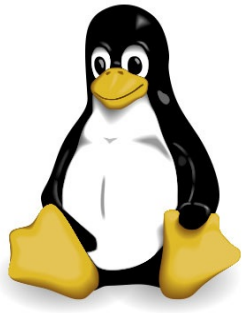


LINUX YAZILIM NOTLARI

Murat Demirten
Serkan Eser

Table of Contents

Giriş

Linux Yazılım Notları kitabında, birbiriyle doğrudan ilişkili olmayan çeşitli konularda yazılım geliştirme üst başlığında derlediğimiz notlarımız yer almaktadır.

FreeTDS ile SqlServer Bağlantısı

Linux tabanlı çözüm kümelerinde her ne kadar pek kullanım alanı bulmasa da, ticari dünyada zaman zaman Microsoft SQL Server veritabanı sunucusuna bağlanmanız ve üzerinde işlem yapmanız gerekebilmektedir.

Java gibi yüksek seviyeli dillerde **ODBC** sürücüleri üzerinden çeşitli çözümler olmakla beraber bu bölümde biz C üzerinden en alt seviyede *native* protokolü kullanarak SqlServer ile haberleşme konusu üzerinde duracağız.

FreeTDS

FreeTDS, **TDS** (Tabular Data Stream) protokolünün **LGPL** lisanslı özgür bir gerçekleştirimidir.

Tabular Data Stream (TDS), bir veritabanı sunucusu ile ona bağlı istemciler arasındaki iletişim ve veri transferini modelleyen, TCP/IP tabanlı bir protokoldür.

Protokol **Sybase Inc.** tarafından geliştirilmiş ve ilk olarak 1984 yılında Sybase SQL Server ürününde kullanılmıştır.

1990 yılında Sybase ve Microsoft firmalarının aralarında yapmış oldukları teknoloji işbirliği anlaşmasını takiben, Microsoft firması da Sybase SQL Server kodunu temel alarak kendi veritabanı sunucusu olan SQL Server ürününü geliştirdi. Bu nedenle Microsoft SQL Server ürününde de TDS protokolü kullanılmaktadır.

Linux platformlarında TDS protokolünün FreeTDS gerçekleştirimi oldukça kararlı durumda olup, C dilinin yanı sıra Php, Ruby, C++ vb. dillerde de alt katmanda FreeTDS kullanan farklı kütüphane alternatifleri mevcuttur.

TDS protokolünün **5.0** versiyonu Sybase tarafından dokümante edilmiş olmakla birlikte, diğer versiyonlarına dair bilgiler genel kullanıma açılmamıştı. 2008 yılında Microsoft, daha önce hayata geçirdiği **Open Specification Promise** doğrultusunda TDS protokol detaylarını genel kullanıma açtı ve bu tarihten sonra kütüphaneler daha güvenilir hale geldi.

> Not: TDS 5.0 versiyonu ile Sybase sunuculara bağlanılabiliyor olmasına karşın bu versiyon Microsoft tarafından desteklenmemektedir. Microsoft SQL Server bağlantıları için protokolün **7.X** versiyonları kullanılmalıdır.

Konsol İstemcisi - Sqsh

Linux sistemlerde kullanılmak üzere, Sybase tarafından geliştirilen **isql** konsol arayüzündeki temel fonksiyonaliye ve ek olarak kullanım kolaylığı açısından bazı yeni fonksiyonlara sahip **sqsh** uygulaması geliştirilmiştir. Uygulamayı paket yöneticinizle aşağıdaki gibi sisteminize kurabilirsiniz:

```
$ sudo apt-get install sqsh
```

Sqsh ile bir sunucuya bağlanırken temel olarak aşağıdaki parametreler kullanılır:

Parametre	Açıklama
-S	Sunucu adresi
-U	Kullanıcı Adı
-P	Parola (parametre olarak girilmez ise konsolda tekrar sorulacaktır)
-D	Veritabanı Adı

Örnek olarak *172.16.2.139* ip adresindeki **example_db** veritabanına *testuser* kullanıcı adı ve *tstpwd123* parolasıyla

bağlanalım ve *bolgeler* tablosundaki kayıtları görelim:

```
$ sqsh -S 172.16.2.139 -U testuser -P tstpwd123 -D example_db
sqsh-2.1.7 Copyright (C) 1995-2001 Scott C. Gray
Portions Copyright (C) 2004-2010 Michael Peppler
This is free software with ABSOLUTELY NO WARRANTY
For more information type '\warranty'

1> select * from bolgeler
2> go
 id          isim
 ----------- --------------------------------------
           1 Akdeniz B�lgesi
           2 Dogu Anadolu B�lgesi
           3 Ege B�lgesi
           4 G�neydogu Anadolu B�lgesi
           5 I� Anadolu B�lgesi
           6 Marmara B�lgesi
           7 Karadeniz B�lgesi
```

Yukarıdaki sonuç kümesine baktığımızda bazı karakterlerin düzgün görüntülenmediğini, bazılarının ise değiştirildiğini görmekteyiz (ğ -> g vb.)

Sorunun çözümü için sunucuya bağlantı kurarken kullanılacak karakter seti kümesi olarak UTF-8'i belirtmemiz gereklidir. Her ne kadar sqsh uygulamasının yardım sayfasında **-J UTF-8** gibi bir parametre geçirmek suretiye bu işlemin yapılabildiği yazsa da kullandığımız versiyonda (2.1.7) bu şekilde çözüm üretemedik. Karakter problemini, sunucu bazlı genel ayarlamaların yapılmasına imkan veren `freetds.conf` dosyası üzerinden yapacağımız tanımlamalarla çözeceğiz.

freetds.conf

FreeTDS kütüphanesi ile çalışırken öntanımlı olarak `/etc/freetds/freetds.conf` dosyası okunmaktadır.

Bu dosyada genel olarak kütüphanenin davranışını değiştirebilecek tanımlamalar bulunmaktadır. Ayrıca belirli bir SQL sunucu için özel ayarların da buradan yapılmasına imkan verilmektedir.

Dosyanın genel içeriği ve örnek sunucu bazlı tanımlamalar aşağıdaki gibidir:

```
[global]
        # TDS protocol version
;       tds version = 4.2

        # Whether to write a TDSDUMP file for diagnostic purposes
        # (setting this to /tmp is insecure on a multi-user system)
;       dump file = /tmp/freetds.log
;       debug flags = 0xffff

        # Command and connection timeouts
;       timeout = 10
;       connect timeout = 10

        # If you get out-of-memory errors, it may mean that your client
        # is trying to allocate a huge buffer for a TEXT field.
        # Try setting 'text size' to a more reasonable limit
        text size = 64512

# A typical Sybase server
[egServer50]
        host = symachine.domain.com
        port = 5000
        tds version = 5.0

# A typical Microsoft server
[egServer70]
        host = ntmachine.domain.com
        port = 1433
```

```
        tds version = 7.0

[mssql]
        host = 172.16.2.139
        port = 1433
        tds version = 7.0
        client charset = UTF-8
```

Yukarıda anlaşılabileceği üzere, tüm sunucuları etkileyecek ayarlar `[global]` bölümü altında yer almakta, aynı zamanda `[mssql]` örneğindeki gibi belirli bir sunucua isim verilerek (DNS ismi olması gerekmiyor), sunucu bazlı ek ayarlamalar yapma şansı da bulunmaktadır.

Örneğimizde **mssql** adında bir sunucu ismi tanımladık ve **client charset** değerini UTF-8 olacak şekilde değiştirdik.

Bu tanım sonrasında hem **sqsh** uygulamasından hem de **freetds** kullanan diğer uygulamalarda, sunucu isim/ip parametresinde **mssql** ismini kullanabilir ve konfigürasyon dosyasında bu bölümde belirtilmiş ayarların aktif olmasını sağlayabiliriz. Bir önceki **select** örneğimizi tekrar edelim:

```
$ sqsh -S mssql -U testuser -P tstpwd123 -D example_db
1> select * from bolgeler
2> go
 id          isim
 ----------- --------------------------------------
           1 Akdeniz Bölgesi
           2 Doğu Anadolu Bölgesi
           3 Ege Bölgesi
           4 Güneydoğu Anadolu Bölgesi
           5 İç Anadolu Bölgesi
           6 Marmara Bölgesi
           7 Karadeniz Bölgesi
```

.sqshrc

Sqsh ile çalışırken kullanım ortamınızı daha konforlu hale getirmek için ek ayarlamaları ev dizininiz altındaki `.sqshrc` dosyası üzerinden tanımlayabilirsiniz (henüz hiç ayar yapılmadı ise dosyanın oluşturulması gerekecektir)

Örneğin yukarıdaki çıktı formatı yerine öntanımlı MySQL konsol arayüzündekine benzer bir fomrmat kullanılmasını istiyorsanız, **go** komutunu **-m pretty** parametresi ile çalıştırmalısınız. Bu komutu her çalıştırdığımızda parametresini girmek zorunda kalmamak için bir **alias** tanımlayabiliriz.

Aşağıdaki satırı `~/.sqshrc` dosyanıza girin:

```
\alias go='\go -m pretty'
```

Şimdi tekrar bölge listesini sorgulayalım:

```
$ sqsh -S mssql -U testuser -P tstpwd123 -D example_db
1> select * from bolgeler
2> go
+=============+====================================================+
|          id | isim                                               |
+=============+====================================================+
|           1 | Akdeniz Bölgesi                                    |
+-------------+----------------------------------------------------+
|           2 | Doğu Anadolu Bölgesi                               |
+-------------+----------------------------------------------------+
|           3 | Ege Bölgesi                                        |
+-------------+----------------------------------------------------+
|           4 | Güneydoğu Anadolu Bölgesi                          |
```

```
+-------------+--------------------------------------------------+
|           5 | İç Anadolu Bölgesi                               |
+-------------+--------------------------------------------------+
|           6 | Marmara Bölgesi                                  |
+-------------+--------------------------------------------------+
|           7 | Karadeniz Bölgesi                                |
+-------------+--------------------------------------------------+
```

Diğer bazı kullanışlı örnekler için http://www.sypron.nl/sqsh.html adresine bakabilirsiniz.

Kütüphane Kullanımı

FreeTDS kütüphanesini C uygulamalarında kullanabilmek için aşağıdaki komutla geliştirme paketini sisteminize yükleyebilirsiniz:

```
$ sudo apt-get install freetds-dev
```

Aşağıdaki örnek uygulamayı mssql_connect.c adıyla kaydedip şu şekilde derleyebilirsiniz:

```
$ gcc -o mssql_connect mssql_connect.c -lsybdb
```

Örnek kodumuzu listeleyip önemli yerlerini detaylandırmaya çalışalım:

```c
/* mssql_connect.c */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sybfront.h>
#include <sybdb.h>
#include "../common/debug.h"

struct mssql_column {
    char *name;
    char *buffer;
    int type;
    int size;
    int status;
};

void display_usage (const char *name)
{
    printf("Usage: %s SERVER USER PASS DATABASE QUERY\n", name);
}

int database_mssql_errhandler (DBPROCESS * dbproc, int severity, int dberr,
        int oserr, char *dberrstr, char *oserrstr)
{
    (void) dbproc;
    (void) oserr;
    (void) oserrstr;

    if (dberr) {
        errorf("Sqlserver Msg %d, Level %d", dberr, severity);
        errorf("%s", dberrstr);
    } else {
        debugf("%s", dberrstr);
    }
    return INT_CANCEL;
}

int main (int argc, char *argv[])
{
    LOGINREC *login;
    DBPROCESS *dbproc;
```

```
    struct mssql_column *columns = NULL;
    struct mssql_column *pcol = NULL;
    int row_code;
    int ncols;
    int nrows;
    int ret;
    int c;

    if (argc != 6) {
        display_usage(argv[0]);
        exit(1);
    }

    const char *server      = argv[1];
    const char *username    = argv[2];
    const char *password    = argv[3];
    const char *database    = argv[4];
    const char *query       = argv[5];

    if (dbinit() == FAIL) {
        errorf("Couldn't init MSSQL library");
        exit(1);
    }
    /* Maximum 5 seconds for sql login */
    dbsetlogintime(5);

    /* Set error handler callback function */
    dberrhandle(database_mssql_errhandler);

    login = dblogin();
    dbsetluser(login, username);
    dbsetlpwd(login, password);
    if ( (dbproc = dbopen(login, server)) == NULL) {
        errorf("Couldn't open sqlserver db connection");
        exit(1);
    }
    if (dbuse(dbproc, database) == FAIL) {
        errorf("Couldn't change to db: %s", database);
        exit(1);
    }
    if (dbfcmd(dbproc, query)  == FAIL) {
        errorf("Couldn't create sql statement");
        exit(1);
    }
    if (dbsqlexec(dbproc) == FAIL) {
        errorf("Couldn't execute sql query");
        exit(1);
    }

    ncols = dbnumcols(dbproc);
    infof("%d columns found", ncols);

    while ( (ret = dbresults(dbproc)) != NO_MORE_RESULTS) {
        if (ret == FAIL) break;

        if ( (columns = calloc(ncols, sizeof(struct mssql_column))) == NULL) {
            errorf("Couldn't allocate columns");
            break;
        }

        for (pcol = columns; pcol - columns < ncols; pcol++) {
            c = pcol - columns + 1;
            pcol->name = dbcolname(dbproc, c);
            pcol->type = dbcoltype(dbproc, c);
            pcol->size = dbcollen(dbproc, c);
            if (pcol->type != SYBCHAR) {
                pcol->size = dbwillconvert(pcol->type, SYBCHAR);
            }
            debugf("col: %d, type: %d, size: %d, name: %s", c, pcol->type, pcol->size, pcol->name);
            if ( (pcol->buffer = malloc(pcol->size + 1)) == NULL) {
                errorf("Couldn't allocate space for row buffer");
                break;
            }
            if (dbbind(dbproc, c, NTBSTRINGBIND, pcol->size + 1, (BYTE*)pcol->buffer) == FAIL) {
                errorf("Couldn't bind to %s", pcol->name);
                break;
            }
            if (dbnullbind(dbproc, c, &pcol->status) == FAIL) {
```

```
                errorf("Couldn't make null bind to %s", pcol->name);
                break;
            }
        }

        while ((row_code = dbnextrow(dbproc)) != NO_MORE_ROWS) {
            switch (row_code) {
            case REG_ROW:
                for (pcol=columns; pcol - columns < ncols; pcol++) {
                    char *buffer = pcol->status == -1 ? "NULL" : pcol->buffer;
                    printf("%s: %s\t", pcol->name, buffer);
                }
                printf("\n");
                break;

            case BUF_FULL:
                errorf("buffer full");
                break;

            case FAIL:
                errorf("failed");
                exit(1);
                break;

            default:
                printf("Data for computeid %d ignored\n", row_code);
            }

        }
    }

    /* Free metadata and data */
    for (pcol = columns; pcol - columns < ncols; pcol++) {
        free(pcol->buffer);
    }
    free(columns);

    /* Get row count if available */
    if ( (nrows = dbcount(dbproc)) > -1) {
        debugf("Affected rows: %d", nrows);
    }

    dbclose(dbproc);
    dbfreebuf(dbproc);
    dbloginfree(login);

    return 0;
}
```

Kütüphanenin İlklendirilmesi: `dbinit`

FreeTDS kütüphanesinin kullanıldığı uygulamalarda, kütüphane içerisinden herhangi bir fonksiyon çağrılmadan önce, `dbinit()` fonksiyonunun çağrılmış olması şarttır.

`dbinit()` dahili bazı veri yapılarının doldurulmasını ve yerel spesifik tarih vb. format bilgilerini okumak için freetds içerisinden çıkan -varsa- `/etc/freetds/locales.conf` dosyasını okur.

> `locales.conf` dosyasının bu şekilde okunması *deprecated* bir özellik olmuştur. Güncel kütüphane versiyonları sistemin yerel (locale) ayarlarından bu bilgileri temin etmektedir. Ancak gene de `locales.conf` dosyası bulunursa işlenmektedir.

Bu işlemin uygulamanın *main* fonksiyonu içerisinde yapılmasında fayda vardır. Ancak herhangi bir sebeple *dbinit* işleminin bir fonksiyon içerisinden koşullu olarak sonradan yapılması gerekiyorsa, mutlaka statik bir değişkenle kütüphanenin ilklendirme işleminin daha önce yapılıp yapılmadığını tutmanız zorunludur. İlklendirme işleminin tekrar edilmesi, takibi zor hatalara yol açabilmektedir.

