Contents

Türkmen Wikiye Hoşgeldiniz

Bu wikiye katkı sağlamak için https://gitlab.com/turkman/devel/doc/wiki/ adresine pull request gönderebilirsiniz.

Bu dokümana https://turkman.gitlab.io/devel/doc/wiki/rst2pdf.pdf adresinden çevirimdışı pdf formatında da erişebilirsiniz.

Giriş:

Türkmen Linux bağımsız tabanlı bir GNU/Linux dağıtımıdır. Kendisine ait YMP (Yerli ve Milli Paket) sistemini kullanır. Türkmen Linux Türkiye originli bir dağıtımdır. □□ Geliştirilme sebebi yeni nesil teknolojileri kullanan bağımsız bir dağıtım oluşturmak ve kullanım kolaylığı sağlamaktır.

- Bütün paketler header dosyaları ve static kütüphaneler ile beraber gelmektedir. Bu sayede derleme yapması diğer dağıtımlardan daha kolaydır.
- Hızla çalışan karışık (ikili ve kaynak) paket sistemine sahiptir. İsteyenler kaynak koddan paket kurarken isteyenler derlenmiş paketleri kullanrabilirler.

Logonun anlamı

Türkmen linuxun logosu mavi zemin üzerine 3 hilalden oluşmaktadır. Logodaki 3 hilal osmanlı döneminde (1517-1793 arasında) kullanılan bayraktan esinlenilmiştir. Mavi renk (#1aa3ff) ise türkmen mavisidir.

Kaynak kod yansılarımız:

- https://github.com/turkman-linux
- https://gitlab.com/turkman

Ymp Paket Sistemi

Ymp paket sisteminin kurulumu

Ymp paket sistemini kaynak koddan derlemek için şu adımlar izlenmelidir.

Kaynak kodun indirilmesi

Kaynak koda https://gitlab.com/turkman/devel/sources/ymp adresinden ulaşabilirsiniz.

Öncelikle kaynak kodu *git clone* komutu ile indirelim.

```
$ git clone https://gitlab.com/turkman/devel/sources/ymp.git
```

Gerekli paketlerin kurulması

Debian tabanlı dağıtımlar için aşağıdaki paketler kurulmalıdır. (testing/unstable için)

```
# Derleyiciler
$ apt install meson gcc valac --no-install-recommends -y
# Derleme bağımlılıkları
$ apt install libarchive-dev libreadline-dev libcurl4-openssl-dev \
    libbrotli-dev --no-install-recommends -y
```

Yapılandırma

Kaynak kod *meson* komutu ile yapılandırılır.

```
$ meson setup build --prefix=/usr
    -Dshared=true
# Burada -Dshared=true veya -Dstatic=true belirtilmelidir.
```

İstenilen özellikleri *-Dözellik=durum* şeklinde belirterek ayarlayabilirsiniz. Özelliklerin listesine kaynak kod içerisinde bulunan meson_options.txt dosyasından ulaşabilirsiniz.

Derleme ve sisteme kurulma işlemi

ninja komutu kullanılarak derleme işlemi yapılır. Derleme sonrasında *ninja install* komutu ile sisteme kurulur. Son olarak *ldconfig* komutu kullanılarak kütüphanesini sistem kütüphane önbelleğini güncellenir.

```
# Derleme işlemi
$ ninja -C build
# Kurulma işlemi
$ ninja install -C build
$ ldconfig
```

Ymp komut satırı kullanımı

Ymp terminal üzerinden komut kullanarak çalıştırarak kullanılır.

```
$ ymp <işlem adı> <parametreler>
```

Yardım çıktısı alma

Tüm ymp işlemlerinin listesine ulaşmak için **ymp help** komutu kullanılır.

```
$ ymp help
...
-> install : Install package from source or package file or repository
...
```

Örneğin ymp kullanarak *git* paketini yükleyelim. Bunun için *ymp install git* komutunu çalıştırmamız gerekmektedir. Komuta ait parametreleri listelemek için **--help** parametresi eklememiz gereklidir.

```
$ ymp install --help
-> Aliases:install / it / add
-> Usage: ymp install [OPTION]... [ARGS]...
-> Install package from source or package file or repository
-> Options:
->   --ignore-dependency : disable dependency check
->   --ignore-satisfied : ignore not satisfied packages
->   --sync-single : sync quarantine after every package installation
->   --reinstall : reinstall if already installed
->   --upgrade : upgrade all packages
->   --no-emerge : use binary packages
-> Common options:
```

```
->   -- : stop argument parser
->   --allow-oem : disable oem check
->   --quiet : disable output
->   --ignore-warning : disable warning messages
->   --debug : enable debug output
->   --verbose : show verbose output
->   --ask : enable questions
->   --no-color : disable color output
->   --no-sysconf : disable sysconf triggers
->   --sandbox : run ymp actions at sandbox environment
->   --help : write help messages
```

Ymp işlemlerinin kısa adları da bulunur. Bu sayede komutu daha kısa şekilde yazıp kullanmamız mümkündür.

```
# Bu ifade ile bir sonraki aynı anlama gelir.
$ ymp it git
$ ymp install git
```

Ymp değişkenleri

Ymp çalışırken kendi içerisinde çeşitli değişkenler tanımlar ve bunları kullanarak çalıştırılacak işlemin özelliklerini belirler. Örneğin Ymp paketleri kurarken derleme yapılması istenmiyorsa **no-emerge** değişkeni **true** olarak ayarlanmalıdır. Değişkenler 2 şekilde ayarlanabilir. İlk olarak komut çalıştırılırken parametre eklenebilir.

Not: Tanımlanmamış olan tüm değişkenler **false** olarak kabul edilir.

Diğer yol ise **/etc/ymp.yaml** dosyasında ymp bölümüne ekleyebilirsiniz.

Ymp kabuğu

Ymp kendi içerisinde komut satırına sahiptir. Bu sayede istenilen işi bir betik dosyasına yazıp çalıştırmanız mümkündür. Ymp kabuğu başlatmak için **ymp shell** komutu kullanılır.

```
$ ymp shell
-> Ymp >> install git
```

Ymp kabugu ymp komutlarını ve parametrelerini kullanarak çalışır. Kabuk üzerinde normal işlemlerin yanında fazladan sadece kabukta çalışan ek işlemler bulunur. Bunlarla ilgili bilgi almak için **ymp help --all** komutu çıktısından yararlanabilirsiniz.

Ymp kabuğu kendi içerisinde basit bir programlama dili yorumlayıcısı barındırır. Bunu kapsamlı bir programlama dili olarak düşünmemekte fayda vardır. Çünkü Bir programlama dilinin sahip olduğu özelliklerin çoğundan mahrumdur ve sadece ymp komutlarının çalıştırılması üzerine tasarlanmıştır.

Açıklama satırları

ile başlayan satırlar açıklama satırıdır. Bununla birlikte **:** komutu işlevsizdir ve açıklama satırı olarak kullanılabilir.

```
# Bu bir açıklama satırıdır.
: Bu da bir açıklama satırıdır.
```

Değişken tanımlama

Değişken tanımlamak için **set** kullanılır. **read** komutu ise klavyeden alınan değeri değişken olarak atar. Bir değişkenin değerini almak için **get** kullanılır. Değişkenler kullanılırken başlarına **$** işareti konulur.

Not: Değişkenlerin herhangi bir karakter kısıtlaması bulunmamaktadır. Sayı, karakter veya türkçe karakter içerebilirler. (Emoji bile kullanabilirsiniz :D)

Koşul tanımlama

if ifadesi ile başlayan satır koşul satırıdır ve **endif** ifadesi gelene kadarki satırlar koşul sağlandığında çalıştırılır.

```
read var
if eq 12 $var
    echo sayı 12ye eşit
endif
```

Etiket tanımlama

label ifadesi ile kabuk betiğinde etiket tanımlanabilir. Daha sonra **goto** ifadesi kullanılarak bu etikete gitmek mümkündür. ymp kabuğunda döngüler bu şekilde sağlanır. Aşağıda siz 0 yazana kadar yazdığınızı ekrana yazan ymp kabuğu betiği mevcuttur.

```
label test
read var
if eq $var 0
    exit
endif
echo $var
goto test
```

Ymp paket yapımı

Ymp paketleri 3 türe ayrılır.

- kaynak paketler
- ikili paketler
- derleme talimatları

Derleme talimatları bizim tarafımızdan yazılan **ympbuild** dosyalarıdır. Bu dosyalar sayesinde ymp hangi kaynak kodun kullanılacağı, sürüm numarası, adı gibi bilgileri anlayabilir.

ympbuild dosyaları aslında birer bash betiğidir. Bu sayede kolay yapılıdır ve bash programlama bilgisi paket yapımı için yeterli olur.

ympbuild dosyası oluşturmak için **ymp template** komutundan yararlanabilirsiniz. Bu komut size şablon olarak ympbuild dosyası üretir. Bu sayede sadece üzerinde düzenleme yaparak kolayca paket yapabilirsiniz.

Paket formatı

Derleme talimatlarından kaynak ve ikili paketler üretmek için **ymp build** komutu kullanılır. Bu işlem önce kaynak kodu indirir ve doğrular. ardındar istenilen komutlara göre derleme işlemi yapıp paket üretir.

Örnek **ympbuild** dosyası aşağıdaki gibidir

```
#!/usr/bin/env bash
name=example
version=1.0
release=1
url='https://example.org'
description='example package'
email='your-name@example.org'
maintainer='linuxuser'
depends=(foo bar)
source=("https://example.org/source.zip"
        "some-stuff.patch"
)
arch=(x86_64 aarch64)
md5sums=('bb91a17fd6c9032c26d0b2b78b50aff5'
    'SKIP'
)
license=('GplV3')
prepare(){
    ...
}
setup(){
    ...
}
build(){
    ...
}
package(){
   ...
}
```

Burada paketin bilgileri ve nasıl derleneceğini tanımlayan işlevleri görebilirsiniz.

Derleme işlemi ve sandbox

Bir ympbuild dosyasını derlemek için aşağıdaki gibi bir komut kullanılmalıdır.

```
$ fdir=./repo/stuff-package/
$ ymp build "$fdir" --sandbox --shared="$fdir"
```

Burada ilk önce paketin derleneceği dizini değişkene atadık. Paket derlenirken **sandbox** özelliğini açmak için **--sandbox** parametresi ekledik ve sandbox içerisine paketin derleneceği dizini erişilebilir hale getirmek için **--shared** parametresine dizinin konumunu yerleştirdik.

Bu komut sandbox olmadan şu şekilde görünürdü.

```
$ ymp build ./repo/stuff-package/
```

Burada sandbox zorunlu değildir fakat paketlerin düzgünce derlenmesi için kullanmanızı şiddetle öneririm.

ympbuild

Değişkenler

- **name** : Paketin adını belirtir.

- **version** : Paket sürümünü belirtir.
- **release** : Paket numarasını belirtir. Ymp güncellik kontrolü için sadece bu değere bakar.
- **url** : Paketin anasayfasını belirtir. Bu değer kullanılmaz.
- **description** : Paket açıklamasını belirtir.
- **email** : Paketçinin email adresidir.
- **maintainer** : Paket bakımcısının adıdır. (veya nickname)

Diziler

- **depends** : Paket bağımlılıklarını belirtir
- **source** : Paket kaynak kodları listesini belirtir
- **md5sums** : Paket md5sum değeri listesidir. **SKIP** olan elemanları görmezden gelinir.
- **uses** ve **uses_extra** : use flag listesidir.
- **arch** : Desteklenen mimari listesidir.

İşlevler

- **prepare** : Hazırlık aşamasıdır. Burada kaynak kod yamaları uygulanır.
- **setup** : Kaynak kod yapılandırma aşamasıdır.
- **build** : Kodun derlendiği aşama burasıdır.
- **package** : Kaynak kodun paketleme dizinine kurulduğu aşamadır.

Derleme dizinleri

Her derlemenin **/tmp/ymp-build/<build-id>** içinde kendi derleme dizini vardır. build-id aslında ympbuild dosyasının md5sum'udur, bu nedenle ympbuild'i değiştirirseniz build-id değişir. Derleme dizini **HOME** çevresel değişkeni oarak tanımlanır. Bu sayede sadece **cd** komutunu kullanarak derleme dizinine geri dönebilirsiniz.

Derlenen kaynak kodlar paketlenirken **/tmp/ymp-build/<build-id>/output** dizinine kurulmalıdır. Bu dizin **installdir** ve **DESTDIR** çevresel değişkeni ile tanımlanır. Bu sayede **make install** gibi komutlara herhangi bir ek parametre vermenize gerek kalmaz.

Not: /tmp dizini genellikle ramdisk olarak bağlı olduğu için derleme sırasında ram dolabilir. Bunu engellemek için aşağıdaki gibi bir komut kullanabilirsiniz.

```
$ rm -rf /tmp/ymp-build
$ mkdir /home/linuxuser/ymp-build
# Bunu sistemi her başlattığınızda tekrarlamanız gerekebilir.
$ ln -s /home/linuxuser/ymp-build /tmp/ymp-build
```

Use flag kavramı

Paketlerde özellik tanımları yapmak için **uses** ve **uses_extra** dizileri tanımlayabilirsiniz. Bu özellikler isteğe bağlı açılıp kapatılabilirler. Bu sayede isteyenler paketleri istedikleri özelliklerle kullanabilirler.

```
...
uses=(foo bar)
uses_extra=(bazz)
```

```
foo_depends=(foo bazz)
...
setup(){
    ../configure --prefix=/usr \
    $(use_opt foo --with-foo --without-foo)
}
...
package(){
    ...
    if use bar ; then
        install stuff ${DESTDIR}/bin/stuff
    fi
}
```

Bağımlılıklar

Tanımlanan her özellik için **xxx_depends** şeklinde dizi tanımlayarak o özelliğin ek bağımlılıkları belirtilebilir. Bu sayede özelliği açtığımızda hangi ek paketlere ihtiyaç duyduğumuzu anlamamız mümkün olur.

İşlevler

Burada **use_opt** özelliğin açık olup olmama durumuna göre çalışır. Kullanımı şu şekildedir:

```
use_opt <özellik> <açık-olma-durumu> <kapalı-olma-durumu>
```

use ise yine özelliğin açık olup olmama durumunu belirtir fakat karşılığında çıktı üretmek yerine **if** ile kullanılır. Kullanımı şu şekildedir:

```
if use <özellik> ; then
    <açık-olma-durumu>
else
    <kapalı-olma-durumu>
fi
```

Özellik açma

Özellikler **USE** çevresel değişkeni ile veya **--use** parametresi veya ayar dosyasında belirtilir.

```
# --use=xxx yöntemi
$ ymp build ./repo/stuff-package --use="foo bar"
# USE=xxx yöntemi
$ USE="foo bar" ymp build ./repo/stuff-package
```

Eğer özellik listesi olarak **all** belirtirseniz **uses** dizisindeki tüm özellikler, **extra** belirtirseniz ise **uses_extra** dizisinin tümü kullanılır.

Not: Use flag sadece kaynak paketler ve derleme talimatları için kullanılabilir.

Not: sistemimizin mimarisi ile aynı adda use flag otomatik olarak tanımlanır ve kullanılır. Bu sayede tek bir ympbuild dosyası ile birden çok mimariye uyumlu paket üretilebilir.

Sistem

Servis Yönetimi

OpenRC

Türkmen varsayılan olarak **openrc** kullanır.

Kurulum

ymp kullanarak openrc yüklemek için aşağıdaki komutu kullanabilirsiniz:

```
$ ymp install openrc
```

Kaynak koddan derlemek için aşağıdaki adımları izlemelisiniz:

```
$ git clone https://github.com/OpenRC/openrc
$ cd openrc
$ meson setup build --prefix=/usr
$ ninja -C build install
```

Basit kullanım

Servis etkinleştirip devre dışı hale getirmek için **rc-update** komutu kullanılır.

```
# servis etkinleştirmek için
$ rc-update add udhcpc boot
# servisi devre dışı yapmak için
$ rc-update del udhcpc boot
# Burada udhcpc servis adı boot ise runlevel adıdır.
```

Servisleri başlatıp durdurmak için ise **rc-service** komutu kullanılır.

```
$ rc-service udhcpc start
# veya şu şekilde de çalıştırılabilir.
$ /etc/init.d/udhcpc start
```

Servislerin durmunu öğrenmek için **rc-status** komutu kullanılır. Ayrıca sistemdeki servislerin sonraki açılışta hangisinin başlatılacağını öğrenmek için ise parametresiz olarak **rc-update** kullanabilirsiniz.

```
# şu an hangi servislerin çalıştığını gösterir
$ rc-status
# sonraki açılışta hangi servislerin çalışacağını gösterir
$ rc-update
```

Sistemi kapatmak veya yeniden başlatmak için **openrc-shutdown** komutunu kullanabilirsiniz.

```
# kapatmak için
$ openrc-shutdown -p 0
# yeniden başlatmak için
$ openrc-shutdown -r 0
```

Docker içinde servis çalıştırma

Docker veya farklı bir ortamda sistem başlatılmadığı için servisler normal olarak çalıştırılmayacaktır. Fakat aşağıdaki adımları uygulayarak servis başlatmamız mümkündür.

```
# Önce /run/openrc dizini oluşturulur
$ mkdir -p /run/openrc
# Ardından boş softlevel dosyası oluşturulur
$ touch /run/openrc/softlevel
```

Bu işlemden sonra servis başlatmamız mümkün hale gelmektedir. Servisi aşağıdaki komut ile başlatabiliriz.

```
$ rc-service sshd start
```

Servis dosyası

Openrc servis dosyaları basit birer **bash** betiğidir. Bu betikler **openrc-run** komutu ile çalıştırılır ve çeşitli fonksiyonlardan oluşabilir. Servis dosyaları **/etc/init.d** içerisinde bulunur. Servisleri ayarlamak için ise **/etc/conf.d** içerisine aynı isimle ayar dosyası oluşturabiliriz.

Çalıştırılacak komut komut parametreleri ve **pidfile** dosyamızı aşağıdaki gibi belirtebiliriz.

```
description="Ornek servis"
command=/usr/bin/ornek-servis
command_args=--parametre
pidfile=/run/ornek-servis.pid
```

Bununla birlikte **start**, **stop**, **status**, **reload**, **start_pre**, **stop_pre** gibi fonksiyonlar da yazabiliriz.

```
...
start(){
    ebegin "Starting ${RC_SVCNAME}"
    start-stop-daemon --start --pidfile "/run/servis.pid" --exec /usr/bin/ornek-servis --parametre
}
...
```

Servis bağımlılıklarını belirtmek için ise **depend** fonksiyonu kullanılır.

```
...
depend() {
  need localmount
  after dbus
}
...
```

Openrc teorik olarak sysv-init betiklerini de çalıştırabilir. Fakat kesinlikle tavsiye edilmemektedir.

Ağ ve İnternet

udhcpc

Busybox tarafından sağlanan dhcp client uygulamasıdır.

Kullanımı aşağıdaki gibidir.

Öncelikle ağ arayüzünü etkinleştirelim.

```
# burada eth0 ağ arayüzü adıdır.
# ağ arayüzleri listesine /sys/class/net/ içerisinden ulaşabiliriniz.
$ ip link set eth0 up
```

```
# -s ile belirtilen betiğin varsayılan konumu /usr/share/udhcpc/default.script
$ udhcpc -i eth0 -s udhcpc.sh
```

udhcpc çalışırken bir betiğe ihtiyaç duyar. Bu betik sayesinde ağ bağlantısı kurulur.

Betik içeriği aşağıdaki gibi olmalıdır.

```
#!/bin/sh
ip addr add $ip/$mask dev $interface
if [ "$router" ] ; then
  ip route add default via $router dev $interface
fi
```

Burada değişkenler şu şekildedir.

- **$ip**: ip adresi
- **$mask**: netmask değeri
- **$interface**: ağ donanımı adı
- **$route**: route adresi (tanımlı olmayabilir)

OpenRC Servisi

Türkmen **udhcpc** için **openrc** servisi sağlar. Bu servisini etkinleştirmek için aşağıdaki komutu kullanabilirsiniz. Bu sayede sistem açıldığında ağ bağlantınız udhcpc ile kurulmuş olur.

```
rc-update add udhcpc boot
```

NetworkManager

networkmanager paketi tarafından sağlanan ağ yönetim uygulamasıdır.

Kurulumu

ymp ile kurmak için aşağıdaki komutu kullanabilirsiniz.

```
$ ymp install networkmanager
```

Kaynak koddan derlemek için aşağıdaki işlemleri uygulayabilirsiniz.

```
# Tüm seçenekler için meson_options.txt dosyasına bakın
$ meson setup build \
    -Dsystemd_journal=false \
    -Dsystemdsystemunitdir=no \
    -Dsession_tracking=no \
    -Dsession_tracking_consolekit=false ...
$ ninja -C build
$ ninja -C build install
```

Openrc servisi

Türkmen networkmanager için openrc servisi sağlar. Bu sayede ağ bağlantısı sistem açıldığında otomatik olarak yapılmış olur. Servisi aşağıdaki gibi etkinleştirebiliriz.

```
$ rc-update add networkmanager
```

Servisi elle başlatma

NetworkManager komutu kullanılarak servis elle başlatılabilir. Bunun için önce **dbus** etkinleştirilmelidir. Eğer **-d** parametresi ile başlatırsanız servis hata ayıklama modunda başlatılacaktır.

```
# dbus servisini açmak için
$ rc-service dbus start
# openrc kullanmadan çalıştırabiliriz.
$ mkdir -p /var/run/dbus
$ dbus-daemon --system &
# servisi başlatma
$ NetworkManager
```

Wifi

Wifi bağlantısı sağlamak için wpa-supplicant kullanılır. Ayrıca wifi kartının sürücüsü için linux-firmware gerekmektedir.

