PROGRAMLAMA SANATI VE TEKNİKLERİ

Önsöz

Bu doküman ile daha esnek ,daha kararlı ve hata oranı en az indirgenmiş programların nasıl yazmanız gerektiğini örnekler ve ipucları vererek anlatmaya çalışacağız .*Program yazmanın bir sanatı olur mu ?* diyebilirsiniz belki;ama yazdığınız kodu belirli kıstaslara göre yazarsanız gerçekten yazması ve okunması güzel kodlar ortaya çıkartabilirsiniz. Çoğu programcı başka kişilerin yazdığı kodları tam anlayamamakta ve yenisini kendim yazarım mantığında hareket etmektedir .*Ve piyasada programlama dilleri üzerine sayısız kitap mevcut iken nasıl program yazılması gerektiği hakkında ve algoritmalar önemi ile alakalı kitap sayısı is yok denecek kadar az* .Bu doküman sizlere nasıl *program yazılmaması* gerektiğini gösteren bir doküman niteliğinde olacaktır.

Unutmayın ki ,Herkes bir program yazabilir ama iyi ,esnek ve hata oranı en az olan kodları yazmak gerçekten emek,alışkanlık isteyen bir konudur.

Bu doküman temel aldığı diller C/C++ olacaktır .Neden başka diller(C#,Java,Delphi) değil diyebilirsiniz .Burada her hangi bir dil tartışmasına girmek niyetinde değilim .Burada okuyacağınız çoğu kavram diğer diller içinde birebir olmasa da uygulanabilir yöntemler ile sağlanabilir . Bu doküman için C/C++ dillerini anlayabilecek kadar bilgi birikiminiz,yeri geldiğinde biraz matematik bilginizin olması bu dokümanı anlamanız için yeterli olacaktır . Platform olarak ben ***linux*** ve ***gcc 3.2.2*** üzeri bir derleyici kullanmaktayım. Hata bulma aracımız ***gdb*** olacaktır. Bu dokümanda ki çoğu örnek Ansi C/C++ için geçerli olmakla birlikte yeri geldiğinde platform tabanlı örnekler verilecektir .Değinilmesi gereken bir noktada bu dokümanda ***?*** işareti ile başlayan kod blokları güvenilir veya anlaşılır olmayan kod bloklarını ifade etmektedirler.

Bu dokümanın her seviye programcı için yararlı olacağına inanmaktayım .

iskender atasoy

BÖLÜM 1

KOD YAZIM STİLİ VE GENEL HATALAR

1.1 Kodlama Stilinde Dikkat Edilecek Unsurlar

1.1.1 Boşluklar Önemlidir ?

Derleyiciler için yazdığınız kodun nasıl olduğu önemli değildir .İster yorum satırı koyun ,ister binlerce değişken koyun,binlerce çeşit değişken ismi verin derleyici için fark etmez .O sadece görevini yapan bir ara birimdir. Yani her şey sizin elinizde.

Yeni bir dile başlayan herkesin ilk yaptığı örnek olan selam dünya'nın iki değişik versiyonu size sunmak istiyorum hangisi göze daha hoş geliyor.

selamdunya1.c

```
#include <stdio.h>
int main(){printf("Selam dunya\n");return 0;}
```

gcc selamdunya1.c ile derleriz
./a.out programı çalıştırıyoruz.Program çıktısı aşağıdadır.

```
root@localhost# ./a.out
Selam Dunya
```

ikinci örneğimiz

selamdunya2.c

```
#include <stdio.h>

int main(){
        printf("Selam dunya\n");
return 0;
}
```

yukarıdaki gibi derlerseniz iki programda aynı çıktıyı verecektir..Bu belki bazılarınızı şaşırta bilir ama,Derleyici için bu iki kod aynı şeyi ifade eder.Fakat biz insanlar için okunulabilirlik önemlidir.Yani her zaman okunabilir kodlar yazmamız gerekmektedir.İlk kod örneğimiz yazım olarak doğru olsa bile mantıken yanlış bir koddur.Unutmamalıyız ki yazılan kodlardaki boşlukların ve yeni satırların bile gerçekten önemli olduğudur..Biz programcılar için çok şey ifade ettiğini.Unutmamalıyız ki Boşluklar ve Satır Aralıkları önemlidir;Kodunuza Daha anlaşılır ve okunabilir yapar.

1.1.2 Değişken İsimleri !

Program yazarken fonksiyonlarımız ve değişkenlerimiz için değişik bir çok isim kullanmaktayız .Peki bunları seçerken hangi kritere göre seçtiğinizi hiç düşündünüz mü ?,Eğer değişkenlere gelişi güzel isimler veriyorsanız bu kodun okunabilirliğini etkileyebileceğini unutmayınız .Peki değişken isimlerini seçerken neye dikkat etmeliyiz ?

1-) Değişken isimleri mutlaka İngilizce ve değişkenin karakteristiği ile alakalı olmalıdır.

Neden İngilizce der iseniz programlarınız çoğu zaman bir çok programcı inceler,İngilizce programla dilinde bir sabittir. Bu nedenle değişken isimlerini ve yorumları yazacak kadar İngilizce bilmeniz gerekmektedir. Ama derleyici için İngilizce ,Türkçe ayrımı olamadığını hatırlatmak istiyorum. Bu sizin yazdığınız kodun daha fazla kişi için okunulabilir olmasını sağlayacağını unutmayın.

Örneğin:Bir dizinin toplamını bulan bir örnek :

```
?int s,i;
?...
?...
?for(i=0;i<10;++i)
?s+=i;
```

bu kod işleyiş açısından doğru olmakla birlikte **s** değişkeni ne ifade etmekte. Amaç değişken tanımlamalarına mümkün olduğunca bakmadan işleve uygun isimler seçmek gerektiğidir .Burada **s** değişkeni akılda kalır bir özelliği bulunmamaktadır. Kod'a defalarca bakıp **s** değişkeninin ne yaptığını düşünmenize sebep olacaktır. Bu kodu aşağıdaki şekilde daha anlaşılır bir şekilde yazılabilir :

```
int sum=0,i;
…
…
for(i=0;i<10;++i)
sum+=i;
```

programı okuyan kişi **sum** değişkeninin bir dizinin toplamı çağrıştırmaktadır .İstediğiniz isimde değişken kullanabilirsiniz .Ama işlev ile alakalı isimler kullanmanız kodun okunabilirliğini arttırmaktadır.

2-) Global Değişkenler için *tanımlayıcı*,Yerel değişkenler için *kısa isimler* kullanmamız gerekir.

Global değişkenler bir program veya programın modüllülerinde her tarafından erişilebilen adı üstüne global olan değişkenlerdir. Global değişkenler fonksiyon,yapı(struct),değişken olabilir.

Global değişkenlerin avantajlarını ve dezavantajlarını ilerleyen bölümlerde açıklanacaktır .Yerel Değişkenler için ise kısa ve tanımlayıcı kelimeler seçmemiz gerekmektedir. Örneğin

```
?for(the_arrays_numbers=0;the_arrays_numbers<10;++i)
?sum+=the_arrays_numbers;
```

bu örnekte **the_arrays_numbers** değişkeni kısaca sayaç demekteyiz.Genel programlama kanunu olarak diyebiliriz ki döngüler için ***i,j,k*** tarzında değişken isimleri programcıların hepsi tarafından genel olarak tercih edilen bir yöntemdir.Yukarıdaki örneği aşağıda daha yaygın ve anlaşılır kullanımı görmektesiniz.

```
for(i=0;i<10;++i)
sum+=i;
```

3-) Değişkenleri tanımlarken türlerini açıklayıcı ön ekler tercih etmeliyiz.

Değişkenlerle uğraşacağımız için çoğu zaman gösterici,işaretli ve işaretsiz değişkenler kullanabiliriz .Gösteriler belki de C/C++ için en karmaşık gelen ve aynı zamanda en zevkli kısmı olduğunu söyleyebilirim .Bir Değişkenin gösterici olup olmadığını belirtecek ön tanımlamalar yapabiliriz.
Örneğin göstericilerin(**Pointer** ingilizcesi) baş harfi olan ***p*** değişkenlerimiz için açıklayıcı olabilir .Yani

```
?char *name;
```
yerine
```
char *p_name veya *pname;
```

olarak değiştirirsek ve dökümanımızda **p_** veya **p** ile başlayan her ifadenin pointer olacağı şeklinde bir kural tanımlayabiliriz.Bu şekilde **char** türündekiler **c** ile başlayabilir.Yani

```
char *pc_name veya *pcname;
```

Aynı zamanda C/C++ için typedef kelimesi ile kendi tanımlı türlerimizi oluştura biliriz.Örneğin

```
typedef char        *string;
typedef unsigned    char   u8;
typedef unsigned    short u16;
typedef signed      short i16;
typedef unsigned    int    u32;
typedef unsigned    long  ul32;
```

tarzında tür tanımlamaları ile daha anlaşılır kodlar yazılabilir.

Unutulmamalıdır ki detayda yapacağınız ince ayrıntılar sizin programcılık veya program yazma seviyesini göstermektedir.

4-) Tanımlanan Fonksiyonlar ve Yapılar/Sınıflar İşleviyle Alakalı olmalıdır.Kullandığımız Fonksiyon isimleri onların işlevi ile alakalı olmalıdır .Örneğin:

```
?if(checkname("ali can") ){
?...//birşeyler yap..
?}
?else
?...// birşeyler yap..
```

yukarıdaki fonksiyonun işlevinin isimler dizisinden **ali can** isimli bir kişinin olup olmadığına baksın eğer kullanıcı var ise doğru ,yok ise yanlış versin .Bu fonksiyonu daha anlaşılır kullanımı şöyle olabilir.

```
if(isname("ali can") ){
?...// birşeyler yap..
}
else
?...// birşeyler yap..
```

Genellikle dogru veya yanlış ifadelerinde(0 veya 1) şeklinde "***is***" takısı ile kullanılmaktadır.örnek kullanım ile alakalı Ansi C fonksiyonları **ctype.h**(***isascii(..)*** gibi) içerisinde inceleyebilirsiniz.

Hatırlatma !

