Örs Kılavuz

Toggle navigation sidebar

Toggle in-page Table of Contents

İçerik:

Theme by the [Executable Book Project](https://ebp.jupyterbook.org)

Contents

İçerik:

* [Giriş](index.html#document-bolumler/giris)
  + [Hedefler](index.html#document-bolumler/giris/onsoz)
  + [Tanıtım](index.html#document-bolumler/giris/tanıtım)
  + [Neden yeni bir yazılım dili ?](index.html#neden-yeni-bir-yazilim-dili)
  + [Örs tasarım ilkeleri](index.html#ors-tasarim-ilkeleri)
  + [Kurulum](index.html#document-bolumler/giris/kurulum)
  + [Derleme](index.html#document-bolumler/giris/derleme)
* [Tanıtım](index.html#document-bolumler/hızlı_tanıtım)
  + [Lügât](index.html#document-bolumler/tanıtım/lügat)
  + [Yorumlar](index.html#document-bolumler/tanıtım/yorumlar)
  + [Değerler](index.html#document-bolumler/tanıtım/degerler)
  + [İfadeler](index.html#document-bolumler/tanıtım/ifadeler)
  + [Satırlar](index.html#document-bolumler/tanıtım/satırlar)
  + [Hafıza](index.html#document-bolumler/tanıtım/hafıza)
  + [İşlemler](index.html#document-bolumler/tanıtım/islemler)
  + [Türler](index.html#document-bolumler/tanıtım/turler)
  + [Birimler](index.html#document-bolumler/tanıtım/birimler)
  + [Yine Yeniden İşlemler:](index.html#document-bolumler/tanıtım/yine_yeniden_islemler)
* [Acemi Ocağı](index.html#document-bolumler/acemi_ocagı)
* [Algoritmalar](index.html#document-bolumler/algoritmalar)
* [Unix Ortamı](index.html#document-bolumler/unix_ortami)
* [Tasarım](index.html#document-bolumler/tasarım)

# Örs Yazılım Dili Kullanım Kılavuzu

## Contents

İçerik:

* [Giriş](index.html#document-bolumler/giris)
  + [Hedefler](index.html#document-bolumler/giris/onsoz)
  + [Tanıtım](index.html#document-bolumler/giris/tanıtım)
  + [Neden yeni bir yazılım dili ?](index.html#neden-yeni-bir-yazilim-dili)
  + [Örs tasarım ilkeleri](index.html#ors-tasarim-ilkeleri)
  + [Kurulum](index.html#document-bolumler/giris/kurulum)
  + [Derleme](index.html#document-bolumler/giris/derleme)
* [Tanıtım](index.html#document-bolumler/hızlı_tanıtım)
  + [Lügât](index.html#document-bolumler/tanıtım/lügat)
  + [Yorumlar](index.html#document-bolumler/tanıtım/yorumlar)
  + [Değerler](index.html#document-bolumler/tanıtım/degerler)
  + [İfadeler](index.html#document-bolumler/tanıtım/ifadeler)
  + [Satırlar](index.html#document-bolumler/tanıtım/satırlar)
  + [Hafıza](index.html#document-bolumler/tanıtım/hafıza)
  + [İşlemler](index.html#document-bolumler/tanıtım/islemler)
  + [Türler](index.html#document-bolumler/tanıtım/turler)
  + [Birimler](index.html#document-bolumler/tanıtım/birimler)
  + [Yine Yeniden İşlemler:](index.html#document-bolumler/tanıtım/yine_yeniden_islemler)
* [Acemi Ocağı](index.html#document-bolumler/acemi_ocagı)
* [Algoritmalar](index.html#document-bolumler/algoritmalar)
* [Unix Ortamı](index.html#document-bolumler/unix_ortami)
* [Tasarım](index.html#document-bolumler/tasarım)

# Örs Yazılım Dili Kullanım Kılavuzu[#](#ors-yazilim-dili-kullanim-kilavuzu)

## Giriş[#](#giris)

### Hedefler[#](#hedefler)

#### Eğitim hedefleri:[#](#egitim-hedefleri)

Örs dilinin nasıl çeşitli yazılımları hazırlamakta verimli ve etkili bir şekilde kullanılacağına dair kullanım kılavuzu hazırlanması.

Orta sınıf öğrencilerin ve üstünün matematik ve bilgisayar bilimlerindeki doğal terimlerle aşina edilmesi.

Doğal terimlerle hazırlanmış, elaltı tasarım ilkesine uyan yazılım arayüzü e ilgili yazılım kütüphanelerinin ve kullanım kılavuzlarının hazırlanması.

Çeşitli vidyo paylaşım sitelerinde yazılımın temelleri, algoritmalar, bilgisayar bilimleri ve terimleri ile alakalı, herşeyden önce aşina olmayana yönelik eğitim vidyoları ve kısa tanıtım dizilerinin hazırlanması.

Dilin orta okul, lise ve üniversite denginde eğitimleri için derste okutulabilir kitabının yazılması.

Gençlerimizin yazılım keyfini tatmaları, onlara ileri seviye bir kavram olan ama temel bir yeti olan yazım mimarisi kavramının üstü kapalı olarak aşılanması için oyun motoru hazırlanması.

Çocuk ve gençlerimize temel seviye bilişim okur yazarlığı için dağınık olan kaynakların uzmanlarınca bir araya getirilmesi ve onların bilgi işlem başıbozukluğundan kurtarılması.

Bilgisayar, matematik, mühendislik konularında açık kaynak, yani uzmanların tartışmasına ve katkısına açık, bir terimler ve deyimler sözlüğü oluşturulması.

#### Altyapı Hedefleri[#](#altyapi-hedefleri)

Windows, Unix ailesi işletim sistemleri ile iletişim kuracak arayüzlerin hazırlanması.

Hiç değilse insan yetiştirmek amacı güden işletim sistemi çekirdeği yazmak.

Hiç değilse insan yetiştirmek amacı güden dosya sistemi yazılımları ve algoritmaları üzerine çalışma yapmak.

Örs yazılım dili söz dizimi ile uyumlu, Microsoft C# Linq tasarımını ilkelerine sadık, SQL, NOSQL veritabanı ağ arayüzleri ile iletişim kurabilen veritabanı dili geliştirilmesi.

İşletim sistem ile iletişim kurmaya yarayan ingilizce “shell” denilen, Örs dili tasarım ilkelerine, yani sadelik, anlaşılabilirlik, okunabilirlik, basitlik, sadık metin arayüzünün geliştirilmesi

Bir sistem yazılımı dili olan Örs’ün aksine nesne tabanlı, işlevsel (yani lamda ifadelerine doğal destek veren), çeşitli yazılım ortamlarında çalışabilecek, piyasadaki alanda iş yapan şirketlerin üretkenliğini arttıracak bir yazılım dili daha tasarlanması.

Unix ailesi işletim sistemleri hakkında bilinçlendirme, farkındalık çalışmaları yapmak. Bu konuda çeşitli yayınlar hazırlamak.

#### Türkçe hedefleri[#](#turkce-hedefleri)

Dilimizin kısır, işlevsiz, kulağa çirkin gelen plaza diline karşı elini güçlendirmek. Sadece bu herşeye değer. Kurumsal hedefler:

Yukarıdaki hedeflere sadık kalacak, eğitim odaklı vakıf kurmak.

#### Faydalar[#](#faydalar)

Bilgi işlem düzenekleri tasarımı konusundaki eğitim sınırını orta okula kadar çekmek fazlasıyla mümkün.