Wpa-supplicant kurulumu

Öncelikle aşağıdaki gibi gerekli paketi yükleyelim.

```
$ ymp install wpa_supplicant
```

Ardından servisini başlatalım.

```
# servisi başlangıca ekleyelim
$ rc-update add wpa_supplicant
# servisi başlatalım
$ rc-service wpa_supplicant start
```

Bağlantının kurulması

Öncelikle wifi kartımızın donanım adını aşağıdaki komut ile tespit edelim.

```
$ ip addr
...
3: wlan0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group default qlen 1000
   link/ether 8e:d2:82:0e:96:41 brd ff:ff:ff:ff:ff:ff permaddr 08:5b:d6:0c:45:bd
...
```

Daha sonra wifi kartımızı çalıştıralım.

```
$ ip link set wlan0 up
```

Ardından bağlantı için ayar dosyasını oluşturalım.

```
$ wpa_passphrase 'SSID' 'parola' > /etc/wpa_supplicant/wpa_supplicant.conf
```

Daha sonra **wpa_supplicant** komutunu kullanarak ayar dosyasına göre bağlantı sağlayalım.

```
# -B arkada çalışması için
# -i donanım adını belirtmek için
# -c ayar dosyası konumu için
$ wpa_supplicant -B -i wlan0 -c /etc/wpa_supplicant/wpa_supplicant.conf
```

Bağlantı kurulduktan sonra ip adresi almak için udhcpc kullanalım.

```
$ udhcpc -i wlan0
```

Network Manager ile kullanılması

wpa_supplicant servisini etkinleştirdikten sonra networkmanager servisini yeniden başlatalım.

```
$ rc-service networkmanager restart
```

Ardından wifi donanımımızı açalım.

```
$ nmcli radio wifi on
```

Ardından wifi ağlarını listeleyelim.

```
$ nmcli device wifi list
```

Ardından wifi ağına bağlanalım.

```
# Parolayı sizden yazı girdi olarak sorması için --ask kullanılır.
$ nmcli device wifi connect "SSID" --ask
# Parolayı doğrudan komuta ekleyebilirsiniz
$ nmcli device wifi connect "@" password "password"
```

Bluetooth

Bluetooth bağlantısı sağlamak için bluez kullanılır. Ayrıca buletooth kartının sürücüsü için linux-firmware gerekmektedir.

Kurulum

Öncelikle **bluez** kurulumu yapalım.

```
$ ymp install bluez
```

Ardından servisi etkinleştirelim.

```
$ rc-update add bluetooth
$ rc-service bluetooth start
```

Bağlantının kurulması

bluetoothctl komutu ile bağlantı kurmamız mümkündür. Öncelikle kullanılabilir aygıtları görüntüleyelim.

```
$ bluetoothctl scan on
...
[NEW] Device 1A:AA:D4:9C:8D:F5 Xiaomi Mi 6X
...
```

Arama modundan **ctrl+c** ile çıkalım. Çıktıdan bağlanmak istediğimiz aygıtın adresini alalım ve aşağıdaki gibi eşleştirelim.

```
$ bluetoothctl pair 1A:AA:D4:9C:8D:F5
```

Şimdi de bağlanalım.

```
$ bluetoothctl connet 1A:AA:D4:9C:8D:F5
```

Not: Özellikle dual boot kullanıyorsanız donanım sürekli olarak farklı şekilde haberleşmeye çalıştığı için bağlanma sorunları oluşabilir. Bu durumun üstesinden gelmenin en basit yolu aygıtı kaldırıp tekrar eklemektir.

Otomatik bağlanma

Her seferinde bağlantıyı elle yapmayıp aygıta güvenmek için aşağıdaki komutu uygulayabilirsiniz.

```
$ bluetoothctl trust 1A:AA:D4:9C:8D:F5
```

Aygıtı kaldırma

Bağlantıyı kesmek için öncelikle tanınan aygıt listesine bakalım.

```
$ bluetoothctl devices
```

Şimdi aygıtın bağlantısını keselim.

```
$ bluetoothctl disconnect 1A:AA:D4:9C:8D:F5
```

Ardından aygıtı silelim.

```
$ bluetoothctl remove 1A:AA:D4:9C:8D:F5
```

Oturum ve Başlangıç

Kullanıcılar

Kullanıcı yönetimi

Her kullanıcının kendisine ait bir uid ve gid değeri bulunur. Bu değer sistem kullanıcıları için 1000 den küçük normal kullanıcılar için ise 1000 ve daha büyük bir değere sahiptir.

Not: uid değeri 0 ise kullanıcı tam yetkilidir. (root yetkisi)

Bu uid değerleri **/etc/passwd** dosyası içerisinde bulunur.

```
root:x:0:0:root:/root:/bin/ash
pingu:x:1000:1000:Linux User:/home/pingu:/bin/bash
# pingu : kullanıcı adı
# x : bir anlamı yok
# 1000 : uid değeri
# 1000 : gid değeri
# Linux User : kullanıcının gözüken adı
# /home/pingu : kullanıcı ev dizini
# /bin/bash : kullanıcı kabuğu
```

Not: Türkmen diğer dağıtımlardan farklı olarak **/home** yerine **/data/user** dizini kullanır. Bu yüzden kullancı ev dizinini uygun olarak ayarlamanız gerekir.

Kullanıcı ekleme ve silme

Kullanıcı ekleme

Yeni kullanıcı eklemek için **useradd** veya **adduser** komutu kullanılır.

```
$ useradd -d /data/user/pingu -m -u 1000 -g 1000 -s /bin/bash pingu
# -d ev dizini
# -m ev dizini oluşturmak için (yoksa)
# -u uid değeri
# -g gid değeri
# -s varsayılan kabuk
$ adduser -D -h /data/user/pingu -u 1000 -s /bin/bash pingu
# -D oluştururken kullanıcı parolası sormaması için
# -h ev dizini konumu
# -u uid değeri
# -s varsayılan kabuk
$ echo "pingu:x:1000:1000:Linux User:/data/user/pingu:/bin/bash" >> /etc/passwd
$ mkdir -p /data/user/pingu
$ chown pingu /data/user/pingu
# Bu şekilde de kullanıcı ekleyebilirsiniz fakat tavsiye edilmez.
```

Sysconf yardımı ile kullanıcı ekleme

Türkmen içerisinde **/etc/passwd.d** dizini bulunur. Bu dizin sayesinde paketler kurulurken kullanıcı eklemek mümkün olur. Bunun için **/etc/passwd.d/paketadı** dosyası oluşturup içerisine passwd dosyasına eklenmesi gereken satırlar yazılabilir. Bu sayede paket kurulurken sysconf bu dosyayı algılayarak kullanıcıyı otomatik olarak sisteme dahil eder.

Kullanıcı silme

Kullanıcı silmek için **userdel** veya **deluser** komutu kullanılır.

```
$ userdel -r pingu
# -r ev dizinini silmek için
$ deluser --remove-home pingu
# --remove-home ev dizinini silmek için
$ sed -i "/^pingu:x:.*/d" /etc/passwd
$ rm -rf /data/user/pingu
# Bu şekilde de kullanıcı silebilirsiniz fakat tavsiye edilmez.
```

Parola ayarlama

Kullanıcılar arası geçiş yapmak için **su** komutu kullanılır. Bu komut geçilecek olan kullanıcının parolasını sorar. Bu yüzden kullanıcılara bir parola tanımlamamız gerekmektedir.

Parola tanımlamak için **passwd** komutu kullanılır.

```
$ passwd pingu
# kullanıcı adı belirtmezseniz root kabul edilir.
```

Parola ayarlamanın diğer bir yolu ise **usermod** komutu kullanmaktır. Bu yöntemde **openssl** komutundan yararlanılır.

```
# önce hash elde edelim
$ openssl passwd -6 'parola'
-> $6$GBPcPGqQLyLcYkKl$1z5BOQB36E31.VIJyGJXwCc6invR2WgeaSI9Jz7QZU/QZbffEm.J8edQkyIBtRWpSa.VFob3p/BH84Unag1Y60
# -6 sha512 formatında hash üretmek için.
# elde ettiğimiz değer ile parola beirleyelim.
$ usermod -p <hash-değeri> pingu
# Şu şekilde de tanımlayabilirsiniz.
$ usermod -p "$(openssl passwd -6 'parola')" pingu
```

Not: Özel karakterler ile parola oluşturma durumuna karşı tek tıknak (') işareti içerisine yazmanız gerekmektedir.

Not: Parolanın kabuğun history bölümünde gözükmesi güvenlik sorunlarına sebep olabilir. İşlem bittikten sonra history dosyasını temizlemenizi öneririm.

Kullanıcılar arası geçiş

Root yetkisi

Root yetkisi sistemde tam erişime sahip yetki düzeyidir. Bu yetki sayesinde sistemde değişiklik yapılabilmektedir. Örneğin paket kurulumu ve kaldırma gibi işlemler için root yetkisine ihtiyacımız vardır.

Root yetkisi **root** kullanıcısına aittir. Bu kullanıcının **UID** ve **GID** değeri 0dır. Ev dizini ise **/root** dizinidir.

Root yetkisinin alınması

Kullanıcı değiştirmek için **su** komutu kullanılır. Eğer bu komuta parametre vermezseniz **root** kullanıcısına geçiş yapılmış olur.

```
$ su
-> Password:
$ id
-> uid=0(root) gid=0(root) groups=0(root)...
```

suid kavramı

su komutunun çalışabilmesi için **suid** iznine sahip olması gereklidir. Bunu aşağıdaki gibi kontrol edebiliriz.

```
# s harfi suid iznini ifade eder.
$ ls -la /bin/su
-> -rws--x--x 1 root root 72816 Jan 14 10:25 /bin/su
```

suid izni vermek için **chmod u+s** izni geri almak için ise **chmod u-s** komutu kullanılır. Bu komutlar sadece root kullanıcısı tarafından çalıştırılabilir.

```
# yetki vermek için
$ chmod u+s /bin/su
# yetkiyi geri almak için
$ chmod u-s /bin/su
```

setuid sistem çağrısı

suid izni verilen dosyalar **setuid()** ve **setgid()** sistem çağrısını kullanabilirler. Örneğin aşağıdaki gibi bir C kodumuz olsun.

```
#include <unistd.h>
#include <stdlib.h>
int main(){
    setuid(0);
    setenv("USER","root",1);
    return system("sh");
}
```

Bu C kodunu gcc ile derleyelim ve **suid** izni verelim. **suid** yalnızca root kullanıcısı ayarlayabileceği için bu işlem root kullanıcısı ile yapılmalıdır.

```
$ gcc -o main.c main
$ chmod u+s main
```

Derlenen ve **suid** izni ayarlanan dosyamızı normal kullanıcımız ile çalıştırdığımızda root yetkisi alacaktır. **su** komutumuz bundan yararlanarak çalışmaktadır.

Not: suid iznine sahip dosyalar potansiyel güvenlik açığı oluşturabilir.

suid engelleme

Dosya sisteminde **suid** iznini engellemek için **nosuid** seçeneği etkinleştirilebilir. **/etc/fstab** dosyamızda ilgisi satır şu şekilde olabilir.

```
...
# filesystem     mountpoint   type   options           dump/pass
/dev/nvme0n1p2   /            ext4   default,rw,nosuid 0 0
...
```

busybox su

Busybox bize **su** komutu sağlayabilmektedir. Bu komutu kullanmak için öncelikle busyboxun kopyası oluşturulmalı ve ona **suid** yetkisi verilmelidir. Türkmen varsayılan su komutu olarak busyboxu kullanmaktadır.

```
$ install /bin/busybox /bin/su
$ chmod u+s /bin/su
```

Kullanıcı Kabuğu

Kullanıcı oturum açtığı anda kullanıcının varsayılan kabuk uygulaması (shell) başlatılır.

Bu varsayılan kobuk konumu **/etc/passwd** dosyasında belirtilmiştir.

```
pingu:x:1000:1000::/data/user/pingu:/bin/ash
```

Burada **/bin/ash** kabuk konumudur.

Kabuğu değiştirme

Öncelikle değiştirmek istediğiniz kabuğun konumunu **/etc/shells** içerisine eklemeniz gerekmektedir. Bu işlemi yapmazsanız kullanıcıyla giriş yapamazsınız.

Daha sonra **/etc/passwd** dosyasından kabuğun konumunu değiştirmemiz gerekir.

Not: Kabuk konumu parametre alamaz ve tam konum olmak zorundardır.

Kabuk değiştirildikten sonra tekrar giriş yapmanız gerekebilir.

Eğer kabuk konumu olarak **/sbin/nologin** kullanırsanız kullanıcının giriş yapmasını engellemiş olursunuz. Bu genellikle servislerin oluşturduğu kullanıcılar için kullanılır.

Unix Kabuğu

/bin/sh sistem tarafından kullanılan genel kabuktur. Bu kabuk debian tabanlılar için **/bin/dash**, alpine linux ve türkmen linux için **/bin/busybox**, diğerleri için **/bin/bash** konumuna sembolik bağlıdır.

Bu kabuk sistem açılırken kullanılır. Aşağıdaki gibi bir C kodu ile durumu örnekleyebiliriz.

```c
#include <stdio.h>
int main(){
    system("echo $0");
    return 0;
}
```

Bu kod çalıştırıldığında ekrana **sh** yazacakdır. Çünkü system komutu şununla eşdeğer şekilde çalışır.

```c
execl("/bin/sh", "sh", "-c", command, (char *) NULL);
```

Bu yüzden /bin/sh kabuğunu iyi seçmek sistem tasarımı açısından önemli olabilir. Farklı unix kabukları ve avantaj/dezavantajları aşağıdaki gibi özetlenebilir.

- /bin/dash : Debian tarafandan geliştirilir. Sadece kullanıcılar tarafından fakat yazılımlar tarafından ihtiyaç duyulmayan ek özellikler bulunmaz. (Tab tuşu ile tamamlama gibi). Bu sayede küçük boyutludur ve hızlı çalışır. Bash uyumlu değildir.
- /bin/ash : Busybox tarafından sağlanır. Ek pakete gereksinim duymaz. Basit seviyede özelliklere sahiptir. Kısmen bash uyumludur.
- /bin/bash : Bash GNU/Linux dağıtımlarının genelinde varsayılandır ve Çok az sorun çıkarır.

Unix kabuğunu değiştirmek için ayağıdaki gibi bir yol izleyebilirsiniz.

```
$ rm -f /bin/sh
$ ln -s bash /bin/sh
```

Oturumlar

Elogind

elogind systemd projesinde bulunan **logind** uygulamasının systemd'den bağımsız çalışabilen halidir. Amacı kullanıcı oturumlarını yönetmektir. **Pam** kullanarak çalışır.

Kurulumu

Ymp kullanarak aşağıdaki gibi kurulum yapabilirsiniz.

```
$ ymp install elogind
```

Veya kaynak koddan derlemek için aşağıdaki komutları kullanabilirsiniz.

```
# Seçenekler için meson_options.txt dosyasına bakın.
$ meson setup build -Dpam=true ...
$ ninja -C build
$ ninja -C build install
```

Ardından openrc servisini etkinleştirelim. Bunun için aşağıdaki komuttan yararlanabiliriz.

```
$ rc-update add elogind
```

elogind **pam** ile çalıştığı için pam yapılandırmasına eklememiz gerekmektedir. Bunun için **/etc/pam.d/system-auth** dosyasına aşağıdaki satırı ekleyelim.

```
# /etc/pam.d/system-auth dosyası içine en alta ekleyin.
session    include    elogind-user
```

agetty servisinin ayar dosyasında **login** komutu ayarlamak gerekebilir. Bunun sebebi **login** komutunun varsayılan olarak pam kullanmadan çalışan busybox tarafından sağlanmasıdır. Bunun için **/etc/conf.d/agetty** içerisini aşağıdaki gibi değiştirelim.

```
...
agetty_options="-l /usr/bin/login"
...
```

Oturumların listelenmesi

Oturum listelemek için **loginctl** komutunu kullanabilirsiniz. Bu komut aşağıdaki gibi çıktı verir.

```
SESSION UID USER SEAT  TTY
      1   0 root seat0 tty1

1 sessions listed.
```

Oturumu kontrolü

Oturumu kapatmak için:

```
$ loginctl terminate-session <session-id>
```

Ekranı kilitlemek için:

```
$ loginctl lock-session <session-id>
```

Sistem kontrolü

Sistemi kapatmak için:

```
$ loginctl poweroff
```

Sistemi yeniden başlatmak için

```
$ loginctl reboot
```

Sistemi uyku moduna almak için

```
$ loginctl suspend
```

Not: Uyku modu bazı donanımlarda düzgün çalışmayabilir.

Dosya Sistemleri

ext4

Ext4 standart dosya sistemdir ve çoğu linux dağıtımında varsayılan olarak tercih edilir.

Diski bağlama

ext4 diskleri bağlamak için öncelikle ext4 çekirdek modülünün etkin olması gerekmektedir. **lsmod | grep ext4** komutunu kullanarak etkın olup olmadığını öğrenebilirsiniz.

```
$ lsmod | grep ext4
```

Daha sonra aşağıdakı komut yardımı ile diski bağlayabiliriz.

```
# diski yazılabılır şekilde bağlamak için rw salt okunur bağlamak için ro kullanılır.
$ mount -t ext4 -o defaults,rw /dev/sda1 /baglama/noktasi
```

Ext4 biçimlendirme

mkfs.ext4 komutu e2fsprogs paketi ile sağlanır. Öncelikle e2fsprogs yükleyelim.

```
$ ymp install e2fsprogs
```

Daha donra diski biçimlendirelim.

```
$ mkfs.ext4 /dev/sda1
```

Not: diski biçimlendirmek verileriniza kalıca hasar verebilir.

Günlüklemeyi kapatma

Ext4 dosya istemi günlükleme özelliğine sahiptir. Bu özelliği aşağıdaki gibi kapatabilirsiniz.

```
$ tune2fs -O ^has_journal /dev/sda1
```

Eğer disk biçimlendirirken kapatmak isterseniz aşağıdaki gibi komut kullanabilirsiniz.

```
$ mkfs.ext4 -O ^has_journal /dev/sda1
```

Fuse

Fuse kullanıcıların root erişimi olmadan disk bağlayabilmelerine imkan tanır. Örneğin bir çıkarılabilir aygıt takıldığında içeriğine erişmek için fuse kullanılabilir.

Kurulumu

Türkmen linuxta fuse yüklemek için aşağıdaki komutu kullanabilirsiniz.

```
$ ymp install fuse
# Eğer libfuse2 gerektiren bir uygulama çalıştıracaksanız şunu da yüklemelisiniz.
$ ymp install fuse2
```

Daha sonra fuse servisini çalıştıralım.

```
# Açılışa ekleyelim
$ rc-update add fuse
# Çalıştıralım
$ rc-service fuse start
```

Eğer servis yerine elle başlatmak isterseniz aşağıdaki gibi çalıştırabilirsiniz.

```
# Önce modülü yükleyelim.
$ modprobe fuse
# Şimdi connections kısmını bağlayalım.
$ mount -t fusectl none /sys/fs/fuse/connections
```

Son olarak fusermount komutuna suid biti ayarlayalım.

```
$ chmod u+s /usr/bin/fusermount
```

Kurulum

Basit Kurulum

Bu bölümde **Ext4** dosya sistemine grub kullanarak kurulum anlatılacaktır. Anlatım boyunca **/dev/sda** diski üzerinden örnekleme yapılmıştır. Siz kendi diskinize göre düzenleyebilirsiniz.

Uefi - Legacy tespiti

/sys/firmware/efi dizini varsa uefi yoksa legacy sisteme sahipsinizdir. Eğer uefi ise ia32 veya x86_64 olup olmadığını anlamak için **/sys/firmware/efi/fw_platform_size** içeriğine bakın.

```
[[ -d /sys/firmware/efi/ ]] && echo UEFI || echo Legacy
[[ "64" == $(cat/sys/firmware/efi/fw_platform_size) ]] && echo x86_64 || ia32
```

Disk Bölümlendirme

Uefi kullananlar ayrı bir disk bölümüne ihtiyaç duyarlar. Bu bölümü **fat32** olarak bölümlendirmeliler.

Bu anlatımda kurulum için **/boot** dizinini ayırmayı ve efi bölümü olarak aynı diski kullanmayı tercih edeceğiz.

Öncelikle **cfdisk** veya **fdisk** komutları ile diski bölümlendirelim.

```
$ cfdisk /dev/sda
```

Ardından boot bölümünü ve kök dizini formatlayalım.

```
$ mkfs.vfat /dev/sda1
$ mkfs.ext4 /dev/sda2
```

Not: ext4 dosya sistemi araçları **e2fsprogs** ile sağlanır.

