

Yalova Üniversitesi
Bilgisayar Mühendisliği

Sistem Programlama

Ders Notları

Yazar-Çizer-Döver: Sezer BOZKIR
Ders Hocası: Necla Bandırmalı
2016

POSIX;

Unix türevlerine sistemlerin kendi aralarında bir standarta oturtulmaya çalışılmıştır. Bunun içinde POSIX standartları oluşturulmuştur POSIX-1 ve POSIX-2 diye ikiye ayrılmıştır. POSIX-1'de sistem çağrılarını yapıyor. POSIX-2 'dede sistem araçlarına dair standartlar mevcuttur.

GCC;

C ve C++ derleyicisidir. Açılımı GNU C Compiler ya da GNU Compiler Collection olarak geçmektedir. C ve C++ 'ı default(Türkçesi aklıma gelmedi özür) olarak derleyebilmektedir. G++; C++ derleyicisidir aslında. GCC C++ kodlarını derlen G++ kullanıyor. GCC yazılarak da G++ yazılarak da C++ kodları derlenebilmektedir.POSIX; örnek; bir işletim sistemi posix standardına sahipse printf her daim çalışır. ANSI-C standart olarak olduğundan, bu standartlarda aynı şekilde çalışmakta.

ELF;

C ve C++ da kod yazdığımızda, GCC'nin nereden başlayacağı, ne yapacağı gibi sisteme bilgi veren dosya sistemidir. ELF bir dosya sistemidir. Executable and Linkable(Linking) File açılımıdır. GCC ile programı derlediğimizde karşımıza bir dosya çıkar, bu dosya object dosyadır. GCC ile dosya derlemesi aşama aşama derlenebilir. Bu dosya ELF formatında ve standardındadır. Ortak bir standart dosya formatıdır. Linux yada Unix türevli işletim sistemleri için (Windows için değil, windowsta COF veya PE sistemi kullanılmakta) dosya formatı ELF Header, Header Data, Program Header Data, Text RO Data gibi sectionlara(bölmelere) ayrılmaktadır.

Linkable bağlanabilir anlamında kullanılmıştır. Yeniden konumlandırılabilir olması ise başka sistemlerde object dosyanın kullanılabilmesini kastetmektedir. Object file System bir standart, object hale gelmiş dosyanın yapısı standart olarak aynıdır.

Compile ettiğimizde dosyayı hangi aşamalardan geçmektedir?

Bir process'in belleğinin nasıl organize edildiğini önceden biliyoruz. (Başlık, heap alanı, stack alanı, kod alanı). Bunların nasıl organize edildiği açıklanacaktır.

-Programın çalışan(run edilmiş) haline process nedir.

Program Nedir?

Programları bilgisayarda farklı farklı programlar ile şekillendirebilir. İşlemciye C++ kodu verdiğimizde direkt çalışmaz, makine koduna dönüştürülmesi gerekir. Yani 1'ler ve 0'lardan anlar. Hangi dilde yazılırsa yazılsın, o dilde yazdığınız kod en nihayetinde makine koduna dönüştürülür. Makine kodunda ise 32 tane binary sayı mevcut. Bunlardan ilk 8'i op-code diğer kodlar ise operand kod olarak isimlendirilir.

Op-code: Op code farklı komutları icra eder. Ekleme, taşıma, çarpma çıkarma gibi işlemleri icra eder. Buradaki komutlar yapılacak işlemleri tetiklerler. Yani Operand kısmındaki veriye ne olacağını makineye op-code söyler. İşlemleri direk 1'ler ve 0'lar olarak yazmak yerine Assembly olarak vermek daha kolay olmaktadır. (örnek; 10001000 = ADD) .

“Bilmeniz gereken yegâne şey, ne dilde olursa olsun, yazılan kod makine koduna dönüştürüldüğünde işlemci onu anlayabiliyor ve çalıştırabiliyor.”

Her işlemcide Assembly yapısı ve instruction set aynı değildir. Mesela x86 mimarisinde yazılan bir program Motorola'nın instruction yapısı farklı olduğundan çalışmayabiliyor. aynı programı farklı mimarilerde çalıştırabilmek için o mimarinin instruction setine göre yazmak gerekir. Kısaca işlemciye bağlı dillerle program geliştirmek pekte avantajlı değildir. Biz de bu nedenle yüksek seviyeli diller kullanıyoruz. Bu sayede işlemciden bağımsız kod yazabiliyoruz.

