun olan bir tanesi yakalar. Eğer hata yakalanamaz ise standart *abort* işlevi çağrılarak program sonlandırılır.

throw Anahtar Sözcüğü ve Hata Nesnesinin Gönderilmesi

throw C++ dilinin bir anahtar sözcüğüdür. *throw* anahtar sözcüğünü bir ifade izler. *throw* anahtar sözcüğünü izleyen ifade gönderilen hata nesnesinin türünü ve değerini belirler. Gönderilen hata nesnesini bir hata işleyen kod parçası yani ingilizce ismiyle bir *handler* yakalar. *throw* ifadesi herhangi bir işlevin içinde yer alabilir. Aşağıdaki kod parçasını inceleyin:

```
int &Array::at(int index);

{

    if (index >= m_size)
        throw index;


    return pd[index];

}
```

Yukarıda *Array* sınıfının *at* isimli üye işlevinin tanımı yer alıyor. İşlevin kaynak kodu içinde, dışarıdan gelen *index* değerinin geçerli olup olmadığı sınanıyor. *index* değeri geçerli değilse, *int* türden bir hata nesnesi gönderiliyor.

try Bloğu

try C++ dilinin bir anahtar sözcüğüdür. *try* anahtar sözcüğünü bir bloğun izlemesi zorunludur. İçinden bir hata nesnesi gönderilme olasılığı bulunan kod parçası bir *try* bloğu içine alınabilir:

```
#include <iostream>

#include "array.h"


int main()

{

    Array myarray(10);
    int index;


    std::cout << "bir index degeri girin ";
    std::cin >> index;
```

```
    try {

        int x = myarray.at(10);

    }

    //...
    return 0:

}
```

Yukarıdaki *main* işlevinde çağrılan *at* işlevi içinden bir hata nesnesi gönderilebileceği için

```
int x = myarray.at(10);
```

deyimi bir *try* bloğu içine alınmış.

Bir *try* bloğu içinde C++ dili sözdizimine uygun herhangi bir deyim bulunabilir. *try* bloğu içinde bildirim ya da tanımlama yapılabilir. Bir *try* bloğu yerel bir bilinirlik alanı oluşturur. Bir *try* bloğu içinde bildirimi yapılan değişkenler bu bloğun dışında görülmez.

catch Anahtar Sözcüğü ve catch Blokları

catch C++ dilinin bir anahtar sözcüğüdür. *catch* anahtar sözcüğünü açılan ve kapanan bir ayraç izler. Bu ayraç içinde, işlevlerdeki parametre değişkeni bildirimine benzer bir şekilde değişken bildirimi yapılabilir. Ayraçlardan sonra yine bir blok yerleştirilir. *catch* anahtar sözcüğünü bir ayracın izlemesi ve ayracın kapanmasından sonra da bir bloğun yerleştirilmesi sözdizimsel bir zorunluluktur.

catch bloğu hatayı işleyen bloktur(*handler*). *catch* blokları *try* bloklarını izler. Bir *try* bloğundan sonra herhangi bir *catch* bloğunun yer almaması geçersizdir. Eğer bir *catch* ayracı içinde tanımlanmış nesnenin türü ile gönderilen hata nesnesinin türü tamamen aynı ise, *catch* parametresine gönderilen hata nesnesinin değeri aktarılır. Programın akışı *catch* bloğu içinde akmaya yönlendirilir. Bu duruma "gönderilen hata nesnesinin yakalanması" denir. Bir *try* bloğu ayrı bir bilinirlik alanı belirtir. Bu blok içinde tanımlanan nesneler *try* bloğunu izleyen *catch* blokları içinde bilinmez.

try bloğu ile *catch* bloğu arasında başka bir kod olamaz. Bir *try* bloğundan sonra istenilen sayıda *catch* bloğu yer alabilir. Şimdiye kadar anlatılanları bir kod parçası ile örnekleyelim:

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main()
```

```
{
    int val;


    cout << "pozitif bir sayi giriniz : ";
    cin >> val;


    try {
        if (val < 0)
            throw val;
    }
    catch(int x) {
        cout << x << " negatif! program sonlaniyor!" << endl;
        exit(EXIT_FAILURE);
    }
    cout << val << " degeri girildi!" << endl;


    return 0;
}
```

Bir *try* bloğunu birden fazla *catch* bloğu izleyebilir. Bu durumda *catch* parametrelerinin türlerinin farklı olması gerekir. Aynı türden nesne bekleyen birden fazla *catch* parametresinin bulunması geçersizdir. Aşağıdaki örneği inceleyin:

```
#include <iostream>

void may_raise_exception();
int main()
{
    try {
        may_raise_exception();
    }
    catch (int) {
        //...
    }
```

```
    catch (double) {
        //...
    }
    catch (long)  {
        //...
    }
    std::cout << "programın akisi bu noktadan devam edecek!" << std::endl;
    //...
    return 0;
}
```

Yukarıda tanımlanan *main* işlevi içinde bir *try* bloğu oluşturuluyor. Oluşturulan *try* bloğu içinde *may_raise_exception* işlevi çağrılıyor. *try* bloğunu üç ayrı *catch* bloğu izliyor. Birinci *catch* bloğu ya da başka bir deyişle birinci hata işleyen blok, *int* türden bir hata nesnesini yakalamaya çalışıyor. İkinci hata işleyen blok, *double* türden, -üçüncü hata işleyen blok, *long* türden bir hata nesnesini yakalamaya çalışıyor. Herhangi bir *catch* bloğu gönderilen hata nesnesini yakalarsa, programın akışı içinde ilgili *catch* bloğunun içindeki deyimler yürütülür. Programın akışı *catch* bloğundan sonra en son *catch* bloğundan sonra yer alan deyimle sürer.

catch blokları *break* deyimiyle sonlanan *case*'ler gibidir. Uygun bir *catch* bloğu bir hata nesnesini yakaladığında programın akışı bu *catch* bloğunun içine girer. Bu *catch* bloğunun kodu yürütüldükten sonra programın akışı son *catch* bloğunu izleyen kod parçasının yürütülmesiyle sürer.

try bloğu içine alınan kod parçası içinde gönderilen hata nesnesinin yakalanması için

throw işleminin doğrudan bu blok içinde yapılması gerekmez.

```
try {
    func();
}
catch (int) {
    //...
}
```

Yukarıdaki *try* bloğu içinde *func* işlevi çağrılıyor. *func* işlevi içinde çağrılan bir işlevin de, *int* türden bir hata nesnesi göndermesi durumunda, *catch* bloğu bu hata nesnesini de yakalar.

throw İfadesiyle catch Parametresinin Uyumu

Bir *catch* bloğunun gönderilen hata nesnesini yakalayabilmesi için *throw* ifadesi ile *catch* parametresinin türlerinin tamamen aynı olması gerekir. *catch* parametresinin önünde bulunan *const* ya da *volatile* niteleyicileri tam uyumu bozmaz. Ancak diğer otomatik tür dönüşümü ve yükseltme kuralları burada geçerli değildir. Örneğin *throw* ifadesi *char* türden *catch* parametresi *int* türden ise, tamsayıya yükseltme (*integral promotion)* kuralı işletilmez. Benzer şekilde gönderilen hata nesnesinin türünün *float* türü olduğunu kabul edelim. Parametresi *double* türden olan bir *catch* bloğu *float* türden bir hata nesnesini yakalayamaz.

Gönderilen Hata Nesnesinin Bir Sınıf Türünden Olabilmesi

İçinde çalışma zamanını hatasının oluşabileceği kod parçasının, hatayı işleyecek kod parçasına gönderdiği hata nesnesi, bir sınıf türünden de olabilir. Bu durumda şüphesiz bu hatayı yakalayacak *catch* bloğunun parametresi aynı sınıf türünden bir nesne ya da aynı sınıf türünden bir referans olmalıdır.

Hata nesnesi oluşturmada çoğunlukla doğal türler yerine sınıf nesneleri kullanılır. Böylece hata oluştuğunda gönderilen sınıf nesneleri *catch* parametresine alındıktan sonra, sınıfın üye işlevleri kullanılarak oluşan hata ele alınır. Sınıf kütüphanelerinin çoğunda çok biçimliliği destekleyen hata işleme sınıfları vardır. C++ standart kütüphanesinde de hiyerarşik hata işleme sınıfları tanımlanmıştır.

Bir kütüphane içinden gönderilecek hata nesnelerinin bir sınıf hiyerarşisi biçiminde organize edilmesinin önemli bir faydası da, kütüphaneye yeni hata sınıflarının eklenmesinden sonra hata işleyen kodlarda bir değişikliğin gerekmemesidir.

catch all Bloğu

İstenirse, ne türden olursa olsun gönderilen bir hata nesnesinin bir *catch* bloğu tarafından yakalanması sağlanabilir. Bu durum *catch* bloğu ayracı içine üç nokta atomu (*elipsis*) yerleştirilmesiyle sağlanır:

```
try {
    func();
}
catch (…) {
    //…
}
```

Yukarıdaki örnekte, *try* bloğu içinde çağrılan *func* işlevi içinden ne türden hata nesnesi gönderilirse gönderilsin, *try* bloğunu izleyen *catch all* bloğu bu hata nesnesini yakalar.

catch Bloklarının Uygun Sırası

Uygulamalarda genellikle *catch* bloklarının parametreleri, en özel olandan en genel olana doğru yukarıdan aşağıya doğru seçilir. Yani türemiş sınıf parametreli *catch* blokları taban sınıf parametreli *catch* bloklarından daha yukarıda yer alır. Çünkü *catch* blokları yukarıdan aşağıya doğru taranır. Aşağıdaki kodu inceleyin:

```
#include <stdexcept>
#include <iostream>


using namespace std;

int main()

{

    try {

        char * buff = new char[100000000];

        //...

    }

    catch(bad_alloc &r) {

        cout << "memory allocation failure";

        //...

    }

    catch(exception &r) {

        cout << r_exception.what() << endl;

    }

    catch(...) {

        cout << "bilinmeyen hata nesnesi" << endl;

    }

    return 0;

}
```

main işlevinde yer alan *try* bloğu içinde, *new* işleciyle büyük bir dinamik blok elde edilmeye çalışılıyor. *try* bloğunu izleyen ilk *catch* bloğu *bad_alloc* sınıfı türünden hata nesneleri yakalanmaya çalışılıyor. *bad_alloc* türünden bir hata nesnesi yakalandığında daha aşağıdaki *catch* bloklarına hiç bakılmaz. İkinci *catch* bloğu ile standart *exception* sınıfından türemiş herhangi bir sınıf türünden hata nesnesi yakalanabilir. *catch* bloğu içinde sınıfın *what* üye işleviyle, hatanın niteliğini gösteren bir yazının ekrana yazdırıldığını görüyorsunuz. Son *catch* bloğu *catch all* bloğudur. Bu bloğa kadar yakalanamayan bir hata nesnesi mutlaka *catch all* bloğu tarafından yakalanır. Eğer proje içinde gönderilen tüm hata nesneleri *exception* sınıf hiyerarşisine ait sınıflardan ise, normal olarak *catch all* bloğu tarafından hiç bir hata nesnesinin yakalanamaması gerekir. Bu blok tarafından bir hatanın yakalanması projeyi yazanların kontrolü dışında bir hata nesnesinin gönderilmiş olduğunu gösterir.

Yakalanamayan Hata Nesnesi

Gönderilen bir hata nesnesinin bir *catch* bloğu tarafından yakalanamaması durumuna

"yakalanamayan hata nesnesi" (*uncaught exception*) denir.

Yakalanamayan hata durumunda *terminate* isimli standart bir işlev otomatik olarak çağrılır. *terminate* işlevinin önceden belirlenmiş (*default*) davranışı, standart *abort* işlevini çağırarak programı sonlandırmaktır. Standart *terminate* işlevinin kodunun aşağıdaki gibi olduğunu düşünebilirsiniz:

```
void(*fp)() = abort;

void terminate()

{

    (*fp)();

}
```

Ancak istenirse, *terminate* işlevinin *abort* işlevi yerine başka bir işlevi çağırması sağlanabilir. *terminate* işlevinin programcı tarafından belirlenen başka bir işlevi çağırmasını sağlamak için *set_terminate* isimli standart işlev çağrılabilir. Hem *terminate* işlevinin hem de *set_terminate* işlevlerinin bildirimleri *<exception>* başlık dosyası içindedir:

```
typedef void(*terminate_handler)();

terminate_handler set_terminate(terminate_handler fp);
void terminate()
```

Yukarıdaki bildirimden şu anlam çıkartılır:

terminate_handler parametre değişkeni olmayan ve geri dönüş değeri üretmeyen bir işlevi gösteren gösterici türünün yeni tür ismidir.

set_terminate işlevi *terminate_handler* türünden bir geri dönüş değerine ve *terminate_handler* türünden bir parametre değişkenine sahiptir. Daha uzun bir ifadeyle *set_terminate* işlevinin geri dönüş değeri, geri dönüş değeri üretmeyen ve parametre değişkeni olmayan bir işlev türünden adrestir. *set_terminate* işlevinin parametre değişkeni geri dönüş değeri üretmeyen ve parametre değişkeni olmayan bir işlevi gösteren göstericidir. Bu durumda *set_terminate* işlevine bu türden bir işlevin adresi geçilmelidir. işlev isimlerinin işleme sokulduğunda otomatik olarak ilgili işlevin başlangıç adresine dönüştürüldüğünü hatırlayın:

```
#include <iostream>
#include <exception>
#include <cstdlib>
```

```
using namespace std;


void may_raise_exception()

{

    throw "gönderilen hata nesnesi yakalanamayacak!";

}


void myterminate()

{

    cout << "Ben myterminate isleviyim" << endl;

    cout << "gonderilen hata nesnesi yakalanamadigi icin ben cagrildim" <<
endl;

    cout << "exit islevini cagirarak programi sonlandiracagim!";
    exit (EXIT_FAILURE);

}


int main()

{

    set_terminate(myterminate);
    try {

        may_raise_exception();

    }

    catch (int) {

        //...

    }

    catch (double) {

        //...

    }

    catch (long)  {

        //...

    }

    cout << "Hata nesnesi yakalanirsa program buradan devam edecek\n";

    //...
```

```
    return 0;

}
```

Yukarıdaki örnekte, *main* işlevinde önce *set_terminate* işlevine daha önce tanımlanmış olan *myterminate* işlevinin adresi geçiriliyor. Böylece *terminate* işlevi çağrıldığında, *terminate* işlevinin önceden belirlenmiş davranış olan *abort* işlevini çağırması yerine, *myterminate* işlevini çağırması sağlanıyor.

try bloğu içinde çağrılmış olan *may_raise_exception* işlevi *const char ** türden bir hata nesnesi gönderiyor. Ama *try* bloğunu izleyen *catch* bloklarının hiçbirinde bu türden bir hata nesnesi yakalanamıyor. Bu durumda yakalanamayan hata durumu oluşuyor.

Otomatik olarak *terminate* işlevi çağrılıyor. *terminate* işlevi de *myterminate* işlevini çağrıyor. *myterminate* işlevi de ekrana bulgu iletilerini yazdıktan sonra, *exit* işlevini çağırarak programı sonlandırıyor.

Yığının Dengelenmesi

Bir hata nesnesinin gönderilmesi ve gönderilen hata nesnesinin yakalanması işlev çağrı mekanizmasına benzer gibi görünse de aslında durum işlev çağrısından çok farklıdır.

Aşağıdaki örneği dikkatle inceleyin:

```
#include <iostream>


using namespace std;

void func1();

void func2();
void func3();



int main()

{

    func1();

    //...
    return 0;

}


void func1()

{

    func2();

    //...
    return;
```

```
}


void func2()

{

    func3();

    //...
    return;

}

void func3()

{

    //...
    return;

}
```

Yukarıdaki programda, *main* işlevi *func1* işlevini, *func1* işlevi *func2* işlevini, *func2* işlevi de *func3* işlevini çağrıyor. Programın akışı *func3* işlevi içinde ilerlerken *return* deyimiyle karşılaşıldığında, akış *func2* işlevinde kaldığı yerden sürer. *func2* işlevinde *return* deyimi ile karşılaşıldığında ise bu durumda *func1* işlevinde kalınan yerden akış sürer. Nihayet *func1* işlevinin çalışması sona erdiğinde, bu kez programın akışı *main* işlevinde kaldığı yerden sürer.

Oysa *func3* işlevi içinde bir hata nesnesi gönderilmiş olsaydı, programın akışı doğrudan bu hata nesnesinin yakalandığı *catch* bloğuna yönlenirdi. Bu durumu sağlamak için gerçekleştirilen işlemlere yığının dengelenmesi (*stack unwinding*) denir. Aşağıdaki kodu inceleyin:

```
#include <iostream>

void func1();

void func2();
void func3();



int main()

{

    try {

        func1();

    }

    catch(int x) {
```

```
        //...
    }
    //...
    return 0;
}

void func1()
{
    func2();
    //...
}

void func2()
{
    func3();
    //...
}

void func3()
{
    throw 5;
    //...
}
```

Yukarıdaki programda *func3* işlevi içinde *throw* anahtar sözcüğüyle bir hata nesnesi gönderildiğinde, programın akışı bu noktadan doğrudan *main* işlevi içindeki *catch* bloğuna yönlendirilir.

İşlev çağrılarında çağrılan işlevin kodunun yürütülmesi sonunda, çağrılan işleve geri dönülmesi normal olarak yığına(*stack*) yazılan bilgilerle gerçekleşir. Programın çalışması sırasında çağıran işlevin komutu yığın alanına yazılır. Çağrılan işlevin kodunun yürütülmesi sona erdiğinde, yani *return* deyimi ile karşılaşıldığında program yığın alanından programın akışının süreceği noktanın adresini alarak programın akışını bu noktaya yönlendirir. Çağıran işlevin çağrılan işleve göndermiş olduğu çağrılar da yığına yerleştirilir. Çağıran işlevin kendisi de yerel değişkenler tanımlanmış ise bunlar için de yığın alanında yer ayrılır. Eğer çağrılan işlevin kendisi de bir işlevi çağırmışsa, yine geri dönüş noktasına ilişkin adres yığına yazılır. Bir işlevin yürütülmesi sona erdiğinde, yığından geri dönüş adresi çekilerek programın akışı o noktaya yönlendirilir. Yığındaki yerel değişkenlerin de ömrü sona erince bu değerler de yığından çıkartılır. Yığında yer alan otomatik ömürlü nesne eğer bir sınıf nesnesi ise bu sınıf nesnesinin sonlandırıcı işlevi çağrılır.

Bir işlevin kodunun yürütülmesi *return* deyimi ile değil de bir hata nesnesi gönderilmesi nedeniyle sonlandırılırsa ne olur? Bu durumda yine sırasıyla yığındaki değerler yığından çıkarılır. Ancak programın akışı bu durumda geri dönüş adresine yönlendirilmez.

Programın akışının devam edeceği noktanın belirlenmesi için bir *try* bloğu içinde yer alan geri dönüş adresi araştırılır. Böyle bir geri dönüş adresi bulunduğunda programın akışı geri dönüş noktasını izleyen ilk deyime değil, söz konusu *try* bloğunu izleyen *catch* bloklarına yönlendirilir. Bu işleme "yığının dengelenmesi" (*stack unwinding)* denir.

Yığın dengelenmesi mekanizmasında, işlevden yapılan normal geri dönüşlerde olduğu gibi yığındaki sınıf nesneleri için sonlandırıcı işlevler çağrılır. Bir işlevin geri dönüş mekanizması ile yalnızca o işlev tarafından yığına yerleştiren nesneleri ele alınırken, bir hata nesnesi gönderilerek yığının dengelenmesi durumunda ilgili *catch* bloğuna ulaşılıncaya kadar yığında yer alan tüm nesneler ele alınır. Bu nesnelerin bir sınıf türünden olması durumunda bu sınıf nesneleri için sonlandırıcı işlevler çağrılır. Yığın dengelenmesi mekanizması olmasaydı, *throw* noktası ile *catch* noktası arasında ömürleri sürmekte olan yerel sınıf nesneleri için sonlandırıcı işlevler çağrılmazdı. Bu da bu sınıf nesnelerinin tuttuğu kaynakların (*resources*) serbest bırakılmaması ya da sisteme geri verilmemesi anlamına gelirdi.

Dinamik sınıf nesneleri yığın alanında değil *"free store"* alanında yer alır. Bir hata nesnesi yakalandığında dinamik sınıf nesneleri için sonlandırıcı işlev çağrılmaz. Bunun uygulamadaki önemi şudur: Eğer programın kullandığı kaynaklar dinamik sınıf nesnelerine bağlanmışlarsa, bir hata oluşması durumunda bu nesnelerin sonlandırıcı işlevleri çağrılmayacağı için, bu kaynaklar geri verilemez. Dinamik nesnelerin akıllı gösterici nesnelerine bağlanmasıyla bu sorun çözülebilir. Akıllı gösterici sınıflarına ileride değineceğiz.

throw İfadesinin catch Parametresine Kopyalanması

throw işlemi ile *throw* ifadesi önce derleyici tarafından oluşturulmuş bir geçici bölgeye alınır. Geçici bölgeden *catch* parametresine kopyalanır. Yani *throw* ifadesinin *catch* parametresine kopyalanması doğrudan değil, geçici bölge ile iki aşamada yapılır.

Derleyicinin oluşturduğu bu geçici bölge *throw* ifadesi ile aynı türdendir. Uygulamada *throw* ifadesi genellikle sınıf türlerine ilişkin olur. *throw* ifadesi bir sınıfa ilişkinse, geçici bölgenin devreye girmesi anlam karmaşıklığına yol açabilir. Bu karmaşıklığı engellemek için çeşitli durumları tek tek ele alalım:

İ) *throw* ifadesi bir sınıf türündendir. *catch* parametresi de aynı türden bir sınıf nesnesidir. Bu durumda önce geçici bölge için kopyalayan kurucu işlev çağrılarak geçici bölgeye atama yapılır. Sonra *catch* parametresi için kopyalayan kurucu işlevin çağrılmasıyla, geçici bölgeden *catch* parametresine kopyalama yapılır. Programın akışı *catch* bloğunu bitirdiğinde, ters sırada yani önce *catch* parametresi için sonra geçici bölge için sonlandırıcı işlev çağrılır. *throw* işleminde sınıf nesnesi kullanılmış ise yığın dengelemesi sırasında geçici bölgeye kopyalama işleminden sonra bu nesne için sonlandırıcı işlev çağrılır. Bu biçimdeki bir hata işleme mekanizması çok fazla kopyalayan kurucu işlev çağrıldığı için, genellikle tercih edilmez.

ii) *throw* ifadesi bir sınıf türündendir. *catch* parametresi aynı sınıf türünden bir referans olur. Bu durumda geçici bölge için kopyalayan kurucu işlev çağrılır. Fakat daha sonra geçici bölgenin adresi *catch* parametresi olan referansa aktarılır. Geçici nesne *catch* bloğu boyunca yaşar. *catch* bloğunun sonunda geçici bölge için sonlandırıcı işlev çağrılır. Bu sık kullanılan bir tekniktir. Örneğin C++'ın standart kütüphanesindeki hata nesnesi yakalama tekniği böyledir.

iii) *throw* işlemi bir sınıf türünden adres ile yapılır. *catch* parametresi de aynı sınıf türünden bir gösterici değişken olur. Bu durumda geçici bölge de aynı sınıf türünden bir gösterici olacağına göre, ne geçici bölge için ne de *catch* parametresi için herhangi bir kurucu işlev çağrılmaz. Tabii *throw* ifadesindeki adres yerel bir nesneye ilişkin olamamalıdır. Bu teknikte genellikle programcı *new* işleci ile dinamik olarak elde ettiği hata nesnesini gönderir. Örneğin:

```
throw new Sample();
```

throw işlemi yapılmış olsa bile dinamik nesneler yaşamaya devam eder. Böyle bir *throw* işlemi ile geçici bölgeye dinamik nesnenin adresi aktarılır. Bu adres buradan da *catch* parametresine aktarılır:

```
throw new Sample();
```

gibi bir ifade için

```
catch(Sample *p)
{
    //...
    delete p;
}
```

işlemi yapılmalıdır. Bu yöntem geçici bölge ve *catch* parametresi için kopyalayan kurucu işlev çağrılmadığı için, en verimli yöntemdir. Ancak bu yöntemde *throw* işleminde yaratılan geçici nesnenin boşaltılması *catch* bloğunun sonunda programcı tarafından yapılmalıdır. Örneğin *MFC* kütüphanesi bu yöntemi kullanmaktadır. Oluşturulan hata nesnelerinin yaratılmasını, *catch* parametresine kopyalanmasını ve yok edilmesini izlemek için aşağıdaki kod parçasını derleyerek çalıştırın:

```
#include <iostream>

using namespace std;
```

```
class Sample {
public:

    Sample(int a):m_a(a){cout << "Sample::Sample(int) " << endl;}

    Sample(const Sample &r){m_a = r.m_a; cout << "Sample::Sample(const
sample &) " << endl;}

    ~Sample(){cout << "Sample::~Sample() " << endl;}
    void display()const {cout << "a = " << m_a << endl;}

private:

    int m_a;

};


void func1()

{

    Sample a(1);
    throw a;

}


void func2()

{

    throw new Sample(2);

}


int main()

{

    try {

        //func1();
        func2();

    }

    catch(Sample a) {
        a.display();

    }

    catch(Sample *ptr) {
        ptr->display();
        delete ptr;

    }
```

```
    cout << "end" << endl;


    return 0;
}
```

Türemiş sınıf nesnesinin türü ile ilgili taban sınıf nesnesinin türü arasında tam uyum olduğu kabul edilir. Yani türemiş sınıf nesnesinin adresiyle *throw* işlemi yapıldığında, taban sınıf türünden bir göstericiye sahip *catch* bloğu bunu yakalayabilir. Uygulamaların çoğunda sınıf kütüphanesinde bir türetme hiyerarşisi içinde hata işleme sınıfları bulunur. *MFC*'de kütüphane içerisinde normal dışı bir durumla karşılaşıldığında, ilgili hata sınıfı türünden bir nesne dinamik olarak yaratılır ve dinamik hata nesnesinin adresiyle *throw* işlemi yapılır. Programcı kütüphanedeki bütün *throw* işlemlerinin bir hata sınıfı nesnesinin adresiyle yapıldığını bilir. Dolayısıyla oluşabilecek her türlü hata durumunu aşağıdaki gibi ele alabilir:

```
try {
    //...
}
catch (CException *pcException)
{
    //...
    delete pCException;
}
```

Hata sınıfları da sanal işlevlere sahip olabilir. Böylece çokbiçimliliği destekleyecek bir şekilde tasarım yapılabilir. Taban sınıfın sanal işlevlere sahip olması çok biçimli bir hata ele alma işlemini sağlayabilir. Örneğin, yukarıdaki *catch* bloğunda *CException* sınıfının sanal *display()* isimli bir işlevi çağrılabilir. Böylece oluşan hata durumuna ilişkin hata iletisi ekrana ya da bir dosyaya yazdırılabilir.

Bir sınıf kütüphanesi kullanılarak proje geliştiriliyorsa, programcı var olan hata sınıflarından türetme yaparak kendi hata sınıflarını oluşturmalıdır. Örneğin *MFC* de bir seri port uygulaması geliştirildiğini düşünelim. Bu durumda *CException* sınıfından *CSerialException* gibi bir sınıf türetilerek, aşağıdaki gibi bir *throw* işlemi yapılabilir:

```
try {
    throw new CSerialException;
    //...
}
```

```
catch (CException *pCException) {
    pCException->Disp();

    delete pCException;

}
```

CException sınıfından türetilmiş *CMemoryException* gibi bir sınıf olduğunu düşünelim. Aşağıdaki gibi *try-catch* blokları oluşturulmuş olsun.

```
try {

    //...

}


catch (CMemoryException *pException) {

    //...

}


catch (CException *pException) {

    //...

}
```

Burada *CMemoryException* sınıfından yapılan *throw* işlemleri birinci *catch* bloğu tarafından, diğerleri ikinci *catch* bloğu tarafından yakalanır. Burada *catch* bloklarının yerleşimi önemlidir. Yani *CException* sınıfına ilişkin *catch* bloğunun aşağıda olması gerekir.

içiçe try blokları

Bir *try* bloğu başka bir *try* bloğunun içinde yuvalanmış olabilir *(nested try blocks).* Bu durumda her bir *try* bloğunu ayrı *catch* blokları izlemelidir.

Aşağıdaki örneği inceleyelim :

```
#include <iostream>

void func1()

{

    throw 1;

}
```

```
void func2()
{
    throw 2.3;
}

int main()
{
    try {
        func1();
        try {
            func2();
        }
        catch (double x) {
            std::cout << "hata yakalandi : " << x << std::endl;
        }
    }
    catch (int x) {
        std::cout << "hata yakalandi : " << x << std::endl;
    }

    std::cout << "hata yakalandiktan sonra program devam ediyor!" <<
std::endl;

    return 0;
}
```

Yukarıdaki örnekte dıştaki *try* bloğunun içinde başka bir *try* bloğu daha yer alıyor. İçteki *try* bloğundaki kodun çalışması sırasında bir hata nesnesi gönderilmesi durumunda, yani *func2* işlevi tarafından bir hata nesnesinin gönderilmesi durumunda, gönderilen hata nesnesi önce içteki *try* bloğunun *catch* bloğu tarafından yakalanmaya çalışılır. Eğer içteki *try* bloğuna ilişkin *catch* bloğu gönderilen hata nesnesini yakalayamaz ise, bu kez gönderilen hata nesnesini dıştaki *try* bloğuna ilişkin *catch* bloğu yakalamaya çalışır.

Şüphesiz, yalnızca dıştaki *try* bloğu içinde yer alan kod parçasının çalışması durumunda, eğer bir hata nesnesi gönderilirse, yani *func1* işlevi içinden bir hata nesnesi gönderilirse, bu hata nesnesini yalnızca dıştaki *catch* bloğu yakalamaya çalışır.

Farklı Derinliklerdeki try catch Blokları

try ve *catch* blokları farklı derinliklerde yer alabilir. Bir *try* bloğu içinde bir hata nesnesi gönderildiğinde, bu hata nesnesini yakalama olasılığı bulunan *catch* bloklarının araştırılması aşağıdan yukarıya doğru yapılır. Yani ilk önce *throw* deyimine en yakın durumda olan ve *throw* deyimini kapsayan *try* bloklarının *catch* bloklarına bakılır. Hata buradaki *catch* blokları tarafından yakalanamamamışsa, bu kez sırasıyla daha yukarıdaki *catch* blokları araştırılır. Aşağıdaki örneği dikkatli şekilde inceleyiniz, programı derleyerek çalıştırın:

```
#include <iostream>
#include <cstdlib>


void func1();
void func2();
void func3();


int main()

{

    try {

        func1();

    }

    catch (double) {

        std::cout << "hata main islevi icinde yakalandi!" << std::endl;
        exit(EXIT_FAILURE);

    }

    return 0;

}


void func1()

{

    try {

        func2();

    }

    catch (long) {

        std::cout << "hata func1 islevi icinde yakalandi!" << std::endl;
        exit(EXIT_FAILURE);

    }

}
```

```
void func2()

{

    try {

        func3();

    }

    catch (int) {

        std::cout << "hata func2 islevi icinde yakalandi!" << std::endl;
        exit(EXIT_FAILURE);

    }

}


void func3()

{

    try {

        //throw 'A';

        //throw 1;

        //throw 1L;
        throw 1.5;

    }

    catch (char) {

        std::cout << "hata func3 islevi icinde yakalandi!" << std::endl;
        exit(EXIT_FAILURE);

    }

}
```

func3 işlevi içinde gönderilen hata nesnesinin türünü değiştirerek programı yeniden derleyip çalıştırın. Programın çalışması sırasında oluşan farklılığı gözleyin.

İşlevi Kapsayan try Bloğu

İstenirse tanımlanan bir işlevin tüm ana bloğu bir *try* bloğu görevi yapabilir. Bu durumda işlev parametre ayracının kapatılmasından sonra, ancak işlevin ana bloğunun açılmasından önce *try* anahtar sözcüğü kullanılır. Aşağıdaki örneği inceleyin:

```
void func(int x)
try
```

```
{
    //...
}

catch (int ) {
    //...
}

catch (double) {
    //...
}
```

İşlevi kapsayan *try* bloğuna ilişkin *catch* bloğu akış bakımından işlevin içinde kabul edilir. Yukarıdaki sözdizim ile, eğer *func* işlevi içinde herhangi türden bir hata nesnesi gönderilirse programın akışı işlevin ana bloğunu izleyen *catch* bloklarına sıçrar. *try* bloğunun başının aynı zamanda *func* işlevin ana bloğunun başlangıcı ve *try* bloğunun sonunun aynı zamanda işlevin ana bloğunun sonu olduğuna dikkat edelim. İstenirse *main* işlevi de bu şekilde bir *try* bloğu içine alınabilir:

```
int main()
try
{
    //...
}
catch (int ) {
    //...
}

catch (double) {
    //...
}
```

Hata Nesnesi Belirlemeleri

Bir işlev içinde bir hata nesnesinin gönderilip gönderilmeyeceği, gönderilecekse gönderilecek hata nesnesinin hangi türlerden olabileceği özel bir sözdizim ile işlevin bildiriminde ve tanımında belirtilebilir. Buna "hata nesnesi belirlemesi" (*exception specification*) denir.

İşlev bildiriminde parametre ayracının kapanmasından sonra *throw* anahtar sözcüğü yazılır. Bu sözcüğü izleyen ayracın içine işlevin gönderebileceği hata nesnesinin/nesnelerinin türü/türleri yazılabilir:

```
double func()throw (Exception);
```

Yukarıdaki bildirimde, geri dönüş değeri *double* türden olan, parametre değişkenine sahip olmayan *func* işlevinin *Exception* sınıfı türünden bir hata nesnesi gönderilebileceği bildiriliyor.

Böyle bir bildirimle iki işlev sağlanır.

1. İşlevin *Exception* sınıfı türünden bir hata nesnesi gönderebileceği bilgisi verilir. Böylece kaynak kodun okunabilirliği artar. İşlevi çağıran programcı, işlev çağrı kodunu bir *try* bloğu içine alarak, işlev içinde bir hata nesnesinin gönderilmesi durumunda, bu hata nesnesini yaklamak ve hatayı işlemek için bir önlem alabilir.

2. Derleyiciye işlevin ne türden bir hata nesnesi gönderebileceği bildirilmiş olur. Böylece derleyicinin ürettiği kod sonucunda, programın çalışma zamanında eğer başka bir türden hata nesnesi gönderilirse, standart bir işlev çağrılarak programın akışı sonlandırılır.

Bir işlevin içinden birden fazla türden farklı hata nesneleri de gönderilebileceği için bu durumun da işlev bildiriminde belirtilmesine olanak sağlanmıştır. Bu durumda işlev bildirimindeki *throw* ayracının içine virgüllerle ayrılmış biçimde gönderilebilecek hata nesnelerinin türleri yazılır:

```
double func()throw (bad_alloc, MathException);
```

Yukarıdaki işlev bildiriminden *func* işlevinin *bad_alloc* ya da *MathException sınıfı*

türlerinden bir hata nesnesi gönderebileceği belirtilmektedir.

```
throw (bad_alloc, MathException)
```

bildiriminin işlev tanımında da yazılması zorunludur. Yani birinde bulunup diğerinde bulunmaması geçersizdir. *func* işlevi aşağıdaki gibi tanımlanmalıdır:

```
double func(double x) throw (bad_alloc, MathException)
{
    //...
}
```

Yine okunabilirliği artırma amacıyla, bir işlevin herhangi bir hata nesnesi göndermediği bilgisi de işlev bildiriminde verilebilir. Bu durumda işlev bildiriminde yer alan *throw* anahtar sözcüğünü içi boş bir ayraç izler.

```
int foo(int) throw();
```

Yukarıdaki *foo* işlevinin bildiriminde, *foo* işlevinin herhangi bir hata nesnesi göndermediği belirtiliyor.

Bir işlevin bildiriminde *throw* anahtar sözcüğünün yer almaması, işlevin herhangi bir hata nesnesi göndermediğine işaret etmez. Derleyici tarafından bu durum, işlevin bir hata nesnesi gönderip göndermediği ya da hangi türden bir hata nesnesi gönderebileceği konusunda bir bilgi verilmediği biçiminde yorumlanır.

```
int process(int);
```

Yukarıdaki işlev bildirimine göre *process* işlevinin bir hata nesnesi gönderip göndermeyeceği konusunda bir bilgi verilmiyor. *process* işlevi herhangi bir türden hata nesnesi gönderebileceği gibi bir hata nesnesi göndermeyedebilir.

Hata nesnesi belirlemesi, bir işlevin imzasının bir parçası değildir. Parametrik yapıları aynı hata belirlemeleri farklı aynı isimli iki işlev aynı bilinirlik alanında yer alamaz:

```
void func(int) throw(int);
void func(int) throw(double);  //func işlevinin yeniden bildiriminde hata
```

Hata belirlemesi, işlevin türünün bir parçası olmadığından *typedef* bildiriminde de yer alamaz. Aşağıdaki *typedef* bildirimi derleme zamanında hata oluşumuna neden olur:

```
typedef void (*PVPTR)(int) throw(MathException);     //Geçersiz!
```

Beklenmeyen Hata

Bir hata nesnesi göndermediği belirtilen bir işlev içinden *throw* ifadesiyle bir hata nesnesi gönderilmesi, ya da belirli türden bir hata nesnesi gönderdiği belirtilen bir işlevin başka türden bir hata nesnesi göndermesi derleme zamanında kontrol edilmez . Bu durummda derleme zamanında bir hata oluşmaz. Bu kontrol programın çalışma zamanında yapılır.

Bir işlevin bildiriminde *throw* anahtar sözcüğü kullanılmışsa, yani gönderebileceği hata nesnesi ya da nesnelerinin türleri hakkında bilgi verilmişse o işlevin belirtilmeyen bir türden hata nesnesi göndermesi durumuna "beklenmeyen hata" (*unexpected exception*) denir.

Bir işlevin içinde beklenmeyen bir hata durumu oluştuğunda, yani işlev bildiriminde belirtilmeyen bir türden hata nesnesi gönderildiğinde, *unexpected* isimli standart bir işlev çağrılır. *unexpected* işlevinin önceden belirlenmiş davranışı standart *terminate* işlevini çağırmaktır. *terminate* işlevinin de önceden belirlenmiş davranışının, *abort* işlevini çağırmak olduğu daha önce açıklanmıştı.

Nasıl *terminate* işlevinin önceden belirlenmiş davranışını değiştirmeye yönelik *set_terminate()* işlevi var ise, *unexpected* işlevinin de önceden belirlenmiş davranışını değiştirebilmek için *set_unexpected* işlevi tanımlanmıştır:

Bu işlev *<exception>* başlık dosyası içinde aşağıdaki gibi bildirilmiştir:

```
typedef void(*terminate_handler)();

terminate_handler set_unexpected(terminate_handler fp) throw();
void unexpected()
```

set_unexpected işlevinin bildiriminden aşağıdaki bilgiler çıkarılabilir:

terminate_handler parametre değişkeni olmayan, geri dönüş değeri üretmeyen bir işlevi gösteren gösterici türünün yeni tür ismidir.

set_unexpected işlevi *terminate_handler* türünden bir geri dönüş değerine ve

terminate_handler türünden bir parametre değişkenine sahiptir.

set_unexpected işlevinin geri dönüş değeri, geri dönüş değeri üretmeyen ve parametre değişkeni olmayan bir işlev türünden adrestir. *set_unexpected* işlevinin parametre değişkeni geri dönüş değeri üretmeyen ve parametre değişkeni olmayan bir işlevi gösteren göstericidir. Bu durumda *set_unexpected* işlevine böyle bir işlevin adresi geçilmelidir. İşlev isimlerinin birer adres olduğunu hatırlayın:

```
void func();

//...
set_unexpected(func);
```

set_unexpected işlevine adresi gönderilen işlev aşağıda tanımlanan işlevlerden birini yapmalıdır:

terminate işlevini çağırarak programı sonlandırmak (zaten bu önceden belirlenmiş (default) davranıştır.

abort işlevini çağırmak.

exit işlevini çağırmak.

Bir hata nesnesi göndermek

Bir işlevin tanımı içinde hata belirlemesiyle bildirilen sınıfların dışında bir hata nesnesi gönderildiğinde, ilgili işlev bu hatayı kendi içinde ele alabilir. Bu durumda *unexpected* işlevi çağrılmaz. Aşağıdaki örneği inceleyin:

```
void func(int val) throw(Exception)
{
    try {
        //...
        throw string("Hata!");
    }
    catch (string){
        //...
    }
    //...
}
```

Yukarıdaki örnekte *func* işlevinin *Exception* sınıfı türünden bir hata nesnesi gönderilebileceği bildiriliyor. Ancak işlev içinde, standart *string* sınıfı türünden bir hata nesnesi gönderiliyor. Gönderilen hata nesnesinin, işlev içinde yer alan *catch* bloğu tarafından yakalandığını görüyorsunuz. Bu durumda *unexpected* işlevi çağrılmaz.

Hata belirlemelerine ilişkin kontrol derleme zamanında değil, programın çalışma zamanında yapılır. Aşağıdaki kodu inceleyin:

```
void func(double)throw(string, Exception);
void process (int val) throw (string)
{
    //...
    func(7.5);
```

```
    //...
}
```

Yukarıda *func* isimli işlevin bildiriminde, işlevin *string* ya da *Exception* sınıfı türünden bir hata nesnesi gönderebileceği bildiriliyor. Aşağıda tanımlanan *process* işlevinin tanımında ise, *string* sınıfı türünden bir hata nesnesi gönderilebileceği bildiriliyor. Oysa *process* işlevi içinde, *Exception* sınıfı türünden bir hata nesnesi gönderme olasılığı bulunan *func* işlevi çağrılıyor. Bu durumda derleme zamanında bir hata oluşmaz. Ancak programın çalışma zamanı sırasında *process* işlevinin kodu çalıştığında, *func* işlevi çağrıldığında gerçekten *Exception* sınıfı türünden bir hata nesnesi gönderilirse, bu durumda *unexpected* işlevi çağrılır.

Bu durumun sağlanması için *process* işlevin tanımının aşağıdaki gibi yapıldığını düşünebelirsiniz.