Eğer /boot bölümünü ayırmayacaksanız grub yüklenirken **unknown filesystem** hatası almanız durumunda aşağıdaki yöntemi kullanabilirsiniz.

```
$ e2fsck -f /dev/sda2
$ tune2fs -O ^metadata_csum /dev/sda2
```

Dosya sistemini kopyalama

Türkmen linuxta kurulum medyası **/cdrom** dizinine bağlanır. Kurulacak sistemin imajını bir dizine bağlayalım.

```
$ mount /cdrom/live/filesystem.squashfs /source
```

Şimdi de bölümlerimizi bağlayalım.

```
# /target yoksa oluşturun.
$ mount -t ext4 /dev/sda2 /target
$ mkdir -p /target/boot
$ mount -t vfat /dev/sda1 /target/boot
```

Ardından dosyaları kopyalayalım.

```
# -p dosya izinlerini korur
# -r alt dizinlerle beraber kopyalar
# -f soru sormayı kapatır
# -v detaylı çıktıları gösterir
$ cp -prfv /source/* /target
```

Daha sonra diski senkronize edelim.

```
$ sync
```

Bootloader kurulumu

Sisteme **ymp chroot** komutu ile girelim.

```
$ ymp chroot /target
# Bunun yerine aşağıdaki gibi de girilebilir.
for dir in /dev /sys /proc /run /tmp ; do
        mount -bind /$dir /target/$dir
done
$ chroot /target
```

Şimdi de eğer uefi kulanıyorsanız efivar bağlayalım.

```
$ mount -t efivarfs efivarfs /sys/firmware/efi/efivarfs
```

Grub paketini yükleyelim.

```
$ ymp install grub
```

Son olarak grub kurulumu yapalım.

```
# biz /boot ayırdığımız ve efi bölümü olarak kullanacağız.
# uefi kullanmayanlar --efi-directory belirtmemeliler.
# kurulu sistemden bağımsız çalışması için --removable kullanılır.
$ grub-install --removable --boot-directory=/boot --efi-directory=/boot /dev/sda
```

Grub yapılandırması

Öncelikle uuid değerimizi bulalım.

```
$ blkid | grep /dev/sda2
/dev/sda2: UUID="..." BLOCK_SIZE="4096" TYPE="ext4" PARTUUID="..."
```

Şimdi aşağıdaki gibi bir yapılandırma dosyası yazalım ve /boot/grub/grub.cfg dosyasına kaydedelim. Burada uuid değerini ve çekirdek sürümünü düzenleyin.

```
search --fs-uuid --no-flopy --set=root <uuid-değeri>
linux /boot/vmlinuz-<çekirdek-sürümü>   root=UUID=<uuid-değeri> rw quiet
initrd /boot/initrd.img-<çekirdek-sürümü>
boot
```

Ayrıca otomatik yapılandırma da oluşturabiliriz.

```
$ grub-mkconfig -o /boot/grub/grub.cfg
```

Fstab dosyası

Bu dosyayı doldurarak açılışta hangi disklerin bağlanacağını ayarlamalıyız. **/etc/fstab** dosyasını aşağıdakine uygun olarak doldurun.

```
# <fs>                  <mountpoint>    <type>          <opts>          <dump/pass>
/dev/sda1       /boot   vfat    defaults,rw     0       1
/dev/sda2       /       ext4    defaults,rw     0       1
```

Not: Disk bölümü konumu yerine **UUID="<uuid-değeri>"** şeklinde yazmanızı öneririm. Bölüm adları değişebilirken uuid değerleri değişmez.

Yapılandırma

Dil ayarlama

Sistem dilini ayarlamak için öncelikle **/etc/locale.gen** dosyamızı aşağıdaki gibi düzenleyelim.

- Dil kodlarına **/usr/share/i18n/locales** içerisinden ulaşabilirsiniz.
- Karakter kodlamalara **/usr/share/i18n/charmaps** içinden ulaşabilirsiniz.

```
tr_TR.UTF-8 UTF-8
```

Not: En altta boş bir satır bulunmalıdır.

Ardından /lib64/locale dizini yoksa oluşturalım.

```
mkdir -p /lib64/locale/
```

Şimdi de çevresel değişkenlerimizi ayarlamak için **/etc/profile.d/locale.sh** dosyamızı düzenleyelim.

```
#!/bin/sh
# Language settings
export LANG="tr_TR.UTF-8"
export LC_ALL="tr_TR.UTF-8"
```

Not: Türkçe büyük küçük harf dönüşümü (i -> İ ve ı -> I) ascii standartına uyumsuz olduğu için **LC_ALL** kısmını türkçe ayarlamayı önermiyoruz. Bunun yerine **C.UTF-8** veya **en_US.UTF-8** olarak ayarlayabilirsiniz.

Son olarak **locale-gen** komutunu çalıştıralım.

```
locale-gen
```

Eğer **/lib64/locale/** dizine okuma iznimiz yoksa verelim.

```
chmod 755 -R /lib64/locale/
```

Son olarak sistemi yeniden başlatalım.

```
openrc-shutdown -r
```

Hostname ayarlama

Makina adımızı ayarlamak için öncelikle belirlediğimiz makina adını **/etc/hostname** dosyasına yazalım.

```
$ echo "sunucu01" > /etc/hostname
```

Ardından hostname servisini yeniden başlatalım.

```
$ rc-service hostname restart
```

Eğer makina adının her açılışta aynı kalmasını istiyorsak servisi etkinleştirelim.

```
$ rc-update add hostname
```

Servis yöneticisini kullanmadan **hostname** komutunu kullanarak makina adını değiştirmek mümkündür.

```
$ hostname sunucu01
```

Not: Makina adı belirlerken belirlerken aşağıdaki kurallara dikkat etmelisiniz: * büyük harf içermemeli * ilk harf sayı olmamalı * 253 karakterden uzun olmamalı * türkçe karakter içermemeli

Araçlar

syslog

Syslog çalışan sistem servislerinin loglarını tutar. Bu loglar /var/log/messages içerisinde bulunur. Bu sayede sistemle ilgili tüm logları buradan takip edebilirsiniz.

```
# logları takip etmek için aşağıdaki komut kullanılabilir.
$ tail -f /var/log/messages
```

Servisin başlatılması

syslogd komutunu kullanarak servisi başlatabilirsiniz. Bu komut busybox tarafından sağlanır.

```
# komutla ilgili detaylara --help parametresi ile ulaşabilirsiniz.
$ syslogd
```

Eğer isterseniz aşağıdaki gibi bir openrc servisi yazıp kullanabilirsiniz.

```
#!/sbin/openrc-run

description="Message logging system"

name="busybox syslog"
command="/bin/busybox"
command_args="syslogd -n"
pidfile="/run/syslogd.pid"
command_background=true

depend() {
      need clock hostname localmount
      provide logger
}
```

Log yazma

logger komutunu kullanarak log yazdırabilirsiniz.

```
$ logger "hello world"
```

bununla birlikte aşağıdaki C kodunu kullanarak log yazmak mümkündür.

```
#include <syslog.h>
#include <unistd.h>
int main(int argc, char** argv){
    setlogmask (LOG_UPTO (LOG_NOTICE));
    openlog ("test", LOG_CONS | LOG_PID | LOG_NDELAY, LOG_LOCAL1);
    syslog (LOG_NOTICE, "Hello World from %d", getuid ());
    closelog ();
```

```
    return 0;
}
```

Not: **/dev/log** konumundaki unix sockete bağlanıp log yollamayı oradan da yapabilirsiniz.

ntpd

Sistem saatini bir sunucudan bakarak eşlemeye yarayan araçtır. **busybox** tarafından sağlanabilir.

```
BusyBox v1.36.1 (2023-06-09 19:35:59 UTC) multi-call binary.

Usage: ntpd [-dnqNwl] [-I IFACE] [-S PROG] [-k KEYFILE] [-p [keyno:N:]PEER]...

NTP client/server

        -d[d]   Verbose
        -n      Run in foreground
        -q      Quit after clock is set
        -N      Run at high priority
        -w      Do not set time (only query peers), implies -n
        -S PROG Run PROG after stepping time, stratum change, and every 11 min
        -k FILE Key file (ntp.keys compatible)
        -p [keyno:NUM:]PEER
                Obtain time from PEER (may be repeated)
                Use key NUM for authentication
                If -p is not given, 'server HOST' lines
                from /etc/ntp.conf are used
        -l      Also run as server on port 123
        -I IFACE Bind server to IFACE, implies -l
```

Servisin başlatılması

Servisi elle başlatmak için *busybox ntpd* komutunu kullanabiliriz.

```
$ busybox ntpd
```

Openrc servisi kullanarak da başlatabiliriz. Bunun için ağağıdaki gibi servis dosyası yazalım.

```
#!/sbin/openrc-run

description="Network Time Sync"

name="busybox ntpd"
command="/bin/busybox"
command_args="ntpd -n"
pidfile="/run/syslogd.pid"
command_background=true

depend() {
      need net
      provide timesync
}
```

Bu servis dosyasını **/etc/init.d/ntpd** dosyasına kaydettikten sonra aşağıdaki komutla etkinleştirelim.

```
# etkinleştirelim
$ rc-update add ntpd
# başlatalım
$ rc-service ntpd start
```

Yapılandırma dosyası

Yapılandırma dosyası **/etc/ntp.conf** konumundadır. Örnek yapılandırma aşağıdaki gibidir.

```
server 0.pool.ntp.org
server 1.pool.ntp.org
server 2.pool.ntp.org
```

Geliştirme ortamı

Python

Python kurulumu

Python yüklemek için **ymp install python** komutunu kullanmalısınız.

```
$ ymp install python
```

Kaynak koddan yüklemek için aşağıdaki adımları takip ediniz.

1. https://python.org adresinden kaynak kodu indirin ve bir dizine açın.
2. kaynak kodun içerisinde aşağıdaki komutları kullanarak derleyin.

```
$ autoreconf -fvi
$ ./configure
$ make
$ make install
```

Türkmen linux size birden çok python sürümünü aynı anda kullanmasına olanak tanır. Bunun için **pydefault** komutu kullanarak varsayılan sürümü değiştirebilirsiniz.

```
$ pydefault 3.10
```

Not: python paketi kurulurken var olan en üst sürümü varsayılan olarak ayarlamaktadır.

Not: En üst sürümü kullanmamak sistemin ve uygulamaların düzgün çalışmamasına sebep olabilir.

Pip etkinleştirilmesi

pip komutunu etkinleştirmek için aşağıdaki komutu kullanın.

```
$ python3 -m ensurepip
$ pip3 install --upgrade pip
```

İlk komut python ile gelen pip modülünü çalıştırarak pip kullanmanıza olanak tanır. İkinci komut ise pip sürümünü güncellemek için kullanılır.

Vala

Vala yazılan kaynak kodu **C** koduna çevirip daha sonra derleyerek çalışan bir programlama dilidir. **valac** kullanılarak derlenir.

Valac kurulumu

Vala derleyicisini (valac) yüklemek için **ymp install vala** komutunu kullanabilirsiniz.

```
$ ymp install vala
```

Kaynak koddan derlemek için ise aşağıdaki adımları takip ediniz:

1. https://gitlab.gnome.org/GNOME/vala adresinden kaynak kodu indirin ve bir dizine çıkarın
2. kaynak kodun içerisine girerek aşağıdaki komutları çalıştırarak derleyin.

```
$ autoreconf -fvi
$ ./configure
$ make
$ make install
```

Nano renklendirme desteği

Kaynak kod yazmak için nano kullanabilirsiniz. Nanoda renklendirme için valaya destek bulunmamaktadır. Vala ile **java** ve **c#** programlama dillerinin renklendirmesi birbirine benzer olduğu için kopyasını kullanarak vala yazarken renklendirme yapabilirsiniz.

Bunun için aşağıdaki yolu izleyin.

1. **/usr/share/nano/** içerisindeki **java.nanorc** dosyasını kopyalayıp aynı dizine **vala.nanorc** olarak kaydedin.
2. Kopyaladığınız dosyayı açın ve java yazan yerleri vala olarak değiştirin.
3. **foreach** ifadesi javada yer almadığı için eksik renklendirilecektir. Bunu nanorc dosyasına elle ekleyebilirsiniz.

C

C en temel programlama dillerinden biridir. C derlemeli bir dil olduğu için **gcc** veya **clang** gibi bir derleyiciye ihtiyaç duyar.

Derleyici kurulumu

Gcc kurulumu

gcc yüklemek için **ymp install gcc** komutunu kullanabilirsiniz.

```
$ ymp install gcc
```

Clang kurulumu

clang yüklemek için **ymp install clang** komutunu kullanabilirsiniz.

```
$ ymp install clang
```

Standart C Kütüphanesi (libc)

Her GNU/Linux dağıtımı bir libc ile gelir. libc, sistemdeki en temel kütüphanedir ve bütün programlar ne ile yazılmış olursa olsun eninde sonunda bir yerlerde libc fonksiyonlarını çağırır. libc fonksiyonlarını kullanmadan program yazmak çok zordur. İşte Türkmen Linux'ta kurulabilen libc implementasyonları:

Glibc

Glibc GNU tarafından geliştirilen ve bakımı yapılan bir libc implementasyonudur ve neredeyse her dağıtım tarafından varsayılan olarak kullanılmaktadır. Türkmen linux için kurmanıza gerek yok çünkü zaten varsayılan olarak gelir ve sistemin çalışması için önemli olduğu için kaldırmaya çalışmanız da pek önerilmez.

Herhangi bir C programını gcc veya clang ile derlediğinizde neredeyse her dağıtımda varsayılan olarak glibc kullanır. Dolayısıyla kullanmak için özel bir çaba sarf etmenize gerek yok.

Musl

Musl, sistemde varsayılan olarak bulunan **glibc** alternatifidir. Musl daha hafiftir fakat dağıtımların geneli glibc kullandığı ve her kaynak kodun düzgün şekilde derlenememesinden dolayı Türkmen linuxta C kütüphanesi olarak glibc tercih edilmiştir.

Yine de musl kullanarak derleme yapılabilmektedir. Bunun için **ymp install musl** komutu ile musl yükleyip sonrasında **musl-gcc** ile derleme yapabilirsiniz.

```
$ ymp install musl
```

Not: tüm sistem kütüphaneleri glibc uyumlu çalıştığı için musl kullanarak yapacağınız derlemelerde sistem kütüphanelerinden yararlanamazsınız.

Derleyici ayarlama

ymp varsayılan olarak **gcc** kullanır. Bunu **/etc/ymp.yaml** içerisinden değiştirebilirsiniz veya **--build:cc=xxx** şeklinde ayarlayabilirsiniz.

Paket yapımı dışında genellikle **CC** çevresel değişkeni kullanılarak derleyici ayarlanabilir.

```
$ export CC=musl-gcc
$ make
```

Donanım

Sürücüler

Linux Firmware

linux-firmware linux çekirdeği ile gelmeyen sürücüleri içerir. Bu sürücüler sayesinde wifi gibi bazı ek donanımlar çalışabilmektedir.

linux-firmware kurulumu

linux-firmware Türkmen deposunda paket olarak yer almaz. Bunun yerine **mklinux** paketi içerisinde yer alan **mkfw** komutu yardımı ile kurulur. Bunun için incelikle mklinux kuralım.

```
$ ymp install mklinux
```

Ardından **mkfw** ile linux-firmware kurulumun gerçekleştirelim.

```
$ mkfw -i
```

Not: Alternatif olarak https://git.kernel.org/pub/scm/linux/kernel/git/firmware/linux-firmware.git adresinden arşivi indirin. Bir dizine açın. Ardından **make install** komutu ile kurun.

Son olarak initramfs imajımızı güncellememiz gerekebilir. Bunun için aşağıdaki komutu kullanabiliriz.

```
$ update-initramfs -u -k <kernel-sürümü>
```

Ekran Kartı

3D controller kapatmak

Çift ekran kartı bulunan laptoplarda **3D controller** bulunur. Bunun yanında bir de **VGA controller** bulunur.

3D controller daha yüksek güç tükettiği ve gündelik kullanımda hiçbir işe yaramadığı için kapatmak istenilebilir.

Bunun için öncelikle **3D controller** aygıtımızı tespit edelim.

```
for dir in /sys/class/drm/card[0-9*] ; do
# 03 Display controller
# 02 3D controller
    if grep "^0x0302" $dir/device/class ; then
      pci="$(basename $(readlink $dir/device))"
      echo "Found 3D controller: ${pci:5}"
    fi
done
```

Daha sonra hbulduğumuz pci adını kullanarak donanımı kapatalım. Bu komutu başlangıçta çalıştırmamız gereklidir.

```
echo 1 > /sys/bus/pci/devices/0000:<pci-adi>/remove
```

Not: Türkmen linuxta bu iş için **disable-secondary-gpu** adında bir servis oluşturulmuştur. Bu servisi etkinleştirmeniz yeterli olmaktadır.

```
$ rc-update add disable-secondary-gpu
```

Çoklu Ortam

Pipewire

Pipewire çoklu ortam yöneticisidir.

Türkmen linuxta pipewire pulseaudio yerine kullanılır ve pipewire-pulse pakete dahildir.

Kurulumu

pipewire paketini yüklemeniz gerekmektedir. Ardından **wireplumber** yükleyip masaüstü ortamı ile çalışmasını sağlamalısınız.

```
$ ymp install pipewire wireplumber
```

Kurulum tamamlandıktan sonra oturumunuzu kapatıp açmanız gerekebilir.

Çalışıp çalışmadığını test etmek için **pactl info** ve **wpctl status** komutlarını kullanarak pipewire ile ilgili bilgi alabilirsiniz.

Not: pactl komutu pipewire-pulse ile sağlanır. wpctl komutu wireplumber ile sağlanır.

Uzak makinaya bağlanma

Öncelikle gerekli modülü sunucu olarak kullanılacak makinada aşağıdaki komut ile etkinleştirelim.

```
# Bu kısım sesi alacak olan makinada çalıştırılır.
# auth-ip-acl parametresini yazmazsanız herkes tarafından elişilebilir olur.
# Sadece bu parametre ile belirtilen ip adresine izin verilir.
# Birden çok ip belirtmek için aralarına ; işareti koyulmalıdır.
$ pactl load-module module-native-protocol-tcp auth-ip-acl=192.168.0.18;192.168.0.15
```

Daha sonra bağlantı kurmak için **PULSE_SERVER** çevresel değişkenini kullanabiliriz.

```
# Bu kısım sesi gönderecek olan makinada çalıştırılır.
$ export PULSE_SERVER=192.168.0.12
$ mpv /home/pingu/test.mkv
```

Ses seviyesi ayarı

Wireplumber üzerinden ses seviyelerini ayarlayabilirsiniz.

Öncelikle **wpctl status** komutu ile mevcut ses aygıtlarının ve uygulamaların id değerlerini bulalım.

```
PipeWire 'pipewire-0' [0.3.67, root@(none), cookie:2586580591]
 └─ Clients:
        31. pipewire-pulse                      [0.3.67, root@(none), pid:4772]
        33. WirePlumber                         [0.3.67, root@(none), pid:4771]
        34. WirePlumber [export]                [0.3.67, root@(none), pid:4771]
        63. Firefox                             [0.3.67, root@(none), pid:4806]
        69. wpctl                               [0.3.67, root@(none), pid:9322]

Audio
 ├─ Devices:
 │      47. Built-in Audio                      [alsa]
 │      55. Built-in Audio                      [alsa]
 │      59. Built-in Audio                      [alsa]
 │
 ├─ Sinks:
 │      32. Built-in Audio Analog Stereo        [vol: 0.39]
 │      51. Built-in Audio Analog Stereo        [vol: 0.40]
 │  *   57. Built-in Audio Analog Stereo        [vol: 1.00]
 │
 ├─ Sink endpoints:
 │
 ├─ Sources:
 │      37. Built-in Audio Analog Stereo        [vol: 1.00]
```

```
  │       38. Built-in Audio Stereo               [vol: 1.00]
  │   *   52. Built-in Audio Analog Stereo        [vol: 1.00]
  │
  ├─ Source endpoints:
  │
  └─ Streams:
         64. Firefox
              44. output_FL       > Generic Analog:playback_FL   [active]
              66. output_FR       > Generic Analog:playback_FR   [active]

Video
  ├─ Devices:
  │
  ├─ Sinks:
  │
  ├─ Sink endpoints:
  │
  ├─ Sources:
  │
  ├─ Source endpoints:
  │
  └─ Streams:

Settings
  └─ Default Configured Node Names:
```

Bulduğumuz id değerini kullanarak ses seviyesini ayarlayabiliriz.

```
# Ses seviyesi 0-1 arası değerde olmalıdır.
# Daha yüksek seviyeler de ayarlanabilir. (tavsiye edilmez)
$ wpctl set-volume 57 0.8
```

Sessiz moda alıp geri açmak için aşağıdaki gibi komut kullanılabilir.

```
# 1 sessiz moda alır. 0 sessiz moddan çıkar.
$ wpctl set-mute 57 0
```

Masaüstü Ortamları

Uygulamalar

AppImage

AppImage paketleri tüm bağımlılıkları içerisinde kurulu gelen tek dosyadan oluşan uygulama paketleridir. Bu paketler sayesinde uygulamaları sürümden bağımsız şekilde ve ek bağımlılık kurmadan kullanabilirsiniz.

AppImage için fuse yüklemeniz gerekmektedir.

AppImage dosyalarına https://appimage.github.io/ ve https://www.appimagehub.com/ adreslerinden ulaşabilirsiniz.

AppImage çalıştırmak

Öncelikle dosyayı çalıştırılabilir yapmalısınız.

```
$ chmod 755 dosya.appimage
```

Daha sonra dosyayı komut çalıştırır gibi çalıştırabilirsiniz.

```
$ ./dosya.appimage
```

Not: AppImage dosyaları herhangi bir güvenlik kontrolünden geçirilmediği için güvenilir olmayan kaynaklardan gelen dosyaları çalıştırmayınız.