#### İletişim[#](#iletisim)

Herhangi bir sorunla karşılaştığınızda [musakalayci@gmail.com](mailto:musakalayci%40gmail.com) adresine bildirebilirsiniz.

### Tanıtım[#](#tanitim)

### Neden yeni bir yazılım dili ?[#](#neden-yeni-bir-yazilim-dili)

### Örs tasarım ilkeleri[#](#ors-tasarim-ilkeleri)

### Kurulum[#](#kurulum)

Gerekli yazlımlar:

#### Freebsd için[#](#freebsd-icin)

$ pkg install llvm[10+] clang[10+] python[3+] cmake[3+] git

Tercihen valgrind kurabilirsiniz.

$ pkg install valgrind

Eğer freebsd’de clang atfı yoksa:

$ ln -s /usr/local/bin/clang[10+] /usr/local/bin/clang

Eğer python atfı yoksa:

$ ln -s /usr/local/bin/python[3+] /usr/local/bin/python

#### Linux için[#](#linux-icin)

Pacman yazılımı kullanan Arch Linux türevleri için:

$ pacman -Syu llvm clang python[3+] cmake git

Ubuntu ve Debian türevleri için:

$ apt install llvm clang python[3+] cmake git

Gerekli yazılımlar yüklendikten sonra

$ git clone https://github.com/musakalayci/ors.git  
$ cd ors  
$ ./baslat.py -d

Uyarı

Eğer ki eğer ‘baslat.py’ belgesi çalıştırılamıyorsa

$ chmod -x ./baslat.py

komutu ile belgeyi çalıştırılabilir hale getirmelisiniz.

Yukarıdaki komutlar ile örs derleyicisini C derleyicisine derlettirip kurabilirsiniz. LLVM arayüzünün çalıştığı her ortamda Örs kullanılabilir hale gelecektir. Şuan itibari ile Örs kaynak kodu C dilinden oluşuyor ama Örs beta aşamasına geçtiğinde kendi kendisini derleyecek yetiye sahip olacaktır ve ilk sürümde hem derleyicinin dili hem de ortamının örtüşmesi sağlanacaktır.

### Derleme[#](#derleme)

Uyarı

Daha hala hazırlık aşamasındayız. Bu illaki kötü bir şey olacak diye bir kaide yok. Eğer yapıcı eleştirilerinizi iletirseniz hazırlık aşamasında katkıda bulunabilir ve Örs dilinin gelişimine kendi isteklerinizi de ekleyebilirsiniz.

İletişim için [musakalayci@gmail.com](mailto:musakalayci%40gmail.com) adresine yazarsanız, konusunu belirtirseniz eleştirilerinizi mümkün olduğunca dikkate alabilirim.

Hatta önerileriniz varsa iletişim kanallarını açık tutarız ve beraber sorunlara çözümler geliştirebiliriz.

Uslu çocuk olursak şirinleri bile görebiliriz.

* [Arama Sayfası](search.html)

## Tanıtım[#](#tanitim)

### Lügât[#](#lugat)

İkinci şahıslar tarafından kullanılamayacak kelimeler her yazılım dilinde olduğu gibi Örs’de de var. Bunlar yazılım dilini oluşturan söz dizisine dahil oldukları için diğer kullanımlara kapalıdırlar.

Birimler:

* birim, dahili

İşlemler:

* iş, sanal,

Birim Erişim:

* küresel, bölgesel, yerel, yaban

İfadeler:

* evet, hayır, doğru, yanlış, boş

Karşılaştırma:

* ve, ya\_da, veya, veyâ

Yapıtaşları:

* eh
* t8, harf, t16, t32, t64, t128, tam
* d8, simge, d16, d32, d64, d128, doğal
* o32, o64, o128, ondalık
* şey, mimari, metin

Türler:

* tür, ortak, kalıp, sayaç

Satırlar:

* değer
* git
* eğer, eğer ki, değilse
* tüm, her, devam, son
* durum, seçim, varsayılan, tekrar, geç,
* dön

Hafıza:

* yeni, temiz, yenile
* sil, boşalt, doldur

### Yorumlar[#](#yorumlar)

Örs dilinde yorumlar C ailesi söz dizimi kuralını uygular.

Tek Satırlı:[#](#term-Tek-Satirli)

//Toplama işlemi için tek satır yorum örneği.  
iş Topla a tam, b tam: tam =>  
 dön a + b; //tek satırlı yorum.

Tek satırlı yorumlar için ‘//’ simgesi ile başlanıp yeni *metin satırında* biter. Burada altını çizmek isterim metin satırları yani ‘/n’ ile simgelendirdiğimiz ascii kodu ‘0x0A’ olan kavram ile Örs derleyicisinin satırlardan anladığı şey bambaşka şeylerdir ve kesinlikle karıştırılmamalıdır. Bu konuya satırlar konusunda değineceğiz.

Çok Satırlı:[#](#term-Cok-Satirli)

/\*  
 Toplama işlemi için  
 çok satırlı yorum örneği.  
\*/  
iş Topla a tam, b tam: tam =>  
 dön a + b; //tek satırlı yorum.

Çok satırlı yorumlar için ‘/\*’ simgesi ile başlanıp ‘\*/’ bitirilir.

Kılavuz:[#](#term-Kilavuz)

Yorumlar kavram olarak yazılımcıların çalışırken ilk önce kendi kendilerine tuttukları notlar olarak değerlendirilebilir. O yüzden derleyici ‘.ors’ belgesindeki yorumları anlamlandırmaz ve çözümleme aşamasında geçilir.

Tabi ki de faydaları tartışılamaz ama kılavuz üretiminde kullanışlı değillerdir. Alakalı önemi ise Örs doğasında açık kaynak destekleyen bir dil ve kılavuzu yazılmamış açık kaynak yazılımların piyasada bir önemi olmamaktadır. Bu yönüyle Örs dilinin kılavuz yetenekleri en az yazılımdaki yetenekleri kadar önemlidir. Çünkü diğer mühendislerin okuyup anlayamadığı ve dolayısıyla kullanamadığı kodun açık kaynak dünyasında pek yaşama şansı yoktur.

{\*  
 kılavuz::örnek\_kütüphane::Topla +=  
 tanıtım: "Tam sayılar için toplama işlemi yapar",  
 değişkenler: {...},  
 örnek: "x := 22;\\n  
 y := 33;\\n  
 z := Topla(x, y)";  
\*}  
iş Topla a tam, b tam: tam =>  
 dön a + b;

Yukarıdaki örnekte olduğu gibi kılavuz açmak için ‘{\*’ simgesi ile başlanır; kılavuz öğesi yazılır ve ‘\*}’ simgesi ile kapatılır.

Kılavuz hazırlama konusunu daha ayrıntılı bir şekilde kılavuz bölümünde ele alacağız.

Kaideler:[#](#term-Kaideler)

**Yorumlar derleyici tarafından satırmış gibi algılanır.** O yüzden ifadeler arasında yorum bulunamaz. Yani aşağıdaki örnek Örs derleyicisi tarafından reddedilecektir.

değer i tam = a \* //bir takım şeyler  
 b;

**Kılavuz yorumları sadece 'birim'lerde yer alabilir.** Zira işlem bedenlerinde kılavuzun anlamı yoktur. Bu bağlamda kılavuz yorumları birim’lere dahil edilir.