Hata İşleme: `dberrhandle`

Kütüphanenin hata ve uyarı durumlarında çağıracağı *callback* fonksiyonunu, `dberrhandle()` fonksiyonu ile belirtilmelidir.

Bu fonksiyonun prototipi aşağıdaki gibidir:

```
typedef int (*EHANDLEFUNC) (DBPROCESS * dbproc, int severity,
            int dberr, int oserr, char *dberrstr, char *oserrstr);
```

Fonksiyon çağrıldığında `dberr` parametresi 0'dan farklı ise, kritik bir veritabanı hatası olduğu anlaşılır.

Bağlantı Kurma ve Veritabanı Seçimi

Bağlantı kurmak için öncelikle kullanıcı adı ve parola bilgileri `LOGINREC` veriyapısı içerisine doldurulmalıdır.

Bunun için öncelikle `LOGINREC` tipinde bir değişken, `dblogin()` fonksiyonu ile ilklendirilir, ardından*dbsetluser* ve *dbsetlpwd* fonksiyonları ile ilgili parametreleri ayarlanır.

Sonraki adımda hazırlanan `LOGINREC` veri yapısı ve sunucu bilgisini (burada IP adresi, DNS üzerinden çözülebilen bir hostname veya `freetds.conf` içerisinde tanımlanmış bir sunucu adı kullanılabilir) parametre olarak alıp, geriye sürecin ilerleyen aşamalarında sürekli kullanılacak olan `DBPROCESS` handle döndürecek olan `dbopen()` fonksiyonu çağrılır.

Herhangi bir sebeple hata alınırsa, ilgili hata mesajının detayı hata işlemeleri için önceden belirlenmiş olan *callback* fonksiyonundan alınabilir.

Bağlantı zaman aşımı süresini kontrol altına almak isterseniz, `dbopen()` fonksiyonu çağırmadan önce `dbsetlogintime(int seconds)` prototipindeki fonksiyonu kullanarak saniye cinsinden bir limit de tanımlayabilirsiniz.

Bağlantı gerçekleştikten sonra `dbuse()` fonksiyonu ile üzerinde çalışılacak olan veritabanı seçimi işlemi yapılmaktadır.

Sorgu Çalıştırma ve Yanıt İşleme

Veritabanı üzerinde çalıştırılacak olan sorgu, öncelikle `dbfcmd()` fonksiyonu ile hazırlanır. Ardından `dbsqlexec()` fonksiyonu ile çalıştırılır.

Sorgu bu şekilde işletildikten sonra geriye dönen değerlerin saklanacağı uygun veri yapıları oluşturulmalıdır. Bunun için uygulama kaynak kodumuzun ilk bölümünde, `struct mssql_column` şeklinde bir yapı tanımladık. Bu yapıyı ihtiyaçlarınız doğrultusunda genişletebilirsiniz.

Tanımladığımız yapıyı, işletmiş olduğumuz sorgunun yanıt kümesindeki her bir sütun ile ilgili veri tipi, uzunluk ve sütun ismi bilgilerini işlemek için kullanacağız.

Örneğimizi geri dönen sütun sayısını önceden bilemeyeceğimiz, her türlü sorgu için çalışacak şekilde hazırladık. Dolayısıyla öncelikle göndermiş olduğumuz sorgu yanıtının kaç sütundan oluştuğunu öğrenmemiz gerekiyor. Bu işlem için `dbnumcols()` fonksiyonunu kullanıyoruz.

Sütun sayısını öğrendikten sonra ilgili bilgileri hazırlamış olduğumuz `struct mssql_column` veri yapısında saklamak üzere bellekte yer ayırıyoruz.

Ardından sorgu yanıtındaki satırları işlemeye geçmeden hemen önce, sütunlarla ilgili sütun ismi, tipi ve veri uzunluğu bilgilerini sırasıyla `dbcolname()`, `dbcoltype()` ve `dbcollen()` fonksiyonlarıyla elde ediyoruz.

Sütun ile ilgili bilgileri bu şekilde öğrendikten sonra, hazırlamış olduğumuz veri yapısında yanıtları saklayacağımız yerleri hazırlıyoruz. Bu noktada kodumuzu kısa tutmak adına, metin dışındaki tiplerin, `dbwillconvert()` fonksiyonuyla metin tabanlı bir formata dönüştürüldüğünde gereken uzunluğu hesaplatıp, metne dönüştüğü zamanki uzunluğu için yetecek kadar bellekte alan açıyoruz. Örnek olarak 4 byte'lık `INT` tipindeki bir sütun için `dbwillconvert()` sonrası sütun boyutunun **11**

olarak geleceğini göreceğiz zira 4 byte'lık bir işaretli INT değerini metne dönüştürüp saklayabilmek için 11 byte uzunluğunda bir alan gereklidir.

Gerçek ortamda gönderdiğiniz sorgularla ilgili bilgi sahibi olacağınızdan, tüm sütunları metin tabanlı dönüşüme zorlamak yerine, `struct mssql_column` veri yapısı içerisindeki genel amaçlı `buffer` değişkenini bir `union` yapısı ile değiştirip, sütun tipine göre `union` içerisinde INT, FLOAT vb. veri tipleri kullanabilir ve metin dönüşümü yapmadan doğrudan bu alanların içerisine yazılmasını sağlayabilirsiniz.

Bellekteki alanlar hazır edildikten sonra her bir sütunu `dbbind()` fonksiyonu ile yanıt setine nasıl bağladığımızı belirtmemiz gerekiyor. Örneğimizde *bind* tipi olarak hep `NTBSTRINGBIND` değerini kullandık. Yukarıdaki ek notumuz doğrultusunda eğer metin dönüşümü uygulamayacaksanız bunun yerine `INTBIND`, `REALBIND`, `BIGINTBIND` vb. diğer veri tipleri için uygun *binding* değerlerini de kullanabilirsiniz.

Her bir sütun için gerekli bind işleminin yanı sıra *NULL* değerler için de `dbnullbind()` fonksiyonuyla bir adet *binding* işleminin daha yapılması gereklidir.

> Not: Sütun ve binding tipleri için sabitler, kütüphane içerisinden çıkan `sybdb.h` dosyası içerisinde yer almaktadır.

Şimdi artık sıra satırları işlemeye geldi. Bunun için `dbnextrows()` fonksiyonu `NO_MORE_ROWS` değeri döndürmediği müddetçe iterasyonla tüm bilgileri alabiliriz.

Örnek uyguladığımızda her bir satırda aldığımız değerleri, sütun ismi ile birlikte ekrana bastırdık.

Yanıt satırlarının işlenmesi tamamlandıktan sonra sistem kaynaklarını serbest bırakıyoruz.

Bağlantının Sonlandırılması

Veritabanı ile ilgili işlemlerimiz tamamlandıysa açık olan bağlantımızı `dbclose()` fonksiyonuyla kapatmamız gerekir.

Son olarak kullandığımız `DBPROCESS` ve `LOGINREC` değişkenlerini de `dbfreebuf()` ve `dbloginfree()` fonksiyonlarıyla da geride artık kalmayacak şekilde temizliyoruz.

Örnek Kullanım

Hazırlamış olduğumuz uygulama ile bir miktar veri içeren `bolgeler` ve `iller` adında 2 tablo üzerinde INNER JOIN sorgusu çalıştıralım:

```
$ ./mssql_connect mssql testuser tstpwd123 example_db \
    "SELECT iller.*, bolgeler.isim AS bolge_adi FROM iller \
    INNER JOIN bolgeler ON iller.bolge_id=bolgeler.id ORDER BY isim"
info: 5 columns found (main mssql_connect.c:91)
debug: col: 1, type: 56, size: 11, name: id (main mssql_connect.c:110)
debug: col: 2, type: 47, size: 8, name: plaka (main mssql_connect.c:110)
debug: col: 3, type: 56, size: 11, name: bolge_id (main mssql_connect.c:110)
debug: col: 4, type: 47, size: 400, name: isim (main mssql_connect.c:110)
debug: col: 5, type: 47, size: 200, name: bolge_adi (main mssql_connect.c:110)
id: 13  plaka: 06  bolge_id: 5  isim: Ankara    bolge_adi: İç Anadolu Bölgesi
id: 11  plaka: 07  bolge_id: 1  isim: Antalya   bolge_adi: Akdeniz Bölgesi
id: 1   plaka: 08  bolge_id: 7  isim: Artvin    bolge_adi: Karadeniz Bölgesi
id: 9   plaka: 16  bolge_id: 6  isim: Bursa     bolge_adi: Marmara Bölgesi
id: 4   plaka: 28  bolge_id: 7  isim: Giresun   bolge_adi: Karadeniz Bölgesi
id: 7   plaka: 34  bolge_id: 6  isim: İstanbul  bolge_adi: Marmara Bölgesi
id: 2   plaka: 53  bolge_id: 7  isim: Rize      bolge_adi: Karadeniz Bölgesi
id: 6   plaka: 55  bolge_id: 7  isim: Samsun    bolge_adi: Karadeniz Bölgesi
id: 15  plaka: 58  bolge_id: 5  isim: Sivas     bolge_adi: İç Anadolu Bölgesi
id: 3   plaka: 61  bolge_id: 7  isim: Trabzon   bolge_adi: Karadeniz Bölgesi
id: 8   plaka: 77  bolge_id: 6  isim: Yalova    bolge_adi: Marmara Bölgesi
debug: Affected rows: 15 (main mssql_connect.c:159)
```

> Not: `debug.h` dosyasını Kaynak Dosyalar bölümünden edinebilirsiniz.

İçsel ve Anonim Fonksiyonlar

İçsel ve onların isimsiz halleri olan anonim fonksiyonlar, başka fonksiyonların içinde tanımlanan fonksiyonlardır. Global fonksiyonlara göre daha dar bir bilinirlik alanına sahip olan bu fonksiyonlar genel olarak davranış değiştirmek ve olay dinlemek amacıyla başka fonksiyonlara *callback* olarak geçirilirler.

Birçok dilde yaygın bir kullanıma sahip bu fonksiyonlar C standartlarında yer almamaktadır. Buna karşın içsel fonksiyonlar GNU C eklentisi olarak desteklenmektedir. Anonim fonksiyonlar ise C++11 standartları ile C++ diline dahil edilmiştir.

Daha önce söylediğimiz gibi anonim fonksiyonlar (*anonymous functions*), bir isme sahip içsel fonksiyonların (*nested functions*) bir formudur. Tanımlanmaları ve çağrılmalarında bazı farklılıklar bulunmaktadır.

Biz ilk önce içsel fonksiyonların GNU C eklentisi olarak nasıl oluşturulduğuna, ardından anonim fonksiyonların C++ dilinde nasıl ele alındığına bakacağız. İncelemelerimizde GNU geliştirme araçlarından faydalanacağız.

İçsel Fonksiyonlar

GNU C eklentilerine göre bir içsel fonksiyon aşağıdaki gibi tanımlanabilir.

```
int main() {
    int local = 0;
    void inner() {
        ++local;
    }
}
```

> **Not:** İçsel fonksiyonlar GNU C tarafından desteklenmesine karşın GNU C++ tarafından desteklenmemektedir.

İçsel *inner* fonksiyonu tanımlandığı fonksiyon bloğunda çağrılabildiği gibi adresi başka bir fonksiyona geçirilerek dışarıdan da çağrılabilir. Sırasıyla bu iki çağırma biçimini inceleyeceğiz.

Örnek kodda görüldüğü gibi *inner* fonksiyonu kendi yerel bilinirlik alanında olmayan, dıştaki *main* fonksiyonunun yerel değişkeninin değerini değiştirmektedir. Bu doğal olmayan kullanım şeklinin nasıl gerçekleştirildiğini incelemek için derleyicinin ürettiği sembolik makina koduna bakacağız. İncelemelerimizde 32 bit mimari hedefli sembolik makina kodu kullanacağız.

> **Not:** 64 bitlik bir sistem kullanıyorsanız derleyicinize **m32** anahtarı geçirerek 32 bitlik kod üretmesini sağlayabilirsiniz. 64 bitlik sistemde 32 bitlik kod üretebilmek ve çalıştırabilmek için ekstradan paketlere ihtiyaç duyulacaktır. Ubuntu 14.04.1 LTS için **libc6-i386** ve **lib32stdc++-4.8-dev** paketleri sisteme kurulmuştur.

İçsel Fonksiyonun Tanımlandığı Blok İçinde Çağrılması

```
#include <stdio.h>
int main() {
    int local = 0;
    void inner() {
        ++local;
    }
    inner();
    printf("%d\n", local);
}
```

Yukarıdaki kodu *inner.c* adıyla saklayıp aşağıdaki gibi derleyebilirsiniz.

```
gcc -oinner inner.c -m32 --save-temps
```

--save-temps anahtarı ile derleyicinin ürettiği ara kodlar uygun ad ve uzantılarla dosya sistemine kaydedilmektedir. *inner.c* için derleyici aşağıdaki dosyaları üretecektir.

Dosya Adı	İçerik
inner.i	Önişlemcinin ürettiği kod
inner.s	Derleyicinin ürettiği sembolik makina kodları
inner.o	Gerçek makina kodlarını içeren ELF formatlı amaç kod
inner	Çalıştırılabilir ELF formatlı kod

> **Not:** Komut satırından kullandığımız **gcc** uygulaması aslında derleyici değil, derleme sürecinde gerekli olan uygulamaları uygun sıra ve parametrelerle çağıran bir sürücü (*driver*) programdır. Bir C kodu çalıştırılabilir hale gelene kadar, temel olarak, aşağıdaki aşamalardan geçmektedir.
>
> - Önişlem aşaması
> - Derleyeci tarafından sembolik makina kodlarının üretilmesi
> - Assembler tarafından gerçek makina kodlarının üretilmesi
> - Linker tarafından çalıştırılan dosyanın üretilmesi
>
> Fakat biz burada detaya girmeden bütün bu süreçten derleme süreci olarak bahsedeceğiz.

Örnek kod derlenip çalıştırıldıktan sonra terminal ekranına **1** değerini basacaktır.

Sembolik makina kodlarını incelemeye başlamadan önce, **binutils** paketinden çıkan **nm** ve **readelf** araçları ile amaç dosyadaki sembollere bakalım.

```
$ nm inner.o

00000000 t inner.1826
0000000e T main
         U printf
```

```
$ readelf -s inner.o

Symbol table '.symtab' contains 12 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS inner.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000    14 FUNC    LOCAL  DEFAULT    1 inner.1826
     6: 00000000     0 SECTION LOCAL  DEFAULT    5
     7: 00000000     0 SECTION LOCAL  DEFAULT    7
     8: 00000000     0 SECTION LOCAL  DEFAULT    8
     9: 00000000     0 SECTION LOCAL  DEFAULT    6
    10: 0000000e    51 FUNC    GLOBAL DEFAULT    1 main
    11: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
```

Yazdığımız *inner* fonksiyonunun başlangıç adresinin **inner.1826** sembolüyle temsil edildiğini görmekteyiz. Derleyici, global isim alanındaki aynı isimli bir fonksiyondan ayırmak için, fonksiyon adının sonuna ürettiği bir sayı eklemiş ve bu sembolü dışsal bağlanıma(*external linkage*) kapatmış. Bu aşamadan sonra sembolik makina kodlarını inceleyebiliriz. **.cfi** ile başlayan assembler direktiflerini göz ardı ettiğimizde *main* fonksiyonu için derleyicinin aşağıdaki gibi bir kod ürettiğini görmekteyiz. Üretilen sembolik makina kodunun **AT&T** sözdiziminde olduğuna dikkat ediniz.

> **Not:** Derleyiciye **-masm=intel** anahtarını geçirerek Intel sözdizimine uygun sembolik makina kodu üretmesini sağlayabilirsiniz.

```
main:
 1          pushl   %ebp
 2          movl    %esp, %ebp
 3          andl    $-16, %esp
 4          subl    $32, %esp
 5          movl    $0, %eax
 6          movl    %eax, 28(%esp)
 7          leal    28(%esp), %eax
 8          movl    %eax, %ecx
 9          call    inner.1826
 10         movl    28(%esp), %eax
 11         movl    %eax, 4(%esp)
 12         movl    $.LC0, (%esp)
```

```
13          call    printf
14          leave
15          ret
```

Baştaki 4 ve sondaki 2 makina kodu derleyici tarafından yazılan başlangıç(*prologue*) ve bitiş(*epilogue*) kodlarıdır. Başlangıç kodları genel olarak yığının ve yazmaçların hazırlanmasından, bitiş kodları ise yazmaçların eski durumlarına yüklenmesinden sorumludur.