Bir C programını assembly olarak görmek istiyorsak; “gcc -S dosyaismi.c” yazarak görebiliyoruz. Bu s
uzantılı bir dosya üretiyor. Ve C kodunun assembly halini görüyoruz:



```
Uçbirim - natgho@Optimist:/home/natgho/deneme
Dosya D enle G ster U birim Sekmeler Yardım
[natgho@Optimist ~]$ cd deneme && cat ornek.s
.file "ornek.c"
.section .rodata
.LC0:
.string "Hello World"
.text
.globl main
.type main, @function
main:
.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
movl $.LC0, %edi
call puts
movl $0, %eax
popq %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
.size main, .-main
.ident "GCC: (GNU) 5.3.0"
.section .note.GNU-stack,"",@progbits
[natgho@Optimist deneme]$
```

***cat** komutu bize o dosyanın i eri ini g sterir.
Bu hal execute hale gelmeden  nceki halidir.
Bu i i assembler yapmaktad r.

Assembler ile  alı manın birinci zorlu u bu  ekilde yazılmasıdır. Yani yazımı zordur.  ikinci sıkıntısı ise adresleme problemidir. Fiziksel adreslerle u ra tı ımızda bellekte kodların nereye yerle ece i ile de u ra mak gerekmekte. Bu bizim u ra mak istemedi imiz bir durum. Y ksek seviyeli diller bizi bunlarla u ra tırmıyor. Di er bir problem assembly ile basit bir programın sat rlarca s rebilmesi. Y ksek seviyeli dilde kısa s ren (bkz:hello world) bir program assembly'de  ok daha uzun olmakta, bu

da aynı iş için daha fazla kod yazmamızı gerektiriyor. Üst düzey dillerin bize sağladığı bir diğer avantaj olan programda kullandığımız kod parçalarını istediğimiz yerde kullanabiliyorken assembly ile bunu yapmamız çok daha zor olmakta.

Compiler Nedir?

Compiler kelime anlamı olarak derleyicidir. Compiler çalıştırma ve dönüştürme işi yapar. C programlarını binary yani ikili makine koduna ya da bir modul şekline dönüştürme işi yapar. Compilerın ve assembler birleşip bize object dosya oluşturur. Ardından linker devreye girmekte.

Linker Ne Yapar?

Linker kütüphanelerle bağlantı sağlar. Kütüphanelerden ilgili fonksiyonları kod içerisine yerleştiriyor. Kodun aslen çalıştırılmasına sağlar. Mesela printf'i çalıştırmak için kütüphaneye gider kütüphaneden printf'i buluyor, o fonksiyonu yerine yerleştiriyor (yani programa bağlıyor) ardından execute dosya oluşmuş oluyor. Linker bağlayıcıdır. Kütüphane dosyaları ile programı bağlar. Execute oluşturulan dosyayı diske kaydeder.

İşlemler sırasıyla Compile --> Assembly--> Linking --> executable dosya

***Linux bize istersek bu ara dosyaları görmemizi sağlar. DEVC++ ya da benzer IDE'ler bu dosyaları saklar, GCC'yede uzantı vermezsek, ara dosyaları görmez, direk executable dosyayı görürüz.

Bir de loader mevcut.

Loader ne yapar?

Loader yükleyici anlamına gelir. Exe dosyamız oluşur, exe dosyamız oluşuktan sonra, programı çalıştırdığımızda, onunla ilgili kodu belleğe yüklüyor.(Diskten alıp ana belleğe yükler). Bunu işletim sistemindeki loader yapmaktadır.

.h dosyaları nedir?

Bu dosyalara başlık dosyaları denir. Başlık dosyalarındaki kod parçaları, kütüphanede dosyalarının içinde bulunur. Linux'da bu dosyalar "usr/includes" içerisinde bulunmaktadır.

Ayrıca başkaları tarafından sağlanan komutlar da h dosyaları içinde

bulunabilmekte. H dosyalarının içerisinde prototipler bulunur.

Programın kendisi başlık dosyasında yazılmaz.