AppImage dosyalarını uygulama menüsüne eklemek

Bunun appimage dosyamısa **--appimage-extract** parametresi vererek içini açalım. Bize **squashfs-root** dizini oluşturulacaktır. Bu dizini istediğimiz bir yere kopyalayıp aşağıdaki gibi bir uygulama başlatıcısı hazırlayalım. Bu başlatıcıyı ~/.local/share/applications dizinine koyalım.

```
[Desktop Entry]
Version=1.0
Name=Firefox
Comment=Web Browser
Exec=/data/user/pingu/firefox/firefox %u
Icon=/data/user/pingu/firefox/icon.png
Terminal=false
Type=Application
MimeType=text/html;
Categories=Network;WebBrowser;
```

Burada MimeType kısmına belirtilen dosyalar birlikte aç menüsünde gözükmesini sağlar. Bir dosyanın mimetype adına **file --mime dosya** komutu ile ulaşabilirsiniz.

Flatpak

Flatpak sistemden bağımsız şekilde uygulama çalıştıran bir altyapıdır. Bu sayede normal yolla kullanmak için çok fazla bağımlılığa ihtiyaç duyan veya kurulması mümkün olmayan uygulamalar çalıştırılabilir.

Kurulum

Öncelikle **flatpak** yükleyelim.

```
$ ymp install flatpak
```

Daha sonra gereken servisleri etkinleştirelim.

```
$ rc-update add devfs
$ rc-update add fuse
$ rc-update add hostname
```

Depo ekleme

Flatpak için uygulama deposu eklememiz gereklidir. Flathub deposunu aşağıdaki gibi ekleyebilirsiniz.

```
$ flatpak remote-add --user --if-not-exists flathub https://flathub.org/repo/flathub.flatpakrepo
# --user parametresi yetkili kullanıcı olmadan eklemek içindir.
```

İsterseniz aşağıdaki gibi alias tanımı yapıp sürekli kullanıcı modunda kullanabilirsiniz.

```
alias flatpak=''flatpak --user
```

Uygulama yükleme

flatpak search kullanarak uygulama arayabilirsiniz. Yüklemek için **flatpak install** kaldırmak için ise **flatpak remove** komutu kullanılır.

Not: Yükleme ve kaldırma için **appid** değeri kullanılır.

```
$ flatpak search  dolphin
Name      Description              Application ID      Version   Branch Remotes
Dolphin   File Manager             org.kde.dolphin     23.04.0   stable flathub
...
$ flatpak install org.kde.dolphin
...
$ flatpak remove org.kde.dolphin
...
```

Uygulamaları güncelleme

flatpak upgrade komutu ile güncelleyebilirsiniz.

```
$ flatpak upgrade
```

Wayland

Weston

Weston basit wayland pencere yöneticisidir. Genellikle test amaçlı kullanılır fakar masaüstü ortamı olarak kullanmak da mümkündür.

Yükleme

Ymp kullanarak weston yüklemek için **weston** paketini yüklemelisiniz.

Ardından **seatd** ve **devfs** servislerini etkinleştirmelisiniz.

```
# paketi kuralım
$ ymp install weston
# servisleri etkinleştirmek için
$ rc-update add seatd
$ rc-update add devfs
# servisleri açmak için
$ rc-service devfs start
$ rc-service seatd start
```

Çalıştırma

Çalıştırmak için tty ekranındayken eğer **seatd** servisi açıksa doğrudan **weston** komutu ile oturum açabilirsiniz. Service açmadıysanız **seatd-launch weston** komutu kullanabilirsiniz.

Not: X11 içerisinde weston çalıştırırsanız weston pence modunda çalıştırılacaktır. Bu sayede X11 üzerinde weston uygulamalarını test etebilirsiniz.

X11

LXDE

Lxde hafif masaüstü ortamıdır. Az kaynak kullanması ve nadiren güncelleme aldığı için tercih edilir.

Kurulumu

Ymp kullanarak lxde yüklemek için aşağıdaki komutu kullanabilirsiniz.

```
$ ymp install @lxde
```

Lxde openbox pencere yöneticisi ile çalışır. Bu yüzden openbox yüklemeniz gerekmektedir.

```
$ ymp install openbox
```

xinitrc ayarları

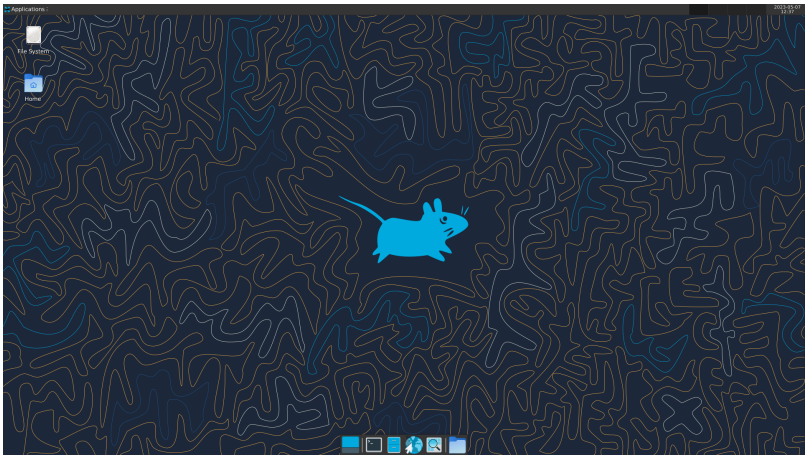
startx komutu çalıştırdığınızda lxde başlatılmasını istiyorsanız **~/.xinitrc** dosyanızın sonuna aşağıdaki gibi ekleme yapmalısınız.

```
exec startlxde
```

Bununla birlikte elogind gerektiği için elogind etkinleştirmelisiniz.

Xfce

Xfce kolay kullanılan ve en çok tercih edilen masaüstü ortamlarından biridir.



Kurulumu

Ymp kullanarak xfce yüklemek için aşağıdaki komutu kullanabilirsiniz.

```
$ ymp install @xfce.base
```

Ek paketleri yüklemek için ise **@xfce.extra** yüklemelisiniz.

xinitrc ayarları

startx komutu çalıştırdığınızda xfce başlatılmasını istiyorsanız **~/.xinitrc** dosyanızın sonuna aşağıdaki gibi ekleme yapmalısınız.

```
exec startxfce4
```

Ayrıca startx yerine doğrudan **startxfce4** komutunu kullanabilirsiniz.

Bununla birlikte elogind gerektiği için elogind etkinleştirmelisiniz.

Blog

Busybox ile Minimal Dağıtım Oluşturma

Busybox tek bir ikili dosya olarak temel linux komutlarını içerisinde barındıran bir dosyadır. Bu dosya ve kernel olduğu zaman sistemimiz açılıçacak temel komutları kullabileceğimiz bir linux elde etmiş oluruz.

İlk olarak busyboxu çalışma dizinimize kopyalayalım. Busyboxun static olarak derlenmiş olduğundan emin olalım.

```
$ mkdir distro
$ cd distro
$ install /bin/busybox ./busybox
```

```
$ ldd ./busybox
-> özdevimli bir çalıştırılabilir değil
```

Ardından initramfs için init dosyamızı aşağıdaki gibi oluşturalım.

```
#!/busybox ash
PATH=/bin
/busybox mkdir /bin
/busybox --install -s /bin
exec /busybox ash
```

initramfs dosyamızı paketleyelim.

```
$ chmod +x init
$ find ./ | cpio -H newc -o > initrd.img
# isterseniz initrd.img sıkıştırabilirsiniz.
$ gzip -9 initrd.img
```

Bu aşamada isterseniz initd.img dosyasını sıkıştırabilirsiniz.

Sıra initrd.img ve kernelin birleştirilmesine geldi. Bunun için aşağıdaki gibi dizin yapısına dosyalarımızı kopyalayalım. vmlinuz dosyamızı kendi sistemimizdeki /boot içinden alabiliriz.

```
iso/vmlinuz
iso/initrd.img
iso/boot/grub/grub.cfg
```

Burada grub.cfg dosyamız bootloader komutlarını içerir. İçerisine aşağıdaki gibi olmalıdır.

```
linux /vmlinuz
initrd /initrd.img
boot
```

Son olarak iso dosyamızı paketleyelim.

```
$ grub-mkrescue iso/ -o distro.iso
```

Minimal sistemimiz hazır. Test etmek için qemu kullanabilirsiniz.

```
$ qemu-system-x86_64 -cdrom distro.iso -m 1G
```

Burada busybox yerine isterseniz static olarak derlenmiş herhangi bir C dosyasını kullanabilirsiniz. Yapmanız gereken init dosyası yerine bu dosyayı kullanmaktır.

Programlama

Bash dersi

Bu yazıda bash betiği yazmayı hızlıca anlatacağım. Bu yazıda karıştırılmaması için girdilerin olduğu satırlar <- ile çıktıların olduğu satırlar -> ile işaretlenmiştir.

Açıklama satırı ve dosya başlangıcı

Açıklamalar **#** ifadesiden başlayıp satır sonuna kadar devam eder. Dosyanın ilk satırına **#!/bin/bash** eklememiz gerekmektedir. Bash betikleri genellikle **.sh** uzantılı olur. Bash betikleri girintilemeye duyarlı değildir. Bash betiği yazarken girintileme için 4 boşluk veya tek tab kullanmanızı öneririm.

Bash betiklerinde alt satıra geçmek yerine **;** kullanabiliriz. Bu sayede kaynak kod daha düzenli tutulabilir.

```
#!/bin/bash
#Bu bir açıklama satırıdır.
```

Bash betiklerini çalıştırmak için öncelikle çalıştırılabilir yapmalı ve sonrasında aşağıdaki gibi çalıştırılmalıdır.

```
chmod +x ders1.sh
./ders1.sh
```

: komutu hiçbir iş yapmayan komuttur. Bu komutu açıklama niyetine kullanabilirsiniz. **true** komutu ile aynı anlama gelmektedir.

Çoklu açımlama satırı için aşağıdaki gibi bir ifade kullanabilirsiniz.

```
: "
Bu bir açıklama satıdırıdır.
Bu da diğer açıklama satırıdır.
Bu da sonunca açıklama satırıdır.
"
```

Ekrana yazı yazalım

Ekrana yazı yazmak için **echo** ifadesi kulanılır.

```
echo Merhaba dünya
-> Merhaba dünya
```

Ekrana özel karakterleri yazmak için **-e** parametresi kullanmamız gerekmektedir.

```
echo -e "Merhaba\ndünya"
-> Merhaba
-> dünya
```

Ekrana renkli yazı da yazdırabiliriz. Bunun için **\033[x;..;ym** ifadesini kullanırız. Burada **x** ve **y** özellik belirtir. Örneğin:

```
# Mavi renkli kalın merhaba ile normal dünya yazdırır.
echo -e "\033[34;1mMerhaba\033[0m Dünya"
```

Aşağıda tablo halinde özellik numarası ve anlamları verilmiştir.

Özellik numarası ve anlamları

Özellik	Anlamı	Özellik	Anlamı	Özellik	Anlamı

0	Sıfırla	30	Siyah yazı	40	Siyah arka plan
1	Aydınlık	31	Kırmızı yazı	41	Kırmızı arka plan
2	Sönük	32	Yeşil yazı	42	Yeşil arka plan
3	İtalik	33	Sarı yazı	43	Sarı arka plan
4	Altı çizili	34	Mavi yazı	44	Mavi arka plan
5	Yanıp sönen	35	Magenta yazı	45	Magenta arkaplan
6	Yanıp sönen	36	Turkuaz yazı	46	Turkuaz arka plan
7	Ters çevirilmiş	37	Beyaz yazı	47	Beyaz arka plan
8	Gizli	39	Varsayılan yazı	49	Varsayılan arkaplan

Çift tırnak (") içine yazılmış yazılardaki değişkenler işlenirken tek tırnak (') içindekiler işlenmez. Örneğin:

```
var=12
echo "$var"
echo '$var'
-> 12
-> $var
```

Parametreler

Bir bash betiği çalıştırılırken verilen parametreleri **$** ifadesinden sonra gelen sayı ile kullanabiliriz. **$#** bize kaç tane parametre olduğunu verir. **$@** ifadesi ile de parametrelerin toplamını elde edebiliriz.

```
echo "$1 - $# - $@"

./ders1.sh merhaba dünya
-> merhaba - 2 - merhaba dünya
```

Değişkenler ve Sabitler

Değişkenler ve sabitler programımızın içerisinde kullanılan verilerdir. Değişkenler tanımlandıktan sonra değiştirilebilirken sabitler tanımlandıktan sonra değiştirilemez.

Değişkenler sayı ile başlayamaz, Türkçe karakter içeremez ve */*[[($* gibi özel karakterleri içeremez.

Normal Değişkenler aşağıdaki gibi tanımlanır.

```
sayı=23
yazi="merhaba"
```

+= ifadesi var olan değişkene ekleme yapmak için kullanılır. Değişkenin türünü belirlemeden tanımlamışsak yazı olarak ele alır.

```
typeset -i a # Değişkeni sayı olarak belirttik.
a=1 ; b=1
a+=1 ; b+=1
```

```
echo "$a $b"
-> 2 11
```

Çevresel değişkenler tüm alt programlarda da geçerlidir. Çevresel değişken tanımlamak için başına **export** ifadesi yerleştirilir.

```
export sayi=23
export yazi="merhaba"
```

Sabitler daha sonradan değeri değiştirilemeyen verilerdir. Sabit tanımlamak için başına **decrale -r** ifadesi yerleştirilir.

```
declare -r yazi="merhaba"
declade -r sayi=23
```

Değişkenler ve sabitler kullanılırken **${}** işareti içine alınırlar veya başına **$** işareti gelir. Bu doküman boyunca ilk kullanım biçimi üzerinden gideceğim.

```
deneme="abc123"
echo ${deneme}
-> abc123
```

Sayı ve yazı türünden değişkenler farklıdır. sayıyı yazıya çevirmek için " işaretleri arasına alabiliriz. Birden fazla yazıyı toplamak için yan yana yazmamız yeterlidir.

```
sayi=11
yazi="karpuz"
echo "${sayi}${karpuz} limon"
-> 11karpuz limon
```

Sayı değişkenleri üzerinde matematiksel işlem yapmak için aşağıdaki ifade kullanılır. *(+-*/ işlemleri için geçerlidir.)*

```
sayi=12
sayi=$((${sayi}/2))
echo ${sayi}
-> 6
```

Bununla birlikte matematiksel işlemler için şunlar da kullanılabilir.

```
expr 3 + 5 # her piri arasında boşluk gerekli
-> 8
echo 6-1 | bc -l # Burada -l virgüllü sayılar için kullanılır.
-> 5
python3 -c "print(10/2)"
-> 5.0
```

Değişkenlere aşağıdaki tabloda belirttiğim gibi müdahale edilebilir. Karakter sayısı 0'dan başlar. Negatif değerler sondan saymaya başlar.

Değişkene müdahale (var="Merhaba")

İfade	Anlamı	Eşleniği
${var%aba}	Sondaki ifadeyi sil	Merh
${var#Mer}	Baştaki ifadeyi sil	haba
${var:1:4}	Baştan 1. 4. karakterler arası	erha
${var::4}	Baştan 4. karaktere kadar	Merha
${var:4}	Baştan 4. karakterden sonrası	aba
${var/erh/abc}	erh yerine abc koy	Mabcaba
${var,,}	hepsini küçük harf yap	merhaba
${var^^}	hepsini büyük harf yap	MERHABA
${var:+abc}	var tanımlıysı abc dördürür.	abc

Diziler

Diziler birden çok eleman içeren değişkenlerdir. Bash betiklerinde diziler aşağıdaki gibi tanımların ve kullanılır.

```
dizi=(muz elma limon armut)
echo ${dizi[1]}
-> elma
echo ${#dizi[@]}
-> 4
echo ${dizi[@]:2:4}
-> limon armut
dizi+=(kiraz)
echo ${dizi[-1]}
-> kiraz
```

Diziler eleman indisleri ile kullanmanın yanında şu şekilde de tanımlanabilir.

```
declare -A dizi
dizi=([kirmizi]=elma [sari]=muz [yesil]=limon [turuncu]=portakal)
for isim in ${!dizi[@]} ; do
    echo -n "$isim "
done
echo
-> turuncu yesil sari kirmizi
for isim in ${dizi[@]} ; do
    echo -n "$isim "
done
echo
-> portakal limon muz elma
echo ${dizi[kirmizi]}
-> elma
```

Klavyeden değer alma

Klavyeden değer almak için **read** komutu kullanılır. Alınan değer değişken olarak tanımlanır.

```
read deger
<- merhaba
echo $deger
-> merhaba
```

Koşullar

Koşullar **if** ile **fi** ile biter. Koşul ifadesi sonrası **then** kullanılır. ilk koşul sağlanmıyorsa **elif** ifadesi ile ikinci koşul sorgulanabilir. Eğer hiçbir koşul sağlanmıyorsa **else** ifadesi içerisindeki eylem gerçekleştirilir.

```
if ifade ; then
    eylem
elif ifade ; then
    eylem
else
    eylem
fi
```

Koşul ifadeleri kısmında çalıştırılan komut 0 döndürüyorsa doğru döndürmüyorsa yalnış olarak değerlendirilir. **[[** veya **[** ile büyük-küçük-eşit kıyaslaması, dosya veya dizin varlığı vb. gibi sorgulamalar yapılabilir. Bu yazıda **[[** kullanılacaktır.

```
read veri
if [[ ${veri} -lt 10 ]] ; then
    echo "Veri 10'dan küçük"
else
    echo "Veri 10'dan büyük veya 10a eşit"
fi

<- 9
-> Veri 10'dan küçük
<- 15
-> Veri 10'dan büyük veya 10a eşit
```

[[yerleşiği ile ilgili başlıca ifadeleri ve kullanımlarını aşağıda tablo olarak ifade ettim. **[** bir komutken **[[** bir yerleşiktir. **[** ayrı bir süreç olarak çalıştırılır. Bu yüzden **[[** kullanmanızı tavsiye ederim.

[[ifadeleri ve kullanımları

İfade	Anlamı	Kullanım şekli
-lt	küçüktür	[[${a} -lt 5]]
-gt	büyüktür	[[${a} -gt 5]]
-eq	eşittir	[[${a} -eq 5]]
-le	küçük eşittir	[[${a} -le 5]]
-ge	büyük eşittir	[[${a} -ge 5]]
-f	dosyadır	[[-f /etc/os-release]]
-d	dizindir	[[-d /etc]]

-e	vardır (dosya veya dizindir)	[[-e /bin/bash]]
-L	sembolik bağdır	[[-L /lib]]
-n	uzunluğu 0 değildir	[[-n ${a}]]
-z	uzunluğu 0dır	[[-z ${a}]]
!	ifadenin tersini alır.	[[! .... veya ! [[....
>	alfabeti olarak büyüktür	[["portakal" > "elma"]]
<	alfabetik olarak küçüktür	[["elma" < "limon"]]
==	alfabetik eşittir	[["nane" == "nane"]]
!=	alfabetik eşit değildir	[["name" != "limon"]]
=~	regex kuralına göre eşittir	[["elma1" =~ ^[a-z]l.*[1]$]]
||	mantıksal veya bağlacı	[[.... || ....]] veya [[....]] || [[....]]
&&	mantıksal ve bağlacı	[[.... && ....]] veya [[....]] && [[....]]

true komutu her zaman doğru **false** komutu ile her zaman yanlış çıkış verir.

Bazı basit koşul ifadeleri için if ifadesi yerine aşağıdaki gibi kullanım yapılabilir.

```
[[ 12 -eq ${a} ]] && echo "12ye eşit." || echo "12ye eşit değil"
#bunun ile aynı anlama gelir:
if [[ 12 -eq ${a} ]] ; then
    echo "12ye eşit"
else
    echo "12ye eşit değil"
fi
```

case yapısı

case yapısı case ile başlar değerden sonra gelen **in** ile devam eder ve koşullardan sonra gelen **esac** ile tamamlanır. case yapısı sayesinde if elif else ile yazmamız gereken uzun ifadeleri kısaltabiliriz.

```
case deger in
    elma | kiraz)
        echo "meyve"
        ;;
    patates | soğan)
        echo "sebze"
        ;;
    balık)
        echo "hayvan"
    *)
        echo "hiçbiri"
        ;;
esac
# Şununla aynıdır:
```

```
if [[ "${deger}" == "elma" || "${deger}" == "kiraz" ]] ; then
    echo "meyve"
elif [[ "${deger}" == "patates" || "${deger}" == "soğan" ]] ; then
    echo "sebze"
elif [[ "${değer}" == "balık" ]] ; then
    echo "hayvan"
else
    echo "hiçbiri"
fi
```

Döngüler

Döngülerde **while** ifadesi sonrası koşul gelir. **do** ile devam eder ve eylemden sonra **done** ifadesi ile biter. Döngülerde ifade doğru olduğu sürece eylem sürekli olarak tekrar eder.

```
while ifade ; do
    eylem
done
```

Örneğin 1den 10a kadar sayıları ekrana yan yana yazdıralım. Eğer echo komutumuzda **-n** parametresi verilirse alt satıra geçmeden yazmaya devam eder.

```
i=1
while [[ ${i} -le 10 ]] ; do
    echo -n "$i " # sayıyı yazıya çevirip sonuna yanına boşluk koyduk
    i=$((${i}+1)) # sayıya 1 ekledik
done
echo # en son alt satıra geçmesi için
-> 1 2 3 4 5 6 7 8 9 10
```

for ifadesinde değişken adından sonra **in** kullanılır daha sonra dizi yer alır. diziden sonra **do** ve bitişte de **done** kullanılır.

```
for degisken in dizi ; do
    eylem
done
```

Ayrı örneğin for ile yapılmış hali

```
for i in 1 2 3 4 5 6 7 8 9 10 ; do
    echo -n "${i} "
done
echo
-> 1 2 3 4 5 6 7 8 9 10
```

Ayrıca uzun uzun 1den 10a kadar yazmak yerine şu şekilde de yapabiliyoruz.

```
for i in {1..10} ; do
    echo -n "${i} "
done
echo
-> 1 2 3 4 5 6 7 8 9 10
```

Buradaki özel kullanımları aşağıda tablo halinde belirttim.

küme parantezli ifadeler ve anlamları

İfade	Anlamı	eşleniği
{1..5}	aralık belirtir	1 2 3 4 5
{1..7..2}	adımlı aralık belirtir	1 3 5 7
{a,ve}li	kurala uygun küme belirtir	ali veli

Fonksiyonlar

Fonksiyonlar alt programları oluşturur ve çağırıldığında işlerini yaptıktan sonra tekrar ana programdan devam edilmesini sağlar. Bir fonksiyonu aşağıdaki gibi tanımlayabiliriz.

```
isim(){
    eylem
    return sonuç
}
# veya
function isim(){
    eylem
    return sonuç
}
```

Burada **return** ifadesi kullanılmadığı durumlarda 0 döndürülür. return ifadesinden sonra fonksiyon tamamlanır ve ana programdan devam edilir.