32 bit sistemlerde yığının tepe noktası *esp* yazmacında tutulmakta ve yığın genel olarak büyük adresten küçük adrese doğru genişlemektedir. Başlangıç kodlarına baktığımızda 4 numaralı komut ile *main* fonksiyonu için yığında 32 byte'lık bir alan ayrıldığını görmekteyiz.

Bu aşamada başlangıç ve bitiş kodları arasındaki kodlar asıl ilgilendiğimiz kısmı oluşturmaktadır. İlk önce *main* sonrasında ise *inner* fonksiyonuna ait kodları tek tek inceleyelim. Sembolik makina kodlarını yorumlamak bir miktar aşinalık gerektirmektedir, burada mümkün olduğunca detaya girmeden komutların yaptıklarıyla ilgileneceğiz.

Sembolik makina kodu	İşlevi
`movl $0, %eax`	eax yazmacına 0 değeri yerleştirilmiş
`movl %eax, 28(%esp)`	eax yazmacındaki 0 değeri, yığının başlangıcından itibaren 28 byte uzaklıktaki güvenli bir bölgeye yerleştirilmiş. Bu alan C kodundaki yerel lokal değişkenine karşılık gelmektedir. Otomatik ömürlü yerel değişkenlerin sabit(hardcoded) adreslere sahip olmayıp, yazmaç göreli(register relative) adreslere sahip olduğunu hatırlayınız
`leal 28(%esp), %eax`	yığında yerel değişken için ayrılmış alanın adresi eax yazmacına atanmış
`movl %eax, %ecx`	eax yazmacının değeri yani yerel değişken adresi ecx yazmacına kopyalanmış
`call inner.1826`	inner fonksiyonu çağrılmış

Buraya kadar olan sembolik makina komutlarının C dilindeki karşılığının aşağıdaki gibi olduğunu söyleyebiliriz.

```
int local = 0;
inner();
```

Bundan sonraki bitiş kodlarına kadar olan komutlar yerel değişkenin değerinin *printf* ile bastırılmasına ilişkindir.

Son durumda *ecx* yazmacında yerel değişkenin adresi bulunmakta ve *inner* fonksiyonu çağrılmakta. *inner* fonksiyonuna ait sembolik makina kodları ise aşağıdaki gibidir.

```
inner.1826:
1           pushl   %ebp
2           movl    %esp, %ebp
3           movl    %ecx, %eax
4           movl    (%eax), %edx
5           addl    $1, %edx
6           movl    %edx, (%eax)
7           popl    %ebp
8           ret
```

Başlangıç ve bitiş kodları arasındaki kodları adım adım inceleyelim.

Sembolik makina kodu	İşlevi

`movl %ecx, %eax`	ecx yazmacındaki değer eax yazmacına kopyalanmış. eax yazmacı artık *main* fonksiyonunun yerel değişkeninin adresini tutmaktadır
`movl (%eax), %edx`	eax yazmacının gösterdiği bellek adresindeki değer, yani main fonksiyonunun yerel değişkeninin değeri, edx yazmacına yazılmış
`addl $1, %edx`	edx yazmacının değeri 1 arttırılmış
`movl %edx, (%eax)`	edx yazmacındaki değer eax yazmacının gösterdiği bellek adresine yani *main* fonksiyonunun yerel değişkenine yazılmış

Özetleyecek olursak, içsel fonksiyonun tanımlandığı blok içinde çağrıldığı durumda, derleyici dıştaki fonksiyona(outer function) ait yerel değişkenin adresini *ecx* yazmacında saklamakta ve içsel fonksiyonda *ecx* yazmacını kullanarak kendini çağıran fonksiyonun yerel değişkeninin adresine ulaşmaktadır.

İçsel Fonksiyonun Dışarıdan Çağrılması

İncelememize örnek bir kod üzerinden başlayalım.

```
#include <stdio.h>

typedef void (*PF) ();

void foo(PF f) {
    //diğer işlemler..
    f();
}

int main() {
    int local = 0;
    void inner() {
        ++local;
    }
    foo(inner);
    printf("%d\n", local);
    return 0;
}
```

Örnekte içsel *inner* fonksiyonunun adresi *foo* isimli başka bir fonksiyona geçirilmekte ve bu şekilde dışsal olarak çağırılmaktadır. Kod derlenip çalıştırıldığında yine bir önceki *1* sonucunu üretecektir.

İçsel fonksiyonun tanımlandığı fonksiyon içinde çağrıldığı durumda, dıştaki fonksiyona ait yerel değişken adresinin *ecx* yazmacında saklandığını ve içsel fonksiyonun yerel değişken adresine *ecx* üzerinden ulaştığını hatırlayınız. Buradaki örnekte ise içsel fonksiyon başka bir fonksiyon tarafından çağırılmakta. Bu durumda içsel fonksiyonun adresinin geçirildiği *foo* fonksiyonunun, içsel fonksiyon çağrısından önce, *ecx* yazmacındaki değeri bozmayacağının bir garantisi yoktur. *foo* fonksiyonu kendisine geçirilen adresin içsel bir fonksiyona ait olup olmadığı bilgisine sahip değildir, kaldı ki *foo* fonksiyonu bir kütüphane fonksiyonu olabilir. Bu durumda *ecx* yazmacına yerel değişkenin adresin yazmak yeterli olmayacaktır.

Örnek kod için derleyicinin ürettiği sembolik makina koduna bakarak oluşan durumu inceleyelim. Örnek uygulama bir öncekine benzer şekilde aşağıdaki gibi derlenebilir.

```
gcc -oinner inner.c -m32 --save-temps -fno-stack-protector
```

Son argümanı derleyicinin yığın taşmalarını(stack overflow) tespit edebilmek için fazladan yazdığı kodları yazmasını engellemek için ekledik. Derleyici içsel fonksiyon için bir öncekiyle aynı kodu yazmasına karşın, *main* fonksiyonunun kodunun bir hayli değiştiğini görmekteyiz.

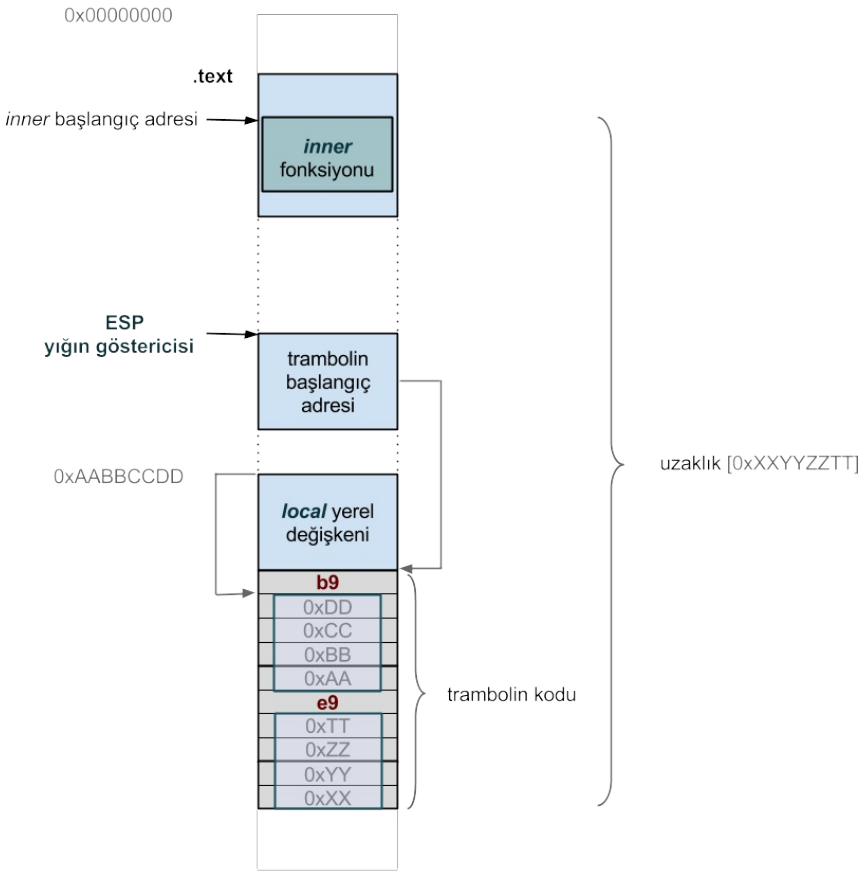
```
main:
        pushl   %ebp
```

```
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $32, %esp
        leal    16(%esp), %eax
        addl    $4, %eax
        leal    16(%esp), %edx
        movb    $-71, (%eax)
        movl    %edx, 1(%eax)
        movb    $-23, 5(%eax)
        movl    $inner.1830, %ecx
        leal    10(%eax), %edx
        subl    %edx, %ecx
        movl    %ecx, %edx
        movl    %edx, 6(%eax)
        movl    $0, %eax
        movl    %eax, 16(%esp)
        leal    16(%esp), %eax
        addl    $4, %eax
        movl    %eax, (%esp)
        call    foo         #foo çağrısı
        movl    16(%esp), %eax
        movl    %eax, 4(%esp)
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        ret
```

main fonksiyonuna bakıldığında **-71** ve **-23** olmak üzere iki adet negatif değerin *eax* yazmacı referans alınarak belleğe yazıldığı görülmektedir. Negatif sayıların bellekte ikiye tümlenmiş halleriyle tutulduğunu hatırlayınız.

> **Not:** Bir sayının ikiye tümleyenini bulmak için, ikili sayı sisteminde temsil edilen sayının, 1 olan bitleri 0 ve 0 olan bitleri 1 yapılarak önce bire tümleyeni alınır. Sonrasında elde edilen sonuç 1 ile toplanarak ikiye tümleyenine ulaşılır.

-71 ve **-23** sayıları, birer byte ile, bellekte sırasıyla **0xb9** ve **0xe9** şeklinde tutulurlar. *main* için yığında yer ayrılmasından, *foo* fonksiyonu çağrısına kadar olan kodlar işletildiğinde yığının son hali aşağıdaki gibi olacaktır.



Dikkat edilecek olursa, beklentinin tersine, *foo* fonksiyonuna argüman olarak **.text** alanında bulunan içsel fonksiyonun başlangıç adresi geçirilmek yerine, yığında içeriği **0xb9** ile başlayan bölgenin adresi geçirilmiştir.

Derleyicinin *foo* için ürettiği kod ise aşağıdaki gibidir.

```
foo:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    8(%ebp), %eax
        call    *%eax
        leave
        ret
```

foo fonksiyonu kendisine argüman olarak geçirilen adresi *eax* yazmacına yazmış ve sonrasında o adrese dolaylı çağrı(indirect call) yapmıştır. Bu durumda *foo* fonksiyonu direkt olarak içsel *inner* fonksiyonuna çağrı yapmak yerine yığında güvenli bir bölgeye çağrı yapmaktadır. Bu noktadan sonra işlemci yığındaki kodları işleyecektir.

> **Not:** Yığındaki bir kodun çalıştırılabilmesi için yığının çalıştırılabilir(*executable stack*) olarak işaretlenmesi gerekmektedir. Bu işlem çoğunlukla bağlayıcı(*linker*) tarafından, sembolik makina kodundaki direktiflere bakılarak yapılır. Bağlayıcı program *ELF* dosya formatı içerisindeki *GNU_STACK* başlık alanına yığının çalıştırılabilir olup olmadığı bilgisini yazar. Yığının çalıştırılabilir olarak işaretlenip işaretlenmediğini sembolik makina komutlarına veya ELF dosyasına bakarak anlayabiliriz. Örneğimiz için aşağıdaki komut çıktılarını inceleyiniz.
>
> ---
>
> $ *cat* inner.s | *grep* -i stack
>
> .section .note.GNU-stack,**"x"**,@progbits
>
> ---
>
> $ *readelf* -lW inner | *grep* -i stack
>
> GNU_STACK 0x000000 0x00000000 0x00000000 0x00000 0x00000 RW**E** 0x10
>
> ---
>
> Ayrıca bağlayıcıya yığının çalıştırılabilir olarak işaretlenip işaretlenmeyeceği açık bir şekilde aşağıdaki gibi de geçirilebilir. İçsel fonksiyon içeren örneğimizde, yığın çalıştırılamaz olarak işaretlendiğinde, bellek üzerinde erişim ihlali oluştuğuna dikkat ediniz.
>
> ---
>
> $ *gcc* -oinner inner.c -m32 --save-temps -fno-stack-protector **-z noexecstack**
>
> $./inner
>
> Segmentation fault (core dumped)
>
> ---
>
> **Not:** Yığın üzerinde bir kodun çalıştırılabilmesinin güvenlik açığı oluşturacağına dikkat ediniz.

Yığındaki çalıştırılabilir bu kod **trambolin**(trampoline) olarak adlandırılır. Trambolin kodu içsel fonksiyona dallanmak(*jump*) için kullanılmıştır. Şekilde, yerel değişkenin başlangıç adresi *0xAABBCCDD* ile, içsel fonksiyonun başlangıcıyla trambolin kodunun sonu arasındaki uzaklık ise *0xXXYYZZTT* ile temsil edilmektedir.

Yığında **b9** ile başlayan 10 byte uzunluğundaki blok trambolin kodunu oluşturmaktadır. **b9**, **x86** mimarisinde *ecx* yazmacına kopyalamaya ilişkin gerçek işlem kodu(*opcode*), **e9** ise göreli dallanma(*relative jump*) işlem kodudur. Trambolin kodu *main* fonksiyonuna ait yerel değişkenin adresini *ecx* yazmacına yazmakta ve sonrasında içsel fonksiyona dallanmaktadır. Bu

şekilde içsel fonksiyon, güvenli bir şekilde, *ecx* yazmacındaki adresi kullanarak *main* fonksiyonunun yerel değişkenine ulaşabilmektedir.

> **Not:** *x86* mimarisinde, *e9* makina komutu göreli dallanma işleminden sorumludur. *e9* makina kodu, operand olarak, hedef adres ile kendinden sonraki makina komutuna ait adresin farkını almaktadır.
>
> **Not:** Burada neden **ecx** yazmacının kullanıldığı gibi bir soru aklınıza gelebilir. C dili için fiili standart(*de facto standard*) çağırma biçimi(calling convention) olan *cdecl(C declaration)* çağırma biçiminde *eax*, *ecx* ve *edx* yazmaçlarının değerleri çağıran kod tarafından saklanmaktadır(*caller-saved*). *ecx* yazmacının değerinin saklanması çağıran tarafın sorumluluğunda olduğundan, bir içsel fonksiyon çağrısından önce çalışan *trambolin* kodunun, *ecx* yazmacının değerini değiştirmesinde bir mahsur yoktur.

Trambolin kodu *main* için ayrılan yığın alanında bulunmaktadır. Bu durumda, içsel bir fonksiyon tanımı içeren dışsal fonksiyon sonlandığında, yığın alanındaki trambolin koduna ait referans geçerliliğini yitirecektir. Bir fonksiyon sonlandığında ona ait yığın alanı geri verilmektedir.

İçsel fonksiyonun adresinin geçirilerek dışarıdan çağrılma durumunda, çağırma işlemi içsel fonksiyonu sarmalayan fonksiyon sonlanmadan yapılmalıdır. Aksi halde, yığının güvenirliliği kalmadığından, belirsiz davranış(*undefined behaviour*) oluşacaktır.

Daha önce içsel fonksiyonların genel olarak başka kodlara geçirildiğini, bu sayede onların davranışlarını değiştirdiğini veya olan bir olaydan haberdar olmayı sağladığını söylemiştik. Birçok dilde bu amaçlar için kullanılabilmelerine rağmen bir GNU C eklentisi olan içsel fonksiyonlar bir olayı dinlemek üzere asenkron çağrılmaya uygun değillerdir. Buna karşın senkron çağrılmaları durumunda diğer fonksiyonlara güvenle geçirilebilirler. Örnek olarak standart bir C fonksiyonu olan *qsort* verilebilir. *qsort* fonksiyonuna karşılaştırma amaçlı kullanması için, global isim alanında görünmeyen, içsel bir fonksiyon son argüman olarak geçirilebilir.

```
    void qsort(void *base, size_t nmemb, size_t size,
               int (*compar)(const void *, const void *));
```

İçsel fonksiyonlar ayrıca iç içe çoklu döngülerde istenilen bir noktada tüm döngülerden çıkmak için kullanılabilir. Aşağıdaki örneği inceleyiniz.

```
#include <stdio.h>

int main() {
    int i, j, k;
    void inner() {
        for (i = 0; i < 100; ++i) {
            for (j = 0; j < 100; ++j) {
                for (k = 0; k < 100; ++k) {
                    /*tüm döngülerden tek hamlede çıkılıyor*/
                    if (k > 0) return;
                }
            }
        }
    }
    inner();
    printf("i: %d\tj: %d\tk: %d\n", i, j, k);
}
```

Tüm döngülerden çıkılmak istendiğinde, birden çok *break* deyimi kullanmaksızın tek bir *return* deyimi kullanılabilir.