Buraya kadar fonksiyon ve değişkenlerimizi İngilizce terimlerden seçtik .Bir kez daha yineliyorum .Genel İngilizceyi bilmenizde fayda bulunmaktadır .İnternetten indireceğiniz hiçbir kodda Türkçe açıklama bulamayacaksınız programın yazarları Türk veya başka ırktan olursa olsun .*Tekrar yazıyorum Genel Kural olarak benimseyiniz kodunuzun daha çok insana hitap etmesini istiyorsanız İngilizce kullanımı ile yazmanızda fayda var*. Ayrıca GNU Standartlarına uygun bir kod yazmanız için en temel koşullardan biri yorumlarınızın İngilizce olması ve Diğer belirlediği kıstaslara uygun olması gerekmektedir .Bunlar yeri geldiğince örneklendirilecektir.

Bir fonksiyona değer aktarırken **set*** ** ,değer alırken **get*** ** tarzında genel kullanım şekillerini tercih etmenizde fayda var. Örnek Bir Yığın yapısının nasıl olabileceğini göstermektedir.

```
class Stack
{
  int item;                          //Bizim elemanımız
  int length=0;                      //Stack yapısının boyu
  Stack*next;                        //Sonraki elemanı gösteren yapı
  public:
  int push();                        //bir eleman çekerken
  void pop(int item);                //bir eleman eklerken
  int getitem(int i);                //i indisli elemanı getirir
  void setitem(int i,int newitem);//i indisli elemanın değerini değiştirir
  int getsize() const;               //stack yapısının toplam eleman sayısını tutar
};
```

bu Yığın yapısının **olamaması** gereken hallerinden biri :

```
? class stackclass
? {
?int the_item_of_stacks;      //Bizim elemanımız
?int the_length_of_stacks=0;//Stack yapısının boyu
?stackclass *next;            //Sonraki elemanı gösteren yapı
?....
?};
```

Genel olarak bu şekilde Stack kullanımı yukarıda ki doğru kullanım diyebiliriz .Burada kadar e kodlama kısmına girmek istemedim .Çünkü bu hali Stack yapısının olabilecek en basit halidir .Çünkü Bu örneği Ara birimler ve şablonlar ile ilerleyen bölümlerde daha yararlı ve istenen şekle getireceğiz .Şu an sadece size bir fikir vermesi adına kullanımını

vermekteyim .Bu şekliyle bile programın yapısı hakkında önemli bilgiler vermektedir.
Değişken kullanımın ve fonksiyon seçimi için Ansi C/C++ fonksiyonlarını örnek verebilirim .Ansi C/C++ kodlarını incelemenizde fayda bulunmaktadır.

Egzersizler:

1-) Aşağıdaki kodu okumaya çalışınız

```
?#define SRMCHCSGFG 0xF0000//Bios Başlangıcı Adresi
?#define FDFGDFGFGG 0xFFFFF//Bios Bitiş Adresi
?
?if(getmemory(SRMCHCSGFG,FDFGDFGFGG,memarias)<0)
```

2-) Aşağıdaki kodu iyileştiriniz.

```
?int _strlen(char *str)
?{
?int theindexs_of_string=0;
?int thelength_of_string=0;
?for(theindexs_of_string=0;str[theindexs_of_string]!=NULL;theindexs_of_str
ing++)
?++thelength_of_string;
?return thelength_of_string;
?}
```

3-) Aşağıdaki kod için düşününüz?

```
?.....
?charbuf[20];
?printf("ismin nedir :");
?scanf("%s",buf);
?printf("Selam Dunya ve %s",buf);
?.....
```

1.2 Karar İfadeleri ve Durumlar

Değişkenler ve fonksiyonlar ile programlarımızın ana iskeletlerini oluştururuz .Karar ifadeleri ile yapmak istediklerimizi gerçekleştiririz .Karar ifadelerini daha anlaşılır bir şekilde oluşturabilirsek programlarımız daha anlaşılır olacaktır .Bunlar aşağıda değinilmiştir.

Karar ifadelerinde Boşlukların Önemi

Karar İfadeleri ile uğraşırken okunulabilirliği arttıracak şekilde boşluk ve yeni satır özelliklerini kullanmalısınız .Örneğin:

```
?for(i=0;str[i]!=NULL;++i);
?*str[i]='\0';
```

bu örnek gerçekten çoğu insanın ilk bakışta çözemiyeceği bir ifadedir.Bunu yerine

```
for(i=0;str[i]!=NULL;++i)
;//bos döngü
*str[i]='\0';
```

şeklinde yazar isek daha farklı bir anlam çıkar. İlk örnek insanlara **str** kelimesinin her karakterini sıfırlıyor manasına gelebilir. Bir tane noktalı virgül gerçekten olayı bazen anlamamızı zorlaştırabilir.

Çoğu zaman for ve if bloklarında açma "{" ve kapama "}" işaretlerini kullanmamız okunulabilirliği arttırır .Örneğin:

```
?if(strcmp(name,"root")==0)
? if(strcmp(password,"123456789")==0)
? root_exec();
?else
? error(name,password);
```

tarzında ifade GNU Standarlarında Yazılırsa Şu şekilde olması gerek.

```
if(strcmp(name,"root")==0)//adı root
{
 if(strcmp(password,"123456789")==0)//şifresi 123456789 ise
 root_exec();//root kullanıcısı aç
 else
 error(name,password);//değil ise hata ver
}
```

Burada en basit yapıda kullanıcı adı ve şifresi sorgulanmaktadır.Yukardaki ifadenin daha iyi şekilde nasıl yapılabileceğini düşününüz ?

GNU Standardında if/else bloklarının durumu şöyle olmaktadır.

```
?if(first_case)
?.....
?else if(second_case)
?....
```

Bu ifade

```
if(first_case)
.....
else
{
 if(second_case)
 ....
}
```

bu şekilde olacaktır.Unutmamalıyız ki tüm bu parantez ,blok açma ve kapama ifadeleri daha anlaşılır program yazmamız için kullanılmaktadır.

1.2.1 Parantez Kullanma ve Önemi

Parantez Kullanmak çoğu zaman operatör kullanımlarında bize fayda sağlamaktadır. Çoğu programcı hangi operatörün önceliği daha fazla hangisinin daha az olduğunu tam bilemeyebilir.
Ama *Tüm programcılar Parantez operatörünün en öncelikli operatör* olduğunu bilmektedir. Yani karışık ifadeler yazmaktansa parantezlerle daha açık anlamlı ifadeler oluşturabiliriz. Örneğin

```
?sum=x+++y;
```

Bu ifade **x++ + y** 'mi yoksa **x+ ++y** 'mi olarak derleyici tarafından çözümlenecek. Bu çıktı derleyicinize göre değişebilir . Yapmak istediğiniz işleve göre birini tercih etmenizde fayda var.

```
sum=(x++)+y; veya sum=x+(++y);
```

olarak değiştirmeniz okunulabilirliği arttırmaktadır. Programı inceleyen bir kişi bu tarz bir ifadeyi daha iyi anlayabilir. Yazdığımız kodları daha anlaşılır bir şekilde yazmamız kodumuza yeni özellikler eklediğimizde neyi niçin yapmıştım sorusunu ortadan kaldırır. Çoğu zaman yazılan bir kod bloğu 2-3 hafta sonra incelendiğinde gerekli yorum satırı veya yazım standartlarına uyulmamış ise kodu yazan kişiler bile neyi niçin yaptığını unutmaktadır.

Başka Bir örnek

```
?x= x1*3-20*y1+2;
```

ile

```
x=(x1*3)-(20*y1)+2;
```

aynı ifadedir.

1.2.2 Komplex İfadeleri Sadeleştiriniz

C/C++ ile program yazarken göstericiler,operatör kullanımı ve bunların sonucunda karışık ifadeler ortaya çıkabilmektedir. İyi bir programın kodu açık ve anlaşılabilir olmalıdır. Yani herkes tarafından kolaylıkla anlaşılabilmelidir. Örneğin

```
?*fx +=(*y=(2*x<x1-x2)?((x*x-(x1-x2)*(x1+x2))):(x-(x1-x2)*(x1+x2)));
```

yukarıda ki ifade ile aşağıdaki ifade aynı anlamlamdadır. Hangi ifade daha açık sizce ?

```
if(2*x<x1-x2)
*y=(x*x-(x1-x2)*(x1+x2));
else
*y=(x-(x1-x2)*(x1+x2));
*fx +=*y;
```

Çoğu zaman göstericilerin adreslerini üzerine işlemler kullanmaktayız .Bazı Durumlarda bunlar kodumuzun karmaşıklığını arttırıcı unsur olmaktadır .Yazılan kodlarda her zaman sadelik ve işlevsellik olması gerekir. Örneğin:

```
?1 int numbers[]={10,12,..,52}; //elimizdeki sayi dizisi
?2 int temp[30];                //şimdilik 30 olsun
?3 int *array1,*array2;
?4
?5 array1=numbers;//array1'e numbers değeri atanıyor
?6 array2=temp;//array2'e 30 dizilik yer atanıyor
?7
?8 while(*array1)
?9 *array2++=*array1++;
```

örneğimizde bir dizinin elemanlarının başka bir diziye aktarılmasını görmekteyiz kafamızı karıştıran 8,9'uncu satırdaki ifadelerdir. İlk bakışta

bile ifade karmaşık bu ifadenin daha açık bir şekilde yazdığımızda okunulabilirliği daha da artacaktır. Ayrıca önemli kod bloklarında ne yaptığımızı yorum satırlarıyla da açıklamayı unutmamalıyız.

```
1 int numbers[]={10,12,..,52};//elimizdeki sayi dizisi
2 int temp[30];//şimdilik 30 olsun
3 int i;//sayaç değerimiz
4 int *array1,*array2;
5
6 array1=numbers;//array1'e numbers değeri atanıyor
7 array2=temp;//array2'e 30 dizilik yer atanıyor
8
9 for(i=0;arrays1[i]!=NULL;++i)
10 array2[i]=array1[i];
```

Sadece 1 satır fazladan ekledik.for döngüsünde daha açık bir şekilde ne yapmak istediğimizi açıkça göstermektedir.Bu kodumuzun okunulabilirliğini arttırmaktadır.

++,-- gibi operatörleri kullanırken dikkatli olamanız gerekir.Örneğin:

```
?printf("%d %d %d",i++,array[i++],i++);
?printf("%d %d",i++,i++);
?array1[i++]=array2[i++]=i;
?array[i++]=i;
```

tüm bu şekilde kullanımlar ciddi problem teşkil etmektedirler.Her zaman aynı fonksiyonu daha basit ve anlaşılır bir şekilde yapmaya çalışmalıyız.Kimse en komplex kodu yazma savaşında değil bunu unutmamalısınız.