```
void process (int val) throw (string)
{
    try {
        //işlevin kodu
    }
    catch (string &) {
        throw;
    }
    catch (...) {
        unexpected();
    }
}
```

Bu durumda

```
void foo(void) throw()
{
    //...
}
```

gibi bir işlev tanımının da gerçekte derleyici tarafından aşağıdaki gibi ele alındığı düşünülebilir, değil mi?

```
void foo(void) throw()
{
```

```
    try {
    //...
    }
    catch(...) {
        std::unexpected();
    }
}
```

bad_exception Sınıfı Türünden Hata Belirlemesi

Bir işlevin bildiriminde yer alan hata belirlemesinde standart *bad_exception* sınıfı kullanılabilir. Bunun anlamı şudur: Eğer söz konusu işlev, hata belirlemesinde belirtilen sınıfların dışında bir türden hata nesnesi gönderirse, yani beklenmeyen hata nesnesi durumu oluşursa bu durumda *unexpected* işlevi çağrılmaz. *bad_exception* sınıfı türünden bir hata nesnesi gönderilir.

Taban Sınıf İle Türemiş Sınıf Arasındaki Hata Belirlemesi Uyumu

C++ dilinin kurallarına göre türemiş sınıfın hata belirlemeleriyle taban sınıfın hata belirlemeleri arasında bir uyum olmak zorundadır. Taban sınıfın sanal işlevini ezen, bir türemiş sınıf işlevi, taban sınıfın gönderdiği hata nesneleri türlerinin dışında bir hata göndermemelidir. Aşağıdaki örneği inceleyin:

```
class Exception{};


class MathException:public Exception {};

class SpecialException {};


class Base {
public:

    virtual void func1() throw (Exception);
    virtual void func2() throw (Exception);
    virtual void func3() throw (MathException);
    virtual void func4() throw (MathException);
    virtual void func5() throw (Exception);
    virtual void func6() throw ();

};


class Der: public Base {

    void func1() throw(MathException);                //Geçerli
```

```
    // void func2() throw(SpecialException);          Geçersiz!
    void func3() throw(MathException);                //Geçerli

    //void func4() throw(Exception);                  Geçersiz!

    //void func5() throw(Exception, SpecialException );  Geçersiz!
    virtual void func6(); //Geçersiz!

};
```

Der sınıfı *Base* sınıfından türetiliyor. *Der* sınıfının, *Base* sınıfının sanal işlevlerini ezen işlevlerinin hata belirlemelerinde, *Base* sınıfının göndermediği bir hata türü kullanılamaz. *Der* sınıfının *func2*, *func4* ve *func5* işlevlerinin bildirimi geçersizdir.

İşlev Göstericileri ve Hata Belirlemeleri

Bir işlev göstericisinin tanımında, göstericinin göstereceği işlevin hata belirlemeleri de belirtilebilir:

```
void (*fp)()throw (Myclass);
```

Yukarıdaki bildirimden şu anlaşılır:

fp geri dönüş değeri üretmeyen, parametre değişkeni olmayan ve *Myclass* sınıfı türünden bir hata nesnesi gönderme olasılığı olan bir işlevi gösterecektir. Böyle bir işlev gösterici değişkeni, parametrik yapısı aynı ama, başka sınıf türünden hata nesnesi gönderme olasılığı olan bir işlevi, gösteremez:

```
void func() throw (Exception);

void (*fp) ()throw (Myclass) = &func;     //Geçersiz!
```

Hata İşleyen Kodun Yeniden Hata Nesnesi Göndermesi

Hata nesnesini yakalayan ve işleyen bir kod yakalamış olduğu hata nesnesini yeniden gönderebilir. Yeniden gönderilen hata nesnesi daha yukarı bir seviyede kapsayan bir *try* bloğu tarafından tutulmaya çalışılır. Böylece hata yakalayan bir *catch* bloğu yakaladığı bir hatayı kısmen ele alabilir. Yapması gerekenleri yaparak, daha sonra hatanın daha yukarı seviyedeki *catch* blokları tarafından yakalanabilmesini sağlamak için, hata nesnesini yeniden gönderebilir.

catch bloğunun içinde, *throw* anahtar sözcüğü yanında bir ifade olmaksızın, yalın olarak kullanılırsa bu işleme "hata nesnesinin yeniden gönderilmesi" *(rethrow)* denir.

```
try {

    func();

}

catch (...) {

    //...
    throw;

}
```

Yeniden *throw* işleminde, hata nesnesi bir dıştaki (kapsayan) *catch* bloğuna gönderilir. Bu işlemde geçici bölge silinmez. Yani ilk *throw* işlemindeki ifade geçici bölgede kalır. Bir başka deyişle gönderilen aynı hata nesnesidir. Aşağıdaki kodu inceleyiniz:

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


enum {SUCCESS, FAILURE};


class File

{

public:

    File (const char *str) {}
    bool is_valid() const {

        //...

        return false;

    }

    int open_new_file() const {return FAILURE;}

};


class Exception {

    //...

};


class FileException: public Exception

{
```

```
public:

    FileException(const char *p) : s(p) {}

    const char * get_message()const {return s.c_str();}
private:

    std::string s;

};


void foo(File &);

using namespace std;


int main()

{

    try {

        File f ("letter.txt");
        foo(f);   // 1

    }

    catch (...) {

        cout << "yeniden gonderilen hata nesnesi yakalandi" << endl;

    }



    return 0;

}

void foo(File &r)

{

    try {

        if (!r.is_valid())

            throw FileException("letter.txt");

    }



    catch(FileException &r_ex) {

        cout << "gecersiz dosya :" << r_ex.get_message() << endl;
        if (r.open_new_file() == FAILURE)

            throw;

    }
```

```
}
```

foo işlevinde yer alan *try* bloğu içinde, *is_valid* işlevi ile dosyanın geçerliliği sınanıyor. Dosya geçersiz ise *FileException* sınıfı türünden bir hata nesnesi gönderiliyor. *try* bloğunu izleyen *catch* bloğu ile gönderilen hata nesnesi yakalanıyor. Ekrana bir hata iletisi yazdırılıyor. Daha sonra yeni bir dosya açılmaya çalışılıyor. Eğer yeni dosya açılamamış ise hata nesnesi yeniden gönderiliyor. *main* işlevi içinde yapılan *foo* işlevi çağrısının bir *try* bloğu içine alındığını bu *try* bloğunu ise bir *catch* bloğunun izlediğini görüyorsunuz.

Yeniden gönderilen hata nesnesi bu kez *catch all* bloğu tarafından yakalanır. Programın ekran çıktısı aşağıdaki gibi olur:

```
gecersiz dosya :letter.txt
yeni dosya acilamiyor!

tekrar gonderilen hata nesnesi yakalandi
```

Çoğu durumda hata nesnesini yakalayarak kısmi olarak işleyen kod parçası, hata nesnesi üzerinde değişiklik de yapar. Böylece yeniden gönderilen hata nesnesini yakalayan kod parçası, hatanın kısmen ele alındığından haberdar olur. İşte *catch* parametrelerinin bir sınıf türünden olması yerine bir sınıf türünden referans olarak seçilmesinin bir nedeni de budur. *catch* parametresi bir sınıf türünden olursa, hata nesnesi üzerinde yapılan değişiklikler yalnızca yerel bir kopya üzerinde yapılır:

Aşağıdaki gibi bir *catch* bloğunu ele alalım:

```
catch (Exception e)

{
    e.set_value(/**/); //catch parametresi üzerinde değişiklik yapılıyor...
    throw;             //Hata nesnesi yeniden gönderiliyor.

}
```

Yukarıdaki *catch* bloğuyla, gönderilen hata nesnesinin kendisi yakalanmaz. Gönderilen hata nesnesi *catch* parametresi olan *e* nesnesine ilkdeğer verir. Dolayısıyla

```
e.set_value(/**/);
```

çağrısıyla değiştirilen, asıl hata nesnesi değil *catch* parametresidir. Yeniden *throw* deyimiyle gönderilen hata nesnesi üzerinde bir değişiklik yapılmamıştır. Ancak *catch* parametresi referans olsaydı, durum değişirdi:

```
catch (Exception &r)

{

    r.set_value(/**/);      //hata nesnesi üzerinde değişiklik yapılıyor...
    throw;                  //hata nesnesi yeniden gönderiliyor.

}
```

Yukarıdaki *catch* bloğu hata nesnesinin kendisini yakalar.

```
r.set_value(/**/);
```

çağrısıyla değiştirilen hata nesnesinin kendisidir. Bu durumda yeniden gönderilen hata nesnesi, üzerinde değişiklik yapılmış olan hata nesnesidir.

Kurucu İşlevlerden Hata Nesnesi Gönderilmesi

Kurucu işlevler için geri dönüş değeri diye bir kavramın söz konusu olmadığını biliyorsunuz. Kurucu işlevler içinde oluşan normal dışı durumların, geri dönüş değeri ile dışarıya iletilmesi mümkün değildir. Kurucu işlev içinden bir hata nesnesi gönderilmesi sık karşılaşılan bir durumdur.

Bir kurucu işlev içinden gönderilen hata nesnesi yakalandığında, kurucu işlevi çağrılmış olan nesne için, yani **this* nesnesi için sonlandırıcı işlev çağrılmaz. Zira nesnenin oluşturulma süreci henüz sona ermeden bir hata nesnesi gönderilmiştir. Bir hata nesnesi yakalandığında, yalnızca yığında yer alan oluşumu tamamlanmış yerel nesneler için sonlandırıcı işlevler çağrılır.

Peki ya içinde *throw* işlemi yapılan kurucu işlev dinamik bir nesne için çağrılmışsa? Bu durumda *new* işleci ile **this* nesnesi için bellekten ayrılmış dinamik alanın *free store* alanına geri verilmesi güvence altına alınmıştır.

Bileşik Nesnelerde Yer alan Elemanların Kurucu İşlevlerinden Hata Nesnesi Gönderilmesi

Şimdi de bir sınıfın başka bir sınıf türünden elemanlara sahip olduğunu düşünelim. Elemanlardan birinin kurucu işlevi içinden bir hata nesnesi gönderildiğinde, ancak oluşumu tamamlanmış elemanlar için sonlandırıcı işlevler çağrılır. Durumu incelemek ve gözlemek için aşağıdaki kod parçasını derleyerek çalıştırın:

```
#include <iostream>


class Mem1 {
public:

    Mem1() {std::cout << "Mem1()::Mem1()" << std::endl;}

    ~Mem1() {std::cout << "Mem1()::~Mem1()" << std::endl;}

};
```

```
class Mem2 {
public:

    Mem2() {std::cout << "Mem2()::Mem2()" << std::endl; }

    ~Mem2() {std::cout << "Mem2()::~Mem2()" << std::endl;}

};


class Owner {
    Mem1 m1;
    Mem2 m2;

public:

    Owner(){std::cout << "Owner::Owner()" << std::endl; throw 1;}

    ~Owner(){std::cout << "Owner::~Owner()" << std::endl;}

};


int main()

{

    try {

        Owner x;

    }

    catch (int x) {

        std::cout << "hata yakalandi!" << "(" << x << ")" << std::endl;

    }

    return 0;

}
```

Owner sınıfının *Mem1* sınıfı türünden *m1,* ve *Mem2* sınıfı türünden *m2* isimli *private* elemanlara sahip olduğunu görüyorsunuz. Tüm sınıfların kurucu ve sonlandırıcı işlevleri, çağrıldıklarında ekrana çağrıldıklarını gösteren bir yazı yazdırıyor. *Owner* sınıfının kurucu işlevi içinde *int* türünden bir hata nesnesi gönderiliyor:

```
//…

Owner() {cout << "Owner::Owner()" << endl; throw 1;}

//…
```

main işlevi içinde *Owner* sınıfı türünden *x* isimli bir nesne yaratıldığını, tanımlama deyiminin *try* bloğu içine alındığını görüyorsunuz. *try* bloğunu izleyen *catch* bloğu ile *int* türden hata nesnesi yakalanır. Hata nesnesi yakalandığında hangi sınıf nesneleri için sonlandırıcı işlevler çağrılır? *m1* ve *m2* sınıf nesnelerinin oluşturulma süreci tamamlanmış olduğu için bu nesnelerin sonlandırıcı işlevleri çağrılır. Ancak *Owner* sınıfının sonlandırıcı işlevi çağrılmaz. Çünkü *Owner* sınıf nesnesinin oluşturulması henüz tamamlanmamıştır.

Programın ekran çıktısı aşağıdaki gibi olur:

```
Mem1()::Mem1()

Mem2()::Mem2()

Owner::Owner()
Mem2()::~Mem2()

Mem1()::~Mem1()
hata yakalandı!(1)
```

Bu kez *Owner* sınıfının kurucu işlevi içinde değil de, *Mem2* sınıfının kurucu işlevi içinden bir hata nesnesi gönderildiğini düşünelim:

```
//…

Mem2() {cout << "Mem2()::Mem2()" << endl; throw 2;}

//…
```

Gönderilen hata nesnesi yakalandığında, yalnızca *Mem1* sınıfının sonlandırıcı işlevi çağrılır. Bu durumda programın ekran çıktısı aşağıdaki gibi olur:

```
Mem1()::Mem1()

Mem2()::Mem2()

Mem1()::~Mem1()
hata yakalandi!(2)
```

Türetmede de benzer bir durum söz konusudur. Bir türemiş sınıf nesnesinin oluşturulması durumunda önce taban sınıf nesnesi oluşturulur değil mi? Türemiş sınıfın kurucu işlevi taban sınıfın kurucu işlevini çağrır. Taban sınıf nesnesinin kurucu işlevi içinden bir hata nesnesi gönderildiğinde hangi sınıfların sonlandırıcı işlevleri çağrılır?

Kurucu İşlevi Sarmalayan try Bloğu

Bir kurucu işlevin tamamı *try* bloğu ile kapsanabilir:

```
Foo::Foo()
try

{

    //...

}

catch (bad_alloc)

{

    //...

}
```

Kurucu işlevde *M.I.L.* sözdizimi kullanılması durumunda da kurucu işlevi sarmalayan bir

try bloğu oluşturulabilir:

```
Foo::Foo()

try : x(a), y(b)

{

    //...

}

catch (bad_alloc)

{

    //...

}
```

Yukarıdaki gibi bir *try* bloğunda, yalnızca kurucu işlev ana bloğu içinden gönderilecek hata nesneleri değil, aynı zamanda elemanların kurucu işlevleri içinden gönderilen hata nesneleri de yakalanabilir.

Kurucu işlevi sarmalayan *try* bloğunun uygulamada bir önemi daha vardır: Böyle bir kurucu işlev hata belirlemesiyle, *public* arayüzünde belirli bir tür dışında hata nesnesi göndermemeyi güvence altına alabilir. Kurucu işlev içinde gönderilen başka türden bir hata nesnesi, her türden hata nesnesini yakalayan bir *catch* bloğu ile yakalanarak, istenilen türden bir hata nesnesinin daha yukarıya gönderilmesi sağlanabilir. Aşağıdaki örneği inceleyin:

```
#include <string>

class Ex{
```

```
    //...
};


class Foo{
    std::string s;
public:
    Foo(const char *) throw (Ex);
};


Foo::Foo(const char *str) throw(Ex)
try: s(str)
{
    //...
}
catch(...)
{
    throw Ex();
}
```

Yukarıdaki örnekte *Foo* sınıfının kurucu işlevi hata belirlemesi ile ancak *Ex* sınıfı türünden bir hata nesnesi göndereceğini bildiriyor. *Foo* sınıfının kurucu işlevinin bir *try* bloğu ile sarmalanmış olduğunu görüyorsunuz. Kurucu işlevi sarmalayan *try* bloğunu, her türden hata nesnesini yakalayabilen bir *catch* bloğu izliyor. Bu *catch* bloğu ile bir hata nesnesinin yakalanması durumunda, *Ex* sınıfı türünden bir hata nesnesi gönderiliyor. Böylece hata belirlemesiyle verilen güvenceye uyulması sağlanıyor.

Kurucu İşlevler Tarafından Elde Edilen Kaynaklar ve Hata Nesneleri

Bir çok sınıfın tasarımında, sınıf nesnesinin kullanacağı bir kaynak, sınıfın kurucu işleviyle sınıf nesnesine bağlanır. Sınıfın sonlandırıcı işlevinin çağrılmasıyla bu kaynak geri verilir. Eğer bu kaynaklar bir göstericiye bağlanırsa, kaynağın bağlanmasından sonra, kurucu işlev içinden bir hata nesnesi gönderilmesi durumunda, bağlanan kaynakların geri verilebilmesi mümkün olmaz.

Bu kaynaklar akıllı gösterici (*smart pointer*) sınıflarından nesnelere bağlanabilir. *STL* içinde bu amaç için tasarlanmış şablon temelli *auto_ptr* isimli standart bir akıllı gösterici sınıfı vardır. "Akıllı göstericiler" başlığı altında bu konuya daha ayrıntılı bir şekilde değineceğiz.

Sonlandırıcı İşlevlerden Hata Nesnesi Gönderilmesi

Sonlandırıcı işlevlerden hata nesnesi gönderilmesi dilin kurallarına göre geçerli olmasına karşın, genellikle önerilmez.

Bir sonlandırıcı işlev üçayrı nedenden dolayı çağrılmış olabilir:

1. Bir sınıf nesnesinin bilinirlik alanı sona ermiştir. Sınıf nesnesinin ömrü sona erdiğinden, sınıfın sonlandırıcı işlevi çağrılır.
2. Dinamik bir sınıf nesnesi *delete* işleci kullanılarak *free store*'a geri verilmek istendiğinde dinamik sınıf nesnesi için de sonlandırıcı işlev çağrılır.
3. Gönderilen bir hata nesnesi bir *catch* bloğu tarafından yakalandığında, yığın dengelenmesi süreci içinde, ilgili *catch* bloğuna ulaşılmasına kadar yığında yer alan tüm yerel sınıf nesneleri için sonlandırıcı işlev çağrılır.

Eğer bir sonlandırıcı işlev yığının dengelenmesi sırasında çağrılmışsa, sonlandırıcı işlev içinden bir hata nesnesi gönderilmesi durumunda programın akışının işlevin dışına çıkmasına izin verilmez. Bu durumda otomatik olarak *std::terminate* işlevinin çağrılması güvence altına alınmıştır.

Sonlandırıcı işlev içinde, hata nesnesi gönderme olasılığı olan bir işlevin çağrılmış olması son derece normal bir durumdur. Ancak çağrılacak işlevler içinden gönderilmesi olasılığı olan hata nesneleri, yine sonlandırıcı işlev içinde yer alan *catch* bloklarıyla yakalanmalı, hatanın daha yukarıya gönderilmesine izin verilmemelidir:

```
Myclass::~Myclass()

{

    //…
    try {

    if (x < 0)

        throw 1;




}

//…

}

}
catch
(int)
{

   /
   /
   …
```

Yukarıdaki örnekte *Myclass* isimli sınıfın sonlandırıcı işlevi içinde gönderilen *int* türden hata nesnesi, yine aynı işlev içinde yakalanıyor.

uncaught_exception İşlevi

Bir başka olanak ise, sonlandırıcı işlev içinde bir hata nesnesi göndermeden önce, yani bir *throw* işlemi yapmadan önce, bir hatanın işlenmekte olup olmadığının sınanmasıdır. Bir hata nesnesinin ilgili bir *catch* bloğu tarafından tutulmasıyla yani bir *catch* bloğuna girilmesiyle, işlenmesinin bitmiş olduğu kabul edilir.

Standart <*stdexcept*> başlık dosyası içinde bildirilen *uncaught_exception* işleviyle bir hatanın işlenmekte olup olmadığı sınanabilir:

```
bool uncaught_exception();
```

İşlev *true* değere geri dönerse bir hata durumu işleniyor demektir. Yani bir hata nesnesi gönderilmiş ancak henüz *catch* bloğuna girilmemiştir. Eğer sonlandırıcı işlevin kodu içinde *uncaught_exception* işlevi çağrılmış ve *true* değere dönmüşse, sonlandırıcı işlevin yığın dengelenmesi sırasında yerel bir sınıf nesnesinin yok edilmesi yüzünden çağrıldığı anlamı çıkar:

```
Sample::~Sample()
{
    //…
    if (x < 0) {
        if (uncaught_exception())
            return;
        throw 1;
    }
return;
}
```

Yukarıdaki kod parçasında *Sample* sınıfının sonlandırıcı işlevi içinde, *x* değişkeninin değerinin *0*'dan küçük olması durumunda önce *uncaught_exception* işlevi çağrılıyor. Ancak işlevin *false* değerine geri dönmesi durumunda bir hata nesnesi gönderiliyor.