Son olarak içsel fonksiyonların bir GNU C eklentisi olan *statement expressions* içinde kullanımından bahsedeceğiz. Bu eklenti ile bir bileşik deyim(compound statement), parentezler içine alınarak bir ifade(expression) gibi ele alınabilir. Birleşik deyimlerin küme parentezlerine alınarak oluşturulduğunu hatırlayınız. Bileşik deyimin en sonundaki noktalı virgül ile sonlandırılmış ifade, tüm yapının değeri olarak ele alınır. Genel formu aşağıdaki gibidir.

```
({deyim veya deyimler; dönüş değeri;})
```

Bir fonksiyondan dönen değerin mutlağını alan örnek bir kod aşağıdaki gibidir.

```
#include <stdio.h>

int get_int() {
    return -111;
}

int main() {
    int abs = 0;
    abs = ({    int a; int b;
                a = get_int();
                b = a < 0 ? -a : a;
                b; });
    printf("%d\n", abs);
    return 0;
}
```

İçsel fonksiyonlar da bu yapı içersinde tanımlanabilir. Aşağıdaki örneği inceleyiniz.

```
#include <stdio.h>

typedef void (*PF) (int);

void foo(PF f) {
    f(111);
}

int main() {
    int local = 0;
    PF pf = ({ void inner (int x) { local = x; } inner; });
    foo(pf);
    printf("local: %d\n", local);
    return 0;
}
```

inner isimli içsel fonksiyon, bileşik ifadenin bir parçası olarak tanımlanmakta ve adresi de bu yapının ürettiği sonuç olarak ele alınmaktadır. *inner* fonksiyonunun adresi önce *pf* göstericisine atanmış, ardından *foo* fonksiyonuna geçirilmiş. Aynı işlem tek hamlede aşağıdaki gibi de yapılabilir.

```
#include <stdio.h>

typedef void (*PF) (int);

void foo(PF f) {
    f(111);
}

int main() {
    int local = 0;
    foo(({ void inner (int x) { local = x; } inner; }));
    printf("local: %d\n", local);
    return 0;
}
```

Önişlemci kullanılarak içsel fonksiyonlar görünüşte anonim fonksiyonlarmış gibi kullanılabilirler. Aşağıdaki örnekte içsel fonksiyon açık bir şekilde, isim verilerek, tanımlanmak yerine bu iş için bir makro kullanılmaktadır. İçsel fonksiyona ait geri dönüş türü ve parametre listesiyle fonksiyon gövdesi *lambda* makrosuna argüman olarak geçirilmiş.

Önişlemcinin ürettiği çıktıyı derleyiciye **-E** anahtarı geçirerek görebilirsiniz.

```c
#include <stdio.h>

#define lambda(return_type, function_body) \
({ \
      return_type _fn_ function_body \
      _fn_; \
})

typedef void (*PF) (int);

void foo(PF f) {
    f(111);
}

int main() {
    int local = 0;
    lambda(void, (int x) { local = x; }) (111);
    printf("local: %d\n", local);
    return 0;
}
```

Anonim Fonksiyonlar

Anonim fonksiyonlar **C++11** standartları ile dile eklenmiş isimsiz içsel fonksiyonlardır.

Anonim fonksiyonlar, ayrıca **lambda fonksiyonları** olarak da adlandırılır ve *lambda ifadeleri (lambda expression)* kullanılarak tanımlanırlar. Bir lambda ifadesi aşağıdaki forma sahiptir.

```
[capture-list] (parameters) -> return_type {function_body}
```

Örnek bir anonim fonksiyon tanımı ise aşağıdaki gibidir.

```
[] (int x, int y) -> int {return x + y}
```

Anonim fonksiyonlar, kendilerini sarmalayan fonksiyonun yerel değişkenlerine ulaşabilmektedir. Bu özelliğin GNU C eklentisi olarak nasıl gerçeklendiğini bir önceki bölümde incelemiştik. Burada ise benzer özellik *fonksiyon nesneleri (function object)* kullanılarak sağlanmaktadır.

Derleyici *lambda* ifadesini kullanarak yeni bir tür tanımlar ve bu tür için bir *operator()* fonksiyonu yazar. Anonim fonksiyona ilişkin işlemler bu nesne kullanılarak yapılır. Bu isimsiz fonksiyon nesneleri ayrıca **closure** olarak da isimlendirilmektedir.

> **Not:** Bir *operator()* fonksiyonu tanımlayarak, fonksiyon çağrı operatorünü (*function call operator*) yükleyen (*overload*) sınıf örneklerine yani bu türden oluşturulan nesnelere fonksiyon nesneleri (*function object*) veya functor denilmektedir.
>
> Fonksiyon nesneleri, söz dizimsel olarak, görünüşte birer fonksiyon gibi kullanılabilir ve başka fonksiyonlara *callback* olarak geçirilebilir.

Fonksiyon nesneleri sahip oldukları veri elemanları sayesinde *durum (state)* bilgisine sahip fonksiyonlar olarak kullanılabilirler. Durum bilgisinin kullanımını göstermek için aşağıdaki örneği inceleyelim.

```
#include <iostream>

using namespace std;

class Functor {
public:
    Functor(int state) {
        m_member = state;
    }
    bool operator() (int a, int b) {
        return (a - b) > m_member;
    }
private:
    int m_member;
};

int main() {
    /*referans karşılaştırma değeri*/
    int reference = 10;
    Functor f_obj(reference);
    bool result = f_obj(11, 0);
    if (result) {
        cout << "Ok" << endl;
    }
    else {
        cout << "Not Ok" << endl;
    }
    return 0;
```

```
}
```

Örneği *functor.cpp* ismiyle saklayıp aşağıdaki gibi derleyebilirsiniz.

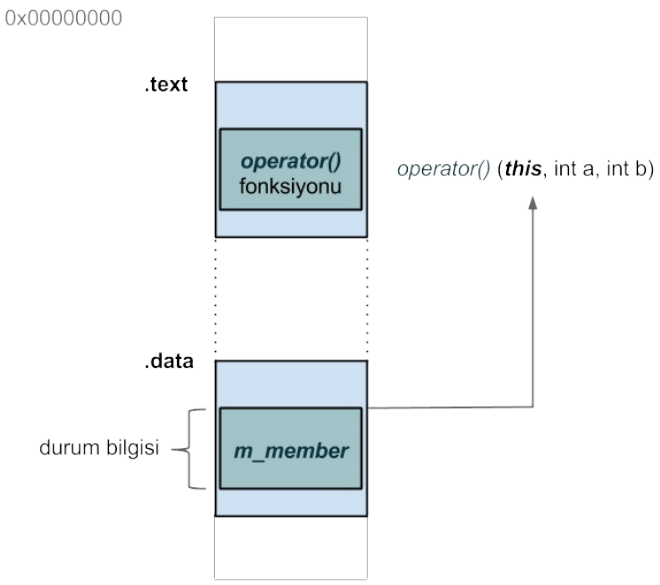
```
g++ -ofunctor functor.cpp
```

Örnekte temel olarak verilen 2 sayı arasındaki farkın bir referans değerden büyük olup olmadığına bakılmıştır. Yapılan işlemleri daha yakından inceleyelim.

```
1.  Functor f_obj(reference);
2.  bool result = f_obj(11, 0);
```

`1.` satırda sınıfın başlangıç fonksiyonuna (*constructor*) referans değeri geçirilerek *f_obj* nesnesi yapılandırılmış, `2.` satırda ise *f_obj* nesnesi üzerinden sınıfın *operator()* üye fonksiyonu çağrılarak karşılaştırma işlemi yapılmıştır.

Sınıfın başlangıç fonksiyonuna geçirilen referans değeri, sınıfın *m_member* üye değişkeninin değerini değiştirerek bir durum (*state*) bilgisini oluşturmaktadır. Daha sonra yapılacak olan karşılaştırma işleminde bu durum bilgisi kullanılmaktadır. Yukarıdaki örnek için bu durum bilgisi görsel olarak aşağıdaki gibi gösterilebilir.



m_member sınıfın data belleğindeki görüntüsünü oluşturmaktadır. *operator()* üye fonksiyonuna *m_member* üye değişkeninin adresi ilk argüman olarak geçirilmektedir.

> **Not:** Üye fonksiyonlara ilk argüman olarak üzerinde işlem yapacakları nesnenin adresinin gizli bir biçimde geçirildiğini hatırlayınız. Bu adrese üye fonksiyon içerisinde **this** anahtar sözcüğü ile ulaşmaktayız.

Burada *m_member* değişkeni durum bilgini tutmakta ve *operator()* fonksiyonunun davranışını değiştirmektedir.

Bölümün başında anonim fonksiyonlar için derleyicinin bir tür yazdığını, bu tür için bir *operator()* üye fonksiyonu tanımladığını ve işlemlerin bu türden oluşturulan isimsiz bir nesne üzerinden yapıldığını söylemiştik. Derleyicinin anonim fonksiyonlar için nasıl bir kod yazdığını ve anonim fonksiyonların fonksiyon nesneleriyle olan ilişkisini görmek için aşağıdaki örnek kodu inceleyelim.

```
#include <iostream>
```

```
using namespace std;

#ifdef FUNCTOR
class Functor {
public:
#ifdef INLINE
    Functor(int& total) __attribute__((always_inline))
        : m_total(total) {}
#else
    Functor(int& total)
        : m_total(total) {}
#endif

    void operator()(int num) {
        m_total += num;
    }
private:
    int& m_total;
};
#endif

int main()
{
int total = 0;
#ifdef FUNCTOR
    Functor fobj(total);
    fobj(111);
    printf("%d\n", total);
#else
    [&total] (int num) { total += num; } (111);
    printf("%d\n", total);
#endif
    return 0;
}
```

main içinde aynı sonucu üreten, önişlemci direktifleriyle ayrılmış, iki adet kod bloğu bulunmaktadır. Derleme işlemine hangi bloğun girecegine *FUNCTOR* makrosunun varlığına göre karar verilmektedir. *INLINE* makrosunu ne amaçla kullandığımızı daha sonra söyleyeceğiz.

> **Not:** Derleyiciye komut satırında **-D** anahtarı geçirerek bir makro tanımlaması sağlanabilmektedir.

Her iki kod bloğunda da *main* fonksiyonunun yerel değişkeninin değeri başka bir fonksiyon tarafından değiştirilmektedir. İlk olarak bu işlemin bizim yazdığımız bir sınıfa ait fonksiyon nesnesiyle nasıl yapıldığını, sonrasında ise bir anonim fonksiyon kullanılarak nasıl yapıldığını inceleyeceğiz.

Fonksiyon Nesnesinin Açık Kullanımı

Örnek uygulamaya *lambda.cpp* ismini verdikten sonra aşağıdaki gibi derleyebiliriz.

```
g++ -olambda lambda.cpp -m32 -std=c++11 -DFUNCTOR --save-temps
```

> **Not:** *-std* anahtarı ile derleyiciye kullanmasını istediğimiz standardı belirtiyoruz.

lambda.s dosyasını adım adım inceleyerek işe başlayalım. Derleyicinin *main* fonksiyonu için aşağıdaki gibi bir kod ürettiğini görmekteyiz.

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $32, %esp
        movl    $0, 24(%esp)
        leal    24(%esp), %eax
```

```
        movl    %eax, 4(%esp)
        leal    28(%esp), %eax
        movl    %eax, (%esp)
        call    _ZN7FunctorC1ERi
        movl    $111, 4(%esp)
        leal    28(%esp), %eax
        movl    %eax, (%esp)
        call    _ZN7FunctorclEi
        movl    24(%esp), %eax
        movl    %eax, 4(%esp)
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        ret
```

main için yığın alanından 32 byte'lık yer ayrılmış.

```
subl    $32, %esp
```

Yığının tepe noktasına 24 byte uzaklıktaki alan *total* yerel değişkeni için ayrılmış ve bu alana 0 değeri atanmış.

```
movl    $0, 24(%esp)
```

Yerel değişkenin adresi ilk önce *eax* yazmacına yazılmış ve oradan yığının tepe noktasına 4 byte uzaklıktaki alana kopyalanmış.

```
leal    24(%esp), %eax
movl    %eax, 4(%esp)
```

Yığının tepe noktasına 28 byte uzaklıktaki alanın adresi önce *eax* yazmacına yazılmış ve oradan yığının tepe noktasından başlayan alana yazılmış.

```
leal    28(%esp), %eax
movl    %eax, (%esp)
```

Sonrasında aşağıdaki gibi bir fonksiyon çağrısına ilişkin sembolik makina kodunu görmekteyiz.

```
call    _ZN7FunctorC1ERi
```

C++ derleyicisinin fonksiyon isimlerini dekore ettiğini hatırlayınız. C++ derleyicisi ürettiği sembolik makina kodunda, kullanıcının tanımladığı isimleri değil, kendi ürettiği aşağı seviyeli *assembler* isimlerini kullanmaktadır.

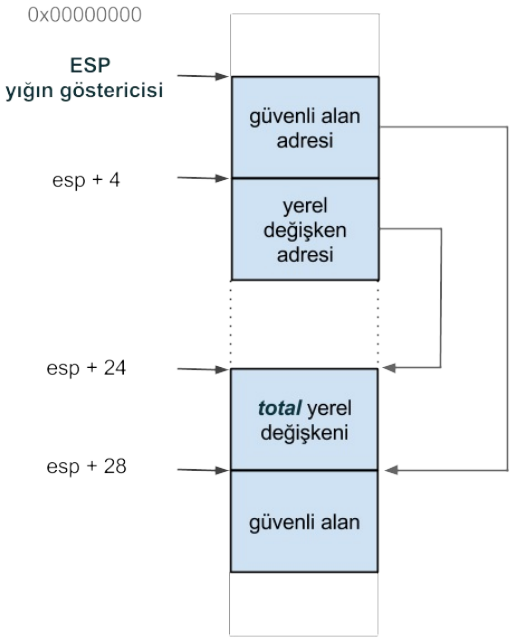
> **Not:** *binutils* paketinden çıkan **c++filt** aracı ile dekore edilmiş isimler kullanıcının tanımladığı isimlere geri dönüştürülebilir.

c++filt ile çağrı yapılan sembolün hangi fonksiyona ait olduğunu bulabiliriz.

```
$ c++filt _ZN7FunctorC1ERi
Functor::Functor(int&)
```

c++filt çıktısından buradaki çağrının sınıfın başlangıç fonksiyonuna (*constructor*) ait olduğunu görüyoruz. Bu aşamada

başlangıç fonksiyonu çağrıldığında yığının durumu aşağıdaki gibidir.



Başlangıç fonksiyonuna ilk argüman olarak yapılandıracağı nesnenin, ikinci argüman olarak ise yerel *total* değişkeninin adresi geçirilmektedir. C++ kodunda, yerel değişken adresinin *referans* yoluyla gizli bir biçimde geçirildiğine dikkat ediniz. Bu durumda yığının tepesinde *güvenli alan adresi* olarak gösterdiğimiz alandaki adres fonksiyon nesnesi için kullanılacak alanı göstermektedir.

> **Not:** gcc derleyicisi, C++ dilinde sınıfın statik olmayan üye fonksiyonları için **thiscall** çağırma biçimini (*calling convention*) kullanmaktadır. *thiscall* çağırma biçimi C dilinde **cdecl** çağırma biçimine oldukça benzemektedir. Çağırılan fonksiyonlara argümanlar yığın yoluyla geçirilmekte ve sağdan sola doğru yığına atılmaktadır. Bu durumda yığının tepesindeki değer çağırılan fonksiyonun en soldaki yani ilk parametresine denk gelmektedir. *thiscall* çağırma biçiminde *cdecl* çağırma biçiminden farklı olarak yığının en tepesi gizli bir *this* göstericisi geçirilmektedir.

Derleyicinin sınıfın başlangıç kodu için ürettiği sembolik makina kodu ise aşağıdaki gibidir.