1.2.3 Döngüler ve Operatörler ile İlgili Bazı İpuçları

- eger sonuza bir döngü kurmak istiyorsanız

  for(;;)
  dan ziyade
  while(1) veya while(TRUE)

  tercih etmelisiniz TRUE ifadesinide

  #define TRUE 1
  #define FALSE 0

tarzında oluşturabilirsiniz.

- Eğer C++,Java vb. bir dil kullanıyorsanız döngülerde

  for(int i=0;i<20;++i)

  tarzında for döngüsü içinde değişken tanımlayabilirsiniz.C için bunu kullanamazsınız. C 'de değişkenler mutlaka tanımlanmalıdırlar.

- Döngülerden for,while döngüsü aynı işleve sahiptir.ama do..while daha

  Özel bir hal olduğunu unutmamalısınız. **İf/else** yapıları ile **switch** bloklarına kolaylıkla dönüştürülebilir. Ama Switch yapısı kodumuzu daha sade bir hale getirebilir. Örnek

```
menuno=getchar();
switch(menuno)
{
case 0 : //ekle menusu
case 1 : //sil menusu
case 2 : //listele menusu
case 3 : //cıkıs menusu
case 4 : //hakkımızda menusu
default: //hatalı tus
}
```

  tarzında düşünülebilir.Bu kodu daha da anlaşılır yapabiliriz.Nasıl mı derseniz ? 0,1 değerlerine #define ile kendi isim sabitlerimizi koyabiliriz yani

```
#define M_EKLE0//Menu_Ekle tipinde
…
```

  şeklinde ekleyebiliriz. Ama bundan daha etkili bir yöntem ise enum tipinde yapı tanımlamak ile daha esnek bir yapı tasarlayabiliriz. Yukarıdaki örnek

```
typedef enum
_MENU{M_EKLE=0,M_SIL,M_LISTELE,M_CIKIS,M_HAKKIMIZD
A}MENU;
…..
…..
MENU menuno;//veya enum _MENU menuno; gibi
```

```
scanf("%d",&menuno);
switch(menuno)
{
case M_EKLE : //ekle menusu
case M_SIL : //sil menusu
case M_LISTELE : //listele menusu
case M_CIKIS : //cıkıs menusu
case M_HAKKIMIZDA : //hakkımızda menusu
default: //hatalı tus
}
```

görüldüğü gibi ifade daha anlaşılır ve geliştirmeye açıktır.

- ++,-- operatörlerini kullanırken dikkatli olmalısınız. Göstericiler ile kullanırken iki kere daha dikkatli olmalısınız.

  Unutmamalısınız ki
  int x=10,y;

  y=++x;
  y=x++;
  ifadeleri aynı manada değildir.
  For döngüsünde ++değişken tarzında kullanmalısınız
  yani
  for(i=0;i<20;i++) den ziyade
  for(i=0;i<20;++i) 'yi tercih etmelisiniz.
  İki for döngüsü de aynı işleve sahiptir.

Yorum Satırları Önemlidir !

Program yazarken mutlaka fonksiyonlar , değişkenler ve kod içi yorumlarımızı eklememiz gerekmektedir. C Dilinde yorum satırları /* *yazılacak ifade*/* ile gösterilmektedir. C++ 'ta ise satır içi yorumlar ve Birden çok satır içi yorumu koyabildiğimiz yorum çeşidi bulunmaktadır. Örnek vermek gerekirse

```
/*Bu satırı hiç bir C/C++ derleyicisi okumaz
Gönül rahatlığıyla yazabilirsiniz.
*/

// Bu tek satırlık yorumdur. C++ destekler.
```

Bu şekilde yorumlarımızı ekleyebiliriz. Genellikle GNU Standardında her program kodunda programın yazarı ve GNU şartlarının olduğu yorum satırı

olmalıdır. Selam Dünya örneğini tekrar yazacak olursak eğer.

Helloworld.c

```
/*
* helloworld.c
*
* Copyright (C) 2005 İskender Atasoy
* mail <iskenderatasoy@yahoo.co.uk>
*
* Eger Program üzerinde eklemeler yapıldı ise kim nezaman
* yaptığını belirtilir.
*
* This program is free software; you can redistribute it and/or modify
* it under the terms of the GNU General Public License version 2 as
* published by the Free Software Foundation.
*
*/

#include <stdio.h>

void write_message(char *);

int main(){
write_message("Selam Dunya");
return 0;
}
/*
English Comment
write_message(char *message) functions takes message argument and
print to the console.
Char *message --> Printed Message
Türkçe Yorumu
write_message(char *message) fonksiyonu message cümlesini konsol ekrana
çıkartır.
Char *message --> Ekrana Çıkacak Mesaj
*/
void write_message(char *message)
{
printf("%s\n",message);
}
```

İyi bir programda tüm fonksiyonların işlevlerinin iyi bir şekilde açıklanması gerekir .Küçük programlar için belki bu dediklerim önemsenmeyecek ayrıntılar olabilir .Peki Yüz binlerce,Milyonlarca Satır Bir programı yazanlar iyi dokümante edilmemiş bir kodda gerçekten çok çok zaman ve emek harcayacakları kesindir .GNU bu probleme kendi koyduğu bir standart ile çözüm bulmaya çalışmaktadır .Her firma kendince bir çözüm üretebilir veya mevcut çözümleri kullanabilir.

1.3 Önişlemci İfadeleri ve Tehlikeleri

Yazdığınız C kodu 2 ayrı modülden oluşmaktadır.(*Ref. A dan Z'ye C Kaan Arslan Kitabından* Alıntıdır.)
1-) Ön işlemci Modülü
2-) Derleme Modülü

Ön işlemci modülü kaynak kod üzerinde bir takım değişiklikler yapan ön bir modüldür. Yani Derlenecek kod ilk önce ön işlemci modüllerine dönüştürülür daha sonrada kullanılan ön işlemciler kaynak kodda değiştirilir .Tüm ön işlemci komutları # işareti ile başlar. Tanımlı ön işlemci ifadeleri:

#define	#elif	#else	#endif
#error	#if	#ifdef	#ifndef
#include	#line	#pragma	#undef

Program yazarken mutlaka dosyalarla işlem yapmaktayız.Bu java olsun,C++ olsun ,Python olsun her dilde mutlaka dosyalarla çalışmaktayız.C/C++ 'da da dosyalarla çalışmaktayız.Dikkat ederseniz C dilinde kullandığımız dosya türlerinin sonu ***.h** ile biter.Biz bu tür dosyalara ***header*** deriz. C programları *.c ve *.h dosya yapılarından oluşmuş bir kümedir. C++ dilinde ise her türlü dosya ismi koda eklenebilir . Header kavramın biraz geliştirilmiş bir hali diyebilirim . Bir header dosyası kendimiz veya var olanlardan kaynak kodumuza ekleyebiliriz .Örnek:

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

Bu stdio.h dosyasını kaynak kodumuza eklemektedir.

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

bu şekilde bir kullanım hata verecektir.

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

şeklinde kullanmamız gerekiyor.#include ifadesinden sonra <header ismi> veya "header ismi" şeklinde kullanımı mevcuttur.

Header yapıları hakkında bilmeniz gereken bir noktada *.h dosyaları içinde programda kullanacağımız fonksiyonların prototiplerin bulunduğudur. Çoğu zaman yazılan kodlar modüllerden oluşmaktadır. Modül kavramını ileride daha detaylı değineceğim. Modüller *.c uzantılı c kodlarıdır. Normal şartlarda tüm kodları ana bir c dosyamıza koyabiliriz. Bu bize kod içinde büyük bir hammallık getirir. Modüler yapıda ise kolay güncellenebilen ve takım çalışması için uygun bir yaklaşımdır.

Bir header(*.h) içinde bulunması gerekenler :

1-) Programın hangi koşullar altında dağıtıldığı yani ufak bir lisan sözleşmesi
2-) Modül veya fonksiyonun işlevi.
3-) Revizyon numarası.
4-) Header adının tanımlanıp tanımlanmadığı kontrol edilmeli.#ifndef
5-) #define ve #include ve diğer gereken ön işlemciler..
6-) Sınıf ve Yapıların genel yapısı hakkında bilgi bulunur.
7-) gerekli tür tanımlamaları
8-) C/C++ için #endif eki.#ifndef başlayan her ön işlemci ifadesi bu şekilde kapatılmalıdır.
Ve sadece prototipler bulunur.

Standart Bir Header yapısı :

stdwrite.h adından

```
/*
Copyright © iskender atasoy
All rights reserved.
write_message(char *message) functions takes message argument and
print to the console.
$ID$
*/

#ifndef _STDWRITE_H_

#define _STDWRITE_H
// Dosyayı Tanımla.

void write_message(char *);

#endif
```

Aynı zamanda bu dosyayı eklememiz gerekecek olan C kodumuz

stdwrite.c kodu

```
#include <stdio.h>
#include "stdwrite.h "


void write_message(char *message)
{
printf("%s\n",message);
}
```

bu bizim ana kodumuz ise

main.c olsun

```
#include "stdwrite.h"

int main()
{
write_message("Selam Dunya\n");
return 0;
}
```

derlerken

gcc -o deneme main.c stdwrite.c dememiz yeterli.

Çalıştırmak için ise **./deneme** demeliyiz.Bu basit kod örneğinde bili **stdwrite** adında basit bir header yazmış olduk.Gerekli makefile'ımızı oluştursaydık sadece gerekli modül/ler kaynak kod üzerinde derlenecekti.Şimdilik kafanızı bu konuya takmayınız modüller hakkında ileride linux üzerine detaylı bir anlatım yapılacaktır.

Birde söylemeden geçmeyeyim.Bu 3 dosya aynı dizindedir.Yani include ederken dizinin yolunu vererekte derleyebiliriz.Derlediğimiz dizinin altında bir çok alt dizin ve bunlarda kodlarımız olabilir.Hiyerarşiyi dikkate alınız.

#define önişlemcisi

Belkide en çok kullanılan önişlemci komutudur.#define önişlemcisi ile yapılabilenler:

1-) Sabit tanımlamaları
2-) Fonksiyon Tanımlamaları ki yerinde kullanıldıgında performansı arttırıcı etkisi olmaktadır.

3-) Yeni tür tanımlamaları
4-) Genel yazılım kuralı olmakla beraber. Büyük harf tercih edilir. Okunabilirlik adına.