Burada prototipler, makrolar bulunur, define tanımlar bulunur, değişkenler bulunur. Programın kendileri C dosyaları içerisinde tutulur.

```
Uçbirim - natgho@Optimist:/usr/include
Dosya Düzenle Göster Uçbirim Sekmeler Yardım

[natgho@Optimist include]$ ls
a52dec      ijs          pcreposix.h
aalib.h     ini_comment.h pcre_scanner.h
accountsservice-1.0 ini_config.h pcre_stringpiece.h
acl         ini_configmod.h pg_config_ext.h
aio.h       ini_configobj.h pg_config.h
AL          ini_valueobj.h pg_config_manual.h
aliases.h   inttypes.h   pg_config_os.h
alloca.h    itcl2TclOO.h pgm-5.2
alpm.h      itclDecls.h  pgtypes_date.h
alpm_list.h itcl.h       pgtypes_error.h
alsa        itclIntDecls.h pgtypes_interval.h
ansidecl.h  itclInt.h    pgtypes_numeric.h
a.out.h     itclMigrate2TclCore.h pgtypes_timestamp.h
apr-1       itclTclIntStubsFcn.h phonon4qt5
archive_entry.h iwlib.h      pipeline.h
archive.h   jack         pixman-1
ares_build.h jansson_config.h plist
ares_dns.h  jansson.h    plugin-api.h
ares.h      jasper       plymouth-1
ares_rules.h jconfig.h    pngconf.h
ares_version.h jerror.h     png.h
argp.h      jmorecfg.h  pnglibconf.h
argz.h      jpegint.h   polkit-1
ar.h        jpeglib.h   polkit-qt5-1
arpa        js-17.0     poll.h
asm         json         poppler
asm-generic json-c       popt.h
aspell.h    json-glib-1.0 postgres_ext.h
ass         kadm5       postgresql
assert.h    kate        potracelib.h
assuan.h    kdb.h       pppd
asyncns.h   keybinder.h pqStubs.h
atasmart.h  keyutils.h pxx
atk-1.0     KF5         pr29.h
atkmm-1.6   KHR         printf.h
atomic_ops  kpathsea   proc
atomic_ops.h krad.h     profile.h
atomic_ops_malloc.h krb5       protocols
atomic_ops_stack.h krb5.h     proxy.h
at-spi-2.0 ksba.h     pspell
at-spi2-atk ksysguard  ptexenc
```

```
Uçbirim - natgho@Optimist:/usr/include
Dosya Düenle Göster Uçbirim Sekmeler Yardım

#if !defined __need_FILE && !defined __need__FILE
# define _STDIO_H 1
# include <features.h>

__BEGIN_DECLS

# define __need_size_t
# define __need_NULL
# include <stddef.h>

# include <bits/types.h>
# define __need_FILE
# define __need__FILE
#endif /* Don't need FILE. */

#if !defined __FILE_defined && defined __need_FILE

/* Define outside of namespace so the C++ is happy. */
struct _IO_FILE;

__BEGIN_NAMESPACE_STD
/* The opaque type of streams. This is the definition used elsewhere. */
```

Örnek başlık dosyası

Derlenmiş dosyalar, exe uzantılı olanlar yani, içerikleri görüntülenemeyen türden dosyalara dönüştürülürler.

*****pwd**: O an bulunduğumuz dizini gösterir.

*****cd** dosya_adi : Bulunduğun dizindeki klasörlerden birine girmek için kullanılır.

*** **ls** (-al -l gibi parametreleri de var); Dizin içerisindeki dosyaları listelemek için kullanılır.

```
Uçbirim - natgho@Optimist:/home/natgho
Dosya Düenle Göster Uçbirim Sekmeler

[natgho@Optimist ~]$ pwd
/home/natgho
[natgho@Optimist ~]$
```

```
Uçbirim - natgho@Optimist:/home/natgho
Dosya Düenle Göster Uçbirim Sekmeler Yardım

[natgho@Optimist ~]$ ls
Belgeler      FoxitSoftware      Müzik           Resimler        'VirtualBox VMs'
deneme        Genel              NetBeansProjects sketchbook       workspace
deneme.txt    github_programlarim 'Örnek Dosyalar' ssh.png
Desktop       Internet_Paylas    PhpstormProjects Şablonlar
Downloads     minicom.cap        PycharmProjects Videolar
[natgho@Optimist ~]$
```

Linux'da bir C programını derleme aşamaları

Linuxda bir C programının derlenişi 4 aşamada gerçekleştirilmektedir.

1. Preprocessing (Ön İşleme)
2. Compalition (Derleme)
3. Assembly (Dönüştürme)
4. Linking (Bağlama)

Yani executable bir dosya oluşturulması için program bu dört işlemten geçiyor.