```
_ZN7FunctorC2ERi:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %eax
        movl    12(%ebp), %edx
        movl    %edx, (%eax)
        popl    %ebp
        ret
```

> **Not:** Başlangıç fonksiyonu *main* içinde *_ZN7FunctorC1ERi* adıyla çağırılmasına karşın fonksiyon tanımı *_ZN7FunctorC2ERi* şeklinde yapılmış. Nedeni konumuzun dışında olduğundan yalnız bu detayı söyleyip geçeceğiz.

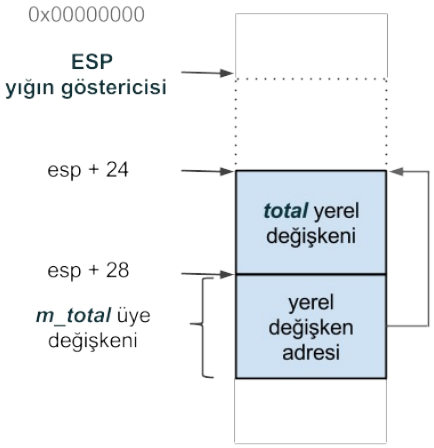
Başlangıç fonksiyonuna geçirilen ilk argüman (*nesnenin adresi*) *eax* yazmacına, ikinci argüman (*yerel değişken adresi*) ise *edx* yazmacına yazılmış.

```
movl    8(%ebp), %eax
movl    12(%ebp), %edx
```

edx yazmacındaki yerel değişken adresi, *eax* yazmacının bellekte gösterdiği alana yazılmış.

```
movl    %edx, (%eax)
```

Bu andan itibaren, fonksiyon nesnesi yapılandırılmış ve *m_total* üye değişkeni *main* fonksiyonunun yerel değişkeninin adresini tutar durumdadır. Başlangıç fonksiyonu döndüğünde yığının durumu aşağıdaki gibidir.



Tekrar *main* fonksiyonuna döndüğümüzde, 111 değerinin ve *m_total* üye değişkeninin adresinin sırasıyla yığına atıldığını görüyoruz. *m_total* değişkeni fonksiyon nesnesinin data belleğinde kapladığı alanı göstermektedir.

```
movl    $111, 4(%esp)
leal    28(%esp), %eax
movl    %eax, (%esp)
```

Sonrasında aşağıdaki fonksiyon çağrısını görmekteyiz.

```
call    _ZN7FunctorclEi
```

Dekore edilmiş sembol adına *c++filt* ile baktığımızda sınıfın *operator()* fonksiyonuna ait olduğunu görmekteyiz.

```
$ c++filt _ZN7FunctorclEi
Functor::operator()(int)
```

operator() fonksiyonuna ait sembolik makina kodu aşağıdaki gibidir.

```
_ZN7FunctorclEi:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %eax
        movl    (%eax), %eax
        movl    8(%ebp), %edx
        movl    (%edx), %edx
        movl    (%edx), %ecx
        movl    12(%ebp), %edx
        addl    %ecx, %edx
        movl    %edx, (%eax)
        popl    %ebp
        ret
```

Makina kodlarına yakından bakalım. Fonksiyona geçirilen ilk argüman değeri ilk önce *eax* yazmacına yazılmış, ardından yazmacın gösterdiği adrese karşılık gelen bellek alanındaki değer tekrar *eax* yazmacına kopyalanmış.

```
movl    8(%ebp), %eax
movl    (%eax), %eax
```

Bu işlem C dilinden aşina olduğumuz *pointer dereference* işlemine karşılık gelmektedir. Son durumda *eax* yazmacında *m_total* değişkeninin değeri yani *main* fonksiyonunun yerel değişkeninin (*total*) adresi bulunmaktadır. Sonraki üç komut ile iki defa *dereference* işlemi yapılarak *ecx* yazmacına *total* yerel değişkeninin değeri yazılmış.

```
movl    8(%ebp), %edx
movl    (%edx), %edx
movl    (%edx), %ecx
```

Son durumda, *eax* yazmacında *total* yerel değişkeninin adresi, *ecx* yazmacında ise değeri bulunmaktadır. Daha sonra *operator()* fonksiyonuna açık olarak geçirilen argüman, örneğimiz için 111, ilk önce *edx* yazmacına atılmış, *total* yerel değişkeninin değeriyle toplanarak yerel değişkenin bellek alanına yazılmış.

```
movl    12(%ebp), %edx
addl    %ecx, %edx
movl    %edx, (%eax)
```

main fonksiyonuna geri döndüğümüzde geri kalan komutların yerel değişkenin değerinin standart çıktıya basılmasıyla ilgili olduğunu görmekteyiz.

Özetleyecek olursak, *main* fonksiyonunun yerel değişkeninin adresi bir fonksiyon nesnesinde *durum* bilgisi olarak saklanmış ve *operator()* fonksiyonuyla bu adrese ulaşılarak yerel değişkenin değeri değiştirilmiştir.

Lamdba İfadeleri

Daha önce derleyicinin *lambda* ifadelerini kullanarak bizim için bir tür yazdığından bahsetmiştik. Şimdi bu duruma daha yakından bakalım. Bir önceki konu başlığında incelediğimiz örnek kodu FUNCTOR makrosu tanımlamaksızın aşağıdaki gibi derleyelim.

```
g++ -olambda lambda.cpp -m32 -std=c++11 --save-temps
```

Bu durumda anonim fonksiyon çağrısı derleme sürecine girecektir. Anonim fonksiyonun tanımlandıktan hemen sonra çağrıldığına dikkat ediniz.

```
[&total] (int num) { total += num; } (111);
```

lambda ifadesinin genel formunu yeniden hatırlatarak daha yakından bakalım.

```
[capture-list] (parameters) -> return_type {function_body}
```

Köşeli parantezler boş bırakılabildiği gibi dışsal değişkenler virgül ile ayrılmış bir liste şeklinde geçirilebilir. Bu dışsal

değişkenlere değer (*capture by value*) veya adres (*capture by reference*) yoluyla erişilebilir. Örnek bazı kullanımlar aşağıdaki gibi verilebilir.

Kullanım	Açıklama
`[ ]`	Dışsal bir değişkene erişim yoktur
`[&]`	Bütün dışsal değişkenlere adres ile erişilir
`[=]`	Bütün dışsal değişkenlere değer ile erişilir
`[x, &y]`	x değişkenine değerle y değişkenine adres ile erişilir
`[&, x]`	x değişkenine değer ile erişilirken diğer tüm dışsal değişkenlere adres ile erişilir

Burada dışsal değişken ile anonim fonksiyonun içinde tanımlandığı fonksiyona ait yerel değişkenleri kastettiğimizi hatırlatalım.

Köşeli parantezlerden sonra parametre değişkenleri ve fonksiyon gövdesi yazılır. Çoğu durumda geri dönüş değerinin türü derleyici tarafından fonksiyon gövdesine bakılarak tahmin edilmektedir. Buna karşın geri dönüş türü açık bir şekilde de yazılabilir.

> **Not:** Aslında bir *lambda* ifadesinin en genel formu aşağıdaki gibidir.
>
> *[capture-list] (params) mutable(optional) exception attribute -> ret { body }*
>
> Biz burada genel olarak anonim fonksiyonların işleyişiyle ilgilendiğimizden detaya girmeyeceğiz.

Köşeli parantezler içinde geçirdiğimiz dışsal değişkenler fonksiyon gövdesi içinde kullanılabilmektedir. Tekrardan örnek koddaki *lambda* ifadesine baktığımızda *total* yerel değişkenine adres yoluyla erişildiğini ve fonksiyon gövdesinde sol taraf değeri olarak kullanıldığını görmekteyiz.

Derlediğimiz kodu çalıştırdığımızda bir öncekiyle aynı sonucu ürettiğini göreceğiz.

Şimdi derleyicinin anonim fonksiyon için ürettiği kodu bir önceki bölümde incelediğimiz kod ile karşılaştırarak inceleyelim. Bir önceki bölümde bir *functor* sınıfı yazmış ve yerel değişkenin değerini bu sınıftan oluşturduğumuz nesne ile değiştirmiştik.

ANONİM

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $32, %esp
        movl    $0, 24(%esp)
        leal    24(%esp), %eax
        movl    %eax, 28(%esp)
        movl    $111, 4(%esp)
        leal    28(%esp), %eax
        movl    %eax, (%esp)
        call    _ZZ4mainENKUliE_clEi
        movl    24(%esp), %eax
        movl    %eax, 4(%esp)
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        ret
```

FUNCTOR

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $32, %esp
        movl    $0, 24(%esp)
        leal    24(%esp), %eax
        movl    %eax, 4(%esp)
        leal    28(%esp), %eax
        movl    %eax, (%esp)
        call    _ZN7FunctorC1ERi
        movl    $111, 4(%esp)
        leal    28(%esp), %eax
        movl    %eax, (%esp)
        call    _ZN7FunctorclEi
        movl    24(%esp), %eax
        movl    %eax, 4(%esp)
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        ret
```

Her iki kodda da yığının tepe noktasından itibaren 24 byte uzaklıktaki alan *total* yerel değişkeni için ayrılmış ve 0 ilk değeri verilmiş.

```
movl    $0, 24(%esp)
```

Functor örneğine baktığımızda bundan sonraki 4 sembolik makina komutunun *Functor* sınıfının başlangıç fonksiyonuna geçirilecek argümanlarla ilgili olduğunu görmekteyiz. Yerel değişkenin ve nesne için ayrılmış alanın adresleri sırasıyla yığına atılmış.

```
leal    24(%esp), %eax
movl    %eax, 4(%esp)
leal    28(%esp), %eax
movl    %eax, (%esp)
```

Sonrasında sınıfın başlangıç kodu çağrılarak nesne için ayrılan alana yerel değişkenin adresi yazılmış. Nesne için yığının başından itibaren 28 byte uzaklıktaki alanın ayrıldığına dikkat ediniz. Bu aşamada anonim fonksiyon örneğine baktığımızda aynı işlemin aşağıdaki gibi yapıldığını görmekteyiz.

```
leal    24(%esp), %eax
movl    %eax, 28(%esp)
```

Gerçekten de *functor* örneğinde başlangıç fonksiyonunu *inline* olarak tanımladığımızda, derleyici bir fonksiyon çağrısı yapmak yerine, buradaki kodun aynısını üretecektir. Bunun için bir önceki örnekte derleyiciye **-DINLINE** argümanı geçirerek bu durumu inceleyebilirsiniz.

Sonrasında her iki kod örneğinde de 111 değeri ve yerel değişkenin adresini tutan alanın (*fonksiyon nesnesi*) adresi yığına aktarılmış ve ardından fonksiyon çağrıları yapılmış. *Functor* örneği için yapılan çağrının sınıfın *operator()* üye fonksiyonuna olduğunu hatırlayınız. Anonim fonksiyon örneğinde ise çağrı aşağıdaki gibidir.

```
call    _ZZ4mainENKUliE_clEi
```

c++filt ile sembolün kullanıcı seviyesindeki karşılığına baktığımızda şöyle bir sonuç ürettiğini görmekteyiz.

```
$ c++filt _ZZ4mainENKUliE_clEi
main::{lambda(int)#1}::operator()(int) const
```

Buradan derleyicinin bizim için *const* bir *operator()* fonksiyonu yazdığını ve çağırdığını anlayabiliriz.

main::{lambda(int)#1} bize derleyicinin bizim için yazdığı tür adını göstermektedir. *lambda* ifadesinin *main* fonksiyonu içinde yazıldığını ve *int* türden parametreye sahip olduğunu hatırlayınız. Derleyicinin yazdığı *operator()* fonksiyonuna baktığımızda daha önce bizim yazdığımız *operator()* fonksiyonuyla aynı olduğunu görmekteyiz.

Burada derleyici, bizim yazdığımız *lambda* ifadesinden yola çıkarak, yerel değişkenin adresini tutan bir fonksiyon nesnesi oluşturmuş, ardından *operator()* fonksiyonu içinde bu yerel değişken adresini ve kullanıcının geçirdiği değeri kullanmış. Daha önce de söylediğimiz gibi burada yerel değişkenin adresini tutan isimsiz nesne **closure** olarak isimlendirilir. İsimsiz fonksiyon nesnesinin otomatik ömürlü olduğuna yani yığında oluşturulduğuna dikkat ediniz.

Bu aşamada anonim fonksiyonların kullanımına birkaç örnek vermek yararlı olacaktır. Anonim fonksiyonlar, şablonlarla (*template*) yoğun bir kullanıma sahip, fonksiyon nesneleri yerine kullanılabilir. Aşağıdaki örneği inceleyiniz.

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

using namespace std;

#ifndef LAMBDA
class AccumulatorFunctor {
public:
    AccumulatorFunctor(int& total)
        : m_total(total) {}

    void operator()(int num) {
        if (num % 2 == 0) {
            m_total += num;
        }
    }
private:
    int& m_total;
};
#endif

template<class InputIt, class UnaryFunction>
UnaryFunction for_each(InputIt first, InputIt last, UnaryFunction f)
{
    for (; first != last; ++first) {
        f(*first);
    }
    return f;
}

int main() {
    vector<int> v = {1, 2, 3, 4, 5, 6};
    int total = 0;
#ifndef LAMBDA
    for_each(v.begin(), v.end(), AccumulatorFunctor(total));
#else
    for_each(v.begin(), v.end(), [&total] (int num) { total += (num % 2 == 0) ? num : 0; });
#endif
    cout << "total: " << total << endl;
    return 0;
}
```

Örnekte, bir vektördeki çift sayıların toplamının yerel *total* değişkenine yazılması hedeflenmiş. Kod içerisinde aynı işin, hem bir fonksiyon nesnesiyle hem de anonim fonksiyon ile nasıl yapıldığının örneği bulunmaktadır. *lambda* ifadesi kullanılarak oluşturulan isimsiz fonksiyon nesnesi (*closure*) fonksiyon şablonu (*function template*) kullanılarak yazılan *for_each* fonksiyonuna *callback* olarak geçirilmiş. İşlemin anonim fonksiyon ile gerçekleştirilmesi için kodu aşağıdaki gibi derleyebilirsiniz.

```
g++ -Wall -olambda lambda.cpp -m32 --save-temps -std=c++11 -DLAMBDA
```

Ayrıca, anonim fonksiyonlar herhangi bir dışsal değişkenle ilişkilendirilmediği durumda (**[]** içinin boş olduğu durum) gizli bir biçimde (*implicitly*) fonksiyon göstericisine dönüştürülerek *callback* olarak kullanılabilir. Aşağıdaki örneği inceleyiniz.