### Değerler[#](#degerler)

Özlerinde amacı bir takım veriler üretmek olan yazılımların başlangıç ve sonuçlarını saklamak için değer kavramına ihtiyacımız var. Bu konuda çeşitli tartışmalar ve aşırıya kaçan çözümler olsa da Örs, C dillerinden yani bilgisayarın kalbine doğrudan erişimi olan dillerden olduğu için, tanımlı tüm değerlerin bilgisayar belleğinde fiziki konumu vardır ve bu konum sanal yollardan erişilebilir.

Örs’de söz dizimi farkıyla iki türlü değer tanımı yapılabilir. Açık ve Pascal tanımı.

Açık tanım:[#](#term-Acik-tanim)

değer sayı tam;.  
değer \_birşeyler metin;  
değer \_utf\_metni \*t8;  
  
değer yaşım tam = 29;  
değer \_mahmut metin = "Ikinci Mahmut";

Burada söz diziminden anlaşılacağı üzere “değer” kelimesi ile değer tanımlandığı tetiklenir, sonrasında değerin benzersiz ismi gelir, sonrasında ise değerin türü. Değer tanımlamak için bu kadarı yeterlidir ama bu şekliyle değerlerin ilk değerleri belirlenmemiştir. Dolayısıyla bu değerlerin hafızada tuttuğu veriye ulaşıldığında elimize ne geçeceğini bilemeyiz. O yüzden isteğe bağlı olarak, yukarıda olduğu gibi, değerler “=” simgesi ile başlatılabilir.

Pascal tanımı:[#](#term-Pascal-tanimi)

gelen := belge::aç();  
yaşım := 29;  
\_mahmut := "İkinci Mahmut";

Pascal tanımlarda ise değerin ilk değerinin olduğu varsayılır. Bu yönüyle tanımın türü en baştan bellidir ve diğer söz dizileri gereksiz kalır. Tasarım amacı okunmayı kolaylaştırmaktır. Görsel kullanışlılığı dışında tüm fiziki, yani bilgisayarın gördüğü, özellikleri “değer” kavramı ile aynıdır. İsmini ise Pascal yazılım dilinden alır. Çeşitli yazılım dillerinde Pascal tanım satırını görebilirsiniz.

### İfadeler[#](#ifadeler)

#### Boş[#](#bos)

Sayılar için 0 değerini; türler için sıfırlanmış üyeleri; konumlar için ise yokluğu tanımlayan saf ifadelerdir. Yani saf halleriyle daha anlamlıdırlar. Anlamları üretim sürecinde yorumlanır.

C dilindeki ‘void’ değillerdir ve bu yönüyle ‘\*boş’ gibi bir ifade hata verir zira olmayan konumun değerini nasıl alacaksınız.

değer i tam = {};  
değer j tam = 0;  
değer k tam = boş;

Örnekteki tanımlı değerlerin hepsinin içeriği 0’dır.

tür ikili  
{  
 a tam;  
 b tam;  
}  
  
iş Örnek  
{  
 değer a\_ve\_b ikili = {};  
 değer b\_ve\_a ikili = boş;  
 //sıradaki ise hata verir;  
 değer hatalı ikili = 0;  
}

Örnek işleminin ilk iki satırında ‘ikili’ türündeki değer yaratılır ama içeriği sıfırlanmıştır. Üçüncü satırda ise hata alacağız çünkü ikili türünün içeriğinde sayı yapıtaşları olsa da kendisi sayısal yapıtaşı değildir.

#### Sayılar[#](#sayilar)

Bilgisayar dünyasındaki tüm veri türleri en sade haliyle ifade edilmiş sabit değerlerdir. Sabitlerdir ve bir değere atanıncaya kadar yaşadıkları konumları yoktur.

Şimdi biraz hayal edin. 16 bit genişliğinde veri türü tanımladığınızda aslında yaptığınız şey; 2^16 farklı birim tanımlamaktır. Çünkü sayılar deyince ortalama bir insanın aklına hesap gelebilir ama bizim için en önemli anlamı bu sayıların etiket gibi kullanılabilecek olmasıdır. Mesela ilk okul numaram 115’di. 115 numarası okul ortamında bana atıfta bulunuyordu. Işte bu yönüyle katı tanımlı sayı türlerinin varlığı sayısal tasarımda anlam kazanıyor. Veritabanlarından, işletim sistemlerine, devlet dairelerindeki belgelerden, okullara kadar her yerde var olan basit bir anlam. O da basitçe özlerinde birer atıflar olmalarıdır. Yani tanımladığımız benzersiz sayılar benzersiz nesne ya da öznelere atıfta bulunurlar.

Örs’de tanımlı tüm sayı yapıtaşları C, C++ dilleri ile llvm aracılığı ile tam uyumludur. Zira öyle olmak da zorundadır. Çünkü yazılım dilleri uzayın derinliklerinde yalıtılmış ortamlarda yaşamıyor. Aksine milyonlarca farklı birimle iletişim halindeler ve bu iletişimi kayıpsız yapabilmenin en basit yolu, tüm dillerde kullanılabilecek sade, bitlerle tanımlabilecek, türlerden geçmektedir. Zaten “dijital tasarım” denilen kavramın özü de tam olarak burasıdır.

/\*pascal haliyle:\*/  
atmışDörtBitlikTamSayı := 55\_st64;  
uzunSayılar := 10\_000\_000\_st32; /\*fark ettiyseniz alt tirelerin yerinin bir önemi yok\*/  
okulNo := 2004\_43\_280\_st32; /\*mesela\*/  
yaşım := 29; /\*t32 varsayılır\*/  
oran32 := 44.00\_o32;  
oran := 12.88 /\*o64 varsayılır ve ondalık kısım nokta ile ayrılır\*/;  
  
/\*açık haliyle\*/  
değer uzun d64 = 11\_000\_000;  
değer uzun64 d64 = 22\_999\_000\_sd64; /\*Aynı şey mi ? Evet. Peki ya sıradaki:\*/  
değer u64 d64 = 22\_990\_1\_st32; /\*u64 değeri 64 bitlik doğal sayı olarak tanımlanıp atanan sabit değer çevrilir.\*/

#### Konumlar ve Diziler[#](#konumlar-ve-diziler)

Sistem dillerinin en çetrefilli ama eğlenceli konusu: Konumlar.

Basitçe hayal edelim; Bir bakkal deposu var ve burada sayılamayacak kadar çok kutu ve içinde de bir takım abur cuburlar var. Tüm bu kutular numaralanmış ve numaralanmış raflara yerleştirilmiş. Kim yerleştirdi bilmiyoruz da umursamıyoruz da. Amacımız bir an önce ‘çuklatalara’ erişmek. İşin güzel yanı laz bakkal bize demiş ki "Al üzerinde ‘LB 22 543’ yazan bu kağıdı takip et, dilediğin kadar çuklataya yumul gitsin."

Ben olsam, kağıttaki bilgiyi yorumlamakla başlardım. Derdim ki, "Tamam LB laz bakkal demek ve zaten depodayız, 22 ve 543 ise ya raf ya da kutu numarası olamalı. İlkin 22 numaralı rafla başlayayım uymazsa tersini denerim." Basit bir strateji ve elimde bir kutu çuklata var.