*** Emacs ile ilgili bir ayrıntı; Dosyalar buffer'da tutulur, kaydedilmediği sürece göremiyoruz. CTRL + X, CTRL + S yapıldığında oluşuyor. Dosya bu arada tamponlarda tutuluyor, biz dosyalar arasında kaydetmeden gezerken bufferları ziyaret ediyoruz.

Bir exe dosyanın tüm aşamalarını görmeye başlıyoruz;

1) Preprocessing:

Bu aşama da ön işleme aşaması olarak tabir edildiğinden, eğer programınızda makrolar varsa makrolar program satırlarının içerisine yazılıyor, komut satırları çıkarılıyor ve include dosyaları genişletiliyor. Konsoldan “gcc -E dosya.c” ile derlenerek bu dosya görülebilir.



```
natgho@Optimist:/home/natgho/sistem_programlama
# natgho@Optimist:/home/natgho/sistem_programlama
x) __attribute__((__nothrow__, __leaf__)); extern long double log2l (long double __x) __attribute__((__nothrow__, __leaf__));

extern long double powl (long double __x, long double __y) __attribute__((__nothrow__, __leaf__)); extern long double powl (long double __x, long double __y) __attribute__((__nothrow__, __leaf__));

extern long double sqrtl (long double __x) __attribute__((__nothrow__, __leaf__)); extern long double sqrtl (long double __x) __attribute__((__nothrow__, __leaf__));

extern long double hypotl (long double __x, long double __y) __attribute__((__nothrow__, __leaf__)); extern long double hypotl (long double __x, long double __y) __attribute__((__nothrow__, __leaf__));

extern long double cbtrl (long double __x) __attribute__((__nothrow__, __leaf__)); extern long double cbtrl (long double __x) __attribute__((__nothrow__, __leaf__));

extern int isnfl (long double __value) __attribute__((__nothrow__, __leaf__)) __attribute__((__const__));

extern int finitel (long double __value) __attribute__((__nothrow__, __leaf__)) __attribute__((__const__));

extern long double drem1 (long double __x, long double __y) __attribute__((__nothrow__, __leaf__)); extern long double drem1 (long double __x, long double __y) __attribute__((__nothrow__, __leaf__));

extern long double significandl (long double __x) __attribute__((__nothrow__, __leaf__)); extern long double significandl (long double __x) __attribute__((__nothrow__, __leaf__));

extern long double copysignl (long double __x, long double __y) __attribute__((__nothrow__, __leaf__)) __attribute__((__const__)); extern long double copysignl (long double __x, long double __y) __attribute__((__nothrow__, __leaf__)) __attribute__((__const__));

extern long double scalbl (long double __x, long double __n) __attribute__((__nothrow__, __leaf__)) __attribute__((__const__));

# 147 "/usr/include/math.h" 2 3 4
# 162 "/usr/include/math.h" 3 4
extern int signgam;
# 203 "/usr/include/math.h" 3 4
enum
{
    FP_NAN =
        0,
    FP_INFINITE =
        1,
    FP_ZERO =
        2,
    FP_SUBNORMAL =
        3,
    FP_NORMAL =
        4
};
# 315 "/usr/include/math.h" 3 4
typedef enum
{
    _IEEE_ = -1,
    _SVID_,
    _XOPEN_,
    _POSIX_,
    _ISOC_
} _LIB_VERSION_TYPE;

extern _LIB_VERSION_TYPE _LIB_VERSION;
# 340 "/usr/include/math.h" 3 4
struct exception
{
    int type;
    char *name;
    double arg1;
    double arg2;
    double retval;
};

extern int matherr (struct exception * __exc);
# 502 "/usr/include/math.h" 3 4
# 3 "islem.c" 2
# 4 "islem.c"
int main(void)
{
    printf("Yazdır bakalım\n");
    [natgho@Optimist sistem_programlama]$
```

Hello world dosyası sonucu oluşturulan dosya.i içeriğinin bir kısmı

Farkedildiği üzere başlık dosyaları açıldığından artık bulunmamakta.

Include satırları genişlediğinden dolayı üst kısım başlık dosyalarının yerleştirilmesinden dolayı bu kadar büyüdü dosya. Makrolar yerleştirilmiş durumda ve komut satırları yok edilmiş durumda. (makro ve yorum satırları olaylarına çok takılma, genel anlamda ne yaptığını bil yeter hacı...)