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

using namespace std;

typedef int (*PF) (int);

void foo(PF f) {
    int result = f(111);
    if (result) {
        cout << "Odd" << endl;
    }
    else {
        cout << "Even" << endl;
    }
```

```
}

int main() {
    int total;
    (void)total;
    foo([] (int arg) { return (arg & 1); });
    return 0;
}
```

Örnekte *total* yerel değişkeninin anonim fonksiyon içinde kullanılmadığına dikkat ediniz. Dışarıya geçirilen içsel bir fonksiyon ile yerel bir değişkene ulaşabilmek için GNU C eklentilerince yığında bir *trambolin* kodu yazıldığını hatırlayınız. Daha önce de belirttiğimiz gibi GNU C++ ise bu eklentiyi içermemektedir.

Son olarak kısaca C++ diline eklenen anonim fonksiyon ya da *closure* kavramını, diğer bazı dillerdeki yakın kullanımlarıyla karşılaştıracağız. Buradaki *closure* ifadesi, *Java* diline *Java 8* ile eklenen ve *javascript* dilinde kullanılmakta olan *closure* ile tam olarak aynı anlamı taşımamaktadır. Daha sınırlı bir kullanıma sahiptir.

Daha önce içsel fonksiyonların dışarıdan *asenkron* çağrılmaları durumunda belirsiz davranış oluşacağından bahsetmiştik. Aynı problem burada anonim fonksiyonlar için de geçerlidir. Anonim fonksiyonun içinde tanımlandığı fonksiyon sonlandığında bu foksiyona ait yığın alanı geri verilmekte ve sonraki anonim fonksiyon çağrıları güvenilir olmayan bir alan üzerinde işlem yapmaktadır. Bu durumu, otomatik ömürlü yerel bir değişken adresini dönen bir fonksiyonun geri dönüş değerinin kullanımına benzetebiliriz. *Java* ve *javascript* gibi dillerde ise içinde anonim fonksiyon tanımlanan fonksiyonlara ait yığın alanı bir şekilde saklanmaktadır. C++ dilinde birçok yönden kullanışlı olan bu özellik maalesef şu haliyle *asenkron* olarak gerçekleşen bir olayı dinlemek için uygun değildir.

Derleyicinin anonim fonksiyonları nasıl ele aldığını bilmek, bizim bu özelliğin sınırlarını bilerek daha doğru kullanmamıza yardımcı olacaktır.

D-Göstericisi

Bu bölümde, d-göstericisinin (**d-pointer**) dinamik C++ kütüphanelerinin uyumluluğunun (*compatibility*) korunmasında nasıl kullanıldığını inceleyeceğiz.

Sırasıyla, kütüphane uyumluluğu ve d-göstericisinden bahsedecek, sonrasında bu yöntemin Qt kaynak kodundaki kullanımına bakacağız.

Kütüphane Uyumluluğu

Dinamik kütüphanelerin gelişimleri sürecinde, dışarıdan erişilebilir arayüzlerinde (*public interface*) ve içsel alanlarında (*private*) değişiklikler olmaktadır. Yapılan değişikliğin seviyesine kütüphanelerin iki grup versiyonu çıkmaktadır:

- **major:** Geçmişe doğru uyumluluğun korunmadığı, kapsamlı değişikliklerin yapıldığı versiyonlardır. Daha önce sağlanan metotların kaldırılması veya parametlerinin değiştirilmesi bu tip bir değişikliğe neden olmaktadır.
- **minor:** Geçmişe doğru uyumluluğunun korunduğu versiyonlardır. Kütüphanenin, sağladığı eski özelliklere ilave, yeni metotlar eklemesi veya var olanların iyileştirmesi durumunda bu tip yeni versiyonlar çıkmaktadır.

> **Not:** İngilizce'de kütüphanelerin dışardan erişime kapalı alanları için *internal* ve *private* kelimelerinin kullanıldığını görmekteyiz. Biz incelememizde bu kelimelere karşılık olarak çoğunlukla *içsel* kelimesini kullanacağız.

Bir kütüphanenin yeni bir *minor* versiyonu çıkması durumunda, kütüphanenin bir önceki versiyonuna bağımlı bir uygulama, yeniden derlenmeksizin, bu yeni versiyonu kullanabilmektedir. Ayrıca bu sayede, çalışabilir bir uygulama yeni bir sisteme taşındığında, yeni sistem önceki sistemdeki kütüphanelerin tam olarak aynılarını bulundurmak zorunda değildir. Kuşkusuz bu durumda uygulama, kütüphanenin sağladığı yeni özelliklerden faydalanamayacak fakat çalışmaya devam edecektir

Bir kütüphanenin *major* versiyonu çıkması durumunda ise bu kütüphaneyi kullanmakta olan uygulamalar üzerinde kaynak kod düzeyinde değişiklik yapılmalı ve uygulamalar yeniden derlenmelidir.

Kütüphanelerin erişebilir alanları dışında bir de gizli içsel (*private*) alanları bulunmaktadır. Bir kütüphanenin içsel alanındaki değişikliklerden onu kullanan uygulamaların etkilenmemesi beklenmektedir.

Konumuzun bundan sonraki bölümünde, uygulamaların kütüphanelerin içsel alanlarına bağımlılığı üzerinde duracağız.

Kütüphanelerin Erişime Kapalı Alanları

Uygulamalar, kütüphanelerin içsel alanlarına direkt olarak olarak erişememelerine karşın bu alandaki değişimden, istenmeyen şekilde, etkilenmektedir. Bir örnek üzerinden bu durumu inceleyelim.

İçerisinde *Class* isimli bir sınıf tanımladığımız kütüphanenin başlık ve kaynak dosyaları aşağıdaki gibi olsun.

test.h:

```
class Class {
public:
    Class();
    int foo();
private:
    int _foo;
    void init();
};
```

test.cpp:

```
#include "test.h"

Class::Class() {
    init();
}

int Class::foo() {
    return _foo;
}

void Class::init() {
    _foo = 111;
}
```

Gizli *_foo* değişkenine yine gizli *init* fonksiyonu ile ilk değeri verilmekte, değeri ise dışsal erişime açık *foo* fonksiyonu üzerinden öğrenilmektedir.

Kütüphane dosyasını, *libtest.so.1* ismiyle, aşağıdaki gibi derleyelim.

```
$ g++ -fPIC -shared -olibtest.so.1 test.cpp
```

Kütüphaneyi kullanacak basit bir uygulama kodunu ise aşağıdaki gibi tanımlayıp derleyebiliriz.

app.cpp:

```
#include <iostream>
#include "test.h"

using namespace std;

int main() {
    Class obj;
    cout << obj.foo() << endl;
    return 0;
}
```

```
$ ln -s libtest.so.1 libtest.so
$ g++ -oapp app.cpp -L. -ltest
```

Derleme sürecini kolaylaştırmak için, kütüphanenin versiyon numarası içermeyen sembolik bir bağlantısını oluşturduğumuza dikkat ediniz.

Uygulamayı aşağıdaki gibi çalıştırabiliriz.

```
$ LD_LIBRARY_PATH=. ./app
111
```

Şimdi kütüphanenin sağladığı *foo* fonksiyonunu iyileştirmek istediğimizi düşünelim, bu durumda kütüphanenin, erişilebilir arayüzüne dokunmaksızın, eskisiyle uyumlu yeni bir versiyonunu çıkarmak isteyebiliriz. *foo* fonksiyonunun yeni alanlara ihtiyaç duyduğunu varsayalım, bu durumu temsil etmek için kütüphanenin içsel alanına *int* türden 100 elemanlı bir dizi daha ekleyip ilklendirelim. Kütüphanenin yeni kodu aşağıdaki gibi olacaktır.

```
class Class {
public:
    Class();
    int foo();
private:
    int _foo;
    int _bar[100];
    void init();
};
```

```
#include "test.h"

Class::Class() {
    init();
}

int Class::foo() {
    return _foo;
}

void Class::init() {
    _foo = 111;
    for (int i = 0; i < 100; ++i) {
        _bar[i] = 0;
    }
}
```

Kütüphaneyi *libtest.so.2* adıyla yeniden derleyelim ve var olan uygulamayı tekrar çalıştıralım. Bu kez sembolik bağlantımız yeni kütüphaneyi göstermeli.

```
$ g++ -fPIC -shared -olibtest.so.2 test.cpp
$ rm libtest.so
$ ln -s libtest.so.2 libtest.so
$ LD_LIBRARY_PATH=. ./app
111
Segmentation fault (core dumped)
```

Uygulamayı yeni kütüphane ile çalıştırdığımızda hata almaktayız. Tekrar derleyip çalıştırdığımızda ise hatasız çalıştığını görmekteyiz.

```
$ g++ -oapp app.cpp -L. -ltest
$ LD_LIBRARY_PATH=. ./app
```

```
111
```

Uygulamaya kütüphaneden herhangi bir kod taşınmamasına karşın, neden uygulama yeniden derlendiğinde sorun ortadan kalkmaktadır? Şimdi bu sorunun cevabını arayalım.

Bir sınıfa ait örnek (*instance*) oluşturulurken, ilk önce bellekte gerekli alan ayrılmakta, sonrasında bu alan üzerinde sınıfın başlangıç fonksiyonu (*constructor*) çalıştırılmaktadır. Örneğimiz için sınıfın *private* alanı sınıf örneğinin bellekteki görüntüsünü oluşturmaktadır.

Uygulama derlenirken, derleyici tarafından sınıfın başlık dosyasındaki bildirimine bakılmakta ve gerekli alanı tahsis edecek kod yazılmaktadır. Kütüphanenin değiştirilmeden önceki ve sonraki halleri için *main* fonksiyonuna ait sembolik makina kodlarının bir kısmı aşağıdaki gibidir.

```
main:
.LFB971:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $32, %esp
        leal    28(%esp), %eax
        movl    %eax, (%esp)
        call    _ZN5ClassC1Ev
```

```
main:
.LFB971:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $432, %esp
        leal    28(%esp), %eax
        movl    %eax, (%esp)
        call    _ZN5ClassC1Ev
```

> **Not:** Uygulamaya ait 32 bitlik sembolik makina kodlarını görmek için uygulamayı aşağıdaki gibi derleyip **app.s** dosyasını inceleyebilirsiniz. Benzer işlemi, m32 anahtarını kaldırarak, 64 bit için de yapabilirsiniz.
>
> $ g++ -oapp app.cpp -L. -ltest -m32 --save-temps
>
> Sembolik makina kodlarındaki assembler direktiflerini ise, sadeleştirme amacıyla, ihmal ediyoruz.

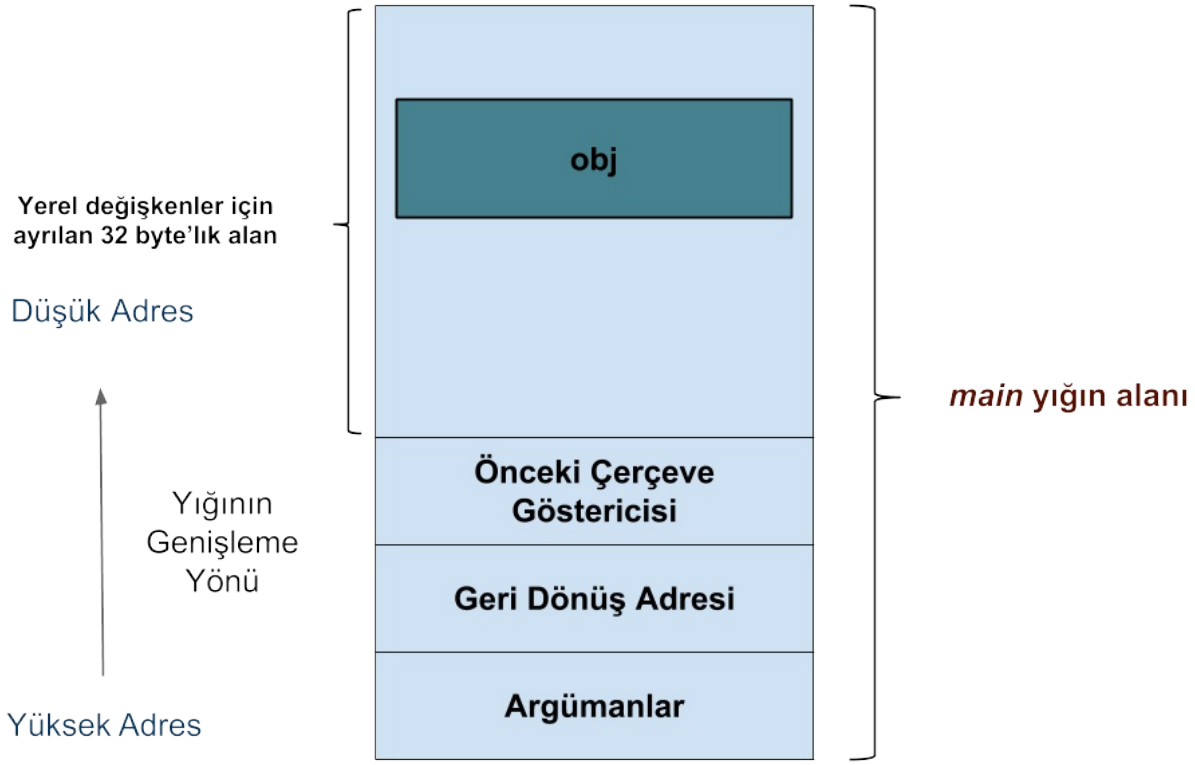
İlk durumda, derleyicinin *main* için yığında (*stack*) **32** byte yer ayırdığını, ikinci durumda ise **432** byte yer ayırdığını görmekteyiz.

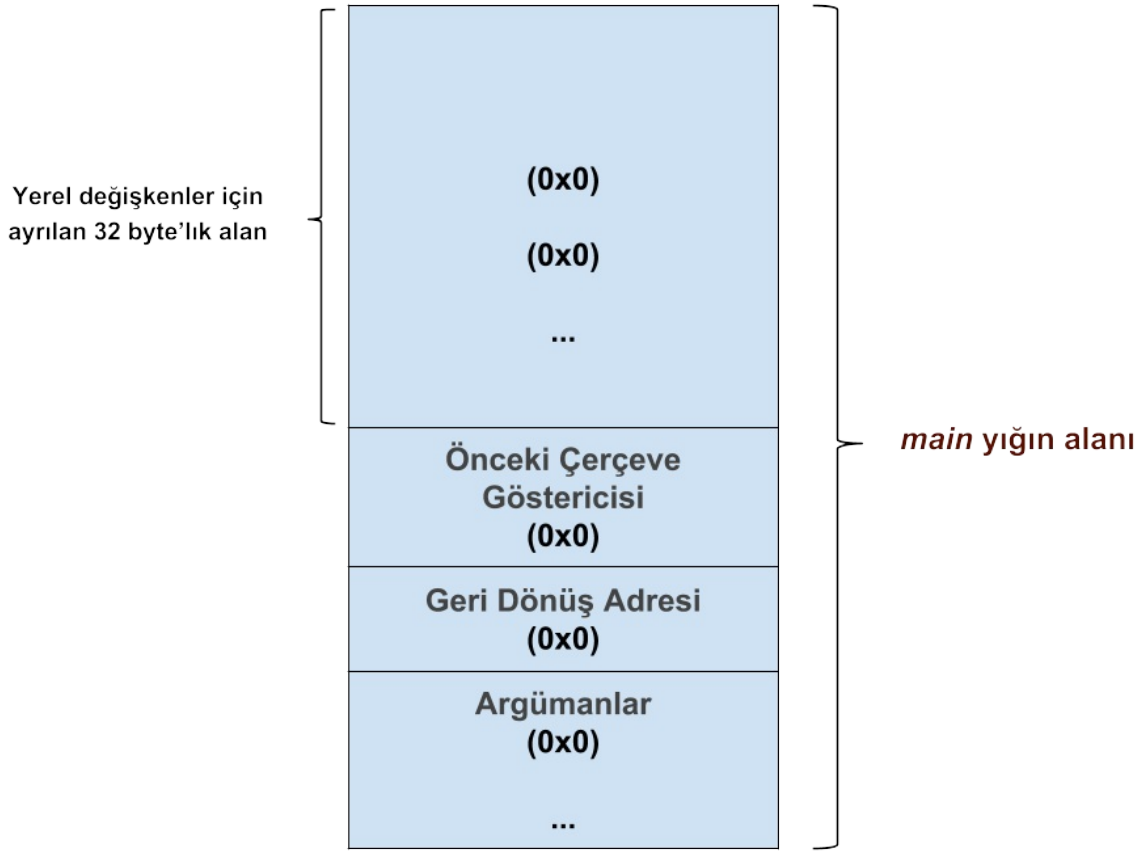
Uygulama, kütüphaneye ilişkin yeni başlık dosyası kullanılarak, ikinci kez derlenirken, *private* alana yeni eklenen **int _bar[100]** dizisi için **sizeof(int) * 100** yani fazladan 400 byte daha yer ayrılmıştır.

Uygulamanın yeniden derlenmeksizin kütüphanenin yeni versiyonuyla çalıştırıldığında neden erişim hatası (*Segmentation fault*) aldığını merak edebilirsiniz.

Fonksiyonların geri dönüş değerleri yığında saklanmaktadır, uygulama kodunda 32 byte yer ayrılmasına karşın kütüphane kodu bu alandan başlayarak fazladan 400 byte uzunluğunda bir alan üzerinde işlem yapmakta ve örneğimiz için bu alanı 0 değeriyle doldurmaktadır. Bu durumda *main* geri dönüş adresi 0. adres olacak ve işlemci bu adrese dallandığında erişim hatası oluşacaktır.

Oluşan hata durumunu görsel olarak aşağıdaki gibi temsil edebiliriz. İlk bellek görünümü, uygulamanın kütüphanenin ilk versiyonuna linklenmiş olduğu durumu sonraki ise kütüphanenin ikinci versiyonuna linklenmesi durumunu temsil etmektedir.





> **Not:** Uygulama içinde, sınıf örneği yığında değilde dinamik olarak *heap* alanında oluşturulsaydı da benzer bir

> problemin oluşabileceğine dikkat ediniz. Sınıfın başlangıç fonksiyonu bu kez heap alanında kendisi için ayrılan alanın dışında işlem yapacaktı.

Yukarıda incelediğimiz probleme ek olarak, kütüphanenin içsel alanındaki değişkenlerin sıralamasının değişmesi de tespit edilmesi zor olan bir probleme neden olabilmektedir. Bir örnek üzerinden bu durumu inceleyelim.

Küçük değişiklikler yaptığımız kütüphane dosyalarımız ve uygulama dosyamız aşağıdaki gibi olsun.

test.h:

```
class Class {
public:
    Class();
    int foo() { return _foo; }
private:
    int _foo;
    int _bar;
    void init();
};
```

test.cpp:

```
#include "test.h"

Class::Class() {
    init();
}

void Class::init() {
    _foo = 111;
    _bar = -1;
}
```

app.cpp:

```
#include <iostream>
#include "test.h"

using namespace std;

int main() {
    Class obj;
    cout << obj.foo() << endl;
    return 0;
}
```

Sırasıyla kütüphane ve uygulama dosyalarımızı daha önce yaptığımız gibi derleyelim ve çalıştıralım.