Global Sınıf Nesnelerinin Kurucu ve Sonlandırıcı İşlevlerinden Hata Nesnesi Gönderilmesi

Global sınıf nesnelerinin kurucu işlevleri *main* işlevin çalışmaya başlamasından önce çağrıldığından, bu nesnelerin kurucu işlevleri içinden gönderilen hata nesnelerinin yakalanma şansı yoktur. Global nesnelerin sonlandırıcı işlevlerinden gönderilen hata nesnelerinin de yakalanma şansı yoktur. Çünkü global nesnelerin sonlandırıcı işlevleri de *main* işlevinden sonra çağrılır.

STL deki Hata Sınıfları

STL deki sınıflar şablon tabanlı olmasına karşın hata sınıfları şablon tabanlı değildir. Hata sınıfları *<exception>* ve <*stdexcept*> başlık dosyaları içinde bildirilmiştir. Standart kütüphanedeki hata sınıfları da bir türetme hiyerarşisi içermektedir. Türetme hiyerarşisinin tepesinde "*exception"* isimli bir sınıf bulunur. *exception* sınıfı standart *exception* başlık dosyası içinde bildirilmiştir:

```
class exception {
public:

    exception() throw();

    exception(const exception& right) throw ();
    exception& operator=(const exception& right) throw();
    virtual ~ exception() throw();

    virtual const char *what() const throw();

};
```

exception sınıfından türetilen bütün standart hata sınıfları, *what* isimli sanal işlemi ezer. Bu işlev, hatayı tanımlayan bir yazının başlangıç adresine dönmektedir.

STL'de yer alan standart sınıflardan bazılarına ilişkin işlevler bir hata durumuyla karşılaştıklarında, *exception* sınıfından türetilmiş sınıf türlerinden hata nesnesi gönderir. *STL* kütüphanesinden gönderilen tüm hata nesneleri *exception* sınıfından türetilmiş sınıflardan oldukları için, bu hata nesnelerinin tümü parametresi *exception* sınıfı türünden referans olan bir *catch* bloğu tarafından yakalanabilir.

standart *exception* sınıfından yine standart *logic_error* ve *runtime_error* isimli sınıflar türetilmiştir.

Bu sınıfların tanımı <*stdexcept*> isimli başlık dosyasında yapılmıştır. *logic_error* sınıfı mantıksal tabanlı hataların ele alınması için düşünülmüştür. *run_time_error* sınıfı ise programın çalışma zamanında ortaya çıkabilecek hatalar için düşünülmüştür. Programcı kendi hata işleme sınıflarını taban sınıf olan *exception* sınıfı yerine bu sınıflardan da türetebilir:

```
class MyException {
public:

    MyException(const char *pMessage):logic_error(pMessage){}
```

```
};
```

logic_error sınıfından da aşağıdaki sınıflar türetilmiştir:

```
domain_error
invalid_argument
length_error
out_of_range
```

bad_cast : *dynamic_cast* işleci tarafından gönderilen hata nesnesi bu sınıf türündendir.

bad_typeid :

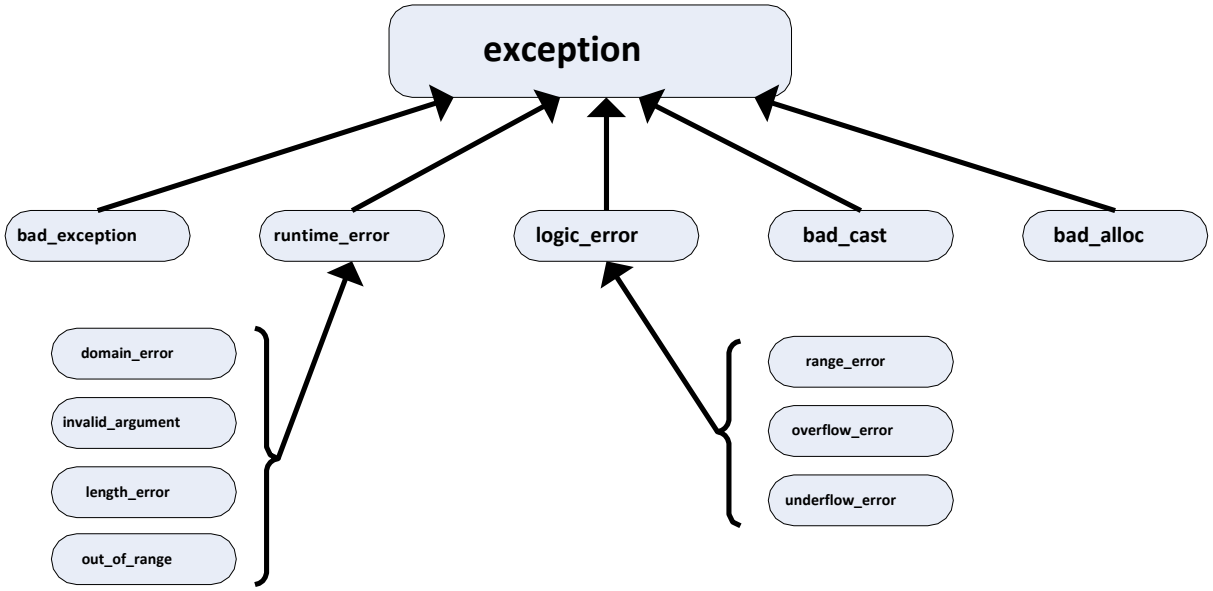
runtime_error sınıfından ise aşağıdaki sınıflar türetilmiştir:

```
range_error
overflow_error
```

bad_alloc : *new* işleci tarafından gönderilen hata nesnesi bu sınıf türündendir. Boş bir sınıf olarak tanımlanmıştır.

Programcı özel bir kütüphane sisteminde çalışmıyorsa hata işleme sınıflarını *STL*"in sınıflarından türeterek oluşturmalıdır. C++ dilinin bazı normal dışı durumlarında gönderdikleri hata nesneleri de *exception* sınıfından türetilmiştir.

STANDART HATA İŞLEME SINIFLARI



ÇALIŞMA ZAMANINDA TÜR BELİRLENMESİ

Çalışma zamanında tür bilgisinin elde edilmesi C++ diline eklenen son özelliklerden biridir. Standartlar öncesi oluşturulmuş derleyicilerde, bu özellik desteklenmeyebilir. Bazı derleyiciler de, bu özelliği isteğe bağlı olarak etkin duruma geçiriyor olabilir. Bu özelliğe kısaca *RTTI* (*Runtime Type Information/Identification*) denir.

RTTI, C++ dili tarafından standart hale getirilmiş, bir nesnenin türünün çalışma zamanında belirlenebilmesine yönelik, bazı araçlara verilen genel isimdir.

Böyle bir aracın standart bir hale getirilmesinden önce, zaten yazılım geliştiren bazı firmalar, çalışma zamanında bir nesnenin türünün belirlenebilmesine yönelik araçları kendileri geliştirip, kendi oluşturdukları sınıf kütüphaneleri için kullanıyorlardı. Ancak bu durum farklı kütüphanelerin kullanılması durumunda bir takım taşınabilirlik sorunlarına yol açıyordu. Çünkü bir firmanın kullandığı bileşen, başka bir firmanın kullandığı bileşen ile uyumsuz olabiliyordu. *RTTI* araçlarının standart hale getirilmesi, gelecekte oluşturulacak farklı sınıf kütüphanelerinin birbirleriyle uyumlu olması konusunda atılmış önemli bir adım kabul edilebilir.

Çalışma Zamanında Bir Nesnenin Türünün Belirlenmesi (RTTI) Nedir?

Mekanizmayı iyi anlayabilmek için önce "aşağı doğru dönüşüm" *(downcast)* konusunu incelemek gerekir. Aşağı doğru dönüşüm ne anlama gelir?

Taban sınıf türünden bir gösterici değişkene, türemiş sınıf türünden bir nesnenin adresi doğrudan doğrudan atanabilir. İngilizcede bu duruma "Yukarı doğru dönüşüm" (*upcasting)* denir. "Aşağı doğru dönüşüm" (*downcasting*) ise, taban sınıf türünden bir adresin türemiş sınıf türünden bir göstericiye, tür dönüştürme işleci kullanılarak atanması işlemidir. Tür dönüştürme işleci kullanılmadan, taban sınıf nesnesinin adresinin, türemiş sınıf türünden bir göstericiye atanması dilin kurallarına göre geçersizdir.

Bir sınıf hiyerarşisi içinde, bir taban sınıftan türemiş çeşitli sınıfların yaratıldığını düşünelim. Türemiş sınıf türünden bir sınıf nesnesinin adresi, taban sınıf türünden bir gösterici değişkene atanabilir. Ya da böyle bir işlem referans kullanılarak yapılabilir.

Peki taban sınıf türünden gösterici ile bir üye işleve çağrı yapıldığında, hangi işlev çağrılır? Eğer çağrılan sanal bir işlev değilse, taban sınıfın işlevi çağrılır. Ancak çağrılan sanal bir işlev ise, taban sınıf göstericisine hangi türemiş sınıf türünden bir nesnenin adresi atanmışsa, o türemiş sınıfa ilişkin işlev çağrılır. Tabi bunun için, türemiş sınıfın devir aldığı sanal işlevi ezmiş (*override*) olması gerekir. Sanal işlevin söz konusu olması durumunda, zaten taban sınıfa ilişkin göstericinin içinde ne türden bir nesnenin adresinin tutulduğunun bilinmesine gerek kalmaz.

Ancak bazı durumlarda, türemiş sınıfa ilişkin bir nesnenin adresi taban sınıf türünden bir nesneye atanmış olsa da, halen bu gösterici yoluyla türemiş sınıf türlerinden birine ilişkin, sanal olmayan bir işlev çağrılmak istenebilir. İşte *RTTI* araçları ağırlıklı olarak böyle durumlarda kullanılır. Bu araç ile, üye işlevi çağrılacak nesnenin tür bilgisi elde edilerek üye işlevin çağrılıp çağrılmayacağına karar verilir.

Söz konusu gereksinimi basit bir örnekle göstermeye çalışalım. Aşağıdaki kodu inceleyin:

```
class Base {

    //..
public:

    virtual void vfunc();

};



class Der1: public Base {

    //...
public:

    virtual void vfunc();
    void foo();

};



class Der2: public Base {
public:

    virtual void vfunc();

    //...

};



class Der3: public Base {
public:

    virtual void vfunc();

    //...
```

```
};


void process(Base *baseptr)
{
    baseptr->vfunc();
    //baseptr Der1 sınıfı türünden ise Der1 sınıfının foo islevinin
    //cagrilmasi gerekiyor.
}
```

Base sınıfının *vfunc* isimli bir sanal üye işlevi var: Bu işlev *Base* sınıfından türeyen *Der1*, *Der2* ve *Der3* sınıfları tarafından eziliyor. Böylece çokbiçimli bir sınıf hiyerarşisi oluşturuluyor. *Der1*sınıfı, *Base* sınıfından devir aldığı bu sanal işlevin dışında, kendisi de *foo* isimli bir işlev tanımlıyor.

Base sınıfı türünden bir gösterici parametre değişkenine sahip, global *process* isimli işlev, çokbiçimli işlev yapmak amacıyla tanımlanıyor. Yani *process* işlevi *Base* türünden bir nesnenin adresi ile çağrılabildiği gibi, *Der1, Der2, Der3* sınıfları türlerinden bir nesnenin adresi ile de çağrılabilir. *process* işlevi içinde, önce taban sınıfın, sanal *vfunc* işlevi çağrılıyor. Bu işlev sanal sanal olduğu için, hangi türden sınıf nesnesinin adresi *process* işlevine geçirilirse o sınıfa ilişkin *vfunc* işlevi çağrılır. *process* işlevine adresi gönderilen nesne *Der1* sınıfı türünden ise, bu sınıfın *foo* isimli işlevinin çağrılması gerektiğini düşünelim.

```
if (baseptr'nin gösterdiği nesne Der1 türünden ise)
    ((Der1 *)baseptr)->foo()
```

Peki ama taban sınıfı içindeki göstericinin, *Der1* sınıfı türünden nesneyi gösterip göstermediği nasıl bilinebilir? Bu bilgi her zaman derleme zamanında elde edilemez. Ancak çalışma zamanı içinde devreye giren bir mekanizma yardımıyla saptanabilir.

Yukarıdaki örnekte taban sınıf türünden gösterici olan *baseptr* göstericisi içindeki adres, tür dönüştürme işleci ile türemiş sınıf türünden bir adrese dönüştürülüp, bu adres ile türetilen *Der1* sınıfının *foo* işlevi çağrılıyor. Ancak bu dönüştürme işlminin çalışma zamanı hatası olmaması için, gerçekten *baseptr* içinde *Der1* sınıfı türünden bir adres olup olmadığının bilinmesi gerekir. Peki *ptr* içinde *Der1* sınıfı türünden bir adres değil de örneğin *Base* sınıfı türünden bir adres varsa ne olur? Bu durumda *foo* işlevine *this* adresi olarak geçirilen adres geçerli bir adres olmaz. *foo* işlevine yapılan çağrı bir gösterici hatasına neden olur.

Aşağı doğru dönüşüm, tür dönüştürme işleçlerinin kullanılarak taban sınıf türünden bir adresten türemiş sınıf türünden bir adresin elde edilmesidir. Ancak bu riskli bir işlemdir. Çünkü tür dönüştürme işlemi yapıldığında, taban sınıf göstericisinin içinde gerçekten türemiş sınıf türünden bir nesnenin adresi olup olmadığı bilinmeyebilir.

İşte *RTTI* araçları bu noktada işe yarar. Bu araçlarla aşağı doğru bir dönüşümün güvenilir bir biçimde yapılması mümkündür.

C++ dilinde *RTTI* araçlarının üç ayrı bileşeni vardır:

1) *dynamic_ca*st tür dönüştürme işleci

dynamic_cast C++ dilinin bir anahtar sözcüğüdür.

Bu işleç, taban sınıf türünden bir adresi -eğer mümkün ise- türemiş sınıf türünden bir adrese dönüştürür. Tür dönüştürme işlemi başarılı olarak yapılırsa işlecin ürettiği değer, hedef tür olan türemiş sınıf türünden bir adrestir. Aksi halde, yani dönüştürme işlemi başarılı olmamışsa, işleç *0* değerini yani *NULL* adresini üretir.

2) *typeid* işleci. Bir nesnenin türünü kesin olarak belirleyen bir işleçtir.

typeid C++ dilinin bir anahtar sözcüğüdür.

3) Bilgi edinilmek istenen tür hakkındaki önemli bilgilerin tutulduğu *type_info* sınıfı.

RTTI araçları ancak sanal işleve sahip bir sınıf hiyerarşisi için kullanılabilir. Eski derleyiciler çalışma zamanında tür belirlenmesini desteklemiyor olabilir. Yeni derleyicilerin çoğunluğu ise *RTTI* araçlarını seçime bağlı olarak etkin duruma geçirir.

Şimdi bu araçları ayrıntılı olarak inceleyelim:

dynamic_cast İşleci

Bu işleç *RTTI* mekanizmasının ağırlıklı olarak kullanılan aracıdır. Bu işlecin kullanılmasıyla, taban sınıf türünden bir adresin, programın çalışma zamanında türemiş sınıf türünden bir adrese güvenilir bir şekilde dönüştürülüp dönüştürülemeyeceği öğrenilebilir.

İşlecin kullanımı daha önce açıklanan yeni tür dönüştürme işleçlerinin (*static_cast, const_cast, reinterpret_cast*) kullanımına benzer.

```
dynamic_cast<Der1 *> (baseptr)
```

Yukarıdaki ifade ile *baseptr* adresi *Der1* sınıfı türünden bir adrese dönüştürülmeye çalışılıyor. Eğer *baseptr* nesnesinin içinde gerçekten *Der1* sınıfı türünden bir nesnenin adresi varsa, dönüşüm başarılı olur. Böylece

işleç *Der1* sınıfı türünden bir adres üretir. Eğer *baseptr* gösterici değişkeni içinde *Der1* sınıfı türünden bir adres yoksa, dönüşüm başarılı olmaz, işleç *0 (NULL)* adresini üretir.

dynamic_cast işlecinin terimi olan adresin -yukarıdaki örnekte *baseptr* göstericisinin- çokbiçimliliği destekleyen bir sınıf türünden olması zorunludur. Yani *baseptr* gösterici değişkenin ait olduğu sınıfın, en az bir sanal işlev içermesi gerekir. Aksi halde işlecin kullanımı geçersizdir.

```
void process(Base *baseptr)

{

    baseptr->vfunc ();


    if (Der1 *der1_ptr = dynamic_cast<Der1 *> (baseptr))
        der1_ptr->foo();

}
```

Şimdi *process* işlevi içinde çalışma zamanında tür belirleniyor. *baseptr* göstericisi *dynamic_cast* işlecinin terimi yapılarak, *Der1 **** türüne dönüştürülüyor. *dynamic_cast* işlecinin ürettiği değer *Der1* türünden gösterici olan *der1_ptr* değişkenine atanıyor. Eğer atanan değer *NULL* adresi değilse, bu gösterici yoluyla, türemiş *Der1* sınıfının *foo* isimli işlevi çağrılıyor.

dynamic_cast işlecinin tür dönüştürme işlemini başarıyla yapıp yapmadığı mutlaka sınanmalıdır.

bad_cast Sınıfı

dynamic_cast işleci ile sınıf nesnesi gösteren bir ifade yani bir sol taraf değeri, sınıf türünden bir referansa da dönüştürülebilir:

```
#include <iostream>

class Base {

    //...

public:

    virtual ~Base(){}

};


class Der : public Base{

    //...

};
```

```
void func(Base &baseref)
{
    Der &r = dynamic_cast<Der &> (baseref);
    //...
}

int main()
{
    Der der;
    func(der);

    return 0;
}
```

Peki dönüştürme işleminin başarısı nasıl sınanabilir? Bir adres türüne dönüştürme yapıldığında, dönüştürme işleminin başarısız olduğu *dynamic_cast* işlecinin *NULL* adresi değeri üretmesiyle anlaşılıyordu. Ancak *NULL* referans diye bir kavram olmadığı için, referansa yapılan dönüşümün başarısı bu şekilde sınanamaz.

Dönüşümün başarısızlığı durumunda *exception handling* mekanizması devreye girer. *dynamic_cast* işleciyle referansa yapılan bir dönüşüm başarılı olmaz ise standart *bad_cast* sınıfı türünden bir hata nesnesi gönderilir. *bad_cast* sınıfı Standart C++ kütüphanesi tarafından tanımlanmış, standart e*xception* sınıfından türetilmiş bir sınıftır. Bu sınıfın tanımı standart *typeinfo* başlık dosyasındadır.

```
void func (Base &baseref)
{
    try {
        Der &r = dynamic_cast<Der &> (baseref);
    }
    catch (std::bad_cast &) {
        //...
    }
}
```

Ne zaman bir adres türü yerine bir referans türüne *dynamic_cast* işleciyle dönüşüm yapılmalıdır? Bu tamamıyla programcının seçimidir. Programcı oluşturduğu tasarım doğrultusunda bir seçim yapar. Referansa yapılan bir dönüşüm söz konusu olduğunda, başarısızlık durumu göz ardı edilemez. Mutlaka hata işleme araçları kullanılmalıdır. Ancak hata işleme araçlarının, programın çalışma zamanı açısından bir maliyeti olduğu için, programcıların çoğu *dynamic_cast* işleciyle bir referans türüne değil bir adres türüne dönüşüm yapmayı tercih eder.

dynamic_cast İŞleci ile static_cast İŞleci Arasındaki Fark *static_cast* işleci ile, bir sınıf türünden adresin, aynı hiyerarşi içinde başka bir sınıf türünden adrese aşağı doğru dönüştürülmesi geçerlidir. Örneğin *A* sınıfından *B* sınıfı, *B*

sınıfından da *C* sınıfını türetmiş olalım. *static_cast* işleci ile *A* sınıfı türünden bir adresi *C*

sınıfı türünden bir adrese dönüştürmek geçerlidir:

```
class A {
    //...
};

class B : public A {
    //...
};

class C : public B {
    //...
};

void func(A *aptr)
{
    C *cptr = static_cast<C *>(aptr);
    //...
}
```

Yukarıdaki kod parçasında, *func* işlevinin parametre değişkeni olan *aptr* göstericisinin değeri, işlevin ana bloğu içinde *static_cast* işleci ile, *C* sınıfı türünden bir adrese dönüştürülüyor. Kod tamamen geçerlidir. Ancak kodun geçerli olması *aptr* gösterici değişkeni içinde, *C* sınıfı türünden bir nesnenin adresi olduğunun güvencesi değildir. Sınama derleme zamanında yapılır. *func* işlevinin *A* sınıfı türünden bir adresle çağrılması gösterici hatasıdır.

static_cast işleci ile aşağı doğru dönüşüm yapılabilir. Ancak dönüşümün güvenli olup olmadığı, programın çalışma zamanında sınanmaz. İşlecin ürettiği değer, her zaman açısal ayraç içine yazılan hedef türden adrestir. Ancak programın çalışma zamanında o adreste o türden bir nesne olacağının hiçbir güvencesi yoktur. Şimdi de dönüşümün *dynamic_cast* işleci ile yapıldığını düşünelim:

```
void func(A *aptr)
{
    if (C *cptr = dynamic_cast<C *>(aptr))
        //...
}
```

Yukarıdaki örnekte *func* işlevi çağrıldığında, *func* işlevine *C* sınıfı türünden ya da *C* sınıfından türeyen bir sınıf türünden adres geçilirse, *dynamic_cast* işleci başarılı olur. İşleç *C* sınıfı türünden bir adres üretir. Aksi halde işleç *NULL* adresi üretir. Sınama programın çalışma zamanında yapılır.

typeid İşleci ve type_info Sınıfı

typeid işleci ile iki nesnenin türünün aynı olup olmadığı, programın çalışma zamanında saptanabilir. *typeid* tek terimli önek konumunda bir işleçtir. İşlecin kullanımı şöyledir:

```
typeid(ifade)
```

Bu işleç ifadenin türünü saptar ve ifadenin türüne uygun

```
const std::type_info &
```

türünden bir değer üretir. Burada belirtilen ifade, *sizeof* işlecinde olduğu gibi herhangi bir tür ismi, ya da normal bir ifade olabilir. İşlecin terimi olan ifade, çokbiçimli bir sınıf türünden bir nesne belirtiyorsa, elde edilecek bilgi çalışma zamanı sırasındaki dinamik türe ilişkindir. Örneğin *baseptr,* çokbiçimli bir türetme şemasında taban sınıf türünden bir gösterici olsun. **baseptr, typeid* işlecinin terimi yapılmış olsun. *baseptr* çalışma zamanında hangi nesneyi gösteriyor ise, o sınıfa ilişkin tür bilgisi elde edilir.

typeid işlecinin ürettiği değer *std::type_info* sınıfı türünden *const* bir referanstır. Bu referans *std::type_info* sınıfı türünden bir nesnenin yerine geçer. *type_info* sınıfının kodlanması derleyiciden derleyiciye değişebilir. Ancak bu sınıfın temel yapısı standart olup, tüm derleyiciler için aynıdır:

```
namespace std{

    class type_info {

    //Derleyiciye bağlı kısım
    private:

        type_info(const type_info &);

        type_info &operator=(const type_info &);
    public:

        virtual ~type_info();

        bool operator==(const type_info &, const type_info &);
        bool operator!=(const type_info &, const type_info &);
        bool before(const type_info &r) const;

        const char *name() const;

    };

}
```

Yukarıdaki sınıf tanımında bazı noktaları inceleyelim :

type_info sınıfı da diğer sınıflarda olduğu gibi *std* isim alanı içinde bildirilmiştir.