*** “-save-temps” uzantısı ile gcc -o çıktı dosya_adi.c: tüm temporary dosyaları ile birlikte derler:

```
Uçbirim - natgho@Optimist:/home/natgho/sistem_programlama
Dosya Düzenle Göster Uçbirim Sekmeler Yardım
[natgho@Optimist sistem_programlama]$ gcc -save-temps islem.c
[natgho@Optimist sistem_programlama]$ ls
a.out islem.c islem.i islem.o islem.s
[natgho@Optimist sistem_programlama]$ ./a.out
Hello World[natgho@Optimist sistem_programlama]$
```

Gcc ile derlendiğinde hangi dosyaları oluşturduğunu görüyoruz. Windows altındada normalde bu ara dosyalar zaten oluşuyor, ama biz görmüyoruz. Linux'dada normalde gözükmemekte, bizim verdiğimiz parametreler sayesinde görüyoruz.

Öncelikle programın “.i” uzantılı çıktısı yani preprocessing işlemi gerçekleştirilmekte.

2)Compiling

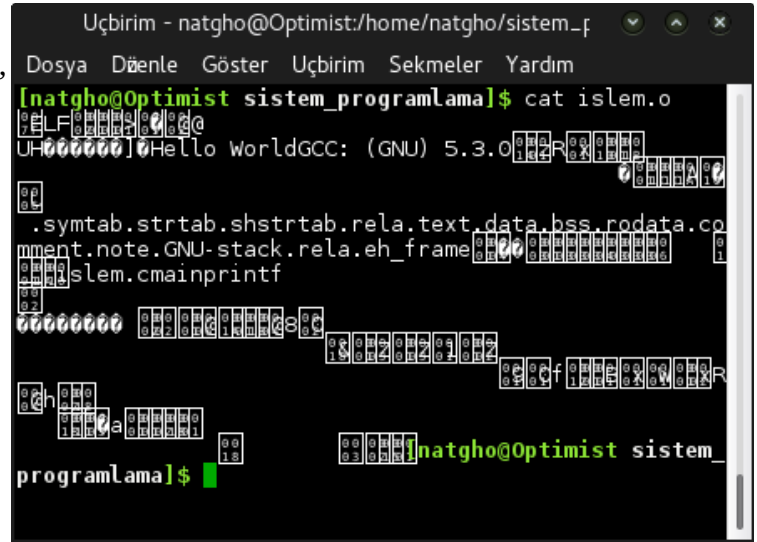
Elimizde .i uzantılı bir dosya var, bu dosya compiling işleminden sonra s uzantılı dosyalara dönüşüyor. “.s” uzantılı dosyalar assembly dosyalarıdır.

Dosyanın adı yazıyor, sectionlardan bahsediyor. Sectionların olduğu kısımlar ve derleyiciye dair bilgiler mevcut. (GCC 5.3.0). Başlık dosyasının yerleştirilmesi, makroların yerleştirilmesi, komut satırlarının çıkarılması gibi işlemlerden sonra ortaya çıkan dosya, “s” uzantılı assembly dosyasına dönüştürülür.

```
Uçbirim - natgho@Optimist:/home/natgho/sistem_
Dosya Düzenle Göster Uçbirim Sekmeler Yardım
[natgho@Optimist sistem_programlama]$ cat islem.s
.file "islem.c"
.section .rodata
.LC0:
.string "Hello World"
.text
.globl main
.type main, @function
main:
.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
movl $.LC0, %edi
movl $0, %eax
call printf
movl $0, %eax
popq %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
.size main, .-main
.ident "GCC: (GNU) 5.3.0"
.section .note.GNU-stack,"",@progbits
[natgho@Optimist sistem_programlama]$
```

3)Assembly:

Assembly işleminde “s” uzantılı dosyamız “o” uzantılı object dosyaya dönüştürülmektedir. Bu dosyaya object file yani nesne dosyası denmektedir. Bu aşamada artık binary dosyadırlar. Bu nedenle “cat” ile içeriğini yazdırdığımızda binary bir dosya olduğu için net görüntüleyememekteyiz.