Konumlar kunusu da bu örneğe benziyor. Konu çetrefilli çünkü konumların gösterdiği yerlerde çuklata yok yine bir diğer sayı var. Eğlenceli çünkü o sayı her anlama gelebilir. Mesela veri tabanındaki en sevdiğimiz çuklata kutusunun depodaki konumunu verebilir ya da o konumda komik vidyo adresi olabilir yani herşey olabilir. Örs dilinde tabi ki de konumlar diye bir özellik yok; hatta diğer sistem dillerinde de yok ama kavram hem sanal dünyada hem de gerçek dünyada temel bir konu olarak yer aldığı gibi Örs’de dahil tüm sistem dillerinde yer alıyor.

Sadece Örs değil diğer sistem dillerinde de konum kavramı bir soyutlamadır. Özlerinde mimari boyutunda doğal sayılardır. Mesela 64 bitlik işlemcideki konumlar da 64 bit genişliğindedir ve bu konumlar basitçe bilgisayar hafızasındaki 64 bitlik hücrelerin numaralarıdır.

Düz mantıkla bilgisayar belleği uç uca dizilmiş boncuklar dizisi gibi düşünülebilir. Mesela benim 64 bitlik bilgisayarın 8gb hafızası var ve bu **düz mantıkla** 0’dan (8gb/8)’e kadar numaralanmış 8 bitlik boncuklar var ve benim öyle bir kavrama ihtiyacım var ki ihtiyaç halinde bu boncuklara ulaşıp içindekini alayım.

Bu kavram konumlar oluyor. Bir önceki **düz mantık** örnekte gerçek konumlar var. Yukarıdaki Laz Bakkal örneğinin aksine elimizde doğrudan değeri gösteren doğrusal konum var. Gerçekte olan şey ise daha karmaşık. İşletim sistemi belleği bölgelere ayırıyor ve bu bölgeleri işlemler ihtiyaç duyduğunda onlara atıyor. Yani siz bir konumda bulunan değere ulaşmak istediğinizde önce işletim sisteminin yazılımıza atadığı bölgeye, sonra değere ulaşıyorsunuz ve bu ucu açık kavrama sanal konum deniyor ki yukarıdaki Laz Bakkal örneğinde, elimizdeki kağıtta yazan konum sanal bir konum.

Örs konumlar konusunu C dilindeki gibi ve Java, C#, OCaml gibi dillerin aksine, yorumsuz ele alır. Zira zaten işletim sistemi sanal konum vereceği için bunun üzerine bir algoritma daha eklemek yazılımın hızını azaltır. Böylece işletim sisteminin bize ayırdığı bölüme doğrudan erişim sağlanabilir. Bu kulağa hoş gelse de aynı zamanda olabildiğince tehlikelidirde. Diğer taraftan konumları yorumlamadan ele almak Örs’ün tasarım ilkeleriyle de çelişmez. Örs’ün tasarım felsefesi bu ve benzeri olaylara "eğer bir hata verilecekse bırakalım verilsin" yargısıyla yaklaşır. Çünkü konumlarla ilgili hatalar konumların doğrudan erişim yeteneğinden ziyade, hafıza yetersizliği, sıralama yanlışlığı, atanan bölge dışına taşma, boş konum ve benzeri sorunlardan kaynaklanır ve benzeri hatalar yazılımın hata ayıklama sürecinde engellenebilir ki yukarıdaki yargının atıfta bulunduğu anlam da budur. Bu yönüyle konumlarla ilgili olası tehlikeler yazılımcının sorumluluğundadır ve ne işletim sisteminin ne de sistem dillerinin sorumluluğunda değildir.

Örs dilinde ve diğer sistem dillerinde konumlar **‘mimari’** bayt genişliğinde yapıtaşlarıdır. Bu özelliğe atıfta bulunmak **‘şey’** yapıtaşının kavramsal görevlerinden de bir tanesidir.

Örs konumları kullanmak için C söz dizisinden yararlanır.

iş KonumÖrneği  
{  
 değer a t64 = 0x0\_47\_46\_45\_44\_43\_42\_41;  
 değer K \*t64 = &a; // a'nın konumunu K'ye geçiriyoruz.  
 stdio::printf(  
 "a: 0x%llx\n"  
 "a'nın konumu: %p\n"  
 "K: 0x%llx\n"  
 "K'nın konumu: %p\n",  
 a, // a değeri  
 &a, // a'nın konumunun alındığı ifade  
 \*K, // K'deki konuma gidildiği ifade  
 K); // ve K değerindeki konum.  
}

Tabi ki de böyle bir örneği gerçek hayatta pek kullanmazsınız. İşte tam burada bir kavramı sizinle tanıştırmam gerekiyor: Dereceler.

##### Konum Dereceleri:[#](#konum-dereceleri)

iş Giriş  
argümanSayisi tam,  
\_argümanlar \*\*t8  
{  
 //argümanları konumlarından yazdır.  
 her Argüman := \_argümanlar; \*Argüman; Argüman++:  
 stdio::printf(  
 "=> konum değeri: %p, konumdaki değer: %s\n",  
 Argüman,  
 \*Argüman);  
}

Yukarıda ise **‘\_argümanlar’** değişkeninde harfler dizisinin dizisinin konumu **Giriş** işlemine verilmiş.

Örnekte görüldüğü üzere **‘\_argümanlar’**’ değişkeninin türünde iki yıldız var. Derece dediğimize göre şunu diyebilir miyiz : "**’\_argümanlar’** değişkeni iki derecelidir" ? Malesef hayır. Değerin yaşadığı konumu saymayı unuturuz. Yani aslında **‘\_argümanlar’** değişkeni 3 dereceli bir değer. Çünkü tüm değişkenler aynı zamanda birer değer. Tanımladığınız tüm değerlere ve işlemlere atıfta bulunulurken, onların fiziki konumlarındaki veri işlem yığınına konulacak. Özet olarak değerlere olan herbir atfınız için konuma gidip derecesini düşürüp veriye öylece erişiliyor.

Bu manada birinci dereceden değerler tüm tanımlı değerler oluyor. Ayrıca, konum boyutu arttıkça derecesi de artıyor.

Burada elimizde kendi kendini gösteren kavram var. Biraz kafa karıştırıcı olsa da iletişim kuramının çıkmaz sokaklarında sıklıkla bu tarz varlıklara rastlanır.

Bu kavramı tanımlama gereği duydum zira hem derleyicide kullanılıyor, hem llvm altyapısında, hem de siz aslında farkında olmadan zihninizde zaten o şekilde canlandırıyorsunuz. Diğer taraftan, kavramı tanımlamak bizi dizinin dizinin disinin dizinin konumu gibi çirkin anlatım kullanmaktan koruyacak.

Peki derecesi olmayan değer olabilir mi ? Tabi ki de hayır. Derecesi yoksa yaşadığı konum da yoktur ve yani o değer yoktur. Olmayan bir değerin derecesi olabilir mi ? Hayır.

Diyelim ki **‘\_argümanlar’** değerine ‘\*\*\_argümanlar’ ifadesi ile atıfta bulundunuz. İlkin **‘\_argümanlar’**’ın yaşadığı konuma, sonra orada tutulan konumdaki diziye, sonra ise dizinin ilk harfine gideceksiniz.

Örneğe tekrar dönelim: Elimizde 3 dereceli değer var. Yani bir dizi konumu var ve o konumda diğer dizilere olan konumlar tutuluyor. Peki ben bir derece indirsem yani **‘\_argümanlar’** değerinin yaşadığı yere gitsem ve bunu farklı bir değere atayıp konumunu artırarak içteki 2 dereceli dizileri yazdıra bilir miyim ? Evet ve yukarıdaki örnek tam olarak bunu yapıyor.