```
$ g++ -fPIC -shared -olibtest.so test.cpp
$ ln -s libtest.so.1 libtest.so
$ g++ -oapp app.cpp -L. -ltest

$ LD_LIBRARY_PATH=. ./app
111
```

Beklediğimiz üzere kütüphane içinde tanımlı *_foo* değerini doğru bir şekilde okuduk.

Kütüphanenin *private* alınında değişiklik yaparken bir nedenden dolayı değişkenlerin sıralamasını değiştirdiğimizi düşünelim. Bu durumda başlık dosyamız aşağıdaki gibi olacaktır.

test.h

```
class Class {
public:
    Class();
    int foo() { return _foo; }
private:
    int _bar;
    int _foo;
    void init();
};
```

Bu şekilde kütüphanenin yeni bir versiyonunu çıkaralım.

```
$ g++ -fPIC -shared -olibtest.so.2 test.cpp
$ rm libtest.so
$ ln -s libtest.so.2 libtest.so
```

Uygulamayı tekrar çalıştırdığımızda *_foo* değil *_bar* değişkeninin değerini okuduğumuzu görmekteyiz.

```
$ LD_LIBRARY_PATH=. ./app
-1
```

İlk bakışta bu tip bir hatayı beklemiyor olabilirsiniz. Kütüphaneyi, *Class* sınıfının *private* alanındaki değişimden sonra yeniden derlendiğimiz için, *foo* fonksiyonunun bu değişen duruma göre yeniden yazıldığını düşünebilirsiniz.

foo fonksiyonu kaynak dosya içinde tanımlansaydı durum tam da böyle olacaktı, fakat *foo* sınıf bildiriminin gövdesinde, yani başlık dosyasında tanımlandığı için **inline** olarak ele alınmaktadır (*implicitly inline*). Bu durumda *foo* kodu uygulamaya taşınmakta ve *foo* için kütüphane çağrısı yapılmamaktadır. Uygulama, kütüphanedeki değişimden bağımsız olarak, derlendiği andaki *foo* fonksiyonunu kullanmaktadır.

Uygulamaya ait sembolik makina komutlarında bu durumu gözleyebiliriz.

```
main:
    pushl   %ebp
    movl    %esp, %ebp
    andl    $-16, %esp
    subl    $32, %esp
    leal    24(%esp), %eax
    movl    %eax, (%esp)
    call    _ZN5ClassC1Ev
    leal    24(%esp), %eax
    movl    %eax, (%esp)
call    _ZN5Class3fooEv
    ...
```

```
_ZN5Class3fooEv:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %eax
    movl    (%eax), %eax
    popl    %ebp
    ret
```

> **Not:** c++filt ile sembolik makina listesindeki isimlerin dekore edilmemiş açık hallerini öğrenebilirsiniz.
>
> $ c++filt _ZN5Class3fooEv

> Class::foo()

Uygulamayı kütühanenin yeni haliyle yeniden derlediğimizde, uygulama kodundaki *foo*, fonksiyonunun değiştiğini görmekteyiz.

```
_ZN5Class3fooEv:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %eax
    movl    4(%eax), %eax
    popl    %ebp
    ret
```

Buraya kadar olan incelemelerimizi özetleyecek olursak, dinamik bağımlılığı olan uygulamaların, kütüphanelerin içsel alanlarındaki değişimlerden etkilendiğini görmekteyiz. Kütüphanelerin içsel alanlarının değişmesi durumunda uygulamalar yeniden derlenmek zorunda kalacaktır.

Böylesi bir durum kütüphane gelişimini kısıtlamaktadır. Kütüphane geliştiricileri, kütüphaneninin erişebilir arayüzüne dokunmadan, bazı özellikleri eklemek veya iyileştirmek için, kütüphanenin içsel alanlarında değişiklik yapabilmelidir. Bu sayede eskisiyle uyumlu yeni versiyonlar çıkarmak mümkün olacaktır.

Şimdi bu kısıtlamayı, d-göstericisi ile nasıl ortadan kaldırabileceğimize bakalım.

D-Göstericisi

Oldukça basit olan bu yöntemde, temel olarak, sınıfın içsel değişkenleri başka bir alanda tutulmakta ve sınıf içerisinde bu alana bir gösterici tutulmaktadır. Kendisi de dışarıdan erişime kapalı olan bu gösterici genellikle **d-pointer** olarak isimlendirilmektedir.

> **Not:** *d-pointer* ismi, bu yöntemi Qt kütüphanelerinde ilk olarak kullanan, Arnt Gulbrandsen'den gelmektedir.
>
> Bu kullanım, ayrıca Pimpl (Pointer to implementation) idiom, opaque pointer, compiler firewall ve Cheshire Cat gibi isimlerle de anılmaktadır.

d-göstericisine, yalnız kütüphane içerisinden erişilebilmekte ve sınıfın içsel alanının boyutları değişse bile uygulama bu değişiklikten etkilenmemektedir. Bir göstericinin, gösterdiği alandan bağımsız olarak daima sabit genişlikte (mimariye göre 4 veya 8 byte uzunluğunda) olduğunu hatırlayınız. İçsel alan üzerindeki tüm işlemler bu gösterici üzerinden yapılmaktadır.

Daha önce hata aldığımız (*Segmentation fault*) örnek için bu kez bu yöntemi kullanalım. İlk durumda kütüphane dosyalarımız aşağıdaki gibi olacaktır.

test.h:

```
class ClassPrivate;

class Class {
public:
    Class();
    ~Class();
    int foo();
private:
    ClassPrivate *d;
};
```

test.cpp:

```
#include "test.h"

class ClassPrivate {
public:
    int _foo;
    void init();
};

void ClassPrivate::init() {
    _foo = 111;
}

Class::Class() {
    d = new ClassPrivate;
    d->init();
}

Class::~Class() {
    delete d;
}

int Class::foo() {
    return d->_foo;
}
```

Örneğimizde, *Class* sınıfının içsel alanını *ClassPrivate* isimli başka bir sınıf üzerinden yöneteceğiz. Bu sayede, kütüphane kullanıcısının dışarıdan erişemeyeceği **init** fonksiyonunu da bu içsel sınıfa taşıyabiliriz. Elbette, sınıfın sonlandırıcı

fonksiyonunda (*destructor*), içsel sınıf için edindiğimiz alanı geri vermeliyiz.

Bu sınıfın bildiriminin başlık dosyasında değil kaynak dosyada yapıldığına dikkat ediniz. Bu yüzden *ClassPrivate* türüne yalnız *Class* sınıfından erişilebilmektedir. Bir diğer yaklaşım ise *ClassPrivate* sınıfını ayrı bir başlık dosyasında yazmak olabilirdi. Kütüphane kullanıcısının görmediği bu tür *private* başlık dosyaları genel olarak **_p.h** ile bitecek şekilde isimlendirilmektedir. Örneğimiz için bu dosya *test_p.h* şeklinde olacaktır. Başka bir yaklaşım ise *ClassPrivate* türünü *Class* sınıfının içsel bir türü olarak yazmak olabilirdi fakat bu kullanım biçimi pek yaygın değildir.

Başlık dosyasında, derleyiciyi bilgilendirme amaçlı olarak, *ClassPrivate* türüyle ilgili *forward declaration* yapıldığına dikkat ediniz. *C* ve *C++* dillerinde, henüz tanımı bilinmeyen türlere ait göstericiler tutulabildiğini hatırlayınız. Elbette bu aşamada, göstericinin gösterdiği adres üzerinde işlem yapılmaya çalışılması derleme hatasına yol açacaktır.

Bu aşamada, daha önce yaptığımız gibi, kütüphanenin ilk versiyonunu çıkaralım ve uygulamamızı çalıştıralım. Bu işlemleri daha önce de yaptığımız için burada tekrarlamıyoruz. Uygulamayı çalıştırdığımızda beklediğimiz sonucu aldığımızı görmekteyiz.

```
$ LD_LIBRARY_PATH=. ./app
111
```

Şimdi daha önce yaptığımız gibi kütüphaneye 100 elamanlı bir *int* dizi ekleyelim. Bu durumda kütüphanenin başlık dosyası aynı kalacaktır.

test.cpp:

```
#include "test.h"

class ClassPrivate {
public:
    int _foo;
    int _bar[100];
    void init();
};

void ClassPrivate::init() {
    _foo = 111;
    for (int i = 0; i < 100; ++i) {
        _bar[i] = 0;
    }
}

Class::Class() {
    d = new ClassPrivate;
    d->init();
}

Class::~Class() {
    delete d;
}

int Class::foo() {
    return d->_foo;
}
```

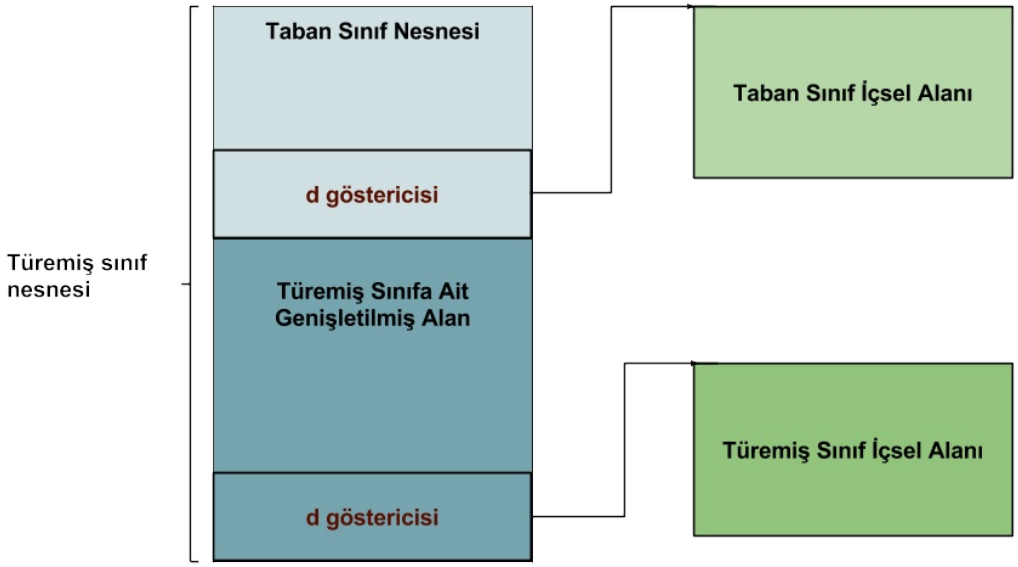
Kütüphaneyi yeniden derleyip çalıştırarak bir hata almadığınızı ve aynı sonuca ulaştığınızı doğrulayınız.

Türetme ve D-Göstericisi

Türetme hiyerarşisi içinde, türeyen tüm sınıfların kendi içlerinde ayrı bir d-göstericisi bulundurmaları istenen bir durum değildir.

Türeyen sınıf, içsel veri alanına yeni bir veri elemanı eklemeksizin, sadece taban sınıfın sanal fonksiyonlarını özelleştirebilir (*override*). Bu durumda, d-göstericisi boşa tutulmuş olacaktır. Ayrıca her sınıfın içsel alanı için bellekte (heap alanında) ayrı bir yer ayrılacak ve bu alanlar gerektiğinde sisteme geri verilecektir. Özellikle türetme hiyerarşinde altlarda bulunan sınıflar için bu yöntem kullanışlı olmayacaktır.

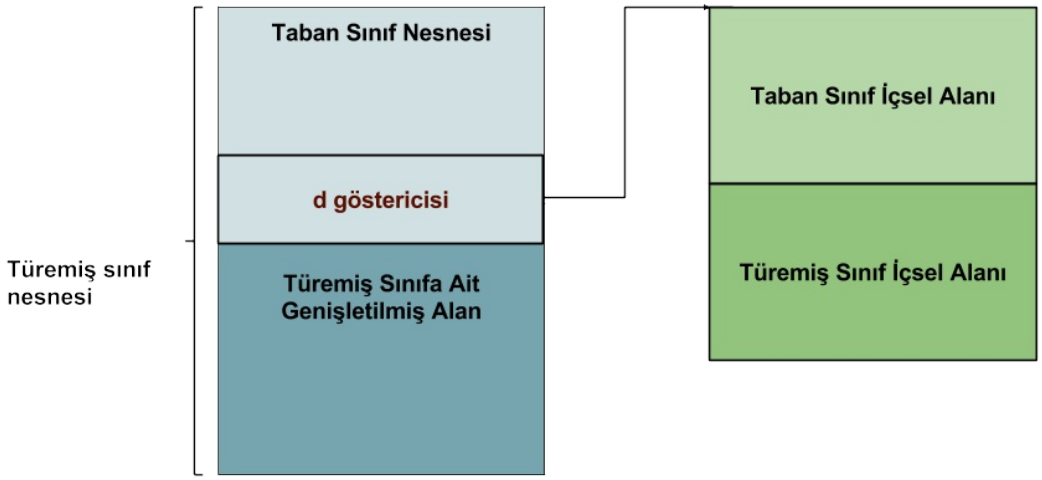
Aşağıdaki şekilde, d-göstericisi kullanan, türemiş bir türe ait nesnenin bellekte kapladığı alan temsil edilmiştir.



Türemiş sınıf nesnesinin içinde aynı zamanda bir taban sınıf nesnesinin bulunduğunu ve ilk olarak taban sınıfa ait başlangıç fonksiyonunun (*constructor*) çağrıldığını hatırlayınız.

Türemiş sınıf nesnesi içinde, taban ve türemiş sınıf içsel alanlarını gösteren, 2 tane d-göstericisi bulunmaktadır. Bu aşamada bu durum makul görünebilir fakat, daha önce de söylediğimiz gibi, yeni sınıflar türetmeye devam edersek çok sayıda bellek alanı ve d-göstericisi kullanmak zorunda kalacağız.

Türlerin kendisinde olduğu gibi, içsel veri alanı türleri üzerinde de türetme yaparak, tek bir d-göstericisi ve bellek alanı kullanmak mümkündür. Hedeflenen model aşağıdaki şekilde temsil edilmiştir.



Taban sınıflara ait içsel veri türlerinden türetme yapılacağından, artık bu türleri kaynak dosyalarda yazmak mümkün olmayacaktır. Bu yüzden, daha önce de bahsi geçen, genellikle *_p.h* biten başlık dosyaları kullanılmaktadır.

Şimdi hedeflediğimiz modeli nasıl gerçekleyebileceğimize daha yakından bakalım.

Türetme hiyerarşisinin en üstündeki taban sınıf için gerekli dosyalar ve kod taslağı aşağıdaki gibi olacaktır.

base.h:

```
class BasePrivate;

class Base {
public:
    Base();
protected:
    BasePrivate *d;
    Base(BasePrivate *d);
};
```

base_p.h:

```
class BasePrivate {
public:
    ...
};
```

base.cpp:

```
#include "base.h"
#include "base_p.h"

Base::Base() {
    d = new BasePrivate;
}

Base::Base(BasePrivate *d) {
    this->d = d;
}
```

Sınıfın, *public* olana ilave olarak, bir de **protected** başlangıç fonksiyonu (*constructor*) barındırdığına ve d-göstericisin de yine **protected** erişime sahip olduğuna dikkat ediniz.

Base sınıfından bir örnek oluşturulduğunda sınıfın *public* başlangıç fonksiyonu çağırılmakta ve içsel bir veri alanı tahsis edilmektedir.

```
Base::Base() {
    d = new BasePrivate;
}
```

Buna karşın, taban sınıftan türeyen bir alt sınıf, ortak bir içsel alan tahsis ederek bu adresi taban sınıfa geçirebilmektedir.