4) Linking

Exe uzantılı dosyamızın olduğu kısım. Binary olduğu için onu da görüntüleyememekteyiz. Burada komut satırları dosyaya eklenerek program asıl şekline dönüşür. Bu nedenle dosya boyutu büyür. Object dosya iken dosyanın boyutu daha azken exe dosya haline kadar dosya boyutu çok daha küçüktür. Çünkü bu aşamaya kadar print komutu işleme sokulup programa eklenmektedir.

Loader: Bir programı çalıştıracığımız zaman diskten o programı okuyup ana belleğe koyan araçtır.

Elf Formatından Önce Kullanılmış Formatlar

a.out: Elfen önce kullanılan formatlardan birisi.

COFF: Yine elften önce kullanılan formatlardan biri, hala bazı yerlerde kullanılmakta.

ECOFF: Üsttekinin aynısı

XCOFF: Üsttekinin aynısı

PE: Portable executable olarak geçmekte. Windows sistemlerinde kullanılmakta.

ELF:

SOM/ESOM: HP ve IBM tarafından kullanılmakta. ((Extended) System Object Module)

Dosyada Bulunan Section'lar ve içerikleri

.text: Bu sectionda genellikle okuma ve yazma izni bulunmakta. Genellikle burada kod bulunmaktadır.

.bss: Henüz başlatılmamış değişkenler bulunmakta. Henüz değişken ataması yapılmamış değişkenler bulunmakta.

.data: Initialize edilmiş değişkenler bulunmakta.

.rdata: Başlatılmış sabitler bulunmaktadır. Yani “int i =0” yani ilk değeri verilmiş sabitler ya da değişkenler bulunmakta. (read-only data)

.reloc: Stores the information required for relocating the image while loading.

Symbol table: Sistem sembol tablosu

Relocation records: (hoca söylemedi)

Çok dosyayı bir arada çalıştırmamız için ne yapmamız gerekir?

Örnek:

Öncelikle set.h diye bir başlık dosyası oluşturuyoruz, içerik olarak sadece ;
int x;

void set(int i);

bulunmakta, yani değişken ve metod prototipi tanımlaması yapılmış durumda.

Ardından set.c oluşturulmuş;

#include "set.h"

void set(int i)

{

x=i;

}

Ve main.c oluşturulmuş durumda;

#include <stdio.h>

#include "set.h"

int main() {

set(1);

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

set(5);

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

}

İşlem olarak main içerisinde set.h include ediliyor, include edilen set ise ilk satırda 1 ve 5 değeri sırayla yazılıp x içerisine atılması için set metoduna gönderiyor, set metodunda kendi içerisinde x değerini dışarıdan gelen i değerine eşitliyor. X değeri global tanımlı olduğundan istediğimiz yerden ulaşabilmekteyiz. Include edildiği için de ona ulaşılabilir.

```
Uçbirim - natgho@Optimist:/home/natgho/
Dosya Dzenle Göster Uçbirim Sekmeler Yardım

[natgho@Optimist set_ornek]$ cat
main.c set.c set.h
[natgho@Optimist set_ornek]$ cat set.h
int x;

void set(int i);

[natgho@Optimist set_ornek]$ cat set.c
#include "set.h"

void set(int i) {
    x=i;
}

[natgho@Optimist set_ornek]$ cat main.c
#include <stdio.h>
#include "set.h"

int main() {
    set(1);
    printf("x=%d\n", x);
    set(5);
    printf("x=%d\n", x);
}

[natgho@Optimist set_ornek]$ gcc set.c main.c
[natgho@Optimist set_ornek]$ ./a.out
x=1
x=5
[natgho@Optimist set_ornek]$ ls
a.out main.c set.c set.h
[natgho@Optimist set_ornek]$
```

Burada birden çok aynı kere set.h include edilmiş durumda, bu problemlere yol açabilmekte. Buna **“include guard”**(include koruması) denir.

Bu korumayı sağlamak için include koruması yapılır, bir makro yazılarak;

```
#ifndef GRANDFATHER_H
#define GRANDFATHER_H

struct foo {
    int member;
};

#endif /* GRANDFATHER_H */
```

yazılır, böylece ifndef ile grandfather_H daha önce tanımlanmış mı diye kontrol eden bir makro oluşturulmuş olur. Tanımlanmamışsa tanımlanmış olur, tanımlanmışsa hatayı engelleyerek tekrar tanımlamasının önüne geçer. Bunun mutlaka yapılması gerekir .(hoca önemsedi, örneği burada detaylı [olarak](#), verdiğim linkten bi bak anlarsın bro)...