Not

Tekrar tekrar belirtmek isterim: Tüm değerler zaten bir hafızadaki bir konumda yaşıyor. Çoğul dereceli değerlerle o konumları her hangi bir değer ile saklayabiliyoruz.

Fark edeceğiniz üzere konumlar ve diziler birbirleri ile bir şekilde ilişkili ama sanki ilişkiyi kuracak kavramı bilmiyoruz ki konum kavramının ve dizilerin birbirlerine karıştırılmasında yatan ana nedenlerden birisi. Konumlar bir kavram. Konumlarla ilgili gerçek olan tek veri özlerinde doğal sayı değerleri olması ama diziler ise yazılımda en az d64 yapıtaşı kadar gerçek bir veri türü.

##### Diziler:[#](#diziler)

Hafızada sıralı ve ardışık yer alan üyelerinin bir önemi olmaksızın doğrusal veri yapısına sahip temel türe denilir.

iş DiziÖrneği  
{  
 stdio::print("\nDizi Örneği:\n");  
 değer tekBoyutlu t8[8] =  
 [65, 66, 67, 68, 69, 70, 71, 0];  
 değer ikiBoyutlu t8[4,2] =  
 [[65, 66],  
 [67, 68],  
 [69, 70],  
 [71, 0]];  
 değer üçBoyutlu t8[2, 2, 2] =  
 [[[65, 66], [67, 68]],  
 [[69, 70], [71, 0]]];  
 stdio::print("-> '%s'\n-> '%s'\n-> '%s'\n",  
 &tekBoyutlu[0],  
 &ikiBoyutlu[0,0],  
 &üçBoyutlu[0,0,0]);  
}

Örnekte yapılan şey gayet basit, **‘tekBoyutlu’** dizisi için **(@tam)\*8** kadar; **‘ikiBoyutlu’** dizisi için **(@tam)\*4\*2** kadar; **‘üçBoyutlu’** dizisi için **(@tam)\*2\*2\*2** kadar bayt hafızası üretim sürecinde ayrılıyor ve dizinin her bir üyesinin içine belirtilen değerler yazılıyor. Arayüzdeki çıktıdan anlayabiliriz ki; hafızada yaşadıkları konum farklı olsa da konumlardaki veri aynı ama tür ifadeleri kullanılarak aynı veri derleyiciye farklı yorumlatılmış. Çünkü üç farklı konumda yaşayan dizinin üçünün de tür bayt genişliği ve sıralaması aynı ve hafızada 32 bayt yer kaplıyor.

C söz dizisinin aksine Örs’de dizi tür ifadesinin sözdizisi farklıdır. C’de yukarıdaki **‘üçBoyutlu’** dizisini tanımlamak için **‘[2][2][2]’** sözdizisini kullanırken; Örs için **‘[2, 2, 2]’** sözdizisini kullanıyoruz. Nedeni ise yazarken virgül koymanın ele daha yatkın olması.

Peki yukarıdaki **‘tekBoyutlu’** dizisinin derecesi kaç ? İki mi ? Külliyen hayır. Orada tek dereceli değer var. Bu değerin içinde sıralı birden fazla değerin olması derecesini arttırmıyor ve konumlar kavramı ile dizi türünün biz tasarımcıların kafasını karıştırıyor. Çünkü işletim sisteminden hafıza alanı istediğimizde onu dizi gibi yorumlayabiliyoruz ve onu en az iki dereceli değerle saklayabiliyoruz. Diğer taraftan diziler ise tek dereceli. Bu nasıl olur ?

Şöyle ki: Varsayalım ki işletim sisteminden doğrusal hafıza alanı istediniz. İşletim sistemi size tek dereceli değeri verdi ve onu yine tek dereceli konuma sakladınız; etti iki dereceli konum değeri. Tanımladığınız dizinin ise yaşadığı hafıza alanının yine bir önceki gibi doğrusal olduğunu varsayalım. Zaten doğrusal bir hafıza alanında tek dereceli olarak tanımlı ve bir değerde saklı; etti bir dereceli dizi değeri.

Anlamsız geliyor olabilir. Bu ayrım o kadar temel ki beyinde oturması yıllar alıyor. Bu manada sizlerle bir diğer hayati kavramı daha tanıştırmalıyım:

##### Boyutlar:[#](#boyutlar)

Geometrideki ya da doğrusal cebirdeki boyut kavramının yazılıma izdüşümü denilebilir ama aynısı denemez. Mesela yukarıdaki **‘\_argümanlar’** konumunda harf dizisi matriksi var ve biz bunun derecesini 2 düşürdüğümüzde (**’\*\*\_argümanlar’**); matriksteki [0,0] hücresindeki harfe erişiyoruz.

Tıpkı nasıl geometride boyutlar var ise konumlarda içlerinde bulundurdukları veriye göre boyut anlamı kazanır ve işletim sistemi bu veriyi yapılandırılmış vereceği için dizilere erişir gibi erişilebilir. Bu yönüyle yukadıraki örneği aşağıdaki gibi yazabiliriz:

iş Giriş2  
argümanSayisi tam,  
\_argümanlar \*\*t8  
{  
 //argümanları konumlarından yazdır.  
 her i := 0; i < argümanSayisi; i++:  
 stdio::printf(  
 "=> konum değeri: %p, konumdaki değer: %s\n",  
 &\_argümanlar[i],  
 \_argümanlar[i]);  
}

Bir önceki örnekle özünde aynı şeyi yapıyor ama fark ettiğiniz üzere ‘argümanSayısı’ değeri bizim gezeceğimiz konumları sınırlıyor ve taşma tehlikesinin de önüne geçiyor. Olmaz ama diyelim ki konumu arttırarak ilerlediğimiz ‘Argüman’ değeri boş konuma denk gelmedi. O zaman işletim sisteminin yazılıma ayırdığı bölgenin dışına taşacağı için meşhur **‘SIGSEGV’** hatasını verecek ve 139 kodu ile sonlandırılacak. Şunu da eklemek gerekir ki; gözlemlerimden çıkardığım sezgisel sonuca göre pek çok yazılımcı ilk dereceyi unutup boyut ve derece kavramını karıştırıyor. Kavramları karıştırarak tasarım yapmaya çalışmak da doğal olarak hatalı sonuçlar doğuruyor. Bunlar göz önünde tutulunca ikinci yöntem daha güvenli. Ayrıca veri türü doğrusal olarak hem derleyici hem yazılımcı tarafından daha yapısal yorumlanıyor ve boyutlar derecelerin aksine elle tutup gözle görebileceğimiz bir kavrama dönüşmeye başlıyor. Doğal olarak tasarım bir kere cisimleşmeye başlayınca; olgunlaşıyor.

Konu konum kavramı iken eklenmesi gereken bir diğer mevzu bu örnekteki ‘i’ değerinin de bir konum olması. Dereceli değerlerle olan farkı dereceli değerler işletim sistiminin belirlediği standartlara uyarak hafızayı konumlandırıp erişirken; ‘i’ değeri doğrusal yaklaşımla hafızaya erişiyor olması. Daha önce bahsettiğimiz sıralı numaralı boncuk dizisi örneğindeki konumlandırma yaklaşlaşımı ile buradaki ‘i’ değerinin yaklaşımı özlerinde aynı ve iki örnekte de doğrusal konumlandırma yapılıyor.