1. *type_info* sınıfının bir varsayılan kurucu işlevi yoktur.

type_info sınıfı türünden bir nesne varsayılan kurucu işlev çağrılacak şekilde yaratılmaz.

2. *type_info* sınıfının kopyalayan kurucu işlevi ve atama işlecini yükleyen işlevi sınıfın *private* bölümüne yerleştirilmiştir. Bu da şu anlama gelir: *typeid* işlecinin ürettiği referansa bir başka *type_info* türünden nesne atanamaz. Bir *type_info* nesnesi başka bir *type_info* nesnesi ile ilk değer verilerek yaratılamaz.

```
#include <typeinfo>

type_info t1;  // Geçersiz!

type_info *ptr = new type_info;   // Geçersiz!

type_info t2(type_id(unsigned int))   //Geçersiz!
```

type_info sınıf türünden bir nesne, ancak *typeid* işleci ile elde edilebilir.

2. == ve *!=* karşılaştırma işleçlerini yükleyen, sınıfın *public* işlevleri tanımlanmıştır. *type_info* türünden nesnelerin eşitliği bu işleçler tarafından sınanabilir. Aşağıdaki örneği inceleyin:

```
#include <iostream>
#include <typeinfo>


class Sample1 {

    //...
public:

    virtual ~Sample1() {}

};


class Sample2 {

    //...
public:

    virtual ~Sample2() {}

};

class Sample3 {

    //...
public:

    virtual ~Sample3() {}

};


int main()

{

    Sample1 s1;
    Sample2 s2;
    Sample3 s3;


    Sample1 &rs1 = s1;
    Sample2 *ptrs2 = &s2;


    if (typeid(Sample1) == typeid(Sample2))
        std::cout << "Ayni tur!" << std::endl;

    else

        std::cout << "Farkli tur!" << std::endl;


    if (typeid(rs1) == typeid(s1))
```

```
        std::cout <<"Ayni tur!" << std::endl;
    else

        std::cout <<"Farkli tur!" << std::endl;


    if (typeid(*ptrs2) == typeid(rs1))
        std::cout <<"Ayni tur!" << std::endl;

    else

        std::cout <<"Farkli tur!" << std::endl;


    return 0;

}
```

3. Sınıfın *char* türden bir adrese geri dönen *name* isimli bir üye işlevi vardır. Bu işlevin geri dönüş değeri olan adreste sınıfın ismi gösteren bir karakter dizisi vardır:

```
std::cout  <<  typeid(Sample1).name()  <<  std::endl;  //class  Sample1
std::cout  <<  typeid(*ptrs2).name()  <<  std::endl;  //class  Sample2
std::cout << typeid(rs1).name() << std::endl;          //class Sample1
std::cout << typeid(int).name() << std::endl;          //int

std::cout << typeid(long double).name() << std::endl;    //long double
```

Ayrıca sınıfın *bool* değere geri dönen *before* isimli bir üye işlevi vardır. Bu işlev *type_info* türünden nesnelerin sıralanabilmesi için tanımlanmıştır. Bu üye işlev ile **this* nesnesine ilişkin türün, arguman olarak alınan *type_info* nesnesi ile karşılaştırılması yapılır.

Karşılaştırma derleyiciye özgü bir biçimde yapılır. Örneğin geri dönüş değeri *true* ise *this nesnesine ilişkin tür, "türetme hiyerarşisinde daha yukarıda" anlamı çıkartılamaz.

typeid işlecinin Çokbiçimli Olmayan Sınıflar İçin Kullanımı

typeid işleci genel olarak çokbiçimli sınıflara ilişkin uygulamalarda kullanılsa da, böyle bir zorunluluk yoktur. Yukarıdaki örnekte görüldüğü gibi *typeid* işlecinin terimi C++ dilinin temel veri türlerinin ismi, ya da bu veri türlerinden nesneler de olabilir.

typeid işlecinin terimi olan nesne taban sınıf türünden ise, *typeid* işlecinin davranışı, söz konusu taban sınıfın çokbiçimli olup olmamasına göre, yani sanal işlev içerip içermemesine göre değişir. Aşağıdaki örneği inceleyelim:

```
#include <iostream>

class Base {
```

```
    //...
public:

};


class Der : public Base{

    //...

};


int main()

{

    Der der;

    Base *base_ptr = &der;

    std::cout << typeid(*base_ptr).name() << std::endl;
    return 0;

}


int main()

{

    Der der;

    Base *base_ptr = &der;

    std::cout << typeid(*base_ptr).name() << std::endl;



    return 0;

}
```

Yukarıdaki örnekte ekrana *class Base* yazılır. *Base* türünden bir göstericiye, *Der* türünden bir nesnenin adresi atanmış olmasına karşın, *typeid* işlecinin üretmiş olduğu referans ile *name* işlevi çağrıldığında, ekrana taban sınıfın ismi yazılır. Çünkü *Base* sınıfı çokbiçimli sınıf değildir. Oysa *Base* sınıfının tanımı bir sanal işlev içerecek şekilde değiştirilirse bu kez ekrana *class Der* yazılır. *Sample* sınıfının tanımını aşağıdaki gibi değiştirip, kaynak kodu yeniden derleyin ve çalıştırın:

```
class Sample {

    //...
public:

    virtual ~Sample(){}
```

```
};
```

Derleyiciler, yukarıda belirtilen ve standartlara göre sınıf tanımı içinde bulunması gereken işlevlerin dışında, *type_info* sınıfı için, başka faydalı üye işlevler tanımlayarak programcının kullanımına sunabilir. Örneğin bazı derleyiciler, sınıf türüne ilişkin üye işlevlerin listesini veren bir üye işlev tanımlamış olabilir. Yine bazı derleyiciler sınıfa ilişkin bir nesnenin bellekteki organizasyonu hakkında bilgi veren bir üye işlev tanımlamış olabilir.

bad_typeid Sınıfı

Birçok uygulamada *typeid* işlecinin terimi, içerik işleci ile oluşturulan bir ifadedir.

Bu durumda, içerik işlecinin terimi *NULL* adresi ise, standart *bad_typeid* sınıfı türünden bir hata nesnesi gönderilir. *bad_typeid* sınıfı, standart *exception* sınıfından türetilmiştir. Sınıfın tanımı *typeinfo* başlık dosyası içindedir.

Aşağıdaki örneği inceleyin:

```
#include <iostream>
#include <typeinfo>



class Base {
    //...
public:
};


void func(Base *ptr)
{
    try {
        std::cout << typeid(*ptr).name() << std::endl;
    }
    catch (std::bad_typeid &) {
        std::cerr << "dereferencing null pointer" << std::endl;
        exit(EXIT_FAILURE);
    }
    //...
}
```

RTTI araçlarının maliyeti

Çalışma zamanında bir nesnenin türünün belirlenmesinin bir maliyeti vardır. *RTTI* araçlarının kodlanması derleyiciyi yazanların seçimine bırakılmıştır. Derleyicilerin çoğu bellekte her bir veri türü için bir *type_info* nesnelik alan ayırır. Daha önce belirtildiği gibi *dynamic_cast* mekanizması yalnızca çokbiçimli nesnelere uygulanabilir. Sistemlerin çoğunda, çokbiçimli her nesne, bu nesne için ayrılmış bellek alanında, ait olduğu çokbiçimli sınıfa ilişkin tutulan sanal işlev tablosunun adresini tutan bir göstericiye sahiptir. Bu göstericiye kavramsal olarak *vptr* (*pointer to virtual functions table*) denir. Daha önceki konularda, sanal işlev tablosu içinde sanal işlevlerin adreslerinin tutulduğu belirtilmişti. İşte çalışma zamanı tür bilgisinin elde edilmesi için, derleyiciler çoğu sanal işlev tablosunun başında, tür bilgilerinin tutulduğu *type_info* sınıf nesnesinin adresini de tutar. Dolayısıyla çalışma zamanı tür bilgisinin elde edilmesi, aslında gösterici işlemleriyle yapılır. Önce nesnenin sanal işlev tablo göstericisine erişilir. Bu göstericiden, sanal işlev tablosunun adresi alınır. Sanal işlev tablosundan da, ilgili *type_info* nesnesinin adresi elde edilir. Maliyetin bir sanal işlev çağrısının maliyetine eşdeğer olduğu düşünülebilir.

Her veri türü için bir *type_info* nesnesinin var olması, özellikle büyük programlar için fazladan yük getirir. Yüzlerce çokbiçimli sınıfa sahip bir programda yine yüzlerce *type_info* nesnesi yaratılır. Tüm bu nesneler bellekte yer kaplar. Bütün sınıf nesneleri gibi *type_info* sınıfı nesnelerinin de yaratılmasının da zamansal bir maliyeti vardır. *RTTI* program içinde hiç kullanılmamış olsa da *type_info* nesneleri yine yaratılır. İşte bu yüzden derleyiciler *RTTI* mekanizmasını bir anahtar ile aktif hale getirilmesine izin verir. *RTTI* mekanizmasının devre dışı bırakılması, çalışabilir kodun küçülmesine ve çalışma hızının artmasına neden olur.

typeid İşleci iled ynamic_cast İşlecinin Maliyet Açısından Karşılaştırılması

typeid işlecinin bir değer üretmesi her nesne için "sabit" bir süre içinde gerçekleşir. Süre her çokbiçimli nesne için, türetme sınıf hiyerarşisi içinde yeri ne olursa olsun sabittir. Bu süre bir sanal işlevin çağrısı için harcanan süreye eşdeğer kabul edilebilir.

```
typeid(nesne)
```

gibi bir ifadenin eşdeğeri

```
*(nesne.vptr[0])
```

gibi bir işleme karşılık geldiği düşünülebilir.

Ancak *dynamic_cast* işlemi için geçen süre, sabit değildir.

```
dynamic_cast<T *>(base_ptr)
```

gibi bir ifadenin değerlendirilmesi için geçen süre, dönüştürülme işlemine sokulan sınıfın, ait olduğu sınıf hiyerarşisinin derinliğinin artmasıyla artar.

Yukarıdaki ifadede *base_ptr* göstericisinin ait oldugu sınıfın çok derin bir türetme zincirinin en tepesindeki sınıf olduğunu düşünelim. Dönüştürülmenin yapılacağı tür de sınıf hiyerarşisi içinde yer almayan bir tür olsun. *dynamic_cast* işlecinin *NULL* adresi üretmesi için türetme zincirinin en alt kademesine kadar gidilip kontrol yapılması gerekir. Tasarım açısından *dynamic_cast* işleci *typeid* işlecine tercih edilmelidir. Çünkü *dynamic_cast* işleci ile oluşturulan kodun esnekliği ve geliştilebiliriliği *type_id* işleci ile oluşturulmuş koda göre daha fazladır. Ancak buna karşı *dynamic_cast* işlecinin maliyeti de *typeid* işlecinin maliyetine göre daha fazladır.

Çalışma Zamanında Tür Belirlenmesinin Sakıncaları

Sanal işlev yapısı her zaman *RTTI* araçlarına tercih edilmelidir. *RTTI* araçlarının, bir zorunluluk bulunmamasına karşın kullanılması, programların karmaşıklığını arttırır, test işlemlerini zorlaştırır. Kaynak kod değişikliklere karşı daha kırılgan hale gelir. Programın çalışması yavaşlayabilir. *RTTI* araçlarının kullanılması konusunda bir zorunluluk varsa, bu durumda tasarım açısından, önce *dynamic_cast* işleci tercih edilmelidir.

C++ dilinde Nesne Yönelimli Programlama Tekniği kullanılarak yazılmış programlarda, program tasarımı, diğer programlama dillerinde olduğundan çok daha fazla öneme sahiptir. Çalışma zamanında tür bilgisinin belirlenmesine yönelik gereksinim, nesne yönelimli programlama tekniğinin iyi bir şekilde uygulanması durumunda, çoğu zaman ortadan kalkar. *RTTI* çoğu zaman programcının nesne yönelimli programlama konusunda yeterli deneyime sahip olmaması nedeniyle başvurulan bir yöntemdir. İyi bir tasarımda yalnızca zorunlu durumlarda kullanılmalıdır. Peki dinamik tür kontrolünün yapılmasında tasarım açısından sorunlu olan durum nedir?

Aşağı doğru dönüşümler güvenli bir şekilde yapılabilmesine karşın, genel olarak aşağıdaki sakıncalardan söz edilebilir:

1. Aşağı doğru dönüşümler, karmaşıklığın hizmet veren kodlardan hizmet alan kodlara, yani kullanıcı kodlarına kaydırılmasına neden olur. Böylece kodların karmaşıklığı artar, buda kodlama maliyetinde bir artışa neden olur.

2. Karmaşıklığın kullanıcı kodlarına kayması kodların genel olarak büyümesine neden olur.

3. Dönüşümün güvenli olup olmadığını sınamak için, her dönüşüm için ayrı bir kod yazmak gerekir. Bu durumda bakım giderleri artar.

4. Bu dönüşümler programın çalışmasında genel bir yavaşlamaya neden olur. Yavaşlama oranı da, sınıf hiyerarşisinin derinliği oranında artar.

5. Programın yapılacak eklemeler zorlaşır. Çoğu zaman, türetme hiyerarşisine eklenen sınıflar için ek sınamalar ya da kontroller yapmak gerekir.

Ana sorun daha önce yazılmış bir kod parçasına ulaşabilmek için, fazladan bir kod yazma zorunluluğudur. Üstelik fazladan yazılan kod, programın genişletilmesi amacında daha sonradan sorunlara yol açabilir. Fazladan yazılan kod, bir sınıf türü için çalışma zamanında, o sınıf türü için belirli işlevlerin çağrılıp çağrılamayacağını sınamak için yazılan kodlardır. Ek kod içinde yapılan sınama olumlu sonuçlanırsa aşağı doğru bir dönüşüm yapılarak, istenen işlev çağrılır.

Eğer daha önce yazılmış olan bir kodu (*server code*) bulmak için programcı ek bir kod yazmış ise, programcının yazmış olduğu ek kod karmaşıktır, çeşitli hatalara açıktır. Oysa nesne yönelimli Programlama tekniğinin ana özelliklerinden birisi karmaşıklığı indirgemektir. Dinamik tür kontrolunun uygulandığı çoğu durumda programcının, ana kodda oluşturulan türetme hiyerarşisi hakkında bilgi sahibi olmasını zorunlu kılar, bu durumda da ileride ana kodda kullanılan sınıf hiyerarşisi yapısında bir değişiklik yapıldığında, yazılan kodların geçersiz bir duruma düşmesine, yeniden yazılmak zorunda kalınmasına neden olabilir, bu durum da yine nesne yönelimli programlama tekniğinin genel felsefesine aykırıdır.

Aşağı Doğru Dönüşümlere Seçenek: Sanal İşlevler

RTTI araçlarını ve aşağı doğru dönüşümleri (*downcast*) kullanmak yerine, dinamik bağlama ve sanal işlevler kullanılabilir. Bu teknikte, yalnızca türemiş sınıflarda kullanılacak üye işlevler genelleştirilerek taban sınıfa kaydırılır. Yani kullanıcı kodları, sanal işlevler biçiminde nesnenin içine kaydırılır. Bu durumda aşağı doğru dönüşümler derleyiciler tarafından hem otomatik olarak hem de güvenli bir şekilde yapılır. Aşağı doğru yapılan bir dönüşüm eğer güvenilir değilse, zaten derleme aşamasında derleyici tarafından saptanır. Ayrıca bu tür bir tasarım, söz konusu programın geliştirilebirliği konusunda da son derece esnek bir yapı sağlar. Yeni bir türetme yapılması durumunda var olan ana sınıf kodlarında bir değişiklik yapılması gerekmez.

Aşağı doğru dönüşüm ve izleyen *if* deyiminden oluşan kod parçacıkları çoğu zaman, sanal işlev çağrılarıyla değiştirilebilir. Buradaki anahtar faktör, yetenek sorgulamasının bir hizmet isteğine dönüştürülmesidir. Nesneye yapılan bir hizmet alma isteği, bir sanal işlev çağrısıdır.

new ve delete İşleçlerinin Yüklenmesi

new işlecinin dinamik bir sınıf bir sınıf nesnesi elde etmek için kullanıldığını düşünelim:

```
class A {

    //...

};


new A *pd = new A;
```

new işlecinin standartlar tarafından güvence altına alınan davranışı şöyledir:

new işleci *operator new* isimli bir global işlevi çağırarak *free store* alanından *sizeof(A)* kadar büyüklükte bir blok elde eder. Daha sonra, elde edilen dinamik bellek bloğunun başlangıç adresi, *A* sınıfının kurucu işlevine *this* adresi olarak geçirilir. Böylece sınıfın kurucu işlevi, dinamik olarak elde edilen bellek bloğunda, *A* sınıfı türünden bir nesne oluşturur.

new işleci ve *operator new* terimleri sıklıkla birbirleriyle karıştırılır. *new,* C++ dilinin bir işlecidir. Programcı hangi işlevi tanımlarsa tanımlasın, *new* işlecinin yukarıda açıklanan davranışını değiştiremez. Ancak programcı isterse, *new* işlecinin *free store* alanından yer elde etmek amacıyla çağırdığı *operator new* işlevini yükleyebilir. Gerçekte yüklenen *new* işleci değil *operator new* işlevidir.

operator new İşlevi

Tek bir sınıf nesnesi için *free store* alanından yer elde etme işlemleri standart bir işlev olan *operator new* tarafından yapılır. Bu işlevin bildirimi

```
void *operator new(size_t);
```

biçimindedir. İşlevin parametresi, *new* işlecinin terimi olmuş ifadenin *sizeof* değeridir. Bu işlev parametresine aktarılan büyüklükte bir dinamik bellek bloğu elde ederek bloğun başlangıç adresini döndürür. *new* işlecinin kullanılması durumunda, işlecin terimi olan türün *sizeof* değeri derleyici tarafından hesaplanarak bu işleve argüman olarak gönderilir.

Yani

```
new A
```

gibi bir ifadenin, derleyici tarafından önce aşağıdaki gibi bir işlev çağrısına dönüştürüldüğünü düşünebilirsiniz:

```
operator new(sizeof(A))
```

Bu çağrıdan elde edilen adresle *A* sınıfının kurucu işlevinin çağrıldığı düşünülebilir. Programcı isterse *operator new* işlevini dinamik bir blok elde etme amacıyla doğrudan da çağırabilir. Aşağıdaki örneği inceleyin:

```
#include <iostream>
#include <cstring>
```

```
using namespace std;

int main()

{

    char str[100];



    cout << "bir yazi girin ";
    cin >> str;

    char *ptr = (char *)operator new(strlen(str) + 1);
    strcpy(ptr, str);

    strrev(ptr);



    cout << "yazi = (" << ptr << ")" << endl;

    //...
    return 0;

}
```

Örnekte, *operator new* işlevi, yerel *str* dizisine girilen yazının uzunluğundan *1* byte daha büyük bir dinamik alan elde ediyor. Bu alanın adresi *void ** türünden bir değer olarak dışarı iletiliyor. C++ da *void* türden adreslerin başka türden göstericilere doğrudan atanamadığını anımsayın. Söz konusu dinamik bellek bloğunun başlangıç adresi, *char ** türüne dönüştürülerek, aynı türden *ptr* göstericisine atanıyor. Bu gösterici değişken yardımıyla ilgili yazı işleniyor.

operator new işlevi de yüklenebilir. Bu işleci global olarak yüklemek mümkün olduğu gibi, bir sınıfın üye işlevi olarak yüklemek de mümkündür. Yüklenen *new* işleci değil *operator new* işlevidir. *new* işlecinin önceden belirlenmiş davranışını değiştirme olanağı yoktur.