Örnek kullanımlar

```
#define TRUE 1
#define FALSE 0
```

veya

```
#defineBIR 1
#define IKI BIR+BIR
#define UC BIR+IKI
```

printf("%d %d %d \n",BIR,IKI,UC); yazdığımızda linker programımızı
printf("%d %d %d \n",1,1+1,1+2); şekline çevirecektir.

Yani ön işlemcide oluşan sabitleri linker programı derlerken yerine yerleştirmektedir.
Başka bir örnek

```
#define HATA "Programınız yanlıs bir islem yaptı\n"
```

şeklindeki bir hatayı devamlı kullanıyorsak ön işlemci olarak tanımlamamızda fayda var.

printf(HATA);
ile
printf("Programınız yanlıs bir islem yaptı\n");

aynı çıktıyı verir.

Başka bir kullanımı

```
#define MAX_SIZE 100
...
int buffer[MAX_SIZE];
int i=0;

for(i=0;i<MAX_SIZE;++i)
buffer[i]=i;
```

şeklinde de dizilerin toplam uzunluklarını belirlemede kullanabiliriz. Bu yaklaşımda MAX_SIZE isteğe göre arttırıp azaltabiliriz. Buda bize daha esnek programlar yazmamıza yardımcı olmaktadır.

Başka Bir Kullanımı

```
#define UZUNCUMLE "Bu cumle gercekten bir satırdan \
                   Fazla sürecektir.Bu tür bir satırdan fazla \
                   ifadelerder ters bölü işareti kullanmaktayız."
```

#define önişlemcisinin makro olarak kullanımı

#define ön işlemcisinin bir diğer önemli kullanımı ise makro şeklinde çağırımıdır. Fonksiyon şeklindeki makrolar ile ön tanımlı fonksiyonlar oluşturabiliriz. Bu belki de en çok kullanılan #define yapısıdır.

```
? #define ABS(x) (x)<0 ? -(x) :(x)
?.....
?printf("%d\n",ABS(10));
?printf("%d\n",ABS(-10));
```

Deniyoruz ve evet çalışıyor. ***Mutluyuz değil mi ?*** Ama sanki ortada bir sorun varmış gibi geliyor. Biraz düşünelim.#define ön işlemcisinin ifadesi aynen koda yansıyordu değil mi ?
Evet. **Hımım**. Hala düşünüyoruz. Sizin aklınıza bir şey gelmedi mi. Ana Kuralımız Bir programcı ne yapmak istediğini gerçekten bilmelidir. Yazdığı kodu defalarca incelemelidir. Buradaki hata deneyimli kullanıcılar tarafından kolaylıkla sezildiğine eminim. Yeni başlayanlar eğer göremediyseler üzülmesinler çünkü ana amaç onlara bu tür eksik kod yazımlarından uzak tutmayı amaç ediniyor. Tekrar dönüyoruz kaldığımız yere.

```
? #define ABS(x) x<0 ? -x :x
?printf("%d\n",ABS(10));
```

ifadesi

```
printf("%d\n",10<0 ? -10 : 10);
```

bu doğru şimdilik. Üstelik derleyici size hata mesajı bile vermedi. Burada tekrarlamak istiyorum. **Derleyiciler bizim ustamız değil,Ancak çırağımız olabilirler.**

eğer biz

ABS(20-30) dersek

```
printf("%d\n",20-30<0 ? -20-30 : (20-30));
```

sonuç -50 dimi

Buradan öğrendiğimiz makro şeklinde fonksiyonlar tanımlarken parantezler çok önemlidir. **C++ programcılarına ise inline** tarzında makrolar tercih etmeliler.

Bu basit kod bile bize bir kabusa dönüşebilirdi.

C programları için :

```
#define ABS(x) ( (x)<0 ? -(x) : (x) )
```

şeklinde tercih etmelisiniz.

C++ programcıları ise

```
inline long abs(long number)
{
if(number<0)
 return -(number);
else
 return number;
}
veya
inline long abs(long number)
{
if(number<0)
 return -(number);
return number;
}
veya
inline long abs(long number)
{
return number<0 ?-(number): number;
}
```

Buradaki basit örnekte şunu belirmek istiyorum. Programlama dünyasında hedefe gidecek yol sayısı çok fazla diyebilirim. Size kalan uygun olanını seçmenizdir. Yukarıdaki 3 C++ kodu da aynı işlevselliğe sahipler. Buradaki felsefeye örnek vermemiz gerekirse.

"*Evinize gidecek sonsusuz yol kombinasyonu bulunmaktadır.Ama sizi eve en kısa götürecek yol kombinasyonu ise daha sınırlıdır. Yazdığınız Kodlarda da sonsuz çeşitlilik sağlayabilirsiniz ama hangisi daha işlevsel olacaktır. Genel amacımız bizi hedefe en çabuk götürecek açık,anlaşılır ve kolay geliştirilebilir*

kodlar yazmak olmalıdır Ki uzmanlık bu tecrübeleri harmanlamadan geçer. Herkes programcı olabilir.Ama iyi programcı herkes kolay kolay olamaz."

Kaldığımız yerden devam edelim genel #define işlemlerindeki hatalarına

```
?define KARE(x) x*x
?...
?1/KARE
```

ifadesi KARE(12) 'de çalışır. Ama KARE(12-20) veya KARE(0-20) de sonuç eklediğimizden farklı olacaktır. Tekrar tekrarlıyorum #define ön işlemcisinde parantezler çok önemlidir.

```
#define KARE(x) ((x)*(x))
```

şeklinde değiştirebiliriz.

```
?#define ROUNT(x) ((int) ( (x) + (((x)>0?0.5:-0.5)))
?...
?item=ROUND(x1+x2+y1+y2);
```

Bazen yazdığımız makro fonksiyonları 1 satırdan uzun olabilir. Bu tür ifadelerde \ ile bir alt satıra inebiliriz. Bu kullanışlı bir hata aktarma tekniği olabilir.

```
#define ERROR(message) (printf("Error was found in !\n" \
"Filename is\t:%s\n" \
"Line is\t:%d\n" \
"Error is\t:%s\n" \
,__FILE__,\
__LINE__,\
message))
```

tarzında hangi dosyada hangi satırda hata olduğunu mesaj ile birlikte veren bir örnektir.

Bunun örnek Kullanımı

```
char *buffer;

buffer=(char*)malloc(sizeof(char)*100);

if(!buffer)
{
ERROR("Not Enough Memory");
```

```
//türkçe olarak kullanıma göre
//ERROR("Yeterli Bellek Bulunmamaktadır.");
exit(0);
}
```

Tabi ki bu ERROR(..) yapısının sonuna ; eklemiyoruz .Kullanırken eklediğimiz için.

#define ön işlemcisi ile dikkat edilmesi gerekenler :

1-) Makro tanımlarken Parantezler çok önemlidir .Dikkatli kullanınız.
2-) C++ için program geliştirecekler inline fonksiyonları tercih etmeleri tasarım ve okunulabilirlik adına daha anlamlı olacaktır.

3-) Genellikle sabit tanımlamalarınızı #define ile yapabiliyoruz. Ama bu tarz ifadeler yerine enum tipinde yapılar tanımlamalıyız. Örneğin

```
#define SOL 1
#define SAG 2
#define UST 3
#define ALT 4
```

tarzındaki bir ifade yerine aşağıdaki gibi bir kullanım daha esnektir.

```
enum YON{SOL=1,SAG,UST,ALT};
```

Ama bir dizinin tanımlıyorsak veya bir tanımlama yapıyorsak örneğin :

```
#define PROC "/proc"
```

yararlı olabilir. Ama yorum size kalmış.

C++ kullanıyorsanız

#define MAX 10 dan ziyade
const int MAX=10**;** tarzında bir kullanım olmalıdır. Çoğu programlama kitabında aynen "***Macros is devil.***" yani makrolar kötüdür .Zorda kalmadıkça kullanmayınız der.

4-) Eğer ön işlemci ile ne yaptığınızı biliyorsanız yapınız. Bu bazı durumlarda performansı arttırıcı unsur bile olabilir. Genellikle makro seçimleri ufak fonksiyon tarzındadır. Eğer uzun ve çok CPU harcayan işlemleri makrolar ile yapmayı düşünüyorsanız dikkatli olun derim.

#error önişlemcisi

Bu önişlemci derleyiciyi kodu derlemesini engeller ve hatayı ekrana gösterir.

```
#error "Bu kod derlenecek mi !"
```

Kontrol İfadeleri
#if , #ifdef , #ifndef , #else , #elif ve #endif en çok kullanılan ön işlemci kontrol deyimleridir.

#if deyiminde önceden tanımlanmış ifadeler var ise istenilen işi yap.Yoksa Başka işleri yap.Yazım Kuralı :

```
#if geçerli bir ifade var mı?
Evet ise bunları yap.
#else
Hayır ise bunları yap.
#endif/*Endif ile her if blogunu kapatmalıyız.*/
```

örneğin :

```
#define SIZE 1024
....
int main()
{
#if SIZE>1023

printf("SIZE değeri 1023'ten büyük\n");
#else
printf("SIZE değeri 1023'ten küçük\n");
return 0;
}
```

En basit olarak kullanılabilecek ifade diyebilirim.Bu ifade blogu #elif ile arttırılabilir.

```
#ifdef ve #ifndef ifadeleri
Bunlar açık anlamıyla
#ifdef ifade /*ifade tanımlı ise*/
ile
if defined ifade aynı manada

#ifndef ifade /*ifade tanımlı değil ise*/
```

ile
if !defined ifade aynı manada dır.

Örnegin :
main.c

```
#include <stdio.h>
#ifdef LINUX
#include <stdlib.h>
#else
#include <conio.h>
#endif

int main()
{
#ifdef LINUX
system("clear");
printf("Bu örnek LINUX mimarisi üzerinde calisiyorENTER\n ");
getchar();
#else
clrscr();
printf("Bu örnek WINDOWS mimarisi üzerinde calisiyor.ENTER\n ");
getch();
#endif
return 0;
}
```

bunu linux'te **gcc main.c** derlediğinizde conio.h 'ı bulamadığını söylemektedir .Çünkü LINUX denen ön işlemci sabiti derlenirken tanımlanmamıştır. Bu soruna nasıl yanıt bulabiliriz sizce ?
GNU derleyicileri ön işlemci sabitlerini ekleyebileceğiniz ***-D**** seçeneğini sunar. Bu arada söylemeden geçemeyeceğim. GCC gerçekten piyasada bulabileceğiniz en mükemmel derleyicilerden biridir. Komut satırında çalışmak çoğu programcı için pek sevilmeyen veya yadırganan bir unsur. Linux'te değişik IDE'ler mevcut. Ama ben bu dokümanda komut satırından çalışan ifadeleri kullanacağım.
Bir kodu derlemek için en temel kullanım.

gcc main.c dersek eğer kodda hata yoksa ***./a.out*** diyerek programımızı çalıştırabiliriz.
Bu şekilde ise istediğimiz adda çalıştırılabilir kod oluşturulur.

gcc -o deneme main.c ./deneme çalıştırmak için.
hatta
gcc -o deneme.exe main.c ./deneme.exe çalıştırmak için.