**‘Giriş2’** işlemindeki tek fark bu da değil. İkinci örnekteki print’in ikinci argümanı için yer alan **(&\_argümanlar[i])** ifadesinde konum alma işlemi kullanılırken, öncekinde üçüncü dereceden **(Argüman)** saf ifadesi yer almış ve ayrıca üçüncü argümandaki **(\*Argüman)** ifadesi ise **‘i’** kadar ilerlenmiş konumun tuttuğu değeri dönüyor. Sanki **‘&’** (konumlandırma işlemi) ile **‘\*’** (derecelendirme işlemi) birbirinin ters işlemi gibi hissediliyor ve yukarıdaki **tanımlı kavramların sınırında**; evet iki işlem birbirinin tersi oluyor. Konumlandırma işlemi var olan değerin konumunu getirirken; **‘\*’** işlemi var olan dereceli değerin tuttuğu konumun derecesini düşürüp; düşürülen değeri dönüyor.

Ama bu ikisi de tekiller ve biz sıradaki konuma ulaşmak için tutulu konumu arttırmak ya da azaltmak zorundayız ki yukarıda anlattığımız gibi yan etkileri olan bir yöntem. Bu bağlamda dizi erişim işlemi gücünü gösteriyor. Dizi erişim işlemiyle, derecelendirme işlemi benzer görevleri paylaşsalarda aynı işlem değiller ama **konumdaki veriler ardışık** ise boyut kavramı hafızanın o **ardışık** parçasına diziymiş gibi erişmemizi sağlıyor. Mesela:

iş BoyutKavramıÖrneği  
{  
 // işletim sisteminden ((t32'nin boyutu) X 20) bayt kadar  
 // hafıza bölgesi istiyoruz.  
 Sayılar := yeni(t32[20]);  
 // işletim sisteminin bize temizlemeden  
 // verdiği bölgedeki t32'leri yazdırıyoruz:  
 her i := 0; i < 20; i++:  
 stdio::print("%d'ıncı pis değer : %d; konumu : %p\n",  
 i, Sayılar[i], &Sayılar[i]);  
 // işimizi bitirdik ve bize atanan hafıza bölgesini  
 // işletim sistemine geri veriyoruz.  
 sil Sayılar;  
}

Derecelendirme işleminin soyutluğu; erişim işlemi ile kayboluyor ve biz zihnimizde işletim sisteminden gelen bölgenin aslında bildiğin t32’li boncuklar dizisi olduğunu canlandırmaya başlıyoruz ve bu noktada konumlar ve dizilerin birbirleri ile ilişkisi belirmeye başlıyor.

Not

Peki şu vaka ile karşılaşabilir miydik ? Biz işletim sisteminden yeni bölge istedik ama gelen bölgedeki verilerin konumları ardışık değil. Linux ve Freebsd’de kesinlikle öyle değil ama diyelim ki öyle biz nasıl dereceler kavramını kullanarak ileri, geri konum aritmetiği yapacaktık ? Basitçe söyleyeyim yapamayacaktık. Eğer veri ardışık değilse zaten konum aritmetiği yapmanın ne anlamı ve güvenilirliği var. Fakat boyutlar ile dereceler ve konumlar ile diziler arasındaki kavram ilişkisi devam edecekti. Yine konumlara derecelendirerek erişecektik ve yine dizilere boyutlandırarak erişecektik. Sadece dereceli erişim yavaş olacaktı ki konum erişimi destekleyen canlı dillerde (C#, OCaml vs) konum erişimi yavaş olacaktı. Çünkü işletim sistemi ile sizin yazılımınız arasında hafıza yönetimi yapan temizlik algoritması var ve her konuma eriştiğinizde zaten işletim sisteminin sanal olan konumlarını bir de temizlik algoritmasının sanallaştırdığı konumları çözümlemeyi eklemek zorunda kalacaktık ve bu bize hız kaybettirecekti.

O yüzdendir ki sistem dillerinin neredeyse tamamı hem temizlik algoritmalarını, hem de sanal makinaları ve diğer aracı algoritmaları reddeder. Çünkü bu seviyede bir hız kaybı işlevsel olarak kabul edilemez. Yazılımlarımızın öngörülebilir olmasını istiyoruz çünkü sistem dilleriyle genelde sürücüler, sanal makineler, işletim sistemleri, veritabanları vs gibi hayatî yazılımların en kötü senaryolarındaki hızları üzerinden hayatî hız hesaplamaları yapıyoruz ve her bir algoritmayı eklediğinizde, bu algoritmanın doğası nasıl olursa olsun, öngörülebilirlikten ödün veriliyor.

Bu durumda şunu diyebilir miydik ? “Zaten işletim sistemleri ardışık verileri bize istediğimizde veriyor, hem sanal makinaların faydalarından yararlanır hem de sistem dillerinin hızlı doğasının tadını çıkarırdık.” Bunu yapmak tabi ki de mümkün ve zaten pek çok dil yapıyor. Yukarıda andığım C# çok iyi tasarlanmış; yıldırım hızında çalışan dillerden bir tanesi ama mesele o değil. Sistem dillerinin tasarım amacı bilgisayarla aracısız konuşmak. Yani malı toptancıdan değil fabrikadan alıyormuşsunuz gibi düşünün.

Daha önemlisi hız kazanmak için olguları ve sorunları olabildiğince doğrusal düşünmek gerekiyor. Yukarıdaki t32X20’lik dizide ne yaparsan yap o diziyi gezmek için mecburen bir döngü kullanacaksınız ve O(n) öngörüsünde çalışacaksınız. Yani veri türünüz algoritmayı belirliyor. 2 boyutlu doğası olan bir tür için zorlasanız zorlasanız O(nlog(n)) senaryosunda algoritma geliştirebilirsiniz ki bu şu anlama geliyor elinizdeki veriye göre algoritma O(n^2) gibi de çalışabilir. Özet olarak kullandığınız tür algoritmanızı belirliyor.

Derleyicide gerçekleşen şey ise çok basit, derecelendirme işlemi konum aritmetiği yapmadan dereceyi düşürüp tutulu konumdaki veriyi getiriyor; dizi gibi yorumladığı konumlar için ise konum aritmetiği yapıp derecesini düşürerek veriyi dönüyor.

##### Çeviriler:[#](#ceviriler)

Boyut ve derece kavramlarının kesiştiği yeri tartıştık ama bir soruyu cevaplamadık. Dizi erişim işlemi bir konumdan diğerine nasıl ilerliyor ya da geriliyor ? Basitçe cevap vereyim; genişliğine ve sıralama katsayısına göre.

::

Bu iki kavramı hafıza yönetimi başlığı altında ayrıntılı bir şekilde tartışacağız ama burada ‘sıralama’ kavramını hafıza yönetimindeki ilişkilerinden bağımsızmış gibi düşünmemiz gerekiyor.

İyi güzel de **‘yeni’** işlemi özünde posix sistem çağrılarından biri olan **‘malloc’** işlemine gönderme yapıyor ve o işlem bize **t8** dizisi dönüyor. Nasıl oluyor da **t8** dizisini bir türmüş ya da diziymiş gibi kullanıyoruz. ? İşletim sisteminden alınan hafıza bölgesini, Örs derleyicisine hedef türün genişlik ve sıralama bilgilerine göre yorumlattırarak. Nesneler derlenirken, Örs onların genişlik ve sıralama bilgilerinin çok sıkı muhasebesini yapıyor. Yukarıda **‘yeni(t32[20])’** işlemi çağrısının sade posix malloc çağrısından farkı bu muhasebeden kaynaklanıyor.