Bu yazı boyunca ilkini tercih edeceğiz.

Fonksionlar sıradan komutlar gibi parametre alabilirler ve ana programa ait sabit ve değişkenleri kullanabilirler.

```
sayi=12
topla(){
    echo $((${sayi}+$1))
    return 0
    echo "Bu satır çalışmaz"
}
topla 1
-> 13
```

local ifadesi sadece fonksionun içinde tanımlanan fonksion bitiminde silinen değişkenler için kullanılır.

Fonksionların çıkış turumlarını koşul ifadesi yerine kullanabiliriz.

```
read sayi
teksayi(){
    local i=$(($1+1)) # sayıya 1 ekledik ve yerel hale getirdik.
    return $((${i}%2))  # sayının 2 ile bölümünden kalanı döndürdük
}
if teksayi ${sayi} ; then
    echo "tek sayıdır"
```

```
else
    echo "çift sayıdır"
fi

<- 12
-> çift sayıdır
<- 5
-> tek sayıdır
```

Bir fonksionun çıktısını değişkene **$(isim)** ifadesi yadımı ile atayabiliriz. Aynı durum komutlar için de geçerlidir.

```
yaz(){
    echo "Merhaba"
}
echo "$(yaz) dünya"
-> Merhaba dünya
```

Tanımlı bir fonksionu silmek için **unset -f** ifadesini kullanmamız gereklidir.

```
yaz(){
    echo "Merhaba"
}
unset -f yaz
echo "$(yaz) dünya"
-> bash: yaz: komut yok
-> dünya
```

Burada dikkat ederseniz olmayan fonksionu çalıştırmaya çalıştığımız için hata mesajı verdi fakat çalışmaya devam etti. Eğer herhangi bir hata durumunda betiğin durmasını istiyorsak **set -e** bu durumun tam tersi için **set +e** ifadesini kullanmalıyız.

```
echo "satır 1"
acho "satır 2" # yanlış yazılan satır fakat devam edecek
echo "satır 3"
set -e
acho "satır 4" # yanlış yazılan satır çalışmayı durduracak
echo "satır 5" # bu satır çalışmayacak
-> satır 1
-> bash: acho: komut yok
-> satır 3
-> bash: acho: komut yok
```

Dosya işlemleri

Bash betiklerinde **stdout stderr** ve **stdin** olmak üzere 2 çıktı ve 1 girdi bulunur. Ekrana stderr ve stdout beraber yazılır.

dosya ifadeleri ve anlamları

İfade	Türü	Anlamı
stdin	Girdi	Klavyeden girilen değerler.

stdout	Çıktı	Sıradan çıktılardır.
stderr	Çıktı	Hata çıktılarıdır.

cat komutu ile dosya içeriğini ekrana yazdırabiliriz. Dosya içeriğini **$(cat dosya.txt)** kullanarak değişkene atabiliriz.

dosya.txt içeriğinin aşağıdaki gibi olduğunu varsayalım.

```
Merhaba dünya
Selam dünya
sayı:123
```

Ayağıdaki örnekle dosya içeriğini önce değişkene atayıp sonra değişkeni ekrana yazdırdık.

```
icerik=$(cat ./dosya.txt)
echo "${icerik}"
-> Merhaba dünya
-> Selam dünya
-> sayı:123
```

grep "sözcük" dosya.txt ile dosya içerisinde sözcük gezen satırları filitreleyebiliriz. Eğer grep komutuna **-v** paraketresi eklersek sadece içermeyenleri filitreler. Eğer filitrelemede hiçbir satır bulunmuyorsa yanlış döner.

```
grep "dünya" dosya.txt
-> Merhaba dünya
-> Selam dünya
grep -v "dünya" dosya.txt
-> sayi:123
```

Aşağıdaki tabloda bazı dosya işlemi ifadeleri ve anlamları verilmiştir.

dosya ifadeleri ve anlamları

İfade	Anlamı	Kullanım şekli
>	çıktıyı dosyaya yönlendir (stdout)	echo "Merhaba dünya" > dosya.txt
2>	çıktıyı dosyaya yönlendir (stderr)	ls /olmayan/dizin 2> dosya.txt
>>	çıktıyı dosyaya ekle	echo -n "Merhaba" > dosya.txt && echo "dünya" >> dosya.txt
&>	çıktıyı yönlendir (stdout ve stderr)	echo "$(cat /olmayan/dosya) deneme" &> dosya.txt

Ayrıca dosyadan veri girişleri için de aşağıda örnekler verilmiştir:

```
# <<EOF:
# EOF ifadesi gelene kadar olan kısmı girdi olarak kullanır:
cat > dosya.txt <<EOF
Merhaba
dünya
EOF
```

```
# < dosya.txt
# Bir dosyayı girdi olarak kullanır:
while read line ; do
    echo ${line:2:5}
done < dosya.txt
```

/dev/null içine atılan çıktılar yok edilir. **/dev/stderr** içine atılan çıktılar ise hata çıktısı olur.

Boru hattı

Bash betiklerinde **stdin** yerine bir önceki komutun çıktısını kullanmak için boru hattı açabiliriz. Boru hattı açmak için iki komutun arasına **|** işareti koyulur. Boru hattında soldan sağa doğru çıktı akışı vardır. Boru hattından sadece **stdout** çıktısı geçmektedir. Eğer **stderr** çıktısını da boru hattından geçirmek istiyorsanız **|&** kullanmalısınız.

```
topla(){
    read sayi1
    read sayi2
    echo $((${sayi1}+${sayi2}))
}
topla
<- 12
<- 25
-> 37
sayiyaz(){
    echo 12
    echo 25
}
 sayiyaz | topla
-> 37
```

Kod bloğu

{ ile **}** arasına yazılan kodlar birer kod bloğudur. Kod blokları fonksionların aksine argument almazlar ve bir isme sahip değillerdir. Kod blokları tanımlandığı yerde çalıştırılırlar. Kod bloğuna boru hattı ile veri girişi ve çıkışı yapılabilir.

```
cikart(){
    read sayi1
    read sayi2
    echo $((${sayi1}-${sayi2}))
}
cikart
<- 25
<- 12
-> 13
{
    echo 25
    echo 12
} | cikart
-> 13
# veya kısaca şu şekilde de yapılabilir.
{ echo 25 ; echo 12 ; } | cikart
-> 13
```

select komutu

select kullanarak basit menü oluşturabiliriz.

```
select deger in ali veli 49 59 ; do
    echo $REPLY # seçilen sayıyı verir
    echo $deger # seçilen elemanı verir
    break
done

-> 1) ali
-> 2) veli
-> 3) 49
-> 4) 59
-> #?
<- 1
-> 1
-> ali
```

Bu örnekde **REPLY** değişkeni seçtiğimiz sayıyı **deger** değişkeni ise seçtiğimiz elemanı ifade eder. **select** komutu sürekli olarak döngü halinde çalışır. Döngüden çıkmak için **break** kullandık.

Birden çok dosya ile çalışmak

Bash betikleri içerisinde diğer bash betiği dosyasını kullanmak için **source** yada **.** ifadeleri kullanılır. Diğer betik eklendiği zaman içerisinde tanımlanmış olan değişkenler ve fonksionlar kullanılabilir olur.

Örneğin deneme.sh dosyamızın içeriği aşağıdaki gibi olsun:

```
mesaj="Selam"
merhaba(){
    echo ${mesaj}
}
echo "deneme yüklendi"
```

Asıl betiğimizin içeriği de aşağıdaki gibi olsun.

```
source deneme.sh # deneme.sh dosyası çalıştırılır.
merhaba
-> deneme yüklendi
-> Selam
```

Ayrıca bir komutun çıktısını da betiğe eklemek mümkündür. Bunun için **<(komut)** ifadesi kullanılır. Aşağıda bununla ilgili bir örnek verilmiştir.

```
source <(curl https://gitlab.com/sulincix/outher/-/raw/gh-pages/deneme.sh) # Örnekteki adrese takılmayın :D
merhaba
merhaba2
echo ${sayi}
-> Merhaba dünya
-> 50
-> 100
```

exec komutu

exec komutu betiğin bundan sonraki bölümünü çalıştırmak yerine hedefteki komut ile değiştirilmesini sağlar. **exec** ile çalıştırılmış olan komut tamamlandığında betik tamamlanmış olur.

```
echo $$ # pid değeri yazdırır
bash -c 'echo $$' # yeni süreç oluşturduğu için pid değeri farklıdır.
exec bash -c 'echo $$' # mevcut komut ile değiştirildiği için pid değeri değişmez
echo "hmm" # Bu kısım çalıştırılamaz.
-> 5755
-> 5756
-> 5755
```

exec komutunu doğrudan terminalde çalıştırırsanız ve komut tamamlanırsa terminaldeki süreç kapanacağı için terminal doğal olarak kapanacaktır.

exec komutunu kullanarak yönlendirmeler yapabilirsiniz.

```
exec > log.txt # bütün çıktıları log.txt içine yazdırır.
echo "merhaba" # ekrana değil dosyaya yazılır.
exec < komutlar.txt # komutlar.txt dosyasındakiler girdi olarak kullanılır.
```

fd kavramı

bash programında birden çok **fd** kullanılabilir. var olan fdlere ulaşmak için **/proc/$$/fd/** konumuna bakabiliriz. 0 stdin 1 stdout 2 stderr olarak çalışır.

Not: $$ mevcut sürecin pid değerini verir.

```
exec 3> log.txt # yazmak için boş bir 3 numaralı fd açmak için.
echo "deneme" >&3
exec 3>& - # açık olan 3 nuramalı fd kapatmak için.
exec 2>&1 # stderr içine atılanı stdout içine aktarır.
exec 4< input.txt # okumak için 4 numaralı fd açmak için.
echo "hmm" > input.txt # girdi dosyamıza yazı yazalım.
read line <&4 # 3 nuramalı fd içinden değir okur.
exec 4<&- # 4 numaralı fd kapatmak için.
```

Hata ayıklama

bash komutuna farklı parametreler vererek kolayca script'inizi derleyebilirsiniz. Örneğin **-n** parametresi kodu çalıştırmayıp sadece hata kontrolü yapacaktır, **-v** komutları çalıştırmadan yazdıracak, **-x** ise işlem bittikten sonra kodları yazdıracaktır.

```
bash -n script_adi.sh
bash -v script_adi.sh
bash -x script_adi.sh
```

C Dersi

Bu derste C programlama dersi anlatılacaktır. Bu dersin düzgünce anlaşılabilmesi için temel düzey gnu/linux bilmeniz gerekmektedir.

Derleme işlemi

C derlemeli bir programlama dilidir. Yani yazılan kodun derleneek bilgisayarın anlayacağı hale getirilmesi gerekmektedir. Derleme işlemini **gcc** veya **clang** kullanarak yapabiliriz.

```
# koddan .o dosyası üretelim
$ gcc -c main.c
# .o dosyasından derlenmiş dosya üretelim.
$ gcc -o main main.o
# kodu çalıştıralım
$ ./main
-> Hello World
```

Yukarıdaki örnekte öncelikle **.o** uzantılı object dosyamızı ürettik. Bu dosya kodun derlenmiş fakat henüz kullanıma hazır hale getirilmemiş halidir. Bu sebeple **.o** dosyalarını linkleme işleminden geçirerek son halini almasını sağlamalıyız.

Not: derleyicimiz **.o** üretmeden de doğrudan derleme yapabilir.

```
$ gcc -o main main.c
```

Açıklama satırı

C kodlarında 3 farklı yolla girintileme yapılabilir.

1. **//** kullanarak satırın geri kalanını açıklama satırı yapabiliriz.

```
// bu bir açıklama satırıdır.
```

2. **/*** ile başlayıp ***/** ile biten alanlar açıklamadır.

```
/* Bu
   bir
   açıklama
   satırıdır */
```

3. **#if 0** ile başlayan satırdan **#endif** satırına kadar olan kısım açıklama satırıdır.

```
#if 0
bu bir
açıklama
satırıdır
#endif
```

Girintileme

C programlama dilinde blocklar **{}** karakterleri ile belirtilir. Kodun okunalı olması için girintilenmesi gereklidir fakat şart değildir. Girintileme için 4 boşluk veya 1 tab kullanabilirsiniz.

Bir block aşağıdaki gibi bir yapıya sahiptir.

```
aaaa (bbbb) {
        cccc;
```

```
  ddd;
}
```

Not: Her satırın sonunda ; işareti bulunmalıdır.

İlk program

C programları çalıştırıldığında **main** fonsiyonu çalıştırılır. Aşağıda örnek main fonksiyonu bulunmaktadır.

```
int main(int argc, char** argv) {
    return 0;
}
```

- **int main** kısmında int döndürülecek değer türü main adıdır.
- **int argc** parametre sayısını belirtir.
- **char **argv** parametre listesini belirtir.
- **return 0** komutu 0 ile çıkış yapar.

Burada **main** fonksiyonunu türünün bir önemi yoktur. **void** olarak da tanımlanabilir. Ayrıca kullanmayacaksak arguman tanımlamaya da gerek yoktur. Kısaca Şu şekilde de yazabilirdik.

```
void main(){}
```

Ekrana yazı yazma

Öncelikle **stdio.h** kütüphanesine ihtiyacimiz olduğu için onu eklemeliyiz. Ardından **printf** fonksiyonu ile ekrana yazı yazabiliriz.

printf fonksiyonunun 1. parametresi yazdirma şablonunu diğerleri ise yazdırılacak verileri belirtir.

```
#include <stdio.h>

int main(int argc, char** argv) {
    printf("%s\n", "Merhaba Dünya!");
    return 0;
}
```

- **include** ile belirttiğimiz dosyalar sistemde **/usr/include** içerisinde bulunur.
- **printf** fonksiyonundaki **%s** yazılar için, **%c** karakterler için, **%d** sayılar için kullanılır.

Değişkenler

C dilinde değişkenler aşağıdaki gibi tanımlanır.

```
...
int sayi = 12;
char* yazi = "test";
char karakter = 'c';
float sayi2 = 12.42;
...
```

Bununla birlikte **#define** kullanarak derlemeden önce koddaki alanların karşılığı ile değiştirilmesini sağlayabilirsiniz. Bu şekilde tanımlanan değerler derlemeden önce yerine yazıldığı için değişken olarak işlem görmezler.

```
#define sayi 12
...
printf("%d\n",sayi);
...
```

Diziler

Diziler iki şekilde tanımlanabilir.

1. Pointer kullanarak tanımlanabilir. Bu konunun detaylarına ilerleyen kısımda değinilecektir. Bu şekilde tanımlanan dizilerde başta uzunluk belirtilmek zorunda değildir.

```
int *dizi = {12, 22, 31};
```

2. Uzunluk belirterek tanımlanabilir. Bu şekilde tanımlanan dizilerin uzunluğu sabittir.

```
int dizi[3] = {12, 22, 31};
```

C dilinde string kavramı bulunmaz. Onun yerine karakter dizileri kullanılır.

```
char *txt = "deneme123";
```

Dizinin bir elemanına erişmek için aşağıdaki gibi bir yol kullanılır.

```
int *dizi = {12, 22, 31};
int c = dizi[1]; // dizinin 2. elemanı
```

Not: Dizi indisleri 0dan başlar.

Bir dizinin uzunluğunu dizinin bellekteki boyutunu birim boyutuna bölerek buluruz. Bunun için **sizeof** fonksiyonu kullanılır.

```
int *dizi = {11, 22, 31};
int l = sizeof(dizi) / sizeof(int);
```

Klavyeden değer alma

Klavyeden değer almak için **scanf** kullanılır. İlk parameter şablonu diğerleri ise değişkenlerin bellek adresini belirtir.

```
int sayi;
scanf("%d\n", &sayi);
```

Not: Bu şekilde değer alma yaptığımızda formata uygun olmayan şekilde değer girilebilir. Eğer böyle bir durum oluşursa değişken **NULL** olarak atanır. yani değeri bulunmaz. Buda kodun işleyişinde soruna yol açabilir. Bu yüzden değişkeni kullanmadan ince **NULL** olup olmadığını kontrol etmelisiniz.

Koşullar

Koşullar için **if** bloğu kullanılır. Block içindeki ifade **0** veya **NULL** olursa koşul sağlanmaz. Bu durumda varse **else** bloğu çalıştırılır.

```
if (koşul1) {
    block 1
} else if (koşul2) {
  block 2
} else {
  block 3
}
```

Örnek olarak girilen sayının çift olup olmadığını yazan uygulama yazalım.

```
#include <stdio.h>

int main(int argc, char** argv) {
    int sayi;
    scanf("%d",&sayi);
    if (sayi == NULL) {
        printf("%s\n", "Geçersiz sayı girdiniz.");
    } else if(sayi % 2) {
        printf("%d tektir.\n", sayi);
    } else {
        printf("%d çifttir.\n", sayi);
    }
    return 0;
}
```

Burada **%** operatörü 2 ile bölümden kalanı bulmaya yarar. Sayı tek ise 1 değilse 0 sonucu elde edilir. Bu sayede tek sayılar için koşul sağlanır çift sayılar için sağlanmaz.

Tek satırdan oluşan koşullarda **{}** kullanmaya gerek yoktur.

```
if (i < 32)
  printf("%s\n","32den küçüktür");
```

Koşul ifadeleri aşağıdaki gibi listelenebilir.

Koşul işleyicileri

ifade	anlamı	örnek
>	büyüktür	121 > 12
<	küçüktür	12 < 121
==	birbirine eşittir	121 == 121
!	karşıtlık bildirir.	!(12 > 121)
&&	logic and	"fg" == "aa" && 121 > 12
||	logic or	"fg" == "aa" || 121 > 12
!=	eşit değildir	"fg" != "aa"
>=	büyük eşittir	121 >= 121

<=	küçük eşittir	12 <= 12

Switch - Case

Bir sayıya karşılık bir işlem yapmak için **switch - case** yapısı kullanılır.

```
      switch(sayi) {
        1:
          // sayı 1se burası çalışır.
          // break olmadığı için alttan devam eder.
        2:
          // sayı 1 veya 2 ise burası çalışır.
          break;
        3:
          // sayı 3 ise burası çalışır.
        default:
          // sayı eşleşmezse burası çalışır.
}
```

Döngüler

Döngüler koşullara benzer fakat döngülerde koşul sağlanmayana kadar block içi tekrarlanır. Döngü oluşturmak için **while** kullanılır.

```
int i=10;
while(i<0){
    printf("%d\n", i);
    i--;
}
```

Yukarıdaki örnekte 10dan 0a kadar geri sayan örnek verilmiştir. En son i değişkeni 0 olduğunda koşul sağlanmadığı için döngü sonlanır.

Aynı işlemi **for** ifadesi ile de yapabiliriz.

```
for(int i=10;i<0;i--){
    printf("%d\n", i);
}
```

Burada for içerisinde 3 bölüm bulunur. İlkinde değer atanır. İkincinde koşul yer alır. Üçüncüsünde değişkene yapılacak işlem belirtilir.

Döngülerde **continue** kullanarak döngünün tamamlanması beklenmeden başa dönülür. **break** kullanarak döngüden çıkılır.

```
      int sayi = 10
      while(1) {
          printf("%d\n",sayi);
    if(sayi < 0) {
              break;
          }
          sayi--;
          continue;
          printf("%s\n","Bu satıra gelinmez.");
}
```

Yukarıdaki örnekte döngü koşulu sürekli olarak devam etmeye neden olur. Sayımız 0dan küçükse döngü **break** kullanarak sonlandırılır. Döngü içinde **continue** kısmına gelindiğinde başa dönüldüğü için bir alttaki satır çalıştırılmaz.

goto

C dilinde kodun içerisindeki bir yere etiket tanımlanıp **goto** ile bu etikete gidilebilir.

```
yaz:
printf("%s\n", "Hello World");
goto yaz;
```

Yukarıdaki örnekte sürekli olarak yazı yazdırılır. Bunun sebebi her seferinde **yaz** etiketine gidilmesidir.

Bundan faydalanarak döngü oluşturulabilir.

```
int i = 10;
islem:
if(i < 0){
    printf("%d\n",i);
    i--;
    goto islem;
}
```

Burada koşul bloğunun en sonunda tekrar başa dönmesi için **goto** kullandık.