Ama bizim kodumuz yukarıdaki şekillerde hata vermekteydi. Tekrar derliyoruz. Artık hangi ön işlemci sabitini kullanmak istediğimizi belirtiyoruz.

gcc -DLINUX -o deneme.exe main.c ./deneme.exe çalıştırmak için.

Ve çalıştı.Değil mi ?

Peki ilk derlediğimde niye çalışmamıştı.Bu ufak kodu yazarken şunu hedeflemiştik .Bu kod LINUX ve WINDOWS sistemlerde derlenebilsin. Windows'ta ön işlemci sabiti tanımlamayacaksanız LINUX'te ise -DLINUX ile derliyeceksiniz. Şimdilik iki platform için çalışabilen bir kod. Ama #ifdef ve #ifndef ifadeleri genelde CPU tipine göre işlem yapılırken veya Derleyicimizin bize sağladığı ön işlemci sabitlerin kullanımında işimize yarar. Bir düşünün yazdığınız en basit kod bile işlemcinin bunu register değerlerinde tutma özelliklerine göre değişmektedir. Yazılan kod tüm platformlar için mümkün olduğunca taşınabilir yapmamız gerekir. Bu tamamen sizin programcılığınıza ve kütüphane bilginize kaldığını unutmamalısınız.

```
#ifdef BIGENDIAN
typedef struct {
u32 h;//yüksek anlamlı biti
u32 l;//düşük anlamlı biti
} u64;
#else
typedef struct {
u32 l; //düşük anlamlı biti
u32 h; //yüksek anlamlı biti
} u64;
#endif
```

Bu şekilde örneğin işlemcimin **BIGENDIAN** ise ayrı bir yapı değilse ayrı bir yapı tanımlanmıştır. Bu kodun amacı 32 ve 64 bitlik işlemciler için işlemci 64 bit ise işlemci yapısına göre veri almaktadır. İşlemci Register yani almaçları verileri saklama konusunda farklılık göstermektedir.

#undef önişlemcisi

Bu ifade ile tanımlanmış ifadelerin geçerlilikleri bu önişlemciden sonra ortadan kaybolur.

```
#define SIZE 1024
…
…
#undef SIZE
…
```
artık SIZE tanımlı değil.

#line önişemcisi

Satır değerini değiştirir.Örneğin

```
#include <stdio.h>
#line 60
int main(){ //61 satır
return 0;//62 satır
}//63 satır
```

#,## önişlemcileri

işlemcisi verilen ifadeyi string yani cümle haline çevirir.

```
#define YAZ(x) # x

printf(YAZ("Selam Dunya\n"));
```
ifadesi
printf("Selam Dunya\n"); şekline dönüşür.

ifadesi iki kelimeyi birleştirir.

```
#define CONCAT(x1,x2) x1##x2

int xy=20; olsun
printf("%d\n",CONCAT(x,y)); ifadesi ile
printf("%d\n",xy); aynı manadadır.
```

Eğzersizler :

1-) Aşağıdaki ifade doğrumudur.Yoksa eksik mi ?

? #define ISDIGIT(ch) ((ch >= '0') &&(ch<=9)) ? 1 : 0

2-) Bu ifade üzerine düşününüz :

? #define isupper(ch) ((ch)>='A' && (ch)<='Z')
? ..
? while(isupper(c=getchar()

3-) Aşağıdaki kod blokları için geçerli macroları yazınız

?if(ch>=65 &&ch<=90)
?...
ve
?if(ch>='A' &&ch<='Z')
?...

#define ***************

4-) Aşağıdaki ifadelerde düzeltilmesi gerekenleri düzeltin:

char *string;
char buffer[20]
double x;
int i=0;

? string=(char *)malloc(sizeof(char)*20);
? strcpy(string,"aaaaaaaaaaaaaaaaaaa");//19 tane a
? string[19]=0;
? ...
? for(;i<=sizeof(string);i++)
? buffer[i]=string[i];
? buffer[i]=0;
? x=20;

5-) Potensiyel Hatalardan kaçınmak için aşağıdaki ifadeleri sizce nasıl yazmanız gerek.

? #define SSS 3.1459
? #define ONE2NUM(x) (1/(x))
? #define NUMBEROFiTEMSLENGTHHASHTABLE 12378

6-) Aşağıdaki koda gerekli açıklamaları ekleyiniz.Açıklamasız bir kodun çok fazla bir değeri yoktur.Ayrıca daha değişik şekillerde yazmaya çalışınız.Bu kod bence C/C++ 'ın gerçek gücünü gösteren bir kod bloğu.Dikkatli

inceleyiniz.Ve geliştirmeye çalışın.

```
? int _strlen(const char *message)
? {
? char *start=(char *)message;
? while(*message)
? ++message;
? return message-start;
? }
```

1.4Hata Tespiti ve Sık Yapılan Hatalar

1.5.1 Fonksiyon Geri Dönüş Değerleri Önemlidir.

Program yazarken yüzlerce değişik fonksiyonu kullanmaktayız ve kullandığımız fonksiyonların geri dönüş değerlerini çoğu zaman önemsememekteyiz. İyi program yazmanın kurallarına yeri geldiği için kısaca değinmek istiyorum:

1-) Yazılacak Program çok çok iyi analiz edilmelidir. Çünkü :

Yapmak istediğiniz şeyin ne olduğunu bilirseniz nasıl yapmanız gerektiğini de bilirsiniz. Analiz iyi bir şekilde yapılırsa Programın yazım süresi azalır,Hata Tespiti kolaylaşır ve maliyeti de azalır. Size bu noktada ilginç bir bilgi vereyim. Yazılım mühendisliğinin uygulandığı projelerde Analiz Kısmının uzun tutulması ile Programın yazımından sonraki güncelleme maliyetleri ki biz buna Test ve kararlı sürüme ulaşmak arasında ters orantı vardır. Yani Analiz Süresi artarsa bu toplam maliyete olumlu katkı sağlamaktadır. Sonuç olarak iyi analiz ,amacı anlamak size daha kararlı projeler geliştirmenize olanak sağlar.

2-) Yazılacak Program çok iyi dökümante edilmelidir.Çünkü :

Bir hafta ,ay veya yıl önceki kodunuza bakıp neyi niçin yaptığınızı tekrar tekrar düşündüğünüzü bir düşünün. Sıkıcı değil mi ? Bu duruma düşmemek için mümkün olduğunca yazdığınız programları iyi dökümante ediniz. **İnsan beyni unutkandır** .Bu yüzden bu dokümanı yazmaya çalışıyoruz .Bu sayede sizlere daha çabuk anlaşılabilir ve kolay güncellenebilen kod parçacıları geliştirmenizi hedeflemekteyiz .Dokümantasyon kısmı iki çeşit olmalıdır. Kod içinde yorumlar ile oluşacak bir dökümtasyon sistematiği ve genel kullanımlı bir yardım dokümantasyonudur. İyi kod yazmak tamamen sizin nasıl alışmanıza bağlıdır. Yani alışkanlıklar önemlidir.

3-) Program Sade ve Kafa karıştırıcı olmamalıdır. Çünkü,

İyi bir programın ne yaptığını programcılar bir bakışta anlayabilmelidir .Hiçbir kafa karıştırıcı unsur kullanılmamalıdır. Program

yazılabilecek en sade ve açık şekilde olmalıdır.

4-) Hata tespiti En üst düzeyde olmalıdır.Çünkü,

Yazdığımız programlar elbet bir gün bir yerden patlak verecektir. Hata ayıklama ve tespiti bu yüzden önemlidir. Bulunan ve Düzeltilen her hatadan sonra programlar sizce Kusursuzlaşır mı ?(Ör: Windows yama üstüne yama çıkıyor ama Hala Problemler Mevcut ; 1. adımın iyi oluşturulmadığı durumlarda oluşacak sonuçlara en net cevap.)

Örneğin Java'da gelişmiş bir hata tespit mekanizması var,C++ ise kararlı ve geliştirilebilir bir hata mekanizması bulunmaktadır. C ise kendi hata isimlendirmenizi yapmanıza izin verir.(Gerçekte hata denetimi yoktur .Size bırakır .Belki de her şey sınırsız olduğu için C bana göre en esnek dildir. Her şey tamamen programcının elindedir.) Hata tespiti önemlidir. **Mutlaka fonksiyonların yardım dokümanları okuma sabrı gösterin** en azından geri dönüş değerlerini. Her şeyin neden sonuç ilişkisi içinde olduğunu unutmayınız.

Fonksiyon geri dönüş değerleri değişmekle birlikte genelde programcılar NULL,0,1,-1 .. tarzında geri dönüş değerleri olmaktadır. Ama siz kullandığınız fonksiyonları geri dönüş değerlerini mutlaka kontrol etmelisiniz. Burada başka bir soru aklınıza gelebilir. Her fonksiyon geri dönüş değeri almalımıdır ? Bu konuda C/C++ 'ta ***void*** tipinde geri dönüş değeri olmayan fonksiyonlar da tasarlanabilir. Örneğin C ' deki ***strcmp(...)*** fonsiyonu iki kelimeyi karşılaştırır. Fonksiyonun geri dönüş değerleri ise 0,sıfırdan büyük,sıfırdan küçük olabilir. Bu şekilde tanımlanmıştır. Ama ***free(..)*** dinamik bellek alanı sıfırlayan fonksiyon ise void tanımlanmıştır. Sizin programcı olarak göreviniz hataya sebebiyet veren kod parçalarını ayıklamaktır. Burada fonksiyonların geri dönüş değerlerini mutlaka kontrol etmelisiniz. Örneğin :

```
? char *buffer;
? ....
? buffer=malloc(20);
? strcpy(buffer,"123456789");
? ...
? free(buffer);
```

Evet bu kod aslında çalışabilen bir kod bloğudur.Yaptığı iş ise bellekten 20 byte yer ayırır.Buffer içine yerleştirir.Peki yer ayıramadığını düşünürseniz ne olur.Yer ayrılmamış bir bellek alanına bellek kopyalar ***core dump***(bellek bozulmasına denen isim dir.) eder.Biz program yazarları mutlaka ve mutlaka yazdığımız programda olabilecek en kötü duruma karşı hazırlıklı olmalıyız.Bu kodda hata olama durumuna göre tekrar düzenlenmelidir.

```
char *buffer;
.....
//Yalnız her sistemde farklı bellek ve tür değişkeni saklanabilir.
//şekilde taşınabilirlik artmaktadır.

buffer=(char *)malloc(sizeof(char)*20);

If(!buffer)
veya
if(buffer==NULL)
{
//Hatamızın türünü stderr buffer alanına yazıyoruz buda ekrana

fprintf(stderr,"yetersiz bellek\n");

exit(0);
}


//örnek olarak free(),close(),strcpy(),strcat() diyebiliriz.
strcpy(buffer,"123456789");
...

free(buffer);
```

Başka bir eksik program örneği.Bu programda bir dosyaya istenen cümle yazılmaktadır.Kullanıcıdan cümle uzunluğunu girmesini istemektedir.