Çeviri işlemi dönülen değere istenilen sıralama ve genişlik bilgilerinin muhasebesini ekliyor ki taşma, yanlış okuma, boş konum vs gibi ciddi sorunlar derleme süresince yakalansın ve eğer muhasebe doğru yapılmıyorsa uyarı verilsin. Bu yönüyle çeviri işlemi gerçek işlem değildir. Derleyicinin genişlik ve sıralama muhasebesinin yönlendirilmesidir. ‘BoyutKavramıÖrneği’ işlemini yeniden yazarak gösterelim:

iş BoyutKavramıÖrneği2  
{  
 Sayılar := <\*t32>stdlib::malloc(20\*@t32);  
 her i := 0; i < 20; i++:  
 stdio::print("%d'ıncı pis değer : %d; konumu : %p\n",  
 i, Sayılar[i], &Sayılar[i]);  
 sil Sayılar;  
}

Birinci örnekle özlerinde aynı ama çeviri işlemiyle **‘Sayılar’** değerinin **t32** konumu olduğu belirtiliyor. Diğer taraftan **‘yeni’** işlemi ise bu görevi gözlerden ırak bir şekilde zaten yapıyor ama gerçekleşen şey, sonuç, çıktı aynı.

Konum çevrilerini sadece kaynak isterken değil, elimizdeki kaynaklar yorumlarken de kullanabiliriz. Mesela:

iş KonumYorumuÖrneği  
{  
 //onaltılık ascii değerleri içeren bir t64 sayısı başlatıyoruz.  
 //ilk sekiz baytın 0 olduğunun altını çizerim.  
 değer sayı t64 = 0x0\_47\_46\_45\_44\_43\_42\_41;  
  
 // t64 konumunu yeni değere atıyoruz.  
 SayıKonumu := &sayı;  
  
 //t64 konumunu, t8 konumuna çeviriyoruz  
 Harfler := <\*t8>SayıKonumu;  
  
 //Sonuçları yazdırıyoruz.  
 stdio::print("Sayı konumu: %p\nHarfler konumu: %p\nHarfler içeriği %s\n",  
 SayıKonumu, Harfler, Harfler);  
}

Burada t64 sayı değerinin konumu harfler dizisiymiş gibi yorumlanıyor. Derleyici ise 8 bayt genişliğindeki ve 8 bayt sıralamasındaki t64 konumunu, 1 bayt genişliğinde ve sıralamasındaki t8 değerleri tutan Harfler konumuna çeviriyor. Bu konunun altı belirgin bir şekilde çizilmeli, çevirme işleminin bilgisayar, işlemci, işletim sistemi için anlamı yok. Derleyiciler için var ki değerlerin konumlarını üretim sürecinde hesaplayabilsinler.

Mesela:

iş ÇeviriÖrneği1  
{  
 değer sayı t64 = 0x0\_47\_46\_45\_44\_43\_42\_41;  
 SayıKonumu := &sayı;  
 Harfler := <\*t8>SayıKonumu;  
 Tamlar := <\*t32>SayıKonumu;  
 Simgeler := <\*t16>SayıKonumu;  
 stdio::printf(  
 "Sayı Konumu: %p\nHarfler konumları:\n [0]:%p\n [1]:%p\n [2]:%p\n [3]:%p\n [4]:%p\n [5]:%p\n [6]:%p\n [7]:%p\n",  
 SayıKonumu,  
 &Harfler[0], &Harfler[1],  
 &Harfler[2], &Harfler[3],  
 &Harfler[4], &Harfler[5],  
 &Harfler[6], &Harfler[7]);  
  
 stdio::printf(  
 "Sayı konumu: %p\nSimgeler Konumları:\n[0]:%p\n[1]:%p\n[2]:%p\n[3]:%p\n",  
 SayıKonumu,  
 &Simgeler[0], &Simgeler[1],  
 &Simgeler[2], &Simgeler[3]);  
  
 stdio::printf(  
 "Sayı konumu: %p\nTamlar Konumları:\n[0]:%p\n[1]:%p\n",  
 SayıKonumu,  
 &Tamlar[0], &Tamlar[1]);  
}  
  
/\*  
 Benim bilgisayarımda böylesi bir çıktı veriyor:  
 Sayı Konumu: 0x7ffcc51683d0  
 Harfler konumları:  
 [0]:0x7ffcc51683d0  
 [1]:0x7ffcc51683d1  
 [2]:0x7ffcc51683d2  
 [3]:0x7ffcc51683d3  
 [4]:0x7ffcc51683d4  
 [5]:0x7ffcc51683d5  
 [6]:0x7ffcc51683d6  
 [7]:0x7ffcc51683d7  
 Sayı konumu: 0x7ffcc51683d0  
 Simgeler Konumları:  
 [0]:0x7ffcc51683d0  
 [1]:0x7ffcc51683d2  
 [2]:0x7ffcc51683d4  
 [3]:0x7ffcc51683d6  
 Sayı konumu: 0x7ffcc51683d0  
 Tamlar Konumları:  
 [0]:0x7ffcc51683d0  
 [1]:0x7ffcc51683d4  
\*/

İşlemin çıktısından farkedeceğiniz üzere, **‘Harfler’** dizisindeki konumlar teker teker, **‘Simgeler’** disisindeki konumlar çifter çifter, **‘Tamlar’** dizisindeki konumlar ise dörder dörder artmış. Derleyici üretim aşamasında genişlik ve sıralama muhasebesini tutuyor ki, dizi erişim ve diğer konum işlemleri canlı çalışırken görevlerini verilerin ardışıklığını ve sıralamasını bozmadan icra edebilsinler.

Not

Bu örnekte bayt genişliği ve sıralamanın, dizi erişiminin ibresiyle birebir hareket etmesi, kullanılan türlerin yapıtaşı olmasından kaynaklanıyor. Türler için bayt genişliği ve sıralama illa ki aynı olacak diye bir kaide yok.

##### Şey:[#](#sey)

**‘şey’** yapıtaşı; **‘mimari’** genişliğinde olan ve içinde herhangi dereceden herhangi konumu tutan veya tutmayan yapıtaşıdır.

Tıpkı nasıl ardışık türlerin ayağını yere bastırmak için dizi türünü tanımladıysak, konum kavramının da ayağını bir yapıtaşı ile yere değdirmemiz gerekiyor.

C dilindeki dengi **‘void\*’** türüdür ama Örs dilinde

##### Sonuç:[#](#sonuc)

Konumlar konusunda ne kadar çok kavram kümesiyle iletişimde bulunduğumuzu gördünüz. Yeni başlayanlar için göze korkutucu gelebiliyor ama ustalar için konumlara ayar çekebilme yeteneği büyük bir güç ve Örs bu konuda üzerinde iyi düşünülmüş araçlara sahip.

#### Metinler[#](#metinler)

##### Harf[#](#harf)

Hepimizin bildiği 8 bit genişliğinde ascii simgeleridir. Unicode ve daha geniş verileri kabul etmezler. Varsayılan türleri harf yani ‘t8’ türüdür.