Fonksiyonlar

C dilinde bir fonksiyon aşağıdaki gibi tanımlanır.

```
int yazdir(char* yazi){
    if(yazi != NULL){
        printf("%s\n",yazi);
        return 0;
    }
return 1;
}
```

Yukarıdaki fonksiyon verilen değişken değere sahipse ekrana yazdırıp 0 döndürür. Eğer değeri yoksa 1 döndürür.

Basit işlemler için **#define** ile de fonksiyon tanımlanabilir. Bu şekilde tanımlanan fonksiyonlar derleme öncesi yerine yazılarak çalışır.

```
#define topla(A,B) A+B

int main(int argc, char** argv){
    int sayi = topla(3, 5);
    return 0;
}
```

Fonksiyonlar yazılma sırasına göre kullanılabilirler. Bu yüzden fonksiyonlar henüz tanımlı değilse kullanılamazlar. Bu durumun üstesinden gelmek için **header** tanımlaması yapılır.

```
void yaz();
int main(){
    yaz();
    return 0;
}
void yaz(){
    printf("%s\n","Hello World");
}
```

Header tanımlamaları kütüphane yazarken de kullanılır. Bunun için bu tanımlamaları **.h** uzantılı dosyalara yazmanız gereklidir. Bu dosyayı **include** kullanarak eklemeliyiz.

yaz.h dosyası

```
void yaz();
```

main.c dosyası

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

int main(){
    yaz();
    return 0;
}

void yaz(){
    printf("%s\n","Hello World");
}
```

Not: include ifadesinde **<>** içine aldığımız dosyalar **/usr/include "**" içine aldığımız ise mevcut dizinde aranır.

Pointer ve Address kavramı

Pointerlar bir değişkenin bellekte bulunduğu yeri belirtir. ve * işareti ile belirtir. Örneğin aşağıda bir metin pointer olarak tanımlansın ve 2 birim kaydırılsın.

```
char* msg = "abcde";
printf("%s\n", msg + sizeof(char)*2 );
```

Bura 2 char uzunluğu kadar pointer kaydırıldığı için ekrana ilk iki karakteri silinerek yazdırılmıştır.

Adres ise bir değişkenin bellek adresini ifade eder. **&** işareti ile belirtilir. Örneğin rastgele bir değişken oluşturup adresini ekrana yazalım.

```
int i = 0;
printf("%p\n" &i);
```

Konunun daha iyi anlaşılması için bir değişken oluşturup adresini bir pointera kopyalayalım. ve sonra değişkenimizi değiştirelim.

```
int i = 0; // değişken tanımladık.
int *k = &i; // adresini kopyaladık.
int l = i; // değeri kopyaladık.
i = 1; // değişkeni değiştirdik.
printf("%d %d\n", i, *k, l);
```

Bu örnekte ilk iki değer de değişir fakat üçüncüsü değişmez. Bunun sebebi ikinci be birinci değişkenlerin adresi aynıyken üçüncü değişkenin adresi farklıdır.

Bir fonksiyon tanımlarken pointer olarak arguman aldırıp bu değerde değişiklik yapabilir. Buna örnek kullanım olarak **scanf** fonksiyonu verilebilir.

```
#include <stdio.h>
void topla(int* sonuc, int sayi1, int sayi2){
    *sonuc = sayi1 + sayi2;
}
void main(){
    int i;
    topla(&i, 12, 22);
    printf("%d\n",i);
}
```

Burada fonksiyona değişkenin adresi girilir. Fonksiyon bu adrese toplamı yazar. Daha sonra değişkenimizi kullanabilirsiniz.

Fonksiyonun kendisini de pointer olarak kullanmak mümkündür. Bunun için aşağıdaki gibi bir yapı kullanılabilir.

```
int topla(int i, int j){
    return i + j;
}

void main(){
    int (*topla_func)(int, int) = topla;
    topla_func(3, 5);
}
```

Ayrıca **typedef** yapısı ile de fonksiyon pointerları oluşturulabilir. Bu konunun detaylarına ilerleyen kısımlarda yer verilmiştir.

```
typedef int (*topla_func)(int, int);
int topla(int i, int j){
    return i + j;
}

void main(){
    topla_func topla_fn = topla;
    topla_fn(3, 5);
}
```

Dinamik bellek yönetimi

Dinamik bellek yönetimi için **malloc**, **realloc**, **calloc**, **free** fonksiyonları kullanılır. Bu fonksiyonlar **stdlib.h** ile sağlanmaktadır.

malloc fonksiyonu belirtilen boyut kadar boş alanı **void*** olarak tahsis eder.

```
// 10 elemanlı sayı dizisi oluşturmak için.
int *sayilar = (int*) malloc(10 * sizeof(int));
// şununla aynı anlama gelir.
int sayilar[10];
```

calloc fonksiyonu malloc ile benzerdir fakat istenen block boyutunu da belirterek kullanılır.

```
// 10 elemanlı sayı dizisi oluşturmak için
int *sayilar = (int*) calloc(10, sizeof(int));
// şununla aynı anlama gelir
int *sayilar = (int*) malloc(10 * sizeof(int));
```

realloc bir değişkenin yeniden boyutlandırılmasını sağlar.

```
// 5 elemanlı dizi tanımlayalım.
int sayilar[5];
// boyutu 10 yapalım
sayilar = (int*) realloc(sayilar, 10*sizeof(int));
```

free fonksiyonu değişkeni bellekten siler.

```
// malloc ile bir alan tanımlayalım.
void* alan = malloc(100);
// bu alanı silelim.
free(alan);
```

Konunun daha iyi anlaşılması için 2 stringi toplayan fonksiyon yazalım.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char* add(char *s1, char *s2){
    int ss = strlen(s1); // ilk arguman uzunluğu
    int sx = strlen(s2); // ikinci arguman uzunluğu
    char* s3 = (char*)malloc(ss+sx*sizeof(char)); // uzunluklar toplamı kadar alan ayır.
    for(int i=0;s1[i];i++) // ilkinin tüm elemanlarını kopyala
        s3[i] = s1[i];
    for(int i=0;s2[i];i++) // ikincinin tüm elemanlarını kopyala
        s3[i+ss] = s2[i];
    s3[ss+sx]='\0'; // stringler '\0' ile sonlanır
    return s3;
}

void main(){
    char *new_str = add( "hello", "world");
    printf("%s\n", new_str);
}
```

Struct

Structure yapıları bellekte belli bir değişken topluluğu oluşturup kullanabilmek için kullanılır. Bu yapılar sayesinde kendi veri türlerinizi tanımlayabilirsiniz.

```c
struct test {
    int num;
    char* name;
};

void main(){
    struct test t1;
    t1.num = 12;
    t1.name = "hello";
}
```

Veri türü adına alias tanımlamak için **typedef** kullanılabilir. Bu sayede değişken tanımlar gibi tanımlama yapmak mümkündür.

```c
typedef struct Test {
    int num;
    char* name;
} test;

void main(){
    test t1;
    t1.num = 12;
    t1.name = "hello";
}
```

typedef kullanarak struct dışında değişken türü tanımlamak da mümkündür.

```c
typedef char* my_string;

void main(){
    my_string str = "Hello World";
}
```

C programlama dili nesne yönelimli bir dil değildir. Bu yüzden sınıf kavramı bulunmaz. Fakat **struct** kullanarak benzer işler yapılabilir. Bunun için fonksiyon pointeri tanımlayıp struct yapımıza ekleyelim. Bir init fonksiyonu kullanarak nesnemizi oluşturalım.

```c
// nesne struct yapısı tanımladık
typedef struct Test {
    // nesne fonksiyonunu tanımladık.
    void (*yazdir)(char*);
    int num;
} test;

// nesne fonksiyon işlevin tanımladık
void test_yazdir(char* msg){
    printf("%s\n",msg);
}

// nesneyi oluşturan fonsiyonu tanımladık.
test test_init(){
    test t1;
    t1.num = 12;
    t1.yazdir = test_yazdir;
```

```
    return t1;
}

void main(){
    test obj = test_init();
    obj.yazdir("Hello World");
}
```

Kütüphane dosyası oluşturma

Kütüphaneler ana kaynak kodun kullandığı yardımcı kodları barındırır. Bu sayede her uygulama için tek tek aynı şeyleri yazmak yerine tek bir kütüphaneden yararlanılabilir.

GNU/Linux ortamında kütüphaneler **.so** uzantılıdır ve **/lib** ve **/usr/lib** dizinlerinde bulunur.

Not: Ek kütüphane dizinlerini **/etc/ld.so.conf** ve **/etc/lo.so.conf.d/*** dosyalarında belirlenir. Bunula birlikte **LD_LIBRARY_PATH** çevresel değişkeni ile kütüphane dizini tanımı yapılabilir.

Bir dosyanın bağımlı olduğu kütüphaneleri **ldd** komutu ile görüntüleyebiliriz.

```
$ ldd /bin/bash
    /lib/ld-musl-x86_64.so.1 (0x7fd299f6d000)
    libreadline.so.8 => /usr/lib/libreadline.so.8 (0x7fd299e5e000)
    libc.musl-x86_64.so.1 => /lib/ld-musl-x86_64.so.1 (0x7fd299f6d000)
    libncursesw.so.6 => /usr/lib/libncursesw.so.6 (0x7fd299e0a000)
```

Kendi kütüphanemizi olşturmak için kaynak kodumuzu **-shared** parametresi ile derlememiz gerekmektedir. Bunu için örneğin aşağıdaki gibi bir kaynak kodumuz olsun.

```
int topla (int a, int b) {
    return a+b;
}
```

Bu kodu derleyelim.

```
$ gcc -c test.c
$ gcc -o libtest.so test.o -shared
```

Şimdi de bu kütüphaneyi kullanabilmek için **test.h** dosyamızı oluşturalım.

```
int topla (int a, int b);
```

Son olarak kütüphaneyi kullanan kodumuzu yazalım.

```
#include <test.h>
void main(){
    int sayi = topla(3, 5);
}
```

Dikkat ettiyseniz **include** kullanırken **""** işareti yerine **<>** kullandık. Bunun sebebi kütüphanelerin kaynak koddan bağımsız olacak şekilde tasarlanmasıdır. Header dosyamızın **/usr/include** içinde ve kutuphanemizin de **/usr/lib** içinde olduğunu varsayarak kodladık.

Kütüphanemizin **kutuphane** adındaki bir dizinde bulunduğunu düşünelim ve aşağıdaki gibi derlemeyi tamamlayalım.

```
$ gcc -c main.c -I ./kutuphane
$ gcc -o main main.o -L ./kutuphane -ltest
```

Kodu kütüphaneyi sisteme yüklemeden derleyebilmemiz için derleyicimize **-I** parametresi eklenir. Bu parametre header aradığı dizinlere belirtilen dizini de ekler. Benzer şekilde derlemenin **linkleme** aşamasında **-l** parametresi ile hangi kütüphanelere ihtiyaç duyulduğu belirtilir. **-L** parametresi ile kütüphanenin aranacağı dizinler listesine belirtilen dizin eklenir.

Gördüğünüz gibi bu parametreler sisteme göre değişiklik gösterebilmektedir. Bu karmaşanın önüne geçebilmek için **pkg-config** kullanılır. Bu dosyada belirtilen değerler kütüphane ile beraber gelmekte olup derlemeye nelerin ekleneceğini belirtir.

Örnek olarak aşağıdaki gibi kullanabiliriz.

```
# derleme parametreleri
$ pkg-config --cflags readline
  -DNCURSES_WIDECHAR
# linkleme parametreleri
$ pkg-config --libs readline
  -lreadline
```

Kaynak kodu derlerken aşağıdaki gibi kullanılabilir.

```
$ gcc -c main.c `pkg-config --cflags readline`
$ gcc -o main main.o `pkg-config --libs readline`
```

pkg-config dosyaları **.pc** uzantılıdır ve **/usr/lib/pkgconfig** içinde bulunur. **pkg-config** dosyaları aşağıdaki formata benzer şekilde yazılır.

```
prefix=/usr
includedir=${prefix}/include

Name: Test
Description: Test library
Version: 1.0
Requires: readline
Cflags: -I{includedir}/test
Libs: -ltest -L{libdir}/test
```

Yukarıdaki örnekte **/usr/include/test/** içerisindeki header dosyamızı ve **/usr/lib/test/** içindeki kütüphane dosyamızı sorunsuzca kullanarak derleme yapabilik.

makefile dersi

makefile formatı yazılan bir kaynak kodu derlemek ve yüklemek için kullanılan ne yaygın derleme talimatı formatlarından biridir.

Bu yazıda sizlere makefile dosyası nasıl yazılır anlatacağım.

Genel bakış

Örneğin aşağıdaki gibi bir **C** kodumuz olsun

```
#include <stdio.h>
int main(){
```

```
    puts("Hello world!\n");
    return 0;
}
```

Bunu aşağıdaki komutu kulanarak derleriz ve kurarız.

```
$ gcc -o hello hello.c
$ install hello /usr/bin/hello
```

Makefile dosyalarının bölüm tanımlamalarında girintileme amaçlı **Tab** kullanılır.

Şimdi aşağıdaki makefile dosyasını inceleyelim.

```
PREFIX=/usr
build:
        $(CC) -o hello hello.c

install:
        install hello $(DESTDIR)/$(PREFIX)/bin/hello
```

Burada **PREFIX**, **CC**, **DESTDIR** gibi parametreler değişkendir. Bu değişkenler derleme esnasında değiştirilebilir.

Bu makefile dosyasını kullanarak derlemeyi ve yüklemeyi aşağıdaki gibi yaparız.

```
$ make
$ make install
```

Görüldüğü gibi derleme ve yükleme işlemi daha kolay ve nasıl derleneceğini basitçe belirtmiş olduk.

Burada kullandığımız değişkenler şu şekilde açılnabilir.

- PREFIX = /usr olarak tanımladık.
- DESTDIR = paket sistemleri paket yaparken bu değişkeni otomatik olarak değiştirir. Kurulacak kök dizin konumudur.
- CC = derleyicinin adıdır. Bu değişkeni ayarlayarak derleyiciyi değiştirebilirsiniz.

Make komutuna eğer hiç parametre verimezsek ilk baştaki bölümü çalıştırır. Biz ilk başta **build** tanımladığımız için make komutu build çalıştırır. make komutuna parametre olarak bölüm verirsek o bölüm çalıştırılır.

Değişken işlemleri

Değişken tanımlamak için **variable=value** şeklinde tanımlanabilir. değişkeni kullanırken de **$()** işareti arasına alınır. Örneğin:

```
yazi=hello world
hello:
        echo $(yazi)
```

Bu değişkeni **make yazi=deneme123** şeklinde komut vererek değiştirebiliriz.

Var olan bir değişkene ekleme yapmak için **+=** ifadesi kullanılır. **:=** ifadesi eğer tanımlama varsa ekleme yapar. **?=** sadece daha önceden tanımlanmışsa ekleme yapar.

```
yazi=hello
yazi+=world
sayi:=$(shell ls)
hello:
	echo $(yazi)
```

Eğer **$** işareti kullanmanız gereken bir durum oluşursa **$$** ifadesi kullanabilirsiniz. Örneğin:

```
hello:
	bash -c "echo $$HOME"
```

Bölümler

Makefile yazarken bölümler tanımlanır ve eğer bölümün adı belirtilmemişse ilk bölüm çalıştırılır. Bölümler arası bağımlılık vermek için aşağıdaki gibi bir kullanım yapılmalıdır.

```
yazi: sayi test
	echo "Hello world"
sayi:
	echo 12
test:
	echo test123
```

Yukarıdaki dosyayı çalıştırdığımızda sırasıyla **sayi** -> **test** -> **yazi** bölümleri çalıştırılır.

Aynı işi yapan birden çok bölüm şu şekilde tanımlanabilir.

```
bol1 bol2:
	echo Merhaba
# Şuna eşittir.
bol1:
	echo Merhaba
bol2:
	echo Merhaba
```

Bölümün adını **$@** kullanarak öğrenebiliriz.

```
bolum:
	echo $@
```

Bölümün tüm bağımlılıklarını almak için için **$^** kullanabiliriz.

```
bolum: bol1 bol2
	echo $^
bol1 bol2:
	true
```

$?** ifadesi **$^ ile benzerdir fakat sadece geçerli bölümden sonra tanımlanan bölümleri döndürür.

```
bol1:
	true
bolum: bol1 bol2
```

```
        echo $?
bol2:
        true
```

$< ifadesi sadece ilk bağımlılığı almak için kullanılır.

```
bol1 bol2:
        true
bolum: bol1 bol2
        echo $<
```

Eğer **xxxx.o** şeklinde bir kural tanımlarsanız bu kural çalıştırıldıktan sonra gcc ile kural adındaki dosya derlenir.

```
main: main.o
main.o: main.c test.c

main.c:
        echo "int main(){}" > main.c
%.c:
        touch $@
```

Burada main.c dosyası var olmayan bir dosyadır ve derleme esnasında oluşturulur. test.c dosyası ise daha önceden var olan bir dosyadır ve o dosyaya bir şey yapılmaz. main.c kuralı sadece main.c için çalıştırılırken **%.c** şeklinde belirtilen kular hem main.c hem test.c için çalıştırılır. **main** ile belirttiğimiz kuralda main.o bağımlılığı olduğu için bi derlemenin sonucu olarak main adında bir derlenmiş dosya üretilmektedir.

wildcard ve shell

Wildcard ifadesi eşleşen dosyaları döndürür.

```
files := $(wildcard *.c)
main:
        gcc -o main $(files)
```

Shell ifadesi ise komut çalıştırarak sonucunu döndürür.

```
files := $(shell find -type f -iname "*.c")
main:
        gcc -o main $(files)
```

Birden çok dosya ile çalışma

make -C xxx şeklinde alt dizindeki bir makefile dosyasını çalıştırabilirsiniz.

```
build:
        make -C src
```

Ayrıca **include** kullanarak başka bir dosyada bulunan kuralları kullanabilirsiniz.

```
# Makefile dosyası
include build.mk
build: main
        gcc main.c -o main
# build.mk dosyası
main:
        echo "int main(){return 0;}" > main.c
```

Koşullar

ifeq ifadesi ile koşul tanımlanabilir. aşağıdaki ifadeşi **make CC=clang** şeklinde çalıştırırkanız clang yazdırır, parametresiz bir şekilde çalıştırırsanız gcc yazdırır. Burada dikkat edilmesi gereken konu **ifeq**, **else**, **endif** girintilenmeden yazılır.

```
build:
ifeq ($(CC),clang)
        echo "clang"
else
        echo "gcc"
endif
```

Komut özellik ifadeleri

Eğer komutun başına **@** işareti koyarsanız komut ekrana yazılmadan çalıştırılır. **-** yazarsanız komut hata alsa bile geri kalan kısımlar çalışmaya devam eder.

```
build:
        @echo "Merhaba dünya"
        -gcc main.c -o main
```

while ve for kullanımı

Bash betiklerinde kullandığımız for ve while yapısı makefile yazarken aşağıdaki gibi kullanılabilir. done dışındaki satırların sonuna **** işareti eklenir, do dışındaki satırların sonuna da **;** işareti koyulur.

```
build:
        @for sayi in 1 2 3 $(dizi) ; do \
            echo $$sayi ; \
            echo "diger satir" ; \
        done
```

SHELL değişkeni

SHELL değişkeni makefile altındaki komutların hangi shell kullanılarak çalıştırılacağını belirtir. Varsayılan değeri **/bin/sh** olarak belirlenmiştir. Örneğin debian tabanlı dağıtımlarda /bin/sh konumu /bin/dash bağlıyken archlinuxta /bin/bash bağlıdır. **dash [[** kullanımını desteklemezken **bash** destekler. Bu sebeple uyumluluğu arttırmak için **SHELL** değişkenini zorla /bin/bash olarak değiştirebiliriz. Aşağıdaki örnekle konuyu daha iyi anlamak için SHELL değişkenini python3 ayarladık ve python kodu yazdık.

```
SHELL=/usr/bin/python3
build:
```

```
        import os ;\
        liste = os.listdir("/") ;\
        print(liste[0])
```

Python dersi

Bu yazıda python programlama dilini hızlıca anlatacağım. Bu yazıda karıştırılmaması için girdilerin olduğu satırlar <- ile çıktıların olduğu satırlar -> ile işaretlenmiştir.

Açıklama satırları

Python programlama dilinde açıklamalar **#** işaretinden sonrası için geçerlidir. Örneğin:

```
#bu bir açıklama satırıdır.
```

Bununla birlikte çoklu açıklama satırı yapmak için **"""** işareti arasına alabiliriz.

```
""" Bu bir açıklama satırı
Bu diğer açıklama satırı
Bu son açıklama satırı """
```

Temel bilgiler

Python programlarının ilk satırında **#!/usr/bin/python3** satırı bulunmalıdır. Bir python programını çalıştırmak için Şunları uygulamamız gereklidir.

```
#!/usr/bin/python3
# Çalıştırma izni vererek çalıştırabiliriz.
chmod +x dosya.py
./dosya.py
# Veya doğrudan çalıştırabiliriz.
python3 dosya.py
```

Python programlama dilinde satır sonuna **;** koyulmaz.