Global operator new İşlevinin Yüklenmesi

İşlevin global olarak tanımlanması durumunda *new* işleciyle yapılan bütün dinamik yer elde etme işlemlerinde, programcı tarafından tanımlanan *new* işlevi çağrılır. Yüklenen işlevin geri dönüş değeri *void ** türünden olmalıdır. Başka türden bir geri dönüş değeri seçilmesi geçersizdir. İşlevin parametre değişkeni *size_t* türünden olmalıdır. Parametre değişkeninin başka bir türden olması geçersizdir. Aşağıda global *operator new* işlevinin yüklenmesine ilişkin bir örnek veriliyor. Programı derleyerek çalıştırın:

```
#include <iostream>
#include <cstdlib>


class A {
```

```
    int a, b;
public:

    A();

};


using namespace std;

A::A()

{

    cout << "A::A()" << endl;

    cout << "this = " << this << endl;
    a = b = 0;

}


void *operator new(size_t size)

{

    cout << "operator new()" << endl;
    cout << "size = " << size << endl;
    void *ptr = malloc(size);

    if (!ptr) {

        cerr << "malloc basarısız" << endl;
        exit(EXIT_FAILURE);

    }

    cout << "elde edilen blogun adresi : " << ptr << endl;


    return ptr;

}

int main()

{

    cout << "sizeof(A) = " << sizeof(A) << endl;
    A *pd = new A;

    cout << "pd = " << pd << endl;

    //...


    return 0;

}
```

Yukarıdaki programda tanımlanan *A* sınıfının kurucu işlevi içinde, işlevin çağrıldığını gösteren bir yazı ile *this* göstericisinin değeri ekrana yazdırılıyor.

Tanımlanan *operator new* işlevinin ana bloğu içinde de, ekrana işlevin çağrıldığını gösteren bir yazı bastırılması sağlanıyor. Bu işlev, parametre değişkeni olan *size* boyutunda bir dinamik bellek bloğu elde ederek bloğun başlangıç adresini döndürüyor. İşlev geri dönüş değerini üretmeden önce dinamik bellek bloğunun başlangıç adresini ekrana yazdırıyor.

main işlevi içinde ise önce ekrana *A* sınıfının *sizeof* değerinin yazdırıldığını görüyorsunuz. Daha sonra *new* işleci kullanılarak bir *A* sınıf nesnesi için bellekte dinamik bellek bloğu elde ediliyor. *new* işlecinin ürettiği değer *A* sınıfı türünden gösterici değişken olan *pd*"ye atanıyor. Son olarak *pd* gösterici değişkeninin değeri ekrana yazdırılıyor.

Çalıştırılan programa ilişkin örnek bir ekran çıktısı aşağıda veriliyor:

```
sizeof(A) = 8
operator new()
size = 8

elde edilen blogun adresi : 0x3d3dc0
A::A()

this = 0x3d3dc0
pd = 0x3d3dc0
```

Global *operator new* işlevi istenirse birden fazla parametre değişkenine sahip olacak şekilde yüklenebilir. Ancak işlevin ilk parametre değişkeninin *size_t* türünden olması zorunludur. Bu durumda işlevin *new* işleci yoluyla çağrılması mümkün olmaz. İşleve çağrı yapmak zorunludur.

operator delete İşlevi

delete C++ dilinin bir işlecidir. *delete* işlecinin terimi bir sınıf türünden adres olduğunda, derleyici nasıl bir kod üretir? Önce *delete* işlecinin terimi olan adres ile sınıfın sonlandırıcı işlevi çağrılır. Daha sonra *operator delete* isimli global bir işlev çağrılarak dinamik olarak elde edilen bloğun *free strore* alanına geri verilmesi sağlanır.

ptr gösterici değişkeninin, *A* sınıfı türünden dinamik bir nesneyi gösterdiğini düşünelim:

```
delete ptr
```

gibi bir ifadenin, derleyici tarafından aşağıdaki gibi bir koda dönüştürüldüğü düşünülebilir:

```
ptr->~A();
```

```
operator delete(ptr);
```

delete işlecinin bu davranışı standartlar tarafından güvence altına alınmıştır. Yazılacak yeni işlevlerle bu davranış değiştirilemez. Ancak istenirse, *delete* işlecinin dinamik bellek bloğunu *free store* a geri vermek amacıyla kullandığı, *operator delete* isimli işlev yüklenebilir. Bu durumda *delete* işleci kullanıldığında, artık programcının yazdığı *operator delete* işlevi çağrılır. İşlevin bildirimi aşağıdaki gibidir:

```
void operator delete(void *);
```

İşlev, parametre değişkenine başlangıç adresi geçilen dinamik bellek bloğunu *free store*

alanına geri verir. İstenirse *operator delete* işlevi doğrudan da çağrılabilir:

```
#include <iostream>

using namespace std;


int main()

{

    cout << "kac tamsayi : ";
    int n;

    cin >> n;

    int *pd = (int *)operator new(n * sizeof(n));
    for (size_t k = 0; k < n; ++k) {

        pd[k] = rand() % 1000;
        cout << pd[k] << " ";

    }

    cout << endl;

    //...

    operator delete(pd);
    return 0;

}
```

Yukarıdaki örnekte *operator new* ile elde edilen bellek bloğu, daha sonra standart

operator delete işlevi ile geri veriliyor.

operator delete İşlevinin Yüklenmesi

Programcının yazacağı *operator delete* işlevinin geri dönüş değeri olmamalı ve işlevin ilk parametre değişkeni *void ** türünden olmalıdır. Daha önce verilen örneğe şimdi aşağıdaki eklemeleri yapıyoruz. Önce *operator delete* işlevini yazıyoruz:

```
void operator delete(void *ptr)
{
    cout << "operator  delete()" << endl;
    cout << "geri verilecek blogun adresi : " << ptr << endl;
    free(ptr);
}
```

Örnekteki *A* sınıfı için, bir de sonlandırıcı işlev ekliyoruz:

```
A::~A()
{
    cout << "A::~A()" << endl;
    cout << "this = " << this << endl;
}
```

main işlevinin sonunda, *new* işleci ile elde edilen dinamik bellek bloğunu geri vermek için

delete işleci kullanılıyor:

```
int main()
{
    cout << "sizeof(A) = " << sizeof(A) << endl;
    A *pd = new A;
    cout << "pd = " << pd << endl;
    delete pd;

    return 0;
}
```

Eklemelerden sonra program yeniden derlenip çalıştırıldığında elde edilen örnek bir ekran çıktısı aşağıda veriliyor:

```
sizeof(A) = 8
operator new()
size = 8

elde edilen blogun adresi : 0x3d3dc0
A::A()

this = 0x3d3dc0
pd = 0x3d3dc0
A::~A()

this = 0x3d3dc0
operator  delete()

geri verilecek blogun adresi : 0x3d3dc0
```

operator new[] İşlevi

Birden fazla sayıda nesne için bir dinamik bellek bloğunun elde edilmesi için *new[]* işleci kullanılır:

```
A *pd = new A[n];
```

new A[10] ifadesi ile önce *operator new[]* isimli global işlev çağrılarak, *A* sınıfı türünden *n* tane nesnenin kaplayacağı kadar bir bellek bloğu elde edilir. Daha sonra *A* sınıfının kurucu işlevi her bir dinamik nesne için ayrı ayrı çağrılır. Yani *new[]* işleci dinamik bellek alanı elde etmek için *operator new[]* işlevini kullanır.

Standartlara göre derleyicilerde böyle bir işlevin tanımlı olması gerekir. İşlevin bildirimi aşağıdaki gibidir:

```
void *operator new[](size_t size);
```

İşlevin geri dönüş değeri ayrılan dinamik bloğun başlangıç adresidir. İşlevin parametre değişkeni ayrılacak dinamik bellek bloğunun büyüklüğüdür. *new[]* işlecinin kullanılması durumunda blok büyüklüğü derleyici tarafından hesaplanarak *operator new* işlevine argüman olarak geçilir. Standart kütüphanedeki *operator new* işlevi kendi tanımı içinde global *operator new* işlevini çağırır:

```
void *operator new[](size_t size)
{
    return operator new(size);
}
```

Yani programcı *operator new[]* işlevini yüklememiş olsa bile *new[]* ile yapılan yer elde etme işlemlerinde yine *operator new* işlevi çağrılır. Ancak isterse programcı *operator new[]* işlevini de tanımlayabilir. Bu durumda

new[] işleci kullanıldığında yer ayırma işlemi programcının yazaacağı *operator new[]* işlevi ile yapılır. Programcının yazacağı işlevin geri dönüş değeri *void ** türünden olmalıdır. İşlevin en az bir parametre değişkeni olmalı ve işlevin ilk paramatre değişkeni *size_t* türünden olmalıdır.

operator delete[] İşlevi

new[] işleci ile elde edilen dinamik bloklar *delete[]* işleci ile geri verilir. *delete[]* işlecinin terimi bir sınıf türünden adres olduğunda derleyici şöyle bir kod üretir: Önce işlece verilen adresten başlayan blok içinde yer alan bütün sınıf nesnleri için tek tek sınıfın sonlandırıcı işlevi çağrılır. Sonra *operator delete* işleviyle, daha önce elde edilmiş olan dinamik blok *free store* a geri verilir. Yani *delete[]* işleci sonlandırıcı işlevi *n* kez çağırdıktan sonra *operator delete[]* işlevini kullanarak ayrılan dinamik bloğu geri verir.

operator delete standart bir işlevdir bildirimi aşağıdaki gibidir:

```
void operator delete[](void *ptr);
```

İşlevin geri dönüş değeri yoktur. İşlev dinamik olarak ayrılmış bir bellek bloğunun başlangıç adresini alır, bloğu *free store* a geri verir. Aslında *operator delete[]* işlevi de kendi içinde *operator delete* işlevini çağırır:

```
void operator delete[](void *ptr)
{
    operator delete(ptr);
}
```

Tabii programcı isterse *operator delete[]* işlevini yükleyebilir. Programcının yazacağı *operator delete[]* işlevinin de geri dönüş değeri olmamalıdır. İşlevin en az bir parametre değişkeni olmalı ve bu parametre değişkeninin türü *void ** olmalıdır.

Daha önce yazılan programın *main* işlevini aşağıdaki gibi değiştirelim:

```
int main()
{
    A *p = new A[4];
    cout << "p = " << p << endl;
    delete[] p;

    return 0;
```

```
}
```

Program yeniden derlenip çalıştırıldığında örnek bir ekran çıktısı aşağıdaki gibi olabilir:

```
operator new()
size = 36

elde edilen blogun adresi : 0x3d3ec0
A::A()

this = 0x3d3ec4
A::A()

this = 0x3d3ecc
A::A()

this = 0x3d3ed4
A::A()

this = 0x3d3edc
p = 0x3d3ec4
A::~A()

this = 0x3d3edc
A::~A()

this = 0x3d3ed4
A::~A()

this = 0x3d3ecc
A::~A()

this = 0x3d3ec4
operator  delete()

geri verilecek blogun adresi : 0x3d3ec0
```

Operator new ve operator delete İşlevlerinin Bir Sınıf İçin Yüklenmesi

Bir sınıf kendisi için yapılacak dinamik yer elde etme işlemlerini kendi üye işlevleriyle yürütebilir. Eğer bir sınıf için *operator new* ve *operator delete* işlevleri çağrılırsa, global işlevler değil programcının sınıf için yazdığı işlevler çağrılır.

Sınıfın operator new() İşlevi

Bir sınıf için *operator new()* işlevi yazılabilir. Derleyici, *new* işlecinin terimi olarak bir sınıfa ilişkin tür isminin verildiğini görürse, önce söz konusu sınıfın, ismi *operator new* olan bir işlevinin bulunup bulunmadığını araştırır.

Sınıfın böyle bir işlevi varsa, *new* işleci dinamik yer elde etme için bu işlevi kullanır. Sınıfın böyle bir işlevi yoksa bu sınıf türü için *new* işlecinin kullanılması sonucu yapılacak dinamik yer elde etme işlemi yine global *operator new* işlevi ile yapılır.

new işlecinin bir sınıf için yüklenmesiyle, bir sınıfa ilişkin dinamik yer elde etme işlemlerinin o sınıfa özgü bir şekilde, diğer sınıf türlerinden farklı bir biçimde yapılması sağlanabilir. Aynı şekilde bir sınıf için *operator new[]* işlevi de tanımlanabilir. Programcı isterse bir sınıfın *operator delete* ya da *operator delete[]* işlevlerini de yazabilir. Böylece belirli bir sınıfa ilişkin dinamik alanı geri verme işlemleri yine o sınıfa özgü bir biçimde yapılabilir.

Sınıf için yazılacak *operator new* işlevinin geri dönüş değeri *void* türden bir adres olmalıdır. Aşağıda örnek bir bildirim görülüyor:

```
class A {
public:

    void *operator new(size_t);

};
```

```
A *ptr = new A;
```

new işlecinin teriminin *A* sınıfı olduğunu gören derleyici, *A* sınıfının *operator new* işlevine sahip olup olmadığını araştırır. *A* sınıfı için *operator new* işlevi tanımlanmış olduğundan bu işlev çağrılır. İşlevin *size_t* türünden olan parametresine *sizeof(A)* değeri geçilir.

Sınıf için *operator new* ve *operator delete* işlevlerinin tanımlanması ya da tanımlanmış işlevlerin kaynak koddan çıkartılması *new* ve *delete* işleçlerinin kullanıldığı kodlarda bir değişiklik yapılmasını gerektirmez. Çünkü global *operator new* işlevi de programcının sınıf için tanımladığı *operator new* işlevi de aynı ifade ile çağrılır. Programcı *operator new* işlevini tanımlamış olsa bile, global olan işlevi çağırabilir:

```
A *ptr = ::new A;
```

A sınıfının *operator new* işlevi olmasına karşın, *new* işlecinin önek konumundaki çözünürlük işleci ile kullanılması sonucu standart kütüphanedeki *operator new* işlevi çağrılır.

Sınıfın operator delete İşlevi

Sınıf için tanımlanacak *operator delete* işlevinin geri dönüş değeri olmamalı, işlevin birinci parametre değişkeni *void ** türünden olmalıdır. *A* sınıfına ilişkin *operator delete* işlevinin bildirimi aşağıdaki gibi olabilir:

```
class A {
public:

    void *operator new(size_t);
    void operator delete(void *);

    //...

}
```

delete işlecine verilen adres bir sınıf türünden ise, derleyici bu sınıfın *operator delete* işlevine sahip olup olmadığını araştırır. Eğer sınıfın *operator delete* işlevi bildirilmiş ise bildirilen işlev çağrılır. Aksi halde global *operator delete* işlevi çağrılır.

```
A *ptr = new A;

//...
delete ptr;
```

ifadeleri için derleyici şöyle bir kod üretir: *ptr*"nin gösterdiği sınıf nesnesi için önce sınıfın sonlandırıcı işlevi çağrılır. Sonra *ptr* adresi *A* sınıfının *operator delete* işlevine argüman olarak gönderilir.

operator new, operator[], operator delete, operator delete[] işlevleri sınıfın statik işlevleridir. Bu işlevlerin bildirimlerinde *static* anahtar sözcüğü yazılmamış olsa bile bunlar derleyici tarafından statik üye işlevler olarak ele alınır. Tabii programcı isterse bu işlevlerin bildirimlerinin önüne static *anahtar sözcüğünü* yerleştirebilir. Bu işlevler statik üye işlevler olduklarından sınıfın statik olmayan elemanlarına doğrudan ulaşamaz. *operator new* ve *operator delete* işlevlerinin bir sınıf için yüklenmesine örnek olarak aşağıda küçük bir program veriliyor. Programı derleyerek çalıştırın:

```
#include <iostream>
#include <cstring>
#include <new>


typedef unsigned char Byte;

class A {

    char s[20]; //yalnızca sınıf nesnesi içinde yer kaplasın diye
    static Byte allocation_pool[];

    static Byte allocation_map[];
public:

    static const size_t max_object_size = 10;
    A(){std::cout << "A()\n";}

    ~A(){std::cout << "~A()\n";}
    void *operator new(size_t);
    void operator delete(void *);
```

```
};


//cpp dosyası

using namespace std;


Byte A::allocation_pool[max_object_size * sizeof(A)];
Byte A::allocation_map[max_object_size] = {0};



void *A::operator new(size_t)

{

    Byte *ptr = (Byte *)memchr(allocation_map, 0, max_object_size);
    if (!ptr) {

        cout << "bellek yetersiz" << endl;
        throw bad_alloc();

    }

    int block_no = ptr - allocation_map;

    cout << block_no << "no lu blok kullanılıyor" << endl;

    *ptr = 1;

    return allocation_pool + (block_no * sizeof(A));

}


void A::operator delete(void *ptr)

{

    if (!ptr)

        return;

    unsigned long block =  reinterpret_cast<unsigned long>(ptr) -
reinterpret_cast<unsigned long>(allocation_pool);

    block /= sizeof(A);

    cout << block << "no lu blok geri veriliyor!" << endl;
    allocation_map[block] = 0;

}



int main()

{

    A *pa[A::max_object_size];
```

```
    try {

        for (size_t i = 0; i < A::max_object_size; ++i)
            pa[i] = new A;

        new A;

    }

    catch (bad_alloc) {

        cerr << "bellek yetersiz!" << endl;

    }



    delete pa[3];

    //geri verilen bellek alnı kullaniliyor!
    A *ptr = new A;

    delete ptr;



    for (size_t i = 0; i < A::max_object_size; ++i)
        delete pa[i];



    return 0;

}
```

Örnek programda *A* isimli bir sınıf tanımlanıyor. Sınıfın elemanı olan *char* türden *s* dizisi yalnızca sınıf nesnesinin belirli bir yer kaplaması amacıyla ekledik. Sınıfın *max_object_size* isimli *const* statik elemanı sınıf türünden dinamik olarak yaratılabilecek en fazla nesne sayısını tutuyor. Sınıfın iki statik dizi elemanı var. *Byte* türden *allocation_pool* isimli dizi sınıf nesnelerinin yaratılabileceği bellek havuzunu belirliyor. En fazla *allocation_pool* dizisinin boyutu kadar sayıda dinamik *A* nesnesi yaratılabilecek. *Byte* türden *allocation_map* dizisi ise *allocation_pool* dizisinin hangi bloklarının kullanımda olduğu bilgisini tutuyor. Yani bir bayrak dizisi olarak kullanılıyor. Dizinin *k* indisli elemanının değerinin *0* olması bellek havuzundaki ilgili bloğun henüz kullanımda olmadığını gösteriyor.

A sınıfı için tanımlanan *operator new* işlevi önce bayrak dizisini tarayarak bellek havuzunda henüz kullanılmayan bir blok olup olmadığına bakıyor. Bellek havuzundaki blokların tümü kullanımda ise yani havuzun tamamı kullanıma sunulmuş ise, *bad_alloc* sınıfı türünden bir hata nesnesi gönderiliyor. Bellek havuzunda en az bir nesnelik yer varsa, ilgili bloğun başlangıç adresi geri dönüş değeri olarak dışarı iletiliyor. Sınıfa ilişkin *operator delete* işlevi ise parametresine geçilen adresin hangi bellek bloğuna ilişkin olduğunu hesaplayarak bayrak dizisinin ilgili elemanının değerini sıfırlıyor. Bloğu yeniden kullanıma açıyor.

Yeri Belirli new İşleci

new işlecinin başka bir biçimi olan yeri belirli *new* işleci ile bir sınıf nesnesi istenilen bir adreste yaratılır. İşlecin kullanılması için *new* başlık dosyasının koda eklenmesi gerekir. Bu işleç kullanıldığında dinamik bir blok elde etmek için bir işlev çağrılmaz. Çünkü zaten sınıf nesnesi işlece verilen adreste yaratılır. İşlecin kullanımının genel biçimi aşağıdaki gibidir:

```
new(adres) tür ismi
```

İşlecin ayraç içindeki terimi herhangi türden adres olabilir. İşlecin ürettiği değer tür ismiyle belirlenen türden bir adrestir. Eğer tür bir sınıf türü ise derleyicinin ürettiği kod sonucunda o sınıfın kurucu işlevi çağrılır. Aşağıdaki kod parçasını inceleyin:

```
#include <iostream>
#include <new>


class A {
    int a, b;
public:
    A():a(0), b(0) {std::cout << "A::A()" << std::endl;}
    ~A(){std::cout << "A::A()" << std::endl;}
    void display()const {std::cout << "a = " << a << "\n" << "b = " << b <<
std::endl;}
};


int main()
{
    char s[sizeof(A)];
    A *ptr = new(s)A;

    ptr->display();
    ptr->~A();
}
```

Örnekte *sizeof(A) byte* boyutunda yerel dizi tanımlanıyor. Bu dizinin başlangıç adresinin *new* işlecine verildiğini görüyorsunuz. İşleç dışarıdan aldığı adresle sınıfın kurucu işlevinin çağrılmasını sağlıyor. Ancak daha sonra sınıfın sonlandırıcı işlevin çağrılması isteniyorsa çağrı açık bir şekilde yapılmalıdır:

```
ptr->~A();
```

deyimiyle *ptr* göstericisinin gösterdiği yapı nesnesinin sonlandırıcı işlevi çağrılıyor. C++ da bir sınıfın sonlandırıcı işlevinin açık bir şekilde çağrılması gereken tek yer burasıdır.