Main.c kodumuz :

```
? #include <stdio.h>
? #include <stdlib.h>
? #include <string.h>
?
? int main()
? {
? char *string;
? FILE *fp;
? int size;
?


? string=(char *)malloc(sizeof(char)*size);///Size uzunluğu kadar yer ayrılıyor
```

```
? //Cümleyi giriniz diyor.
? printf("Please Give the String Total Size is big from %d\n",size);

? scanf("%s",string);
? //dosya açılıyor
? fp=fopen("deneme.txt","w+");

? fwrite(string,size,1,fp);
? //dosya kapatılıyor
? fclose(fp);
? alınan bellek adresi boşaltılıyor
? free(string);
? return 0;
? }
```

derliyoruz
gcc Main.c -o deneme

ve **./deneme** diye çalıştırıyoruz. Programın çalışırken ki hali şu şekildedir :

root@localhost # ./deneme
Total String Size :10
Please Give the String Total Size is big from 10
aaaaa
root@localhost # less deneme.txt diyerek içine bakıyoruz ve 5 tane a yazılmıştır.

Evet çok iyi çalışan ! Bir kod yazdık dimi ? Bu kod gerçekten nasıl program yazılmamasına birebir örnek teşkil etmektedir. Bu kodun eksiklerine bakmadan evvel *Operating System Desing and İplementations* Kitabında Andrew S. Tanenbaum ,Güvenlik kısmında alıntıdır ,bizlere şu tavsiyeleri vermektedir.

Güvenlik Özelliği (The Security Enviroment)

Aslında Güvenlik(Securty) ve sistem koruması(Protections) çoğu zaman birbiriyle ilişkilidir. Güvenlik kavramını bizler genel hataların sonuçları olarak yorumlayacağız. Sistem koruması mekanizması (Protection Mechanisms) ise bu oluşan hataları güvenli bir şekilde engellemek ve en aza indirgemektir .Güvenlik kısmında veri kaybı çeşitleri 3 tür olabilir .Bunlar :

1-) Elimizde olmayan tanrısal sebepler : Deprem,Sel,Savaşlar,Yangın yani doğal afetler diyebiliriz.

2-) Donanım ve yazılım hataları : CPU hataları(Buna en bariz örnek olarak Pentium CPU'larda oluşan 1 Milyarda Bir gözüken bir işlemci hatası idi. Evet normal bir kullanıcı için bu belki de önemli değildir. Ama Saniyede milyonlarca işlem yapan bir süper bilgisayar için çok çok tehlikeli bir durumdur.) ,Disk hataları veya bozulmaları,program hataları vb.. gibi.
3-) İnsan hataları :Yanlış veri girişi ,yanlış program çalıştırmak,aşırı veri yüklenmesi(Serverlar için),program yazarlarının iyi tasarlayamadığı programlar vb..

Genel Sistem Saldırıları Çeşitleri:

Güvenlik sonradan bir sisteme eklenecek bir unsur değildir .Yani baştan tasarlanmalıdır. Sonradan eklenen her yeni yama mutlaka ama mutlaka yeni problemlere ve hatalara sebep olacak. Bu nedenle bu dokümanda özellikle tasarımın önemine ,kodun basitliğine ve anlaşılabilirliğini açıklamaktayız. Sizin yazdığınız program farz edelim hiçbir hata içermeyen bir program olsun. Ama çalıştırdığınız sistemlerde mutlaka oluşacak hatalar sizin programlarınız da güvenli olmasını engelleyecektir. Hiçbir sistem tam güvenli değildir. Ve olamayacaktır. Sadece diğerlerinden daha güvenilirdir diyebiliriz. Genel Saldırı çeşitleri :
1-) Bellek ,hard disk,disk alanı vb sistemlerimi okuyup üzerine yazmak diyebiliriz.

2-) İllegal sistem çağrılarını denenmesi veya legal sistem çağrılarının illegal parametreler ile veya legal parametrelerin kullanımının oluşturduğu sistem hataları.

3-) Sisteme giriş ve girdikten sonra yapılan işlemlerin yedeklenmesi(logging) mevcuttur. Bazı sistemlerde şifre isteyen programlar ölürler(işlevselliğini kaybeder) ve giriş başarılı bir şekilde yapılabilir. Eğer iyi kayıt alınmış sistemlerde kimin neyi ne zaman yaptığı kayıtlıdır. Yazacağınız programlarınızda kendi log(kayıt ve hata günlüğü olması gerekir) dosyalarının olması iyi bir şeydir.

4-) Karmaşık İşletim Sistemi özelliklerinin Kullanıcı bazında kısıtlanarak Kullanıcının anlıya bileceği şekilde kolay anlaşılır bir tasarımda(*User Friendly*) gerçekleştirmek.

5-) Program dokümantasyonlarına iyi kontrol edin.Eğer dokümanda "Bunu şu şekilde denemeyin.. " tarzında bir yazı var ise Tüm Bilgisayar Korsanları sistemi hataya sürükleyecek şekilde hata yapabilecek çözümleri ararlar.

Evet Bunlar gerçekten önemli ifadelerdir. Şunu unutmamalıyız. İnsanlar hata yapabilirler veya **Kasıtlı yapabilirle**r ki Bilgisayar korsanlarının

yaptığı işler bunlarla alakalıdır. Şimdi biraz önceki yazdığımız koda dönelim bu program gerçekten çok önemli hatalar içermektedir. Biraz önce çalıştırdığımız kodun çalışma anındaki görüntüsü aşağıdadır.

root@localhost# ./deneme

Total String Size :10
Please Give the String Total Size is big from 10
aaaaa

En önemli nokta burada programı çalıştıran kötü niyetli kişiler Total String Değerini değiştirerek gösterici hatası oluşturacaktırlar. Cümle uzunluğunu 10 verdiğimiz örnekte ayırdığımız yerden daha uzun bir cümle girersek örneğin 200,100 bin kelimelik ki girebiliriz. Eğer bu şekilde hata denetimi yapmak isek başımıza büyük dertler açabilir.
Bu tüm işletim sistemlerinde en büyük sistem bozulmasına(core dump) sebep veren bir hata çeşitidir . Burada size şiddetle öneriyorum yazdığınız kodun bellek hatalarına ve bu çeşit sistem hatalarına sebep vermemesi için mutlaka hata arama mekanizması oluşturmalısınız. Yani

? string=(char *)malloc(sizeof(char)*size);///Size uzunluğu kadar yer ayrılıyor

kullanıcı size değerini çok fazla büyük girebilir veya negatif deger girebilir...Bu sistem tarafından karşılanmayabilir.Olmıyacak bir şey değil.Her zaman kullanıcıların hata yapma olasılıkarını hesaba katarak düşünmeliyiz.Malloc fonksiyonunun dökümantasyonun okursanız hata olduğunda NULL ,C 'de (void *)0 olan bir hata döndürür.Bizim oluşturmamız gereken algoritma ise :
Başarılı bir şekilde yer ayrıldı mı ?Evet ise işleme devam et,Hayır ise hatayı ver ve gerekiyorsa programdan çıkmanız gerekir. Yoksa yer ayrılmamış bir bellek alanına veri yazarsınız ki yazdığınız çok önemli bir bellek hatasıdır. Ayrıca Size değeri negatif de olmamalıdır.

//aynı zamanda maksimum bir değerdende fazla olmamalıdır.

```
If(size<1)
{
fprintf(stderr,"Negatif Değer Girdiniz Tekrar Deneyiniz.\n");
exit(0);
}
string=(char *)malloc(sizeof(char)*size);///Size uzunluğu kadar yer ayrılıyor

//engellemek için kulanışlı bir fonksiyon.
memset(string,0,size);


{
```

```
fprintf(stderr,"Sizteminizde Yeterli Bellek yok.\n");
exit(0);
}
```

tarzında hata arama kısmını eklememiz gerekiyor.Kodun diğer kısmına geçelim.

```
? printf("Please Give the String Total Size is big from %d\n",size);

? scanf("%s",string);
```

Bu kod ile klavyeden string kelimesi giriyoruz.Eğer Girilen kelime size değişkeninden fazla ise ise bu bir hataya sebebiyet verecektir.Peki çözüm ne olmalıdır.Tabiki size değişkeni kadar keşki kelime klavyeden alsak dediğinizi duyar gibiyim.Evet haklısınız.Bu tarz karakter göstericilerinde klavyeden alınan karakter sayısını sınırlayabiliriz.

```
printf("Please Give the String Total Size is big from %d\n",size);

fgets(string,size-1,stdin);
```

burada belirtmek istiyorum.C/C++ 'de 3 değiş standart dosya akımı(stream) bulunmaktadır.
C/C++ için
stdin /cin : Standart Giriş ,örnek : klavye
stdout /cout : Standart Çıkış ,örnek : Monitör
stderr /cerr : Standart Hata ,örnek : Monitör

C/C++'taki bu tampon dosyalamın çeşitleri ise

1-) Satır tamponlamadır .printf(...)'te \n ile kullanırsanız satır tamponlama yapar.
2-) Tam tamponlama ise tampon dolana kadar bekler .Dolduktan sonra işletim sitemi ekrana bunu aktarır.
3-) Tamponlama yok.