```
Base::Base(BasePrivate *d) {
    this->d = d;
}
```

Bir alt sınıf için ise kod taslağını aşağıdaki gibi oluşturabiliriz.

derived.h:

```
#include "base.h"

class DerivedPrivate;

class Derived : public Base {
public:
    Derived();
protected:
    Derived(DerivedPrivate *d);
};
```

derived_p.h:

```
#include "base_p.h"

class DerivedPrivate : public BasePrivate {
public:
    ...
};
```

derived.cpp:

```
#include "derived.h"
#include "derived_p.h"

Derived::Derived() : Base(new DerivedPrivate) {

}

Derived::Derived(DerivedPrivate *d) : Base(d) {

}
```

Türemiş sınıfta bir d-göstericisinin bulunmadığına dikkat ediniz.

Türemiş sınıftan bir örnek oluşturulduğunda, sınıfın *public* başlangıç fonksiyonu çağırılmakta ve ortak bir içsel veri alanı tahsis edilerek taban sınıfa geçirilmektedir.

```
Derived::Derived() : Base(new DerivedPrivate) {

}
```

Derived sınıfından türemiş başka bir sınıf ise, kendi edindiği içsel alan adresini yine *protected* başlangıç fonksiyonunu kullanarak nihayetinde *Base* sınıfına geçirebilmektedir.

```
Derived::Derived(DerivedPrivate *d) : Base(d) {

}
```

Bu sayede bir sınıf, türetme zincirinin neresinde olursa olsun, kendisi ve türediği tüm sınıflar için gerekli içsel alanı tahsis edip, bu alan adresini tek bir d-göstericisinde saklayabilmektedir. Fakat bu durumda türemiş sınıfların, taban sınıf içindeki d-göstericisine ulaşmaları gerekmektedir.

d-göstericisi **protected** erişime sahip olmasına karşın, en üstteki içsel sınıf türünden olduğundan, türemiş sınıflar içinde direkt kullanılamaz. Örneğimiz üzerinden gidecek olursak, d-göstericisi **BasePrivate*** türünden olduğundan, türemiş sınıf içinde bu adres **DerivedPrivate*** türüne dönüştürülmelidir (*type casting*). Bu amaçla türemiş sınıflar, tür dönüşüm işini yaparak d-göstericisinin değerini dönen fonksiyonlar barındırmaktadır. Örneğimiz için, *Derived* sınıfı içinde, böyle bir fonksiyonu aşağıdaki gibi tanımlayabiliriz.

```
DerivedPrivate *Derived::d_func() {
    return static_cast<DerivedPrivate *>(d);
}
```

Şimdi bu yöntemin Qt kütüphanesinde nasıl kullanıldığına bakalım.

Qt Kütüphanesinde D-Gösterici Kullanımı

Qt kodunda, d-göstericileri yoğun olarak kullanılmakta ve bu amaçla bazı makrolar barındırılmaktadır.

> **Not:** İncelememizde Qt 5.5.0 versiyonunu kullanacağız.

qglobal.h başlık dosyasında tanımlı ilgili makrolar ve fonksiyon şablonları (*function template*) aşağıdaki gibidir.

```
template <typename T> static inline T *qGetPtrHelper(T *ptr) { return ptr; }
template <typename Wrapper> static inline typename Wrapper::pointer qGetPtrHelper(const Wrapper &p) { return p.data();
```

```
#define Q_DECLARE_PRIVATE(Class) \
    inline Class##Private* d_func() { return reinterpret_cast<Class##Private *>(qGetPtrHelper(d_ptr)); } \
    inline const Class##Private* d_func() const { return reinterpret_cast<const Class##Private *>(qGetPtrHelper(d_ptr))
    friend class Class##Private;
```

```
#define Q_D(Class) Class##Private * const d = d_func()
```

Q_DECLARE_PRIVATE ile, biri *const* olmak üzere, iki tane *d_func* fonksiyonu tanımlandığını ve sınıfın içsel fonksiyonuna arkadaşlık (*friend*) verildiğini görüyoruz. *d_func* fonksiyonunun uygun tür dönüşümü yaparak d-göstericisi adresini döndüğünü hatırlayınız. Makro tanımındaki *qGetPtrHelper* fonksiyonuna ve *d_ptr* değişkenine ise daha sonra değineceğiz.

Q_D makrosu ise yazım kolaylığı sağlamakta, d_func geri dönüş değeri **d** isimli *const* bir yerel değişkende saklanmaktadır.

Bu makroların, QLabel sınıfındaki örnek bir kullanımı aşağıdaki gibidir.

qlabel.h

```
private:
    Q_DECLARE_PRIVATE(QLabel)
```

qlabel.cpp

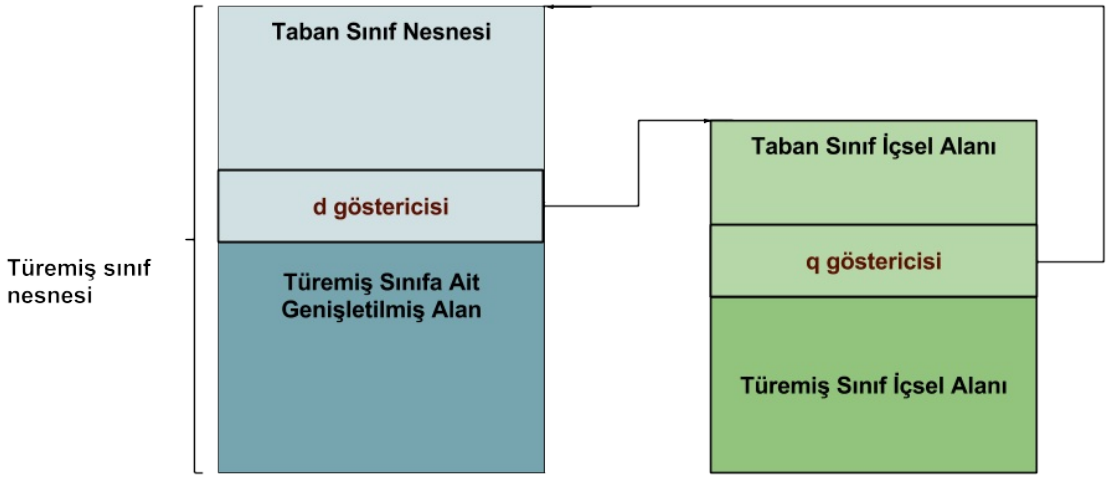
```
QLabel::QLabel(QWidget *parent, Qt::WindowFlags f)
    : QFrame(*new QLabelPrivate(), parent, f)
{
    Q_D(QLabel);
    d->init();
}
```

Bu aşamada d-göstericisi kullanmanın getirdiği bir kısıtlamadan bahsetmek istiyoruz. Daha önce, sınıfların *private* fonksiyonlarının da içsel sınıfa taşındığından bahsetmiştik. Fakat bu durumda, sınıfın *private* fonksiyonları *public* olanları direkt olarak çağıramamaktadır. Bazı durumlarda *private* fonksiyonların *public* olanları çağırması istenmektedir. Bu sebeple *public* sınıfa ait nesnenin adresi, içsel (*private*) sınıfa geçirilmektedir.

Qt kodunda, içsel sınıf içinde, *public* sınıf nesnesinin adresini tutan bir gösterici tutulmaktadır. Bu gösterici **q-pointer** olarak isimlendirilmektedir.

q-pointer

Aşağıdaki şekli inceleyiniz.



Türetme hiyerarşisinin herhangi bir yerindeki bir sınıf, q-göstericisine ulaşmak isteyecektir. Qt bu sebeple, d-göstericisinde olduğu gibi, uygun tür dönüştürme işlemiyle beraber q-göstericisini dönen fonksiyon tanımları içeren makroları barındırmaktdır.

qglobal.h başlık dosyasında tanımlı ilgili makrolar aşağıdaki gibidir.

```
#define Q_DECLARE_PUBLIC(Class)                                    \
    inline Class* q_func() { return static_cast<Class *>(q_ptr); } \
    inline const Class* q_func() const { return static_cast<const Class *>(q_ptr); } \
    friend class Class;
```

```
#define Q_Q(Class) Class * const q = q_func()
```

Şimdi bütün Qt nesnelerinin temel sınıfı olan **QObject** sınıfında bu yöntemin nasıl kullanıldığına bakalım. Qt kaynak kodunda, QObject sınıfı için *qobject.cpp*, *qobject.h* ve *qobject_p.h* olmak üzere 3 adet dosyanın bulundurulduğunu görüyoruz. Şimdi bu dosyalarda ilgilendiğimiz kısımlara bakalım.

qobject.h:

```
class Q_CORE_EXPORT QObjectData {
public:
    ...
    QObject *q_ptr;
    ...
};

class Q_CORE_EXPORT QObject
{
    ...
    Q_DECLARE_PRIVATE(QObject)
    ...
public:
    Q_INVOKABLE explicit QObject(QObject *parent=0);
```

```
protected:
    QObject(QObjectPrivate &dd, QObject *parent = 0);

protected:
    QScopedPointer<QObjectData> d_ptr;
    ...
};
```

Beklediğimiz üzere sınıfın, *public* ve *protected* olmak üzere, iki adet başlangıç fonksiyonu bulundurduğunu ve Q_DECLARE_PRIVATE makrosunu kullandığını görüyoruz. d-göstericisi ise **d_ptr** şeklinde isimlendirilmiş. Q_DECLARE_PRIVATE makrosuna tekrardan bakacak olursak, d_func tanımında *d_ptr* değişkeninin kullanıldığını görmekteyiz.

Başlık dosyasında ayrıca, *q_ptr* değişkenini barındıran, *QObjectData* sınıfının bildirildiğini görüyoruz. *QObject* sınıfının içsel veri türü *QObjectPrivate* bu sınıftan türetilmiştir. *qobject_p.h* dosyasında bu durumu görmekteyiz.

qobject_p.h:

```
class Q_CORE_EXPORT QObjectPrivate : public QObjectData
{
    Q_DECLARE_PUBLIC(QObject)
    ...
};
```

d_ptr'nin, içsel sınıf türünden doğal bir gösterici şeklinde değilde bir sınıf türünden tanımlandığına dikkat ediniz. *d_ptr* bildirimi, daha önceki örneklerimizde kullandığımız şekliyle, aşağıdaki gibi de yapılabilirdi.

```
    QObjectData *d_ptr;
```

Fakat bu durumda, d_ptr'nin gösterdiği alanın geri verilmesi, *QObject* sınıfının sorumluluğunda olacaktı. Burada ise d_ptr bir akıllı gösterici (*smart pointer*) olarak tanımlanmıştır. Bu sayede QObject nesnesi sonlandığında, içsel verilerin tutulduğu alan da beraberinde sisteme verilecektir.

> **Not:** QScopedPointer, akıllı göstericileri türden bağımsız olarak gerçeklemek için bir sınıf şablonu (template class) olarak yazılmıştır.
>
> Akıllı gösterici sınıflarının, **operator*()** ve **operator->()** fonksiyonlarını bulundurduklarını ve bu türden nesnelerin kendi ömürleri sona erdiğinde kullandıkları kaynakları da geri verdiğini hatırlayınız.

d_ptr'nin akıllı gösterici olarak tanımlanması, içsel alanın geri verilmesini kolaylaştırmasına karşın, içsel alan adresine direkt olarak erişimi engellemektedir. Bu sebeple **qGetPtrHelper** fonksiyon şablonu tanımlanmıştır. Q_DECLARE_PRIVATE makrosundaki *qGetPtrHelper* kullanımını hatırlayınız.

```
template <typename Wrapper> static inline typename Wrapper::pointer qGetPtrHelper(const Wrapper &p) { return p.data();
```

qGetPtrHelper ile akıllı gösterici sınıfının **data** fonksiyonu çağırılmakta ve bu sayede tutulan adrese erişilmektedir.

QScopedPointer sınıf şablonunun ilgili alanları aşağıdaki gibidir. Başlangıç fonksiyonuyla geçirilen adrese *data* fonksiyonuyla erişildiğini ve bu adres türünün *pointer* ismiyle *typedef* edildiğini görüyoruz.

```
class QScopedPointer
{
    ...
```

```
public:
    explicit inline QScopedPointer(T *p = 0) : d(p)
    {
    }
    ...
    inline T *data() const
    {
        return d;
    }
    ...
    typedef T *pointer;
    ...
};
```

Ayrıca Qt kodunda aşağıdaki gibi bir *qGetPtrHelper* şablon fonksiyonu daha bulunmaktadır.

```
template <typename T> static inline T *qGetPtrHelper(T *ptr) { return ptr; }
```

d_ptr'nin akıllı gösterici yerine, sınıfın içsel veri türü adresinden tanımlanması durumunda bu şablon kullanılacaktır. d_ptr, *QObjectData* d_ptr* şeklinde tanımlansaydı, derleyici tarafından bu şablon kullanılacaktı. Bu şablon kullanılarak yazılan fonksiyon, kendisine geçirilen değeri geri dönmektedir.

Tekrardan QObject sınıfına dönecek olursak, sırasıyla *public* ve *protected* başlangıç fonksiyonlarının aşağıdaki gibi tanımladığını görüyoruz.

qobject.cpp:

```
QObject::QObject(QObject *parent)
    : d_ptr(new QObjectPrivate)
{
    Q_D(QObject);
    d_ptr->q_ptr = this;
    ...
}
```

```
QObject::QObject(QObjectPrivate &dd, QObject *parent)
    : d_ptr(&dd)
{
    Q_D(QObject);
    d_ptr->q_ptr = this;
    ...
}
```

Beklediğimiz üzere, sınıfın *public* başlangıç fonksiyonunda, içsel veri alanı için yer tahsis edildiğini, *protected* başlangıç fonksiyonunda ise türeyen sınıfın geçirdiği adresin kullanıldığını görüyoruz. Her iki fonksiyonda da içsel veri alanına nesnenin adresi geçirilmektedir.

```
 d_ptr->q_ptr = this;
```

> **Not:** Türemiş sınıftan taban sınıfa olan adres geçirme işlemlerinin referans yoluyla yapıldığına dikkat ediniz.

Son olarak, QObject sınıfından türeyen QWidget sınıfına bakalım.

QWidget içsel sınıfının QObject içsel sınıfından türetildiğini görüyoruz.

qwidget_p.h:

```
class Q_WIDGETS_EXPORT QWidgetPrivate : public QObjectPrivate
{
    ...
};
```

qwidget.cpp içinde tanımlı, *public* ve *protected* başlangıç fonksiyonları aşağıdaki gibidir:

qwidget.cpp:

```
QWidget::QWidget(QWidget *parent, Qt::WindowFlags f)
    : QObject(*new QWidgetPrivate, 0), QPaintDevice()
{
    ...
}
```

```
QWidget::QWidget(QWidgetPrivate &dd, QWidget* parent, Qt::WindowFlags f)
    : QObject(dd, 0), QPaintDevice()
{
    ...
}
```

public başlangıç fonksiyonunda içsel veri alanı için gerekli kaynak alınırken, *protected* başlangıç fonksiyonunda, QWidget'dan türeyen sınıf örneğinin edindiği adres kullanılmaktadır.

Sonuç olarak, Qt kodunda da daha önce incelediğimiz örneklere benzer şekilde bir kullanım olduğunu, fakat akıllı gösterici ve makro kullanımıyla kodun daha yönetilebilir bir şekilde yazıldığını görmekteyiz.

D-Gösterici Kullanımının Avantaj ve Dezavantajları

d-göstericisi ile kütüphanenin içsel alanını izole etmenin temel avantaj ve dezavantajlarını aşağıdaki gibi sıralayabiliriz.

Avantajlar:

- Doküman boyunca incelediğimiz gibi, uygulamaların kullandıkları kütüphanelerin içsel alanlarına bağımlılığı ortadan kalkmakta, bu sayede kütüphanelerin içsel davranışları değiştirilerek eskiyle uyumlu yeni versiyonları çıkarılabilmektedir.
- Sınıf bildirimlerindeki *private* değişken ve fonksiyon bildirimleri çıkartılmaktadır. Kütüphane kullanıcısı, *private* alan ve fonksiyonları kullanamayacağı için başlık dosyasında bu bilgilerin bulunmaması daha anlamlı olacaktır. Bu sayede, başlık dosyaları API referansı olarak kullanılabilir.

Dezavantajlar:

- Özellikle türetme hesaba katıldığında tasarım karmaşıklaşacaktır.
- İçsel alana erişmek için fazladan bir *indirection* işlemi daha yapılmaktadır.