Eğer programcı

```
delete ptr
```

gibi bir deyim yazsaydı bu çalışma zamanına ilişkin bir hata olurdu. Dinamik olarak elde edilen bir blok olmamasına karşın, *operator delete* işleviyle *ptr* değişkeninin değeri olan adres *free store* alanaına geri verilmeye çalışılmış olurdu.

Yeri belirli *new* işlecinin de köşeli ayraçlı biçimi vardır. İşlecin bu biçimi kullanılarak, verilen bir adresten başlayarak birden fazla sınıf nesnesinin yaratılması sağlanabilir. Aşağıda daha önce yazılan *main* bu kez işlevin köşeli ayraç biçimini kullanacak biçimde yeniden tanımlanıyoruz:

```
int main()

{
    char s[3 * sizeof(A)];
    A *ptr = new(s)A[3];


    for (int k = 0; k < 3; ++k)
        ptr[k].display();

    for (int k = 0; k < 3; ++k)
        ptr[k].~A();


    return 0;

}
```

Bu kez yerel *s* dizisini, üç *A* nesnesinin sığacağı büyüklükte tanımladık. Yeri belirli *new* işleci ile bu kez *A* sınıf nesneleri başlangıç adresi verilen blokta yaratıldı. A sınıfının sonlandırıcı işlevinin her nesne için ayrıca çağrıldığına dikkat edin.

nothrow new İşleci

new ve *new[]* işleçlerinin *nothrow* parametreli bir biçimleri de vardır. Bu işleçler dinamik bellek alanı elde edilemediği zaman *bad_alloc* sınıfı türünden bir hata nesnesi göndermek yerine *NULL* adresi üretir. Bu işleçlerin dinamik blok elde etmek için çağırdıkları global işlevlerin bildirimleri şöyledir:

```
void *operator new(size_t, const nothrow_t &) throw();
void *operator new[](size_t, const nothrow_t &) throw();
```

Görüldüğü gibi bu işlevlerin bildirimi hata nesnesi belirleme sözdizimiyle yapılmış, bu işlevler içinden bir hata nesnesi gönderilmediği bildirilmiştir.

nothrow_t işlevlerin imzalarını farklı kılmak için tanımlanan boş bir yapı türünün ismidir.

new başlık dosyası içinde *nothrow_t* türünden ismi *nothrow* olan bir nesne bildirilmiştir.

```
struct nothrow_t {
}nothrow;
```

Bu işleçler aşağıdaki gibi kullanılır:

```
class A{
    //...
};

void func()
{
    A *ptr = new(nothrow)A;
    if(!ptr) {
        //...
    }
}
//...
```

new anahtar sözcüğünden sonra açılan ayraç içinde *nothrow* isminin yazıldığını görüyorsunuz. Daha sonra da bir tür ismi yer alıyor. Derleyicinin ürettiği kodla, dinamik bellek bloğu elde etmek için çağrılan işleve *sizeof(A)*

değeri ve *nothrow* nesnesi geçilir. Yukarıdaki *main* işlevinde *nothrow new* işleciyle, *A* sınıfı türünden bir nesne için dinamik bir blok elde ediliyor. İşlemin başarısı, işlecin ürettiği değerin *NULL* adresi olup olmadığı ile sınanıyor. Eğer dinamik bir blok elde edilirse, derleyicinin ürettiği kodla *A* sınıfının kurucu işlevi çağrılır. Aynı işlecin köşeli ayraçlı biçimiyle birden fazla sınıf nesnesi için de dinamik bir bellek bloğu elde edilebilir:

```
A *ptr = new(nothrow)A[10];
```

Yukarıdaki deyimin yürütülmesi sonucunda dinamik bellek bloğu elde etme girişimi başarılı olmaz ise *ptr* gösterici değişkenine *NULL* adresi atanır. İşlevin başarılı olması durumunda *A* sınıfının varsayılan kurucu işlevi her bir sınıf nesnesi için bir kez, yani toplam *10* kez çağrılır.

set_new_handler İşlevi

new işlecinin başarısız olması durumunda *bad_alloc* sınıfı türünden bir hata nesnesi gönderdiği açıklanmıştır. Ancak hata nesnesinin gönderilmesinden önce programcı tarafından gönderilen bir işlevin çağrılması sağlanabilir. Bu amaçla *set_new_handler* işlevi kullanılır.

set_new_handler işlevi, *new* işlecinin başarısızlığı durumunda bir hata nesnesi gönderilmesinden önce çağrılması istenen işlevin adresini argüman olarak alır. Yani işlevin parametre değişkeni bir işlev göstericisidir. İşlevin geri dönüş değeri de daha önce belirlenmiş bir işlevin adresidir:

```
typedef void (*new_handler)();
new_handler set_new_handler(new_handler);
```

Yukarıdaki bildirimleri inceleyelim:

new_handler geri dönüş değeri ve parametre değişkeni olmayan bir işlev adresi türüne verilen *typedef* ismidir.

set_new_handler işlevinin parametre değişkeni bu türdendir ve işlev de bu türe geri döner.

set_new_handler işleviyle bir işlev belirlemesi yapılmaz ise, *new* işleci başarısız olduğunda hiçbir işlevi çağırmaz doğrudan *bad_alloc* sınıfı türünden bir hata nesnesi gönderir.

operator new işlevi dinamik blok elde etme işleminde başarısız olduğunda, belirlenen işlevi yalnızca bir kez çağırmaz. Gereken bellek alanını buluncaya kadar bir döngü içinde sürekli çağırır. Bu kural *set_new_handler* işlevine sunulacak işlevin tasarımına bazı olanaklar sağlar. Bu işlev aşağıdaki biçimlerde tasarlanabilir:

1. İşlev daha fazla bir bellek alanını *operator new* işlevinin hizmetine sunabilir. Böylece işlev işini bitirdiğinde artık bellek yetersizliği söz konusu olmayacağından, *operator new* işlevi *bad_alloc* sınıfı türünden bir hata

nesnesi göndermez, işini başarıyla tamamlar. Program başladığında büyük bir bellek bloğu elde edilir, işlev bu bellek alanını küçük parçalar halinde *operator new* işlevinin kullanımına sunabilir.

2. *set_new_handler* işleviyle yeni bir işlev kayıt edilebilir. Böylece *operator new* bu kez yeni işlevi çağırır. Yeni kayıt edilen işlev bellek alanı elde etmek için başka iş yapabilir ya da başka bir algoritma kullanabilir, böylece *operator new* işlevi için bir bellek alanı yaratılabilir.

3. İşlev *set_new_handler* işlevini *NULL* adresiyle çağırır. Böylece ortada çağrılması gereken bir işlev kalmadığından *operator new* işlevi *bad_alloc* sınıfı türünden bir hata nesnesi gönderir.

4. İşlevin kendisi *bad_alloc* sınıfı türünden ya da *bad_alloc* sınıfından türetilmiş bir sınıf türünden hata nesnesi gönderir. Bu durumda gönderilen hata nesnesi *operator new* işlevi tarafından yakalanamaz. Böylece gönderilen hata nesnesi, *new* işlecinin kullandığı yerde ya da daha yüksek seviyede yakalanmaya bırakılabilir.

ÇOKLU TÜRETME

Bir sınıfı tek bir türetme işlemiyle birden fazla taban sınıftan türetmeye "çoklu türetme"

(multiple inheritance) denir.

```
class A {
    //…
};

class B {
    //…
};
class C : public A, public B {
    //…
};
```

Yukarıdaki kod parçasında *C* sınıfı, *A* ve *B* sınıflarından çoklu türetme yoluyla türetiliyor. Sözdizimde „:" atomundan sonra her bir taban sınıf isminin erişim belirteciyle birlikte yer aldığına dikkat edin. Eğer türetme işlemi

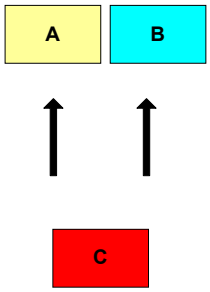
```
class C :public A, B {
    //…
};
```

biçiminde yapılsaydı, *C* sınıfı, A sınıfından *public* türetmesiyle ancak *B* sınıfından *private*

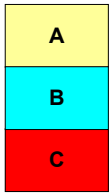
türetmesiyle türetilmiş olurdu.

Bir çoklu türetme işlemine ilişkin taban sınıfların sayısı hakkında bir kısıtlama yoktur. Ancak uygulamaların çoğunda bir sınıf iki ayrı taban sınıftan *public* türetmesi ile türetilir. Çoklu türetme ile elde edilmiş türemiş sınıf nesnesinin içinde her taban sınıfın elemanları da yer alır. Türetilmiş nesnelerin bellekteki yerleşimi standart olarak belirlenmemiştir.

Ancak derleyicilerin çoğu türemiş sınıf nesnesi içinde taban sınıf nesnelerini, önce bildirilen taban sınıf nesnesi daha düşük sayısal adreste olacak biçimde yerleştirir:



Yukarıdaki gibi bir türetme işleminden sonra türemiş sınıf olan C sınıfı türünden bir nesne tanımlandığı zaman, nesnenin bellekte yerleşimi aşağıdaki gibi olabilir:



Çoklu Türetmede Bilinirlik Alanı ve İsim Arama

Çoklu türetme durumunda türemiş sınıf, bütün taban sınıflara erişebilir. Taban sınıflara erişim konusunda, hiçbir taban sınıfın diğerine göre üstünlüğü yoktur. Bütün taban sınıflar aynı bilinirlik alanı içinde kabul edilir. Her iki taban sınıfta da aynı isimli bir eleman ya da üye işlev varsa, elemana erişim çözünürlük işleci ile yapılmamışsa bu durum çift anlamlılık hatasına yol açar. Türemiş sınıf bilinirlik alanında bir isim kullanıldığında bu isim önce türemiş sınıf bilinirlik alanında aranır. Eğer bulunamaz ise bu kez bir sıra gözetmeksizin, taban sınıfların her birinin bilinirlik alanı içinde aranır. Aşağıdaki örneği inceleyin:

```
#include <iostream>


class Base1 {
public:

    void func() {std::cout << "Base1::func()" << std::endl;}

};


class Base2 {
public:

    void func() {std::cout << "Base2::func()" << std::endl;}

};


class Mder : public Base1, public Base2 {
```

```
    //...
};


int main()

{
    Mder md;

    //md.func(); Geçersiz! Cift anlamlılık hatası
    md.Base1::func();    //Geçerli
    md.Base2::func();    //Geçerli


    return 0;

}
```

Yukarıdaki programda *Mder* sınıfı *Base1* ve *Base2* sınıflarından çoklu türetme ile elde ediliyor. Hem *Base1* hem de *Base2* sınıfları içinde *func* isimli işlevin bildirildiğini görüyorsunuz. *main* işlevi içinde tanımlanan *Mder* sınıf türünden *md* nesnesi ile, *func* işlevinin çağrılması derleme zamanında çift anlamlılık hatası oluşumuna neden olur. Ancak istenirse çözünürlük işlecinin kullanılmasıyla, yani ismin nitelenmesiyle çift anlamlılık hatası ortadan kaldırılabilir. İstenilen taban sınıfın *func* isimli işlevinin çağrılması sağlanabilir:

```
md.Base1::func()
```

çağrısıyla *Base1* sınıfının *func* işlevi

```
md.Base2::func();
```

çağrısıyla ise *Base2* sınıfının *func* işlevi çağrılabilir.

Çoklu türetmenin farklı kollarındaki aynı isimli işlevler arasında işlev yüklemesi (*function overloading)* yapılamaz. Çünkü imzaları farklı, aynı isimli işlevlerin bulunması aynı bilinirlik alanına özgüdür. Aşağıdaki kodu inceleyin:

```
#include <iostream>


class Base1 {
public:

    void func(int) {std::cout << "Base1::func(int)" << std::endl;}
```

```
};


class Base2 {
public:

    void func(double) {std::cout << "Base2::func(double)" << std::endl;}

};


class Mder : public Base1, public Base2 {

    //...

};


int main()

{

    Mder md;

    //md.func(3.7);  Geçersiz! Çift anlamlılık hatası
    md.Base1::func(5); //Geçerli.
    md.Base2::func(3.14); //Geçerli.


    return 0;

}
```

Yukarıdaki kodda, *Base1* sınfı içinde *double* türden parametre değişkenine sahip *func* isimli bir işlev tanımlanırken, *Base2* sınıfı içinde *int* parametreli aynı isimli bir işlev tanımlanıyor. *main* işlevi içinde tanımlanan *Mder* sınıf türünden *md* nesnesi ile yapılan

```
md.func(3.7);
```

çağrısı yine çift anlamlılık hatası oluşturur. Çünkü farklı bilinirlik alanlarında bulunan, parametrik yapıları birbirinden farklı olan *func* işlevleri için işlev yüklemesi yapılması mümkün değildir. Ancak istenirse nitelenmiş ismin kullanılmasıyla bu işlevlerden istenilen birinin çağrılması sağlanabilir:

```
md.Base1::func(5);
```

çağrısı ile *Base1* sınıfının *func* isimli işlevi çağrılırken

```
md.Base1::func(3.14);
```

çağrısı ile *Base2* sınıfının *func* isimli işlevi çağrılmış olur.

Taban Sınıfların Kurucu İşlevlerinin Çağrılması

Çoklu türetmede taban sınıfın varsayılan kurucu işlevleri, taban sınıfların bildirim sırasına göre çağrılır. Yani önce bildirilen taban sınıfın kurucu işlevi daha önce çağrılır. Türemiş sınıfın kurucu işlevlerin başına eklenen bir kodla, bildirim sıralarına göre tüm taban sınıfların kurucu işlevlerinin çağrılması sağlanır. Sonlandırıcı işlevlerin çağrısı ters sırada olur. Aşağıdaki örneği derleyerek çalıştırın ve kurucu/sonlandırıcı işlevlerin çağrılma sırasını gözlemleyin:

```
#include <iostream>


class Base1 {
public:
    Base1() {std::cout << "Base1::Base1()" << std::endl;}
    ~Base1() {std::cout << "Base1::~Base1()" << std::endl;}
};


class Base2 {
public:
    Base2() {std::cout << "Base2::Base2()" << std::endl;}
    ~Base2() {std::cout << "Base2::~Base2()" << std::endl;}
};
class Mder : public Base1, public Base2 {
public:
    Mder() {std::cout << "Mder::Mder()" << std::endl;}
    ~Mder() {cout << "Mder::~Mder()" << endl;}
};


void func()
{
    Mder mder;
```

```
}

int main()
{
    func();

    return 0;
}
```

Taban sınıfların varsayılan kurucu işlevlerinin değil de parametreli kurucu işlevlerinin çağrılması isteniyorsa bunun için *MIL* sözdizimi kullanılmalıdır.

Çoklu Türetilmiş Sınıflarda Türemiş Sınıf Taban Sınıf Dönüştürmeleri

Bir türemiş sınıfın *public* türetmesi yoluyla birden fazla taban sınıftan türetilmesi durumunda, türemiş sınıf nesnesi aynı zamanda taban sınıflardan herhangi birinin türünden nesneymiş gibi kullanılabilir. Örneğin bir türemiş sınıf nesnesinin adresi taban sınıflardan herhangi biri türünden göstericiye atanabilir. Şüphesiz taban sınıflardan biri türünden göstericiye türemiş sınıf türünden bir adres atandığında, atanan adres türemiş sınıf nesnesinin ilgili taban sınıf alt nesnesinin adresidir. Aşağıdaki örneği derleyip çalıştırın:

```
class Base1 {
    int b1;

public:

};


class Base2 {
    int b2;

public:

};


class Mder : public Base1, public Base2 {
    int m1;

public:

};

#include <iostream>

int main()

{
```

```
    Mder md;

    Base1 *base1_ptr = &md;
    Base2 *base2_ptr = &md;

    std::cout << "&md = " << &md << std::endl;

    std::cout << "base1_ptr = " << base1_ptr << std::endl;
    std::cout << "base2_ptr = " << base2_ptr << std::endl;


    return 0;
}
```

Programın ekran çıktısından da göreceğiniz gibi, dönüştürme sonunda adresin sayısal bileşeni değişebilmektedir. Daha sonra ters bir dönüşümün yapılması, yani adresin eski haline getirilmesi mümkün olmayabilir.

İki ayrı sınıftan çoklu türetmeyle elde edilmiş türemiş sınıf nesnesinin adresinin her iki taban sınıf türünden göstericiye de atanabilmesi, işlev yüklemesi sırasında çift anlamlılık hatasının oluşmasına neden olabilir.

Aşağıdaki gibi iki işlevin bildirildiğini düşünelim:

```
void display(const Base1 &)
{
    std::cout << "display(const Base1 &)" << std::endl;
}


void display(const Base2 &)
{
    std::cout << "display(const Base2 &)" << std::endl;
}


void foo()
{
    Mder md;

    display(md);    //Geçersiz! Çift anlamlılık hatası
}
```

Türemiş sınıf adreslerinin taban sınıf türlerinden adreslere dönüştürülmesinde, taban sınıfların birbirlerine göre üstünlükleri yoktur. *md* nesnesi ile yukarıdaki örnekte bildirilen her iki *display* işlevi de çağrılabilir. Bu durumda çift anlamlılık hatası oluşur. Çift anlamlılık hatası tür döüştürme işlecinin kullanılması ile ortadan kaldırılabilir:

```
void foo()

{

    Mder md;
    display(static_cast<Base1>(md));
    display(static_cast<Base2>(md));

}
```

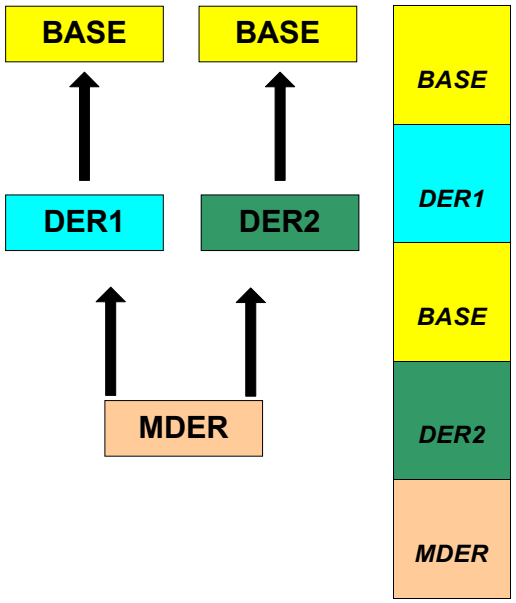
Elmas Oluşumu

Çoklu türetmelerde taban sınıfın bazen iki kopyası bulunur. Bu çoğu kez istenmeyen bir durumdur.

Aşağıdaki örneği inceleyin:

```
class Base {
    int a;

public:

    void func();

};


class Der1 : public Base {
public:

    //...

};


class Der2 : public Base {
public:

    //...

};




class Mder : public Der1, public Der2 {
public:

    //...

};
```

Yukarıdaki örnekte *Der1* ve *Der2* sınıfları *Base* sınıfından türetiliyor. *Mder* sınıfı da çoklu türetme yoluyla *Der1* ve *Der2* sınıflarından türetiliyor. Burada *Mder* sınıfından bir nesne tanımlanırsa nesnenin bellekteki görüntüsü aşağıdaki gibi olur:



Görüldüğü gibi *Base* sınıfı alt nesnesinden iki tane bulunmaktadır. Oysa bir tane bulunması daha istenen bir durumdur.