C/C++ fonksiyonları 3 çeşit tamponlamayı kullanabilir. Ama standartlara göre
stderr *tamponlama kullanmaz direk olarak çıktıyı ekrana yazar.*
Stdin *için genelde satır tamponlama kullanılır.*
Stdout *için genelde satır tamponlama kullanılır.*

Genellikle hataları gösterirken stderr 'e aktarmalıyız. Örnek hata kullanım çeşitleri

1-) fprintf fonksiyonu

fprintf(stderr,"oluşan hata türü",var ise değişkenler);

fprintf fonksiyonunda bizler değikenlerder ekleyebiliriz.

Örnek :

fprintf(stderr,"bir hata oluştu hata kodu %d\n",hatakodu);

2-) perror fonksiyonu

bu fonksiyon fprintf'in arguman almayan halidir.

perror("bir hata var");

fprintf(stderr,"bir hata var");

kodumuzun diğer kısmını inceliyoruz.

```
? //dosya açılıyor
? fp=fopen("deneme.txt","w+");
```

fopen(...) ile dosyayı açıyoruz. Ama işletim sistemi ve kesmeleri veya oluşacak diğer hatalar sebebiyle dosyanın açılmama olasılığı mevcut. Bu nedenle dosya açılmadı ise hata kodunu ver ve çık. ***fopen*** geri dönüş değeri olarak NULL değerini verir.

```
//dosya açılıyor
fp=fopen("deneme.txt","w+");
{
fprintf(stderr,"dosya açılamadı\n");
//açtığımız dinamik bellek alanını boşaltmamız gerekiyor.
free(string);
exit(0);
}
```

Dosyaya yazılıyor.Eğer geri dönüş değeri 0 ise hata var

```
req=fwrite(string,size,1,fp);
if(!req)
{
fprintf(stderr,"dosya %s yazılamadı\n",string);
//açtığımız dinamik bellek alanını boşaltmamız gerekiyor.
free(string);
exit(0);
}
//dosya kapatılıyor
fclose(fp);
//alınan bellek adresi boşaltılıyor
```

```
free(string);
return 0;
}
```

C++ programcıları bu kodda değişiklik yapmak zorundalar ama genel işleyiş olarak C'nin dosya kütüphanelerinde kullanabilir. Ama biz C++ 'ın dosya kütüphanelerini kullanacağız. Bu da kodu biraz değiştirebilir. C++'ta bellek işlemleri farklı bir şekilde ayrılır .***C için malloc,free*** kullanılırken ***C++'ta ise new,delete*** yapısı kullanılır. Tekrar yineliyorum dinamik bellek ile uğraşıyorken dikkatli olmalıyız. En genel hata ayrılan yer alanın boşaltılmamasıdır veya dinamik bellek ayrılıp ayrılmadığının kontrolüdür. Bu konu her zaman unutulabilen basit bir hata çeşitidir. C++ ın bize sunduğu hoş bir hata mekanizması olan *try,catch ,throw* yapıları sunar.

New kaynaklı hata denetimleri:

2 çeşit new kaynaklı hata denetimi bulunmaktadır.

1-) Standart C++'da yer ayırma işlemi başarısız olduğunda new hata denetimi boşluk döndürmektedir .Bu eski tip hata denetimi diyebiliriz. Örneğin :

```
int *pint;

pint=new(nothrow) int(100);
if(!pint)
{
cerr<<"Bellek Hatası\n";
exit(0);
}
...
delete pint;
```

2-) Tip hata denetimi bad_alloc hatasına atar .Bu tip hata denetimi ilerde Standart C++'ta kullanılacak olan hata denetimidir.

```
int *pint;

try
{
pint=new int(100);
}
catch(bad_alloc ba)
```

```
cerr<<"Bellek Hatası\n";
exit(0);
}
...
delete pint;



char *str;
try
{
str=new char[100];
}
catch(bad_alloc ba)
cerr<<"Bellek Hatası\n";
exit(0);
}
...
cout<<"Adın ne :";

cin.getline(str,strlen(str)-1);
cout<<"Selamlar "<<str<<"\n";

delete[] str;
```

tarzında new kaynaklı hata denetimleri kullanabiliriz.

C++'ta try/catch bloğunu istediğimiz değişken türüne göre değiştirebiliriz. Hatta kendi hata sınıfımızı yaratarak kullanabiliriz de. Örneğin :

```
//dosya açılıyor
fp=fopen("deneme.txt","w+");
{
fprintf(stderr,"dosya açılamadı\n");
//açtığımız dinamik bellek alanını boşaltmamız gerekiyor.
free(string);
exit(0);
}
```

tarzındaki bir C kodunu C++'a çevirdiğimizde şu şekilde oluşturabiliriz.

```
try
{
  fp=fopen("deneme.txt","w+");
   throw 0;
}
catch(int i_event)
{
   cerr<<"Dosya açilamadi\n";
   delete[] string;
   exit(0);
}
```

tarzında eğer fp NULL değerinde ise int türünde hata bloğuna dallan diyoruz. Bu şekilde double ,char veya kendi tanımladığımı bir türdeki hata sınıfımızda dallanabiliriz. Örnek :

```
...throw 12;
catch(int i_event)

...throw 12.2;
(double d_event)

...throw "e";
catch(char c_event);

...throw "hata var !";
catch(char *pc_event);

...throw myevent(..);
catch(myevent *my_event);
```

tarzında olabilir.Gördüğünüz gibi C++'ta C'ye göre daha anlaşılır hata mekanizması mevcut.Birde aşağıdaki kodlara bir inceleyiniz.

```
? try
? {
? if(sayi<0) throw 1
? }
? catch(int *pi_event)
? {
? }
```

tarzında bir hata yakalayan bir kod yazdığımız farz edelim. Burada bir tür

uyuşmazlığı var eger catch(int i_event) yaparsak düzelebilir. Aynı zamanda yazdığımız kodda birden çok catch bloğuda olabilir. Burada tanımlanmamış bir catch yapı programınızı sebepsiz yere çıkmanıza sebep olabilir. Bu nedenle C++ bize catch(...) şeklinde bir tanımlanmamış hata yapısı sağlar. Örneğin :

```
try
{
…
}
catch(int ie)
{
}
catch(char ce)
{
}
..
catch(…)
{
cerr<<"Belirlenemiyen bir hata çeşiti oluştu ve yakalandı\n";
}
```

kullanımda olabilir.

Yardım Dokümanlarını İyi okuyunuz .?

Yardım dokümanları fonksiyonlar hakkında bize bilgi ve kullanımlarına ait bilgi verir.*UNIX sistemlerde yardım dokümanları ulaşmak için ***man dokuman_ismi*** deriz.
Örneğin :

```
man 2 printf
```

dersek printf ve türevleri hakkında bize bilgi sunar. man yanındaki 2 bize programlama dokümanları içinden arama yaptırır. Daha fazla bilgi için man man demeniz yeterli. Ayrıca çeşitli dokümanların bulunduğu bir başka yararlı komut ise info komutudur. Şimdi tüm C öğrenenlerin ilk olarak yazdıkları ve kitaplarında bulunan meşhur adınız ne deyip selam dunya isminiz şeklinde olan programı tekrar yazalım.

selamdunya.c

```
#include <stdio.h>

int main()
```

```
{
char name[120];
printf("Adiniz :");
gets(name);
printf("Selam Dunya ! Nasilsin %s",name);
return 0;
}
```

olan kodumuzu derliyoruz

```
[root@localhost /]# gcc selamdunya.c
/tmp/ccx8f45g.o(.text+0x28): In function `main':
: the `gets' function is dangerous and should not be used.
```

Hımm bize bir uyarı veriyor.gets fonksiyonunun tehlikeli ve kullanılmaması gerektiği.Programı çalıştırıyoruz.

```
[root@localhost /]# ./a.out
Adin ne :iskender
Selam Dunya ! Nasilsin iskender
```

şeklide bir çıktı vermekte yani çok iyi çalışıyor ! Dimi.Ama merak iyidir deyip neden bu tarz bir uyarı verdiğini incelemek istiyorum.

```
[root@localhost /]# man gets
```

...
...
BUGS
Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use fgets() instead.

It is not advisable to mix calls to input functions from the stdio library with low - level calls to read() for the file descriptor associated with the input stream; the results will be undefined and very probably not what you want.
...
...

kısmı benim ilgimi çekiyor. Bugs Türkçe karşıtı ile böcekler demek yani programı hataya sürükleyebilecek olan tehlikeliler de diyebiliriz. Ne yazık ki her şeyi tam Türkçeye adapte edemiyoruz.V e kullandığımız çoğu terim İngilizce ama şunu bilmenizi istiyorum. Bir kelimeyi tam olarak çevirebiliyorsak ve karşıtı var ise onu kullanmamız gerekiyor. Ama gidipte HTTP için renkli yazı formatı RYF mi demeliyiz ? Ne yazık ki her şeyi

Türkçeleştiremiyoruz.
Programımızda hataya sebep olabilecek tehlikeler(BUGS kısmının Türkçesi) kısmında aynen yazılanlar tercüme edilmiştir :

HATA OLASILIĞI(BUGS)

Asla gets() fonksiyonunu kullanmayınız. çünkü gets() fonksiyonun aslında kaç karakter okuyabileceğini bilemiyoruz. gets() tampon bellek dolana kadar veri girişi yapmaktadır .Buda tamamiyle kullanım için tehlikelidir. Güvenlik nedeniyle kullanılmamalıdır. Alternatif fgets() 'i kullanınız.

Bu fonksiyonu stdio.h altıdaki aşağı seviyeli sistem çağrılarında read() ile karışık kullanımı tavsiye edilmez. Sonuç çoğunlukla güvenilir değildir
...

açıklamada da anlaşılacağı gibi gets() 'i bizi hiç tavsiye etmiyor.Şimdi derleyicinin niye bize böyle cevap verdiğini anlıyoruz.Her şey neden sonuç ilişkisi ile örüntülüdür.Bize fgets()'i tavsiye etmektedir.İllegal bir kullanım örneği

```
[root@localhost /]# ./a.out
Adin ne
:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Selam Dunya
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Segmentation fault
```

Bellek hatası potansiyel bir hata ya sebebiyet vermiş olduk.