Python programlarında işler işlevler üzerinden yürür. işlevlerin girdileri ve çıktıları bulunur.

```
cikti = islev(girdi1, girdi2)
```

Pythonda girintileme olayı için de **{** ve **}** kullanılmak yerine boşluklandırma kullanılır. Herhangi bir girintilemeye başlanan satırın sonunda **:** işareti bulunur. Örneğin:

```
f = 12 # f sayısını 12ye eşitledik
if f == 12: # f sayısı 12ye eşit mi diye sorguladık
    print("eşit") # ekrana yazı yazdırık
```

Girintileme için 4 boşluk, 2 boşluk veya tek tab kullanabilirsiniz. Bu yazıda 4 boşluğu tercih edeceğiz.

Yazı yazdırma

Pythonda ekrana yazı yazmak için **print** işlevini kullanıyoruz.

```
print("Merhaba Dünya")
-> Merhaba Dünya
```

Birden çok ifadeyi yazdırmak için **print** işlevine birden çok girdi verebilirsiniz. Bu şekilde aralarına birer boşluk koyarak yazdırır.

```
# Yazılar tırnak içine alınır.
# Sayılar tırnak içine alınmaz.
# True ve False doğruluk belirtir.
print("Merhaba",12,"Dünya",True)
-> Merhaba 12 Dünya True
```

Değişkenler

Değişkenler içerisinde veri bulunduran ve ihtiyaç durumunda bu veriyi düzenleme imkanı tanıyan kavramlardır. Değişkenler tanımlanırken **degisken = deger** şeklinde bir ifade kulanılır.

```
i = 12
yazi = "merhaba dünya"
k = 1.2
hmm = True
```

Değişken adları sayı ile başlayamaz, Türkçe karakter içeremez ve sadece harfler, sayılar ve **-** **_** karakterlerinden oluşur.

Değişkenler kullanılırken başına herhangi bir işaret almasına gerek yoktur. Örneğin:

```
i = 12
print(i)
-> 12
```

Değişkenler tanımlanırken her ne kadar türlerini belirtmesek bile birer türe sahip olarak tanımlanır. Bunlar başlıca **integer**, **float**, **string**, **boolean** türleridir.

Bir değişkenin türünü öğrenmek için **type** işlevini kullanabiliriz.

```
veri = "abc123"
turu = type(veri)
print(turu)
-> <class 'str'>
```

Boş bir değişken tanımlamak için onun değerine **None** atayabiliriz. Bu sayede değişken tanımlanmış fakar değeri atanmamış olur.

```
veri = None
tur = type(veri)
print(tur)
-> <class 'NoneType'>
```

String

String türünden değişkenler yazı içerir. " veya ' veya """ arasına yazılırarak tanımlanır.

```
yazi1 = "merhaba"
yazi2 = 'yazım'
yazi3 = """dünya"""
```

String türünden değişkenler **+** işareti ile uc uca eklenebilir.

```
yazi = "merhaba" + ' ' + """dünya"""
print(yazi)
-> merhaba dünya
```

Değişkeni birden çok kez toplamak için * işareti kullanılabilir.

```
yazi = "ali"*5
print(yazi)
-> alialialialiali
```

String türünden bir değişkenin içerisindeki bir bölümü başka bir şey ile değiştirmek için **replace** işlevi kullanılınabilir.

```
veri = "Merhaba"
veri2 = veri.replace("rha","123")
print(veri2)
-> Me123ba
```

Integer

Integer türünden değişkenler tam sayı belirtir. Dört işlem işaretleri ile işleme sokulabilirler.

```
sayi = (((24/2)+4)*2)-1
"""
24/2 = 12
12+4 = 16
16*2 = 32
32-1 = 31
"""
print(sayi)
-> 31
```

Integer değişkenlerin kuvvetlerini almak için ** kullanılır.

```
sayi = 2**3
print(sayi)
-> 8
```

String türünden bir değişkeni integer haline getirmek için **int** işlevi kullanılır.

```
print(int("12")/2)
-> 6
```

Float

Float türünden değişkenler virgüllü sayılardır. Aynı integer sayılar gibi dört işleme sokulabilirler. İki integer değişkenin birbirine bölümü ile float oluşabilir.

```
sayi = 1/2 # sayi = 0.5 şeklinde de tanımlanabilir.
print(sayi)
-> 0.5
```

Bir float değişkenini integer haline getirmek için **int** işlevi kullanılır. Bu dönüşümde virgülden sonraki kısım atılır.

```
sayi = 3.2
print(sayi)
sayi2 = int(3.2)
print(sayi2)
-> 3.2
-> 3
```

Not: float ile string çarpılamaz.

String türünden bir değişkeni float haline getirmek için **float** işlevi kullanılır.

```
print(float("2.2")/2)
-> 1.1
```

Boolean

Boolean değişkenler sadece **True** veya **False** değerlerini alabilir. Bu değişken daha çok koşullarda ve döngülerde kullanılır. iki değişkenin eşitliği sorgulanarak boolean üretilebilir.

```
bool = 12 == 13
"""
== eşit
!= eşit değil
<  küçük
>  büyük
<= küçük eşit
>= büyük eşit
"""
print(bool)
-> False
```

boolean değişkeninin tersini almak için **not** ifadesi kullanılabilir.

```
veri = not True
print(veri)
-> False
```

Bir string türünden değişkenin içinde başka bir string türünden değişken var mı diye kontrol etmek için **in** ifadesi kullanılır. Bu ifadenin sonucu boolean üretir.

```
veri = "ef" in "Dünya"
veri2 = "ny" in "Dünya"
print(veri,veri2)
-> False True
```

Boolean değişkenlerde mantıksal işlemler **and** ve **or** ifadeleri ile yapılır.

```
veri = 12 < 6 or 4 > 2 # False or True = True
print(veri)
-> True
```

Klavyeden değer alma

Python programlarının kullanıcı ile etkileşime girmesi için klavye üzerinden kullanıcıdan değer alması gerekebilir. Bunun için **input** işlevi kullanılır. Bu işlevin çıkışı string türündendir.

```
a = input("Bir değer girin >")
print(a,type(a))
<- 12
-> 12 <class 'str'>
```

String türünden bir ifadeyi bir değişken üretmek için kullanmak istiyorsak **eval** işlevini kullanabiliriz.

```
a = eval("12/2 == 16-10") # string ifade çalıştırılır ve sonucu aktarılır.
print(a)
-> True
```

Not: Bu işlev tehlikelidir. Potansiyel güvenlik açığına neden olabilir! Mümkün olduğu kadar kullanmayın :D

Koşullar

Koşul tanımı yapmak için **if** ifadesi kullanılır. Koşul sağlanmıyorsa **elif** ifadesi ile yeni bir koşul tanımlanabilir veya **else** ifadesi ile koşulun sağlanmadığı durum tanımlanabilir.

```
if koşul:
    eylem
elif koşul:
    eylem
else:
    eylem
```

Örneğin bir integer değişkenin çift olup olmadığını bulalım.

```
if 13 % 2 == 0 : # % işareti bölümden kalanı bulmaya yarar.
    print("Çift sayı")
else:
    print("Tek sayı")
```

Değeri olmayan (None) değişkenler koşul ifadelerinde **False** olarak kabul edilir.

```
veri = None
if veri:
    print("Tanımlı")
else:
    print("Tanımsız")
-> Tanımsız
```

Koşul tanımlamayı alternatif olarak şu şekilde de yapabiliriz:

```
koşul and eylem
""" Şununla aynıdır:
if koşul:
    eylem
"""
koşul or eylem
""" Şununla aynıdır:
if not koşul:
    eylem
"""
```

Bu konunun daha iyi anlaşılması için:

```
12 == 12 and print("eşittir")
12 == 14 or print("eşit değildir")
-> eşittir
-> eşit değildir
```

Diziler

Diziler birden çok elemanı içerebilen değişkenlerdir. Diziler aşağıdaki gibi tanımlanır:

```
a = [1, 3, "merhaba", True, 1.2, None]
```

Dizilerin elemanlarının türü aynı olmak zorunda değildir. Hatta None bile olabilir.

Dizilerde eleman eklemek için **append** veya **insert** işlevini eleman silmek için ise **remove** veya **pop** işlevi kullanılır. Örneğin:

```
a = [22]
print(a)
a.append("Merhaba") # Sona ekleme yapar.
a.insert(0,12) # 0 elemanın ekleneceği yeri ifade eder.
print(a)
a.remove(22) # 22 elemanını siler
print(a)
a.pop(0) # 0. elemanı siler.
print(a)
-> [22]
-> [12, 22, 'Merhaba']
-> [12, 'Merhaba']
-> ['Merhaba']
```

Dizileri sıralamak için **sort** boşaltmak için ise **clear** işlevi kullanılır. Bir dizinin istenilen elemanını öğrenmek için **liste[index]** şeklinde bir ifade kullanılır. Index numaraları 0dan başyan integer olmalıdır. negatif değerlerde sondan saymaya başlar.

```
a = [1, 3, 6, 4, 7, 9, 2]
print(a[2],a[-3])
a.sort()
print(a)
a.clear()
print(a)
-> 3 7
```

```
-> [1, 2, 3, 4, 6, 7, 9]
-> []
```

Dizideki bir elemanın uzunluğunu bulmak için **len** işlevi, elemanın dizinin kaçıncısı olduğunu bulmak için ise **index** işlevi kullanılır.

```
a = [12, "hmm", 3.2]
sayi = len(a)
sayi2 = a.index(3.2)
print(sayi,sayi2)
-> 3
-> 2
```

Dizilerin elemanlarını **+** kullanarak birleştirebiliriz.

```
a = [1, 2, 3]
b = [4, 5, 6]
c = a + b
print(c)
-> [1, 2, 3, 4, 5, 6]
```

Dizilerin bir bölümünü aşağıdakine benzer yolla kesebiliriz:

```
a = [1, 3, 5, 7, 9, 12, 44, 31, 16]
b = a[:2] # baştan 3. elemana kadar.
c = a[4:] # 4. elemandan sonrası
d = a[3:6] # 4. elemandan 6. elemana kadar (dahil)
```

String türünden bir değişkeni belli bir harf veya harf öbeğine göre bölmek için **split** işlevini kullanırız. Ayrıca string türünden bir değişkenin başındaki ve sonundaki boşlukları temizlemek için **strip** işlevini kullanırız.

```
veri="   Bu bir yazıdır   "
veri2 = veri.strip()
print(len(veri),len(veri2))
liste = veri2.split(" ")
print(liste)
-> 20 14
-> ['Bu', 'bir', 'yazıdır']
```

While döngüsü

Döngüler belli bir işi koşul bağlanana kadar tekrar etmeye yarayan işlevdir. Kısaca **while** döngüsü ile **if** arasındaki fark **while** içerisindeki durum tamamlandığı zaman tekrar başa dönüp koşulu kontrol eder.

```
while koşul:
    eylem
```

Örneğin 1den 10a kadar olan sayıları yazalım. Bu durumda *i* sayısı 10 olana kadar sürekli olarak ekrana yazılıp değeri 1 arttırılacakdır.

```
i = 1
while i < 10:
    print(i)
    i+=1 # i = i + 1 ile aynı anlama gelir.
-> 1 2 3 4 5 6 7 8 9 (Bunu alt alta yazdığını hayal edin :D )
```

Bir döngüden çıkmak için **break** ifadesi kullanılır. Döngünün o andi adımını tamamlayıp diğer adıma geçmek için ise **continue** ifadesi kullanılır.

Örneğin aşyağıda siz çift sayı girene kadar sürekli olarak çalışan bir program yazalım.

```
while True:
    sayi = int(input("Sayı girin"))
    if sayi % 2 == 0:
        break
```

For döngüsü

For döngüsü while ile benzerdir fakat koşul aranmak yerine iteration yapar. Bu işlemde bir dizinin bütün elemanları tek tek işleme koyulur. Aşağıdaki gibi bir kullanımı vardır:

```
for eleman in dizi:
    eylem
# Şununla aynıdır
i = 0
toplam = len(dizi)
while i < toplam: # eleman yerine dizi[i] kullanabilirsiniz.
    eylem
    i += 1
```

Örneğin bir integer değişkenlerden oluşan dizi oluşturalım ve elemanlarını 2ye bölerek ayrı bir diziye ekleyelim.

```
a = [2, 4, 6, 8, 10] # dizi tanımladık
b = [] # diğer diziyi tanımladık
for i in a: # a elemanları i içine atılacak
    b.append(i/2) # b içine elemanın yarısını ekledik.
print(b)
-> [1, 2, 3, 4, 5]
```

Eğer dizi yerine string türünden bir değişken verirsek elemanlar bu stringin harfleri olacaktır. Aşağıdaki örnekte string içerisinde kaç tane a veya e harfi bulunduğunu hesaplayalım.

```
veri = "Merhaba Dünya"
toplam = 0
for i in veri:
    if i == "a" or i == "e":
        toplam += 1
print(toplam)
-> 4
```

Şimdiye kadarki anlatılanların daha iyi anlaşılması için asal sayı hesaplayan bir python kodu yazalım:

```
asallar = [2] # ilk asal sayıyı elle yazdık.
i = 3 # Şu anki sayı
while i < 60: # 60a kadar say
    hmm = True # asal sayı mı diye bakılan değişken
    for e in asallar: # asal sayılar listesi elemanları
        if i % e == 0: # tam bölünüyor mu
            hmm = False # asal sayı değildir
            break # for döngüsünden çıkmak için
    if hmm: # Asal sayıysa diziye ekleyelim
        asallar.append(i)
    i += 1 # mevcut sayımızı 1 arttıralım.
print(asallar) # 60a kadar olan asal sayılar dizisini yazalım.
-> [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59]
```

İşlevler

Python programlarken işlev tanımlayıp daha sonra bu işlevi kullanabiliriz. İşlevler aşağıdaki gibi tanımlanırlar:

```
def islev(girdi1,girdi2):
    eylem
```

işlevlerde çıktı sonucu olarak bir değişken döndürmek için **return** ifadesi kullanılır. Örneğin girdideki sayıları toplayan işlev yazalım.

```
def topla(sayi1,sayi2):
    return sayi1 + sayi2
    print("Merhaba") # bu satır çalıştırılmaz
toplam = topla(3,5)
print(toplam)
-> 8
```

Eğer bir değişken sadece işlevin içerisinde tanımlanırsa o değişken işlevin dışında tanımsız olur.

```
def yazdir():
    yazi = "Merhaba"
    print(yazi)
yazdir()
print(yazi)
-> Merhaba
-> Traceback (most recent call last):
->   File "ders.py", line 5, in <module>
->     print(yazi)
-> NameError: name 'yazi' is not defined
```

Bir işlevin ne işe yaradığını öğrenmek için **help** işlevi kullanılır. işlevin ne işe yaradığını tanımlamak için ise ilk satıra """ içerisine yazabiliriz. Bunu tanımlamak programınızı inceleyen diğer insanlar için yararlı olacaktır.

```
def abc(sayi):
    """Girilen sayıyı 10dan çıkartır"""
    return 10-sayi
help(abc)
```

```
->  Help on function abc in module __main__:
->
->  abc(sayi)
->      Girilen sayıyı 10dan çıkartır
```

Bir işlevin birden çok çıktısı olabilir. Bunun için **return** ifadesini virgülle ayrılmış olarak birden çok değişken ile kullanmalıyız.

```
def yer_degistir(a,b):
    """Girilen değişkenlerin yerini değiştirir"""
    return b,a
c = 12
d = 31
c,d = yer_degistir(c,d)
# Bunun yerine doğrudan c,d = d,c kullanılabilirdi.
print(c,d)
-> 31 12
```

Konunun daha iyi anlaşılabilir olması için girilen dizinin sayılarının ortalamasını alan bir fonksiyon yazalım.

```
def ortalama(dizi):
    toplam = 0 # toplam değişkeni tanımladık.
    for eleman in dizi: # for döngüsü oluşturduk.
        toplam += int(eleman) # elemanları topladık.
    return toplam / len(dizi) # toplamı eleman sayısına böldük.

ort = input("Dizi giriniz. aralarına , koyunuz")
print(ortalama(ort.split(",")))
<- 1,34,22,-32
-> 6.25
```

Sınıflar

Sınıf kavramı işlevlerin ve değişkenlerin guruplanarak nesneler haline getirilmesinden meydana gelir. Yani bir sınıf ona bağlı işlevlerden ve değişkenlerden oluşur. Sınıflar aşağıdaki gibi tanımlanırlar.

```
class sinif:
    def __init__(self,girdi):
        eylem
    def islev(self,girdi):
        eylem
```

Burada **__init__** işlevi sınıfı oluştururken çalıştırılan ilk eylemleri tanımlamak için kullanılır. sınıf işlevleri tanımlanırken ilk girdi olarak **self** kullanılmalıdır. Bu ifade sınıfın kendisi anlamına gelir. Örneğin bir sınıf tanımlayalım ve bu sınıftaki işlevler ile girdideki sayılara toplama ve çıkartma işlemi uygulayalım.

```
class sayi_isle:
    def __init__(self,ilk,ikinci):
        self.sayi1 = ilk
        self.sayi2 = ikinci
```

```
    def topla():
        return self.sayi1 + self.sayi2
    def cikart():
        return self.sayi1 - self.sayi2

nesne = sayi_isle(12,3)
a = nesne.topla()
b = nesne.cikart()
print(a,b)
-> 15 9
```

Burada işlevlere **nesne.islev()** ifadesi ile erişebiliyoruz. Aynı zamanda değişkenlere de **nesne.degisken** ifadesi ile erişmemiz ve değiştirmemiz mümkündür. Sınıf içerisinden ise **self.islev()** ve **self.degisken** şeklinde bir ifade kullanmamız gerekmektedir.

Dosya işlemleri

Bir dosyayı açmak için **open** işlevi kullanılır. Açılan dosyadan satır okumak için **readline** işlevi, tamamını okumak için **read** işlevi, tüm satırları okuyup dizi haline getirmek için ise **readlines** işlevi kullanılır.

deneme.txt adında içeriği aşağıdaki gibi olan bir doysamız olsun:

```
Merhaba dünya
Selam dünya
sayı:123
```

Aşağıdaki örnekte bu dosyayı açıp okuyup ekrana basalım.

```
dosya = open("dosya.txt","r") # okumak için r kullanılır.
ilksatir = dosya.readline()
tumu = dosya.read()
satirlar = dosya.readlines()
print(len(satirlar))
-> 3
```

Dosyaya yazmak için ise **write** işlevi kullanılır. Okuma ve yazma işlemleri bittikten sonra **close** işlevi ile dosya kapatılmalıdır. Dosyayı kapatmadan değişiklikleri diske işletmek için **flush** işlevi kullanılır.

```
dosya = open("dosya.txt","w") # yazmak için w eklemek için a kullanılır.
dosya.write("Merhaba dünya\n")
dosya.write("Selam dünya\n")
dosya.write("sayı:123\n")
dosya.close()
```

Modüller

Python programlarında kodların markaşıklaşmasını önlemek ve daha kullanışlı hale getirmek amacıyla modüller bulunur. Modüller **import** ifadesi ile çağırılır. Modüller aslında birer Birer python kütüphanesidir ve sınıf sayılırlar. Örneğin deneme.py dosyamızda aşağıdaki kodlar bulunsun:

```
yazi = "Merhaba"
sayi = 12
def yazdir():
    print(yazi)
class sinif:
    def islev(self):
        print("selam")
```

Şimdi bu modülümüzü çağırıp içerisindeki işlevleri ve değişkenleri kullanalım.

```
import deneme # deneme modülünü çağırdık
print(deneme.yazi) # değişkeni kullandık
deneme.yazdir() # işlevi kullandık.
deneme.sayi = 76 # değişkeni değiştirdik
nesne = deneme.sinif() # sınıftan nesne oluşturduk
nesne.islev() # nesneyi kullandık
-> Merhaba
-> selam
```

Şimdiye kadar anlatılanların daha iyi anlaşılması için aşağıda ini parser örneği yapalım. Örnek bir ini dosyası aşağıdaki gibidir:

```
[bölüm1]
veri1=deger1
veri2=deger2
[bölüm2]
veri3=deger3
veri4=deger4
```

Bir adet modül yazıp bu modül ile ini dosyası okuyup istenilen bölümdeki değeri bulalım ve döndürelim.

```
[Merhaba]
dünya=12
selam=44
[hmm]
veri=abc123x
sayı=44
```

```
# iniparser.py içeriği
dosya = None # boş dosya nesnesi
class inidosya:
    def __init__(self,yol):
        ini = open(yol,"r") # ini dosyasını açtık
        self.icerik = ini.read() # dosya içeriğini okuduk
    def deger_al(bolum,veri):
        etkin = False # istenilen yere gelene kadar etkisiz kal
        for satir in dosya.icerik.split("\n"):
            if "[" + bolum + "]" in satir: # okunan satırda istenilen bölümün başı mı
                etkin = True # etkinleştir
            if etkin and "=" in satir: # etkinse ve satırda = bulunuyorsa
                if satir.split("=")[0] == veri: # = işaretine göre 0. eleman aranan mı
                    return satir.split("=")[0] # = işaretine göre böl . elemanı al
```

```
# main.py dosyası içeriği
import iniparser # modülü yükle
iniparser.dosya = iniparser.inidosya("dosya") # ini dosyasını yükle
deger=iniparser.deger_al("hmm","veri") # değeri al
print(deger.strip()) # değerin başında sonunda boşluk varsa sil ve yaz
-> abc123x
```

Bir modülü diğer bir modülün genişletilmişi olarak tanımlayabiliriz. Gelişletilen modül asıl modüldeki tüm fonksiyonlara ve değişkenlere sahip olur. Aşağıdaki örnekteki gibi **super().__init__()** kullanarak üst modülümüzdeki tüm tanımlamalara sahip olmasını sağlayabiliriz.

```
# ornek.py dosyası
class deneme:
    def __init__(self):
        self.sayi = 13
    def hmm(self,yazi):
        print(yazi)

class genis(deneme):
    def __init__(self):
        super().__init__()
        self.sayi2 = 44
```

```
# main.py dosyası
from ornek import genis as g # ornek.py dosyasındaki genis sınıfını g olarak içeri aldık.
print(g.sayi, g.sayi2)
g.hmm("abc123")
-> 13 44
-> abc123
```

Vala dersi

Bu yazıda vala programlama dilini anlatacağım. Örneklende **...** bulunan satırlar üstünde ve altında başka kodların bulunabileceğini belirtir. Çalıştırılan komutların başında **$** işareti kullanılmıştır.