Bu yapıya "*elmas oluşumu"* denir *(Dreadful Diamond of Derivation).*

Elmas oluşumu bir çok durumda çift anlamlılık hatasına yol açar. Aşağıdaki *main* işlevini inceleyin:

```
int main()

{

    Mder md;

    //md.func();        //Geçersiz! Çift anlamlılık hatası
    md.Der1::func();

    md.Der2::func();
    return 0;

}
```

md.func() çağrısı geçersizdir. Çünkü hangi taban sınıftan alınan *func* işlevine erişildiği belli değildir. Ancak erişim için çözünürlük işleci ile kullanıldığında bir hata oluşmaz.

```
md.Der1::func(); //Der1 sınıfından alınan func işlevi çağrılıyor.
md.Der2::func(); //Der2 sınıfından alınan func işlevi çağrılıyor.
```

Mder sınıfı türünden bir nesne hem *Der1* hem de *Der2* sınıfı türünden nesne ise, normal olarak *Mder* sınıfı türünden bir nesnenin adresi *Base* sınıfı türünden bir göstericiye atanabilir. Ancak elmas oluşumu söz konusu ise böyle bir atama çift anlamlılık hatasına neden olur. Derleyici *Mder* sınıf türünden nesnenin içinde yer alan iki ayrı *Base* sınıfı türünden nesnenin hangisinin adresini *Base* sınıfından gösterici içine yerleştireceği konusunda seçim yapamaz. Aşağıdaki *main* işlevini inceleyin:

```
void func()
{
    Mder mder;
    Base *base_ptr = &mder; //Geçersiz! Çift anlamlılık hatası
}
```

Ancak tür dönüştürme işlemiyle, *Mder* sınıf nesnesi içinde yer alan *Base* sınıfı türünden nesnelerden hangisinin adresinin *Base* sınıf türünden göstericiye atanması gerektiği derleyiciye açıkça bildirilebilir:

```
Base *base_ptr = static_cast<Base *>(static_cast<Der1 *>(&mder));
```

Yukarıdaki ifade ile *mder* nesnesinin adresi önce *static_cast* tür dönüştürme işleciyle *Der1*

*** türüne dönüştürülüyor. Sonra elde edilen *Der1** türünden adres bu kez yine *static_cast*

işleciyle *Base ** türünden bir adrese dönüştürülerek *base_ptr* göstericisine atanıyor.

Elmas oluşumunun yol açtığı bir sorun da şudur: Taban sınıflardan herhangi birinin üye işlevi çağrıldığında bu üye işlevin ortak olan taban sınıfın elemanları üzerinde değişiklik yaptığını düşünelim. Böyle bir değişikliğin yapılmasından sonra, bu kez çoklu türetilmiş sınıf nesnesiyle diğer doğrudan taban sınıf nesnesinin bir üye işlevi çağrıldığında, yapılan değişiklik etkisiz kalır. Çünkü her doğrudan taban sınıf kendi taban sınıf nesneleri üzerinde işlem yapmaktadır. Doğrudan taban sınıfların taban sınıfları tek değildir.

Aşağıdaki örneği inceleyin:

```
#include <iostream>

class Base {
    int m_a;
public:
    Base():m_a(0){}
    void set(int val) {m_a = val;}
    int get() const {return m_a;}
};
```

```
class Der1: public Base {
public:

    void func(int x){set(x);}

};



class Der2: public Base {
public:

    void display()const {std::cout << get() << std::endl;}

};



class MDer: public Der1, public Der2 {
public:

    //…

};



int main()

{

    MDer mder;
    mder.func(135);
    mder.display();



    return 0;

}
```

Yukarıdaki örnekte *main* işlevi içinde *MDer* sınıfı türünden bir nesne tanımlanıyor. Bu nesne ile *MDer* sınıfının *Der1* sınıfından aldığı *func* isimli üye işlevi çağrılıyor. *Der1* sınıfının *func* üye işlevi *Base* sınıfının elemanı olan *m_a* değişkeninin değerini değiştiriyor. Daha sonra bu kez *MDer* sınıfının *Der2* sınıfından aldığı *display* üye işlevinin çağrıldığını görüyorsunuz. Bu işlev *Der2* sınıfının taban sınıfı olan *Base* sınıfının *m_a* elemanının değerini alıyor ve ekrana yazdırıyor.

```
mder.func(135);
```

çağrısı ile *m_a* elemanının değeri *135* yapılmasına karşın, daha sonra yapılan

```
mder.display();
```

çağrısıyla *m_a* elemanının değeri olarak ekrana *0* değeri yazılır. Çünkü *MBase* sınıfının doğrudan taban sınıfları olan *Der1* ve *Der2 Base* sınıfı türünden ayrı taban sınıf alt nesneleri üzerinde işlemler yapmaktadır.

Çoklu Türetme Sınıflarında Sanal İşlevlerin Kullanılması

Çoklu türetilmiş sınıf her bir taban sınıfının sanal işlevini ezebilirBu durumda çoklu türetmeyle türemiş sınıf için, sanal işleve sahip taban sınıf sayısı kadar işlev tablosu bulunması gerekir. Bu tür durumlarda çoklu türetilmiş sınıf nesnesinin adresini taban sınıflardan birine ilişkin göstericiye aktarıp o gösterici yoluyla o sınıfın sanal işlevi çağrıldığında, derleyici o göstericinin gösterdiği yerde sanal işlev tablo göstericisini arar.

```
#include <iostream>


class Base1 {
public:

    virtual void base1_vfunc() {std::cout << "Base1::base1_func()" <<
std::endl;}

};


class Base2 {
public:

    virtual void base2_vfunc() {std::cout << "Base2::base2_func()" <<
std::endl;}

};


class Mder : public Base1, public Base2 {
public:

    virtual void base1_vfunc() {std::cout << "Mder::base1_func()" <<
std::endl;}

    virtual void base2_vfunc() {std::cout << "Mder::base2_func()" <<
std::endl;}

    virtual void mder_vfunc()  {std::cout << "Mder::mder_func()" <<
std::endl;}

};


int main()

{

    Mder md;
```

```
    Base1 *base1_ptr = &md;
    Base2 *base2_ptr = &md;
    base1_ptr->base1_vfunc();
    base2_ptr->base2_vfunc();


    return 0;
}
```

Yukarıda yer alan *main* işlevinde *Mder* sınıfı türünden bir nesne olan *md* nesnesinin adresi hem *Base1* sınıfı türünden gösterici olan *base_ptr* nesnesine hem de *Base2* sınıfı türünden gösterici olan *base2_ptr* nesnesine atanıyor.

Hem *base1_ptr* hem de *base2_ptr* ile yapılan sanal işlev çağrıları sonunda, çağrılan *Mder*

sınıfının işlevleri olur.

Sanal Türetme

Çoklu türetmeye konu olan doğrudan taban sınıflar, ortak bir taban sınıftan türetilerek elde edilmişlerse, çoğul türetilmiş sınıf nesnesinin içinde iki ayrı ortak taban sınıf nesnesi yer alır. Çoğu zaman istenmeyen bu duruma *"Elmas oluşumu"* ismini vermiştik.

Ortak olan taban sınıfın çoklu türetilmiş sınıfta yalnızca bir kez yer alması isteniyorsa “sanal türetme” (*virtual inheritance)* denilen araç kullanılabilir. En son elde edilecek çoklu türetilmiş nesne, yalnızca bir kez yer alması istenen taban sınıftan türetme yapılırken *virtual* anahtar sözcüğünün kullanılmasıyla bu durum sağlanabilir. Aşağıdaki kodu derleyerek çalıştırın:

```
class Base {
    int a;

public:

    void func();

};


class Der1 : virtual public Base {
public:

    //...

};


class Der2 : virtual public Base {
public:
```

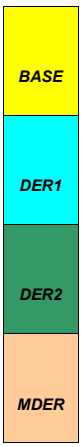
```
    //...
};


class Mder : public Der1, public Der2 {
public:
    //…
};
```

Der1 ve *Der2* sınıfları *Base* sınıfından "sanal" olarak türetiliyor. *virtual* anahtar sözcüğü *„:"* atomundan sonra yer alıyor. *virtual* anahtar sözcüğü *public* anahtar sözcüğünden sonra da yazılabilir.

Yukarıdaki türetme işlemlerinden sonra *Mder* sınıfı türünden bir nesne tanımlandığını düşünelim:

```
Mder md;
```

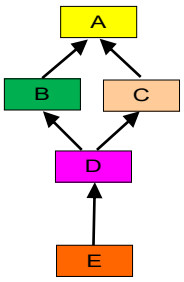
Bu nesnenin bellekte yerleşimi aşağıdaki gibi olur:



Sanal Taban Sınıfın Kurucu İşlevinin Çağrılması

Sanal türetmede kurucu işlevlerin çağrılması durumunda bir farklılık vardır. Çoklu türetilmiş bir sınıf nesnesi yaratıldığında ilk olarak sanal taban sınıfın kurucu işlevi çağrılır. Sanal taban sınıfın kurucu işlevini en alttaki türemiş sınıfın kurucu işlevi çağırmalıdır. En alttaki türemiş sınıf (*most derived class*) nedir?

A sınıfından sanal türetme yoluyla *B* ve *C* sınıflarının türetildiğini daha sonra *B* ve *C* sınıflarından çoğul türetme ile *D* isimli bir sınıfın türetildiğini düşünelim. Son olarak da *D* sınıfından *E* isimli bir sınıf türetilmiş olsun:



B sınıfının kurucu işlevi söz konusu olduğunda en alttaki türemiş sınıf *B* sınıfı olur. Yani *B*

sınıfının kurucu işlevi, ilk olarak *A* sınıfının kurucu işlevini çağırmalıdır.

D sınıfının kurucu işlevi söz konusu olduğunda bu kez en alttaki türemiş sınıf *D* sınıfı olur. Yani *D* sınıfının kurucu işlevi ilk olarak *A* sınıfının kurucu işlevini çağırmalıdır.

E sınıfının kurucu işlevi söz konusu olduğunda en alttaki türemiş sınıf E sınıfı olur. Yani E sınıfının kurucu işlevi ilk olarak *A* sınıfının kurucu işlevini çağırmalıdır.

Sanal taban sınıfın kurucu işlevinin çağrılması gerektiği durumlarda bir sorun oluşmaz. Çünkü derleyici zaten en alttaki türemiş sınıfın kurucu işlevinin koduna, sanal taban sınıfın (Örneğimizde *A* sınıfı) kurucu işlevinin çağrısını ekler. Ancak sanal taban sınıfın parametreli bir kurucu işlevi çağrılacaksa, bu en alttaki türemiş sınıfın kurucu işleviiçinde

M.I.L. sözdizimi ile sanal taban sınıfın parametreli kurucu işlevinin çağrılması ile olur. Aşağıdaki örneği derleyerek çalıştırın:

```
#include <iostream>


class Base {
public:

    Base() {std::cout << "Base::Base()" << std::endl;}

};
```

```
class Der1 : virtual public Base {
public:

    Der1() {std::cout << "Der1::Der1()" << std::endl;}

};

class Der2 : virtual public Base {
public:

    Der2() {std::cout << "Der2::Der2()" << std::endl;}

};


class Mder1: public Der1, public Der2 {
public:

    Mder1() {std::cout << "Mder1::Mder1()" << std::endl;}

};


class Mder2: public Mder1{
public:

    Mder2() {std::cout << "Mder2::Mder2()" << std::endl;}

};


int main()

{

    Mder2 mder2;
    Mder1 mder1;
    Der1 der1;
    Der2 der2;
    return 0;

}
```

Programcı açısından bu durumun önemi şudur: Türetme hiyerarşisi içindeki herhangi derinlikte bir sınıf, sanal taban sınıfın varsayılan kurucu işlevini değil de parametreli bir kurucu işlevinin çağrılmasını isterse, sanal taban sınıfın kurucu işlevini *M.I.L.* sözdizimiyle çağırmalıdır. Aşağıdaki örneği inceleyin:

```
class Vbase {

//...
public:

    Vbase(int);

//...

};
```

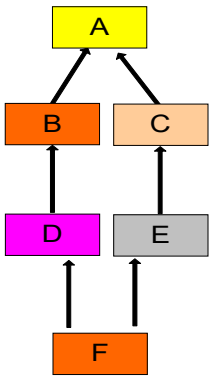
```
class Base1: virtual public Vbase {

//
public:

    Base1():Vbase(0) {}

};


class Base2: virtual public Vbase {

//...
public:

    Base1():Vbase(0) {}

//...

};


class Der1: public Base1, public Base2 {

//...
public:

    Der1():Vbase(0){}

};


class Der2: public Der1{

//...
public:

    Der2():Vbase(0){}

};
```

Yukarıdaki örnekte *Vbase* isimli sınıfın varsayılan kurucu işlevi yok. Sınıfın yalnızca tek parametreli bir kurucu işlevine sahip olduğunu görüyorsunuz. Bu sınıftan sanal türetme yoluyla *Base1* ve *Base2* sınıfları türetiliyor. *Base1* ve *Base2* sınıflarının kurucu işlevleri içinde *M.I.L.* sözdizimiyle sanal taban sınıfın tek parametreli kurucu işlevi çağrılıyor.

Base1 ve *Base2* sınıflarından çoklu türetme yoluyla *Der1* sınıfı türetiliyor. *Der1* sınıfının kurucu işlevinde de tepedeki sanal taban sınıf olan *Vbase* sınıfının tek parametreli kurucu işlevi *M.I.L* sözdizimi kullanılarak çağrılıyor. Aynı durum *Der1* sınıfından türetilen *Der2* sınıfı için de geçerlidir. Hangi kademede olursa olsun türetme hiyerarşisi içinde yer alan bir sınıf sanal taban sınıfın parametreli kurucu işlevini *MIL* sözdizimiyle çağırmak zorundadır. Aksi halde türemiş sınıf türlerinden herhangi biri türünden nesne tanımlanması durumunda derleme zamanında hata oluşur.

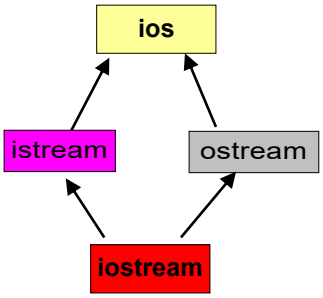
Çoklu Sanal Türetmede Sanal İşlevlerin Durumu

Çoklu sanal türetmede taban sınıflardaki bir sanal işlevin aşağıya doğru tek bir sonlanan işlevi (*final overrider*) olmalıdır. Örneğin aşağıdaki gibi bir türetme şeması söz konusu olsun:



A sınıfının *func* isimli bir sanal işlevinin olduğunu düşünelim. Bu işlev hem *B* sınıfında hem de *E* sınıfında ezilmişse (*override* edilmişse) bu durum hata oluşturur. Yalnızca *E* sınıfında ya da yalnızca F sınıfında ezilmişse bu durum geçerlidir.

C++'ın standart kütüphanesinde giriş çıkış işlemlerini yapacak sınıflarda çoklu türetme uygulanmıştır:



ios sınıfından sanal türetme yoluyla *istream* ve *ostream* sınıfları türetilmiş, *istream* ve

ostream sınıflarından da çoklu türetme ile *iostream* sınıfı türetilmiştir.

Bu durumda *C++'ın* standart *iostream* sınıf sisteminde *ios* taban sınıf alt nesnesinden yalnızca bir tane bulunur.

Burada *istream* sınıfı üzerinde işlem yapıldığında ve bu işlemler *ios* elemanlarını değiştirdiğinde *ostream* bu değişiklikleri görür.

Çoklu Türetmenin Bir Taban Sınıfın Arayüzünün Onarılması Amacıyla Kullanılması

Bazı durumlarda hizmet veren bir taban sınıfın arayüzü programcının elinde olmasına karşın sınıfın kodlama dosyası yani kaynak dosya programcının elinde bulunmaz. Bu durumda böyle bir sınıf üzerinde ancak kısıtlı değişiklikler yapılabilir. Sınıfa normal bir üye işlev eklenebilir ama sınıfa bir sanal işlev eklenemez. Sanal olmayan bir işlev sanal işlev haline getirilemez. Böyle bir durumda, hizmet veren taban sınıfda bu tür değişiklikler yapılması mümkün olmadığı için çoklu türetme aracından faydalanılır.

Onarmak amacıyla yapılan iş özetle şudur: Önce başka bir soyut taban sınıf oluşturulur. Bu soyut sınıfla elde ara yüzü olan hizmet veren taban sınıftan çoğul türetme yoluyla bir sınıf türetilir. Diğer türetmeler artık çoğul türetilmiş sınfıtan yapılır. Aşağıdaki örneği dikkatli bir şekilde izleyin:

Elimizde aşağıdaki gibi bir arayüzün olduğunu düşünelim:

```
////server.h
class ServerBase {
public:
    virtual void vfunc() const;   //sanal işlev
    void nvfunc()const;  //sanal olmayan işlev (sanal olmasını istiyoruz)
    ~Base3rd();        //sanal olmayan sonlandırıcı işlev(tasarım hatası)
};

class ServerDer: public ServerBase {
public:
    void vfunc() const;           //sanal işlev eziliyor
    ~Der3rd();
};

void global_func(const ServerBase3rd &); //Çok biçimli işleme yapan işlev
```

Yukarıda hizmet veren bir kaynak dosyanın arayüzü görülüyor. *ServerBase* sınıfının *vfunc* isimli sanal bir işlevi var. Ancak sınıfın *nvfunc* isimli üye işlev sanal olmayan bir işlevdir. Bu arayüzün onarılarak *nvfunc* işlevinin sanal bir işlev haline getirilmesi isteniyor. Yine sınıfın sonlandırıcı işlevi belki de ihmal sonucu sanal yapılmış. Oysa kullanıcı kodları *ServerBase* sınıfının sonlandırıcı işlevinin sanal sonlandırıcı işlev olmasını istiyor. Bir de belirli bir amacı gerçekleştirmesi amacıyla sınıfa *new_vfunc* isimli bir sanal işlevin eklenmesi isteniyor.

global_func1 işlevi ise taban sınıf referansı yoluyla çok biçimli işlem yapan global bir işlev.

server. *cpp* dosyası elde bulunmadığı için sınıfın kaynak kodlarını değiştirme şansımız olmadığını yeniden anımsatalım.

server.cpp Kaynak dosyasının da aşağıdaki gibi olduğunu düşünelim:

```
void ServerBase::vfunc()const
{
    cout << "ServerBase::vfunc()const" << endl;
    //...
```

```
}

void ServerBase::nvfunc()const
{
     cout << "ServerBase::nvfunc()const" << endl;
    //...
}

ServerBase::~ServerBase()
{
    cout << "ServerBase::~ServerBase()" << endl;
    //...
}

void ServerDer::vfunc()const
{
    cout << "ServerDer::vfunc()" << endl;
    //...
}

ServerDer::~ServerDer()
{
    cout << "Der3rd::~Der3rd()" << endl;
}

void global_func(const Base3rd &r)
{
    r.vfunc();
    r.nvfunc();
}
```

Sınama amacıyla önce ismi *NewBase* olan bir sınıf tanımlayalım:

```
class NewBase {
public:

    virtual void vfunc()const = 0;       //saf sanal işlev
    virtual void nvfunc()const = 0;      //saf sanal işlev

    //yeni arayuze iliskin işlev
    virtual void new_vfunc ()const = 0;

    virtual ~NewBase(){cout << "~NewBase()" << endl;}

};
```

NewBase isimli sınıfta *vfunc*, *nvfunc* ve *new_vfunc* isimli işlevlerin saf sanal işlev olarak bildirildiğini görüyorsunuz. Şimdi de ismi *Join* olan bir sınıfı *NewBase* ve *ServerBase* sınıflarından çoklu türetme yoluyla türeteceğiz:

```
class ServerBase {
public:

    virtual void vfunc()const;   //sanal işlev

    void nvfunc()const;    //sanal olmayan işlev (sanal olmasını istiyoruz)

    ~ServerBase();      //sanal olmayan sonlandırıcı işlev (tasarım hatası)

};


class ServerDer: public ServerBase {
public:

    void vfunc() const;     //sanal işlev eziliyor

    ~ServerDer();

};


#include <iostream>

using namespace std;

void global_func1(const ServerBase &);    //çokbiçimli işleme yapan işlev

//elimizde olmayan kodlama dosyası
void ServerBase::vfunc()const

{

    cout << "ServerBase::vfunc()const" << endl;

    //...
```

```
}

void ServerBase::nvfunc()const
{
    cout << "ServerBase::nvfunc()const" << endl;
    //...
}

ServerBase::~ServerBase()
{
    cout << "ServerBase::~ServerBase()" << endl;
    //...
}
void ServerDer::vfunc()const
{
    cout << "ServerDer::vfunc()" << endl;
    //...
}

ServerDer::~ServerDer()
{
    cout << "ServerDer::~ServerDer()" << endl;
}



void global_func(const ServerBase &r)
{
    r.vfunc();
    r.nvfunc();
}

class NewBase {
public:
    virtual void vfunc()const = 0;      //saf sanal işlev
```

```
    virtual void nvfunc()const = 0;     //saf sanal işlev

    //yeni arayuze iliskin işlev
    virtual void new_vfunc ()const = 0;

    virtual ~NewBase(){cout << "~NewBase()" << endl;}

};


class Join: public NewBase, public ServerBase {
public:

    void vfunc()const {cout << "Join::vfunc()" << endl;
ServerBase::vfunc();}

    void nvfunc()const {cout << "Join::nvfunc()" << endl;
ServerBase::nvfunc();}

    void new_vfunc ()const {cout << "Join::new_vfunc ()" << endl;}

    ~Join(){cout << "Join::~Join()" << endl;}

};



int main()

{

    Join &join_ref = *new Join;
    NewBase &nbase = join_ref;
    nbase.vfunc();

    cout << "*****************************************" << endl;
    nbase.new_vfunc();

    cout << "*****************************************" << endl;
    global_func(join_ref);

    cout << "*****************************************" << endl;
    delete &nbase;

    cout << "*****************************************" << endl;



    return 0;

}
```