Burada en acıklı kısım ise her programlama kitabında ve örneklerde insanlar nedense gets()'i hala kullanmaktadırlar. Programlamaya yeni başlayan ve bu kodu yazan hemen hemen herkes mutlu bir şekilde ilk örneğini yapmanın huzurunda. Ama yazılan ilk örnekte bile :-) çoğu kuralı ihlal etmekteyiz ve yanlış programlar yazmaktayız. Ama çok fazlada karamsar olmamalıyız. Bu kodda ve benzer kodlarda gets()'i kullanmamalıyız. Örneğin doğru hali

selamdunya.c

```c
#include <stdio.h>

int main()
{
char name[120];
```

```
printf("Adiniz :");
fgets(name,sizeof(name)-1,stdin);
printf("Selam Dunya ! Nasilsin %s",name);
return 0;
}
```

Mutlaka ama mutlaka fonksiyonların çalışma mantıklarını iyi derecede bilmelisiniz ki hataları en az indirmelisiniz. Doküman okumak sıkıcı olmakla birlikle çoğu zaman fark edemeyeceğimiz hataların çıkmasını engeller. Bir başka tehlikeli bir fonksiyon

```
?man strcpy
```

bunuda siz inceleyiniz !.

errno değikeni !

Bir bölümde fonksiyonların geri dönüş değerlerinin önemli olduğu hakkında sizlere bilgi vermiştik.Ve geri dönüş değerlerinin değişmesi ile birlikte genelde sistem çağrılarının 0,-1,NULL tarzında oldukları hakkında bilgi vermiştik. Bu bölümde sistem fonksiyonları sonucunda oluşabilecek hataların errno değişkeni ile nasıl kullanılacağı hakkında size örnekler verilecektir.
errno değişkeni çoğu sistem fonksiyonunun başarısız olması durumunda oluşan hata kodunun yerleştirildiği global bir değişkendir. Hata kodları tam sayı değişkenlerdir. Hata kodları büyük E ile başlar. Örneğin

```
#define EPERM 1 /* Operation not permitted */
#define ENOENT 2 /* No such file or directory */
```

tarzında hata çeşitleri oluşturulmuştur.

Errno değişkenini kullanabilmeniz için *errno.h* dosyasını eklemelisiniz. Aynı zamanda GNU/Linux tarfından bu hata kodlarının okunabilir bir şekilde aktarılabilmesi için string.h altında *strerrno(...)* fonsiyonu bulunmaktadır. Burada ayrıca belirtmek istiyorum ki errno değişkeni global bir değişken olup bir hata olduğunda tüm sistem fonksiyonlarının hata kodlarını bu değişkene yüklerler. Bu nedenle oluşan errno değişkeni geçici bir nesneye atanması gerekir. Aşağıdaki örneğimizde bir dosya açılıyor ve kapatılıyor. Bu örnekte özellikle hata oluşması için mevcut olmayan bir dosya ismi verdik.

errno1.c kodu

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
```

```
int main()
{
  int err;
  FILE *fp;

  fp=fopen("deneme","r");

  if(!fp)
  {
    err=errno;
    fprintf(stderr,"Error Done !\n"
            "Error Numbers :%d\n"
            "Error Types :%s\n",err,strerror(err) );
     exit(0);
   }
   fclose(fp);
return 0;
}
```

derliyoruz.

```
Gcc error1.c
```

çalıştırıyoruz .

```
#./a.out
Error Done !
Error Numbers : 2
Error Types : No such file or directory
```

Dosyanın olmama koşuluna bağlı olarak bu tarzında bir çıktı alacağız.Yukarıdaki kodu C++ için değiştirirsek eğer int türünde throw yapısı tanımlıyacağız.errno değişkenini atanacağı yedek bir değere gerek yoktur.Sadece gerekli kısmında değişiklik yapılmıştır.

```
....
try
{
fp=fopen("deneme","r");
if(!fp)
throw errno;
}catch(int err)
cout<<"Error Done !\n"
"Error Numbers :"<<err<<
```

```
"\nError Types :"<<strerror(err)<<"\n";
exit(0);
}
.....
```

Bir başka alternatif hata arama tekniği olarak şunu kullanabilir.

```
err=errno;
switch(err)
{
case EPERM :
case EROFS :
....
}
```

tarzında kullanacağımız fonksiyonun yardım dokümanında errno değişkenine yüklenen değerlere göre istersek switch bloğu içerisinde böyle bir hata ayıklama metodu uygulayabiliriz.

Genel Hatalar !

Diziler ile Yapılan Hatalar !

Diziler üzerinde işlem yaparken farkında olamadan yapabileceğimiz hatalar üstüne bilgi verilecektir. Diziler sabit boyutlu ve sıralı olarak dizilmiş veri kümeleridir. Buradan da anlaşılacağı üzere boyutları sabittir. Örnek bir dizi tanımlaması şöyle yapılabilir :

```
int dizi[20];       //20 boyutlu bir tam sayı dizi
char *pch ;         //dinamik bellek yapısı aslında diziden ibarettir.
char ch[20]         //20 boyutlu bir char dizisi
```

C/C++ için diziler 0 'dan başlarlar.(aslında çoğu dilde 0 'dan başlar.Ama dilden dile değişebilir.Daha detaylı bilgi için kullandığınız dilin dökümantasyonuna bakınız.)

```
? int dizi[20];
? ...
? for(i=0;i<=20;++i)
? dizi[i]=i;
```

burada dizi boyu ihlal edilmiştir.20 boyutlu bir dizi C/C++ için 0 'dan 19'a kadar olan bellek alanını olışturur.Bu örnekte dizi 0'dan başlamıştır.Ama 20. elemanada değer verilmiştir.Böyle bir kodu derlediğinizde derleyici hata vermez.Çünkü C/C++ derleyicileri sizlerin kodda yapacağınız

hatalarla ilgilenmezler onlar kodun doğru oldup olmadığına bakarlar.Bu yüzden C/C++ bu kadar esnek bir dildir.

```
?dizi[100]=100;
```

20 elemanlı bir diziye 100 elemanı eklemek.Son derece yanlış bir hata.Bunun nedeni C 'de diziler üstünde sınırlama yoktur.

```
? int dizi1[20];
? int dizi2[40];
? int sum=0;
? int i=0;
?...
? for( ;i<40;++i)
? sum +=(dizi1[i]+dizi2[i]);
```

Burada iki dizi toplanmaktadır.Ama dizilerden biri farklı uzunluktadır.Burada sonuç yanlış olacağı aşikardır.Eğer birden çok dizi üzerinde işlem yapacaksak dikkatli olmalıyız yoksa istemediğimiz sonuçlara sebebiyet verebiliriz.
Karakter dizileri ile işlem yaparken de dikkatli olmalıyız. Karakter dizilerinde son elemana sonlandırıcı karakter olan NULL((void *)0) yerleştirilir. Ve bu sayede dizilerin toplam eleman sayısı öğrenilebilir. Eğer sonlandırıcı karakterin üzerine bir değer girersek ne olur çok tehlikeli bir hata yapmış oluruz. Diziler ile uğraşırken dikkatli olmalıyız.

```
? char ad[100];
?
? printf("Adin ne :");
? gets(ad);
```

bu çeşit bir hatayı daha önce değinmiştik.Dizilerin mevcut uzunlukları kadar eleman almalıyız. Bu tür hataları fgets ile önleyebiliriz.

```
? char ad[10];
? char temp[]="neden bu kadar uzun bir cümleyi sabit bir diziye aktarıyorsun!";
? ...
? strcpy(ad,temp);
```

Sabit uzunluktaki bir diziye uzunluğundan fazla eleman eklenmesi ile oluşan bir hatadır.Karakter dizileri üzerinde işlem yaparken dikkatli olmalıyız.Çünkü sabit diziler ile uğraşmaktayız.Eğer man strcpy(...) der isek hata kısmında(yani BUGS altında) söyle demektedir.

Man strcpy

....

BUGS

If the destination string of a strcpy() is not large enough (that is,if the programmer was stupid/lazy, and failed to check the size before copying) then anything might happen. Overflowing fixed length strings is a favourite cracker technique.

....

burada şunu demektedir.

HATA OLASILIĞI(BUGS)

Eğer kopyalanacak kelime yeteri kadar büyük değilse (yani programcı aptal veya tembel ve kopyalamadan önce boyutlarına bakmadı ise) herşey olabilir.Sabit uzunluktaki kelimeler için Bellek Taşması en sevilen bilgisayar ele geçirme(bkz. Cracking tekniği) tekniğidir.

Diyor.Yani kontrol etmeden yaptığımız her kelime işlemi aslında çok riskli olduğunu görmekteyiz.Hataya sebebiyet vermemek için mutlaka hata arama ve oluşma olasılıklarını düşünerek kod yazmanız gerekir.

C++ ile yazılım geliştirenlere ise tavsiyem sabit uzunluktaki diziler yerine vector tipinde sınırsız uzunluktaki veri yapıları kullanmalılar.Vector yapısı dinamik olarak artabilen ve dizinin boyunu öğrenebildiğimiz ve dizi şeklinde erişim olan bir şablon sınıftır.Örnek:

```
vector<int> dizi;
int sum=0;

dizi.push_back(1);
dizi.push_back(2);
dizi.push_back(3);
dizi.push_back(4);
dizi.push_back(5);
...

for(int i=0;i<dizi.size();++i)
sum +=dizi[i];

for(int i=0;i<dizi.size();++i)
cout<<i<<" Eleman : "<<dizi[i]<<"\n";
```

Aynı zamanda değişik bir çok tipte vector tanımlayabiliriz.

```
vector<double> d_dizi;
vector<string> str_dizi;
vector<kendi sınıfımız> bizim_dizi;
```

şeklinde geniş bir kullanımı mevcuttur.

C++ programcıları char türündeki kelime dizileri kullanmaları yerine string türünde dinamik uzunluklu sınıfı kullanmalıdırlar.Böylelikle C'deki hataya açık string işlemlerinde kurtulmuş olunur.Örnek kullanım :

```
#include <iostream>
#include <string>
using namespace std;
....
string str1("C++ gerçekten zevkli");
string str2("ve kullanışlı bir dil");
string str3;

str3=str1+str2;

cout<<str3<<"\n";
veya
if(str1==str2)
veya
if(str3==(str1+str2))
```

veya char * türüne aktarmak için ise
str1.c_str();
gibi çok çeşitli kullanımları mevcuttur.C deki karakter dizilerine göre daha esnek ve hatadan arınmış bir yapıdadır.

Doküman Hakkındaki Görüş ve Önerileriniz için mail adresim :
iskenderatasoy@yahoo.co.uk
islenderatasoy@gmail.com