Vala programlama dili derlemeli bir dil olup yazmış olduğunuz kod valanın çeviricisi yardımı ile önce **C** koduna çevrilir ve ardından **gcc** veya **clang** gibi bir derleyici kullanılarak derlenir. Vala kaynak kodunu derlemek için **valac** kullanılır. Örneğin aşağıda basit bir derleme örneği verilmiştir.

```
$ valac main.vala
```

Vala programlarını main fonksiyonu ile başlar. Bu fonksiyon aşağıdaki gibi yazılır:

```
int main(string[] args){
   ...
   return 0;
}
```

Burada **int** fonksiyonun çıktı türünü **main** adını **string[] args** ise parametresini ifade eder. **return 0** ise çıkış kodunu bildirir. Vala programlama dilinde her satırın sonunda **;** bulunmak zorundadır.

Girintileme

Vala girintilemeye dikkat eden bir dil değildir fakat kodun okunaklı olması açısından girintilemeye dikkat etmenizi öneririm. Girintileme ile ilgili aşağıdaki kuralları takip edebilirsiniz. Bu kurallar zorunluluk oluşturmadığı gibi kendinize göre farklılaştırabilirziniz.

- Koşullar döngüler ve fonksiyonlarda 4 boşluk veya tek tab ile girintileyin.
- Virgülden sonra bir boşluk bırakın.
- Koşullar döngüler ve fonksiyonlarda **{** işaretini satır sonuna koyun.
- **else** ve **else if** ifadelerinde **}** ve **{** işaretini satırın başında ve sonunda kullanın.
- İşleyicilerin başına ve sonuna birer boşluk ekleyin.

Örneğin bu kurala uygun girintilenmiş bir kod:

```
int main(string[] args){
    int i = int.parse(stdin.read_line());
    int[] liste = {12, 22, 55, 27};
    if(i == 12){
        i = 33 / 3;
    }else if(i >= 44){
        i = 0;
    }
    return 0;
}
```

Burada da aynı kodun kötü girintilenmiş hali bulunmaktadır.

```
int main(    string[] args ){
int i=int.parse(  stdin.read_line() );
int[] liste={12,22,55,27};
if (i== 2)
  {i=33/3;}
    else if    (i>=44){i=0;
        }return 0;
            }
```

Her ikisi de aslında tamamen aynı kod fakat ilk örnek daha okunaklı ve düzenli gözükmektedir.

Açıklama satırı

Açıklamalar 2 şekilde yapılır. Açıklama bölümleri derleme esnasında dikkate alınmaz.

- \ ifadesinden sonra satır sonuna kadar olan kısım açıklamadır.
- * ile başlayıp *\ ile biten yazılar açıklamadır.

```
// bu bir açıklamadır
/* bu bir açıklamadır */
```

Ekrana yazı yazdırma

Ekrana yazı yazmak için **printf** kullanılır. normal çıktı için **stdout.printf**, hata çıktısı için ise **stderr.printf** kullanılır.

```
int main(string[] args){
    stdout.printf("Merhaba Dünya");
    return 0;
}
```

Değişken türleri

Değişkenler türleri ile beraber tanımlanırlar veya **var** ifadesi kullanılarak tanımlanırlar.

```
...
int num = 0;
val text = "Hello world";
string abc = "fff";
...
```

Bununla birlikte değişkenlere başta değer atamayıp sonradan da değer atama işlemi yapılabilir.

```
...
int num;
num = 31;
...
```

Değişkenler arası tür dönüşümü işlemi için **parse** ve **to_string** kulanılır.

```
...
int num = 15;
string txt = num.to_string(); // int -> string dönüşümü
int ff = int.parse("23"); // string -> int dönüşümü
...
```

Başlıca veri türleri şunlardır:

- **int** tam sayıları tutar
- **char** tek bir karakter tutar.
- **float** virgüllü sayıları tutar.x
- **double** büyük bellek boyutu gerektiren sayıları tutar.
- **bool** doğru veya yanlış olma durumu tutar.
- **string** yazı tutar.

Diziler

Diziler birden çok eleman tutan değişkenlerdir. tanımlanırken **xxx[] yy** şeklinde tanımlanırlar.

```
...
int[] nums = {12,22,45,31,48};
stdout.printf(num[3].to_string()); // Ekrana 31 yazar.
...
```

Yukarıda **{}** kullanılarak dizi elemanları ile beraber tanımlanmıştır. Bir altındaki satırda ise dizinin 4. elemanı çekilmiştir ve string türüne çevirilip ekrana yazılmıştır. Burada 3 olarak çekilme sebebi dizilerin eleman sayılarının 0dan başlamasıdır.

Diziye aşağıdaki gibi eleman ekleyebiliriz.

```
...
    int nums = {14,44,12};
    nums += 98;
...
```

Dizinin boyutunu aşağıdaki gibi öğrenebiliriz.

```
...
    string[] msgs = {"Hello", "World"};
    int ff = msgs.length;
...
```

Vala programlama dilinde diziler basit işler için yeterli olsa da genellikle yetersiz kaldığı için **libgee** kütüphanesinden faydalanılır. Öncelikle kodun en üstüne *Using gee;* eklenir. bu sayede kütüphane içerisindeki işlevler kullanılabilir olur. Bu ifade detaylı olarak ilerleyen bölümlerde anlatılacaktır. **libgee** kullanılırken derleme işlemine *--pkg gee-0.8* eklenir. Bu sayede derlenen programa libgee kütüphanesi dahil edilir.

```
$ valac main.vala --pkg gee-0.8
```

Liste tanımlaması ve eleman ekleyip çıkarılması aşağıdaki gibidir:

```
Using gee;

void test(){
    var liste = new ArrayList<int>();
    liste.add(12);
    liste.add(18);
    liste.add(3);
    liste.remove(18);
}
...
```

Yukarıdaki örnekde **ArrayList** tanımlanmıştır. **add** ile eleman eklemesi **remove** ile eleman çıkarılması yapılır.

Listenin belirtilen index sayılı elemanı aşağıdaki gibi getirilir.

```
...
int num = liste.get(3); // 4. eleman değeri getirilir.
...
```

Listenin istenen bir elemanı aşağıdaki gibi değiştirilebilir.

```
...
liste.set(3,144); // 4. eleman değiştirilir.
...
```

Listenin eleman sayısı aşağıdaki gibi bulunur.

```
...
int boyut = liste.size;
...
```

Klavyeden değer alma

Klavyeden string türünden değer almak için **stdin.read_line()** kullanılır.

```
...
var text = stdin.read_line();
stdout.printf(text);
...
```

Koşullar

Koşul tanımlamak için **if** kullanılır. Bu ifade parametre olarak **bool** türünden değişken alır. Koşulun gerçekleşmediği durumda **else if** kullanılarak diğer koşul karşılanıyor mu diye bakılır. Hiçbiri gerçekleşmiyorsa **else** kullanılarak bu durumda yapılacaklar belirtilir.

```
...
if(koşul){
   ...
}else if(diğer-koşul){
   ...
}else{
   ...
}
...
```

Örneğin klavyeden değer alalım ve bu değerin eşit olma durumuna bakalım.

```
...
string parola = stdin.read_line();
if(parola == "abc123"){
    stdout.printf("doğru parola");
}else{
    stderr.printf("hatalı parola");
}
...
```

Koşullarda kullanılan işleyiciler ve anlamları aşağıda liste halinde verilmiştir.

Koşul işleyicileri

ifade	anlamı	örnek
>	büyüktür	121 > 12
<	küçüktür	12 < 121
==	birbirine eşittir	121 == 121
!	karşıtlık bildirir.	!(12 > 121)
&&	logic and	"fg" == "aa" && 121 > 12

||	logic or	"fg" == "aa" || 121 > 12
!=	eşit değildir	"fg" != "aa"
>=	büyük eşittir	121 >= 121
<=	küçük eşittir	12 <= 12
in	eleman içerme kontrolü	12 in {12, 121, 48, 94}

Koşullar için alternatif olarak şu şekilde de kullanım mevcuttur.

```
koşul ? durum : diğer-durum;
```

Burada **?** işaretinden sonra ilk durum **:** işaretinden sonra da gerçekleşmediği durum belirtilir.

```
...
string parola = stdin.read_line();
parola == "abc123" ? stdout.printf("Doğru parola") : stderr.printf("yanlış parola");
...
```

Döngüler

Döngüler aşağıdaki gibi tanımlanır. döngüde koşul sağlandığı sürece sürekli olarak içerisindeki kod çalıştırılır.

```
while(koşul){
    ...
}
```

Örneğin ekrana 0dan 10a kadar olan sayıları yazdıralım.

```
...
int sayi = 0;
while (sayi <=10){
    stdout.printf(sayi.to_string());
    sayi += 1; // sayi = sayi + 1 ile aynı anlama gelir.
}
...
```

Yukarıdaki örnekte **while** ifadesi sayı 10dan küçük ve eşitse çalışır. sayı 11 olduğunda bu sağlanmadığı için işlem sonlandırılır.

for ifadesi kulanılarak benzer bir döngü yapılabilir. Örneğin:

```
...
for(int i=0; i<=10; i++){ // i += 1 ile aynı anlama gelir
    stdout.printf(sayi.to_string());
}
...
```

Bu örnek while örneğindeki ile aynı işlemi gerçekleştirir.

Bir listenin tüm elemanları ile döngü oluşturmak için ise **foreach** kullanılır.

```
...
int[] i = {31, 44, 78, 84, 27};
foreach(int sayi in i){
    stdout.printf(sayi.to_string());
}
...
```

Burada **sayi** değişkeni her seferinde listenin bir sonraki elemanı olarak tanımlanır ve işleme koyulur.

Döngüden çıkmak için **break** döngünün alt satırlarının çalışmayıp sonraki koşul için başa dönülmesi için ise **continue** kullanılır.

```
...
while(true){
    int txt = stdin.read_line();
    if(txt == "abc123"){
        stdout.printf("Doğru parola");
        break;
    }else{
        stderr.printf("Hatalı parola");
        continue;
    }
    stdout.printf("test 123"); // bu satır çalıştırılmaz.
}
...
```

Fonksiyonlar ve parametreler

Vala yazarken forksiyon tanımlarız ve bu fonksiyonları parametreler ile çağırabiliriz.

```
int main(string[] args){
    write("Hello world");
    return 0;
}
void write(string message){
    stdout.printf(message);
}
```

Bir fonksiyon sadece bir kez tanımlanabilir. Fakat fonksiyonu isim olarak oluşturup daha sonra tanımlamak mümkündür.

```
...
void fff(); // isim olarak tanımlanabilir.
void fff(){
    stdout.printf("hmmm");
}
...
```

Ayrıca fonksiyonu isim olarak tanımlayıp **C** programlama dili ile yazılmış bir fonksiyon kullanabiliriz. Bu sayede kaynak kod C ve Vala karışımından oluşmuş olur. Bunun için **extern** ifadesi kullanılır.

```
// main.vala dosyası
extern void fff(string msg);
int main(string[] args){
    fff("Hello World");
}
```

```
// util.c dosyası
#include <stdio.h>
void fff(char* msg){
    fputs(msg,stdout);
}
```

Yukarıdaki örnekteki 2 dosyayı derlemek için aşağıdaki gibi bir komut kullanılmalıdır.

```
$ valac main.vala util.c
```

C kaynak kodunun gerektirdiği parametreleri **-X** kullanarak ekleyebiliriz. Bu sayede doğrudan gccye parametre eklenebilir.

```
$ valac main.vala util.c -X "-lreadline" # C ile readline kütüphanesini kullanmak için -lreadline gerekir.
```

Vala içinde C kullanabildiğimiz gibi tam tersi de mümkündür. Bunun için C tarafında fonksiyon için isim tanımlamamız yeterlidir.

```
void fff(char* message);
int main(int argc, char *argv[]){
    fff("Hello world");
}
```

```
public void fff(string message){
    print(message);
}
```

Yukarıdaki örnekte C kodu içerisinde vala ile yazılmış bir fonksiyon kullanılmıştır.

Bir fonksiyon normal şartlarda başka bir fonksiyona parametre olarak verilemez. Bu gibi durumlar için **delegate** ifadesinden yararlanılır. Önce delegate ifadesi ile fonksiyonun nasıl tanımlanacağı belirtilir daha sonra bu yeni oluşturulmuş tür parametre olarak kullanılır.

```
delegate void fff(string message);

// delegate ile kullanıma uygun fonksiyon tanımladık.
void f1(string message){
    stdout.printf(message);
}

// delegate çağırmaya yarayan fonksiyon yazdık
void f2(fff function, string message){
    function(message);
}

// main fonksiyonu
void main(string[] args){
```

```
    f2(f1,"Hello World");
}
```

Sınıf kavramı

Vala nesne yönelimli bir programlama dilidir. Bu sebeple sınıflar oluşturabiliriz. Sınıflar **Gtk** gibi arayüz programlamada kullanışlı olmaktadırlar. Sınıf oluşturmak için **class** ifadesi kullanılır.

```
public class test {
    public void write(){
        stdout.printf("test123");
    }
}
int main(string[] args){
    test t = new test();
    t.write();
}
```

Yukarıdaki örnekte sınıf tanımlanmıştır. Daha sonra bu sınıftan bir nesne türetilmiştir ve ardıntan nesneye ait fonksiyon çalıştırılmıştır.

Sınıf içerisinde bulunan bazı fonksiyonların dışarıdan erişilmesini istemiyorsanız **private**, erişilmesini istiyorsanız **public** ifadesi ile tanımlamanız gerekmektedir.

Sınıf içerisinde tanımlanmış değişkenlere ulaşmak için **this** ifadesi kullanılır.

```
...
public class test {
    private int i;
    private int j;

    public void set(int i, int j){
        this.i = i;
        this.j = j;
    }
}
...
```

Super sınıf

Bir sınıfı başka bir sınıftan türetebiliriz. Bunun için sınıf tanımlanırken *class xxx : yyy* yapısı kullanılır.

```
public class hello {
    public void write_hello(){
        stdout.printf("Hello");
    }
}
public class world : hello {
    public void write_world(){
        stdout.printf("World");
    }
    public void write(){
        write_hello();
```

```
        write_world();
    }
}
int main(){
    world w = new world();
    w.write();
    return 0;
}
```

Eğer var olan bir fonksiyonun üzerine yazmak istiyorsak **override** ifadesini kullanabiliriz.

```
...
public class hello {
    public void write(){
        stdout.printf("hello");
    }
}
public class world : hello {
    public override void write(){
        stdout.printf("world");
    }
}
...
```

Bir sınıfı birden fazla sınıfın birleşiminden türetebiliriz.

```
...
public class hello {
    public void write_hello(){
        stdout.printf("Hello");
    }
}
public class world {
    public void write_world(){
        stdout.printf("World");
    }
}
public class helloworld : hello, world {
    public void write(){
        write_hello();
        write_world();
    }

}
...
```

Signal kavramı

Valada sinyal tanımlayarak bir sınıftaki bir işlevin nasıl çalışması gerektiği ayarlanabilir. Bunun için isim olarak tanımlanan fonksiyonun başına **signal** ifadesi yerleştirilir.

```
public class test {
    public signal void sig1(int i);

```

```
    public void run(int i){
        this.sig1(i);
    }
}
int main(string[] args){
    test t1 = new test();
    t1.sig1.connect((i)=>{
        stdout.printf(i.to_string());
    });
    t1.run(31);
    return 0;
}
```

Namespace kavramı

Valada kodları alan adlarına bölerek yazmamız mümkündür. Bu sayede alan adı içine tanımladığımız fonksiyonları alan adı ile beraber çağırarak kullanabiliriz. Bunun için **namespace {}** ifadesi kullanılır.

```
namespace test {
    void print(){
        stdout.printf("Test");
    }
}
...
int main(string[] args){
    test.print();
}
```

Namespace iç içe tanımlanabilir.

```
namespace test1 {
    namespace test2 {
        void print(){
            stdout.printf("Test");
        }
    }
}
...
int main(string[] args){
    test1.test2.print();
}
```

Bir namespace alanını kaynak kodda içeri aktararak kullanmak için **using** ifadesi kullanılır. Bu ifade sayesinde belirtilen alan adındaki tüm fonksiyonlar kaynak kodda doğrudan kullanılabilir hale gelir.

```
using Gtk;

int main(string[] args){
    // İsterseniz yine de namespace adı ile kullanabiliriz.
    Gtk.init (ref args);
    // Gtk.Window yerine Window kullanabiliriz.
    var win = new Window();
```

```
    // Şununla aynı anlama gelir
    // var win = new Gtk.Window();
    ...
    // Aynı isimde var olan bir fonksiyonu namespace adı olmadan kullanmak mümkün değildir.
    Gtk.main ();
    return 0;
}
```

Sınıfları tanımlarken namespace belirterek tanımlamak mümkündür. Bunun için sınıfın adının başına namespace adını belirtmek yeterlidir.

```
public class test.cls {
    public void print(){
        stdout.printf("Test");
    }
}
...
int main(string[] args){
    var tcls = new test.cls();
    tcls.print();
    return 0;
}
```

Kütüphane oluşturma

Vala kaynak kodu kullanarak kütüphane oluşturabiliriz. Bunun için kodu aşağıdaki gibi derleyebiliriz.

```
// library.vala dosyası
public int test(){
    stdout.printf("Hello World");
    return 0;
}
```

Vala kaynak kodunu önce C koduna çevirmemiz gerekmektedir. Daha sonra gcc ile derleyebiliriz. Vala programlama dili **glib-2.0** kullanarak çalıştığı için bu kütüphaneyi derleme esnasında eklememiz gerekmektedir. Ayrıca glib-2.0 derlenirken **-fPIC** parametresine ihtiyaç duyar.

```
# Önce C koduna çevirelim
$ valac -C -H libtest.h --vapi libtest.vapi library.vala
# Sonra gcc ile derleyelim.
$ gcc library.c -o libtest.so -shared \
    `pkg-config --cflags --libs glib-2.0` -fPIC
```

Alternatif olarak aşağıdaki gibi de derleyebilirsiniz. Bu durumda C kaynak koduna çevirmeye gerek kalmadan kütüphanemiz derlenmiş olur.

```
$ valac -H libtest.h --vapi libtest.vapi \
    -o libtest.so -X -shared -X -fPIC library.vala
```

Şimdi aşağıdaki gibi bir C kodu yazalım ve kütüphanemizi orada kullanalım. Oluşturulmuş olan **library.h** dosyamızdan yararlanabiliriz.

```
// main.c dosyası
#include <libtest.h>
int main(){
    gint i = test(); // vala değişken türleri glib kütüphanesinden gelir.
    return (int) i;
}
```

Ve şimdi de C kodunu derleyemlim.

```
$ gcc -L. -I. -ltest main.c `pkg-config --cflags --libs glib-2.0` -fPIC
```

Bununla birlikte **libtest.vapi** dosyamızı kullanarak kütüphanemizi vala ile kullanmamız da mümkündür.

```
// main.vala dosyası
int main(string[] args){
    int i = test();
    return i;
}
```

Şimdi vala kodunu derleyelim.

```
$ valac --vapidir ./ main.vala --pkg libtest
```

Gobject oluşturma

Gobject kulanarak yazdığımız kütüphaneyi farklı dillerde kullanmamız mümkündür. Bunun için önce aşağıdaki gibi bir kaynak kodumuz olsun. Burada bin namespace tanımlayalım.

```
namespace hello {
    public void print(){
        stdout.printf("Hello World\n");
    }
}
```

Şimdi bu kodu aşağıdaki gibi derleyelim.

```
# Önce C koduna çevirelim ve gir dosyası oluşturalım.
$ valac -C main.vala \
    --gir=hello-1.0.gir \
    --library=hello \
    -H libhello.h
# C kodunu derleyelim.
$ gcc main.c -o main -shared \
    `pkg-config --cflags --libs glib-2.0` -fPIC
```

Burada yazdığımız nameplace alanına ait fonksiyonları ve sınıfları parametreleri ile birlikte listeleyen şablonumuz oluşmuş oldu. Şimdi bu şablondan typelib dosyası oluşturalım.

```
$ g-ir-compiler hello-1.0.gir --shared-library=libhello --output=hello-1.0.typelib
```

Son olarak dosyaları sistemimize kuralım.

```
$ install libhello.so /usr/lib/
$ install hello-1.0.typelib /usr/share/girepository-1.0/
```

Bunun yerine aşağıdaki 2 çevresel değişkeni ayarlayarak test etmemiz mümkün dür.

$ export GI_REPOSITORY_PATH=/home/user/gobject-ornek $ export LD_LIBRARY_PATH=/home/user/gobject-ornek

Şimdi yazdığımız kütüphaneyi python ile çalıştıralım. Bunun için aşağıdaki gibi bir python kodu yazabiliriz.

```
import gi
gi.require_version('hello', '1.0')
from gi.repository import hello
hello.print()
```