Tıpkı diğer C ailesi üyeleri gibi ‘a’ şeklinde ifade edilirler. Mesela:

değer a harf = 'a';  
b := 'b';  
c := 0x63\_st8;

##### Harfler[#](#harfler)

Boyutu durağan belirlenmiş harf dizileridir. Utf8 ve Unicode kabul ederler. Sıralandıklarında tek bir dizi gibi kabul edilirler.

değer ö \*t8 = "örs dili";  
sıralı := "Sıralandıklarında"  
 "tek bir dizi gibi"  
 "kabul edilirler";

Örs’de harfler türü mimari türünün bayt boyutunun katlarıdır ve sıralıdır.

#### Temel İşlemler[#](#temel-islemler)

Yazılım dünyasında işlevi yönünden özüne indirgenmiş, görevi önceden tasarlanmış ifadelere denir.

Bu indirgemedeki hedef temel işlemin en az makina dili ile işlemci için ifade edilmesi, eğer indirgenemiyorsa mümkün olduğunca durağanlaştırılması; yani derleme sürecinde o işlemlerin sonuçlandırılmasıdır. Özüne indirgenmiş deyimiyle kastedilen budur.

Bazı ifadeler ise yazılımlarda tekrar eder. Mesela çok fazla geçirme işlemi yapan algoritmalarda bu geçirme işlemi görevleri önceden tasarlandığı haliyle temel işlem olarak kullanılabilir. Görevi önceden tasarlanmış deyimiyle ise kastedilen budur.

Canlı temel işlemler; işlemci tarafından çalıştırılabilirken; durağan temel işlemler üretim sürecinde sonuçlandırılır. İlki işlemciyi hedef alırken; durağan temel işlemler derleyici hedef alır.

Canlı temel işlemlerin makina dilinde birebir karşılığı bazen olabilir. Matematik, Mantık, Konum, Erişim sınıfındaki işlemlerin genelde işlemci komutu karşılığı vardır. Diğer sınıfta canlı işlemler malesef aynı görevi üstlenmiş işlemci komutu almayabilir. Mesela **‘+=’** önce değer ile yapıtaşını toplayacak sonra ise sağdaki değere geçirecek. Bu temel işlemin işlemci komutlarında karşılığı olsaydı zaten yazılım dillerine gerek olmazdı.

Uyarı

Tekrar belirtelim; yapısal dillerdeki temel işlemlerin hepimizin bildiği matematikteki temel işlemlerle alakası bile yoktur.

Hatta matematikten ödünç alınmış işlemlerin bile bize lise ve üniversitede öğretilen işlemlerle alakası yoktur. Bize sürekli matematik öğretilmiş iken bilgisayar dünyasındaki matematik işlemleri sıralı matematiğin alanına (ingilizcesi:discrete mathematics) girer.

Örs dili C ailesinin temel işlemlerini değiştirmemesine rağmen kendi içinde yeni temel işlemler de ekler.

C++ dilinin aksine Örs temel işlemleri yeniden tanımlatmaya izin vermez. Nedeni ise yazılım dünyasında bu konuda bir anlaşmanın olmaması ve herkesin fevri simgelerle bir takım işlemleri kolaylaştırma çabasının kodu okunamaz hale getirmesidir.

C dil ailesinde tüm temel işlemler sola doğru döndüğü için Örs’de bu kaideye uyar.

Temel işlemler çeşitli özelliklerine göre sınıflandırılabilir. Aşağıda söz dizimine göre sınıflandırarak bu işlemleri tanıtacağız.

##### İkiz işlemler:[#](#ikiz-islemler)

**‘(ifade) işlem (ifade)’** söz dizisiyle sağ ve sol ifadeler bekleyen işlemlerdir:

**Aritmetik:**

* **‘+’**, **‘-’** simgeleri sağ ve sollarındaki sayısal ifadeleri sırasıyla toplar ve çıkarır.
* **‘\*’**, **‘/’** simgeleri sağ ve sollarındaki sayısal ifadeleri sırasıyla çarpar ve böler.
* **‘%’** simgesi solundaki sayı türündeki değer sağındaki sayı türündeki değere bölününce kalan değerini verir.

**Bit:**

* **‘&’**
* **‘|’**
* **‘^’**
* **‘~’**
* **‘<<’**
* **‘>>’**

**Mantıksal:**

* **‘&&’**, **‘ve’** simgeleri sağ ve sollarındaki sayısal değerleri mantıktaki ‘ve’ işlemi ile karşılaştırır.
* **‘||’**, **‘veya’** simgeleri sağ ve sollarındaki sayısal değerleri mantıktaki ‘veya’ işlemi ile karşılaştırır.

**Karşılaştırma:**

* **‘<’**: sayı türündeki değerleri niceliklerine göre karşılaştırır ve sağı küçükse 1 döner.
* **‘>’**: sayı türündeki değerleri niceliklerine göre karşılaştırır ve sağı büyükse 1 döner.
* **‘<=’**: sayı türündeki değerleri niceliklerine göre karşılaştırır ve sağı küçükse 1 döner.
* **‘>=’**: sayı türündeki değerleri niceliklerine göre karşılaştırır ve sağı büyükse 1 döner.
* **‘==’**: sayı türündeki değerler eşit ise 1 döner.
* **“!=”**: sayı türündeki değerler eşit değil ise 1 döner.

**Atama:**

* **‘+=’**: sağındaki değeri solundakiyle toplar ve soldaki değere geçirir.
* **‘-=’**: sağındaki değeri solundakiyle çıkarır ve soldaki değere geçirir.
* **‘\*=’**: sağındaki değeri solundakiyle çarpar ve soldaki değere geçirir.
* **‘/=’**: sağındaki değeri solundakiyle böler ve soldaki değere geçirir.
* **‘%=’**: sağındaki değeri solundakiyle kalma işlemini uygular ve soldaki değere geçirir.
* **‘<<=’**: sağındaki değeri solundakiyle sola kaydırma işlemini uygular ve soldaki değere geçirir.
* **‘>>=’**: sağındaki değeri solundakiyle sağa kaydırma işlemini uygular ve soldaki değere geçirir.
* **‘||=’**: sağındaki değeri solundakiyle ‘veya’ işlemini uygular ve soldaki değere geçirir.
* **‘&&=’**: sağındaki değeri solundakiyle ‘ve’ işlemini uygular ve soldaki değere geçirir.
* **‘~=’**: sağındaki değeri solundakiyle ‘bit tersleme’ işlemini uygular ve soldaki değere geçirir.
* **‘&=’**: sağındaki değeri solundakiyle ‘bit ve’ işlemini uygular ve soldaki değere geçirir.
* **‘|=’**: sağındaki değeri solundakiyle ‘bit veya’ işlemini uygular ve soldaki değere geçirir.
* **‘^=’**: sağındaki değeri solundakiyle ‘ters bit veya’ işlemini uygular ve soldaki değere geçirir.

**Hafıza:**

* **‘<-’**: Sağındaki konumun değerini dönüp; solundaki değere geçirir.
* **‘<=>’**: Sağ ve soldaki konumların değerlerini değiştirir.

**Arama:**

* **‘::’**: Birimde tanımlı nesneleri arar ve bulursa döner. Bulamazsa hata verip derlemeyi sonlandırır.

**Erişim:**

* **‘->’**: Soldaki türün konumuna erişir ve sağındaki aranan ifadeyi tür üyesi olarak bulabilirse döner.
* **‘.’**: Soldaki türün değerine erişir ve sağındaki aranan ifadeyi tür üyesi olarak bulabilirse döner

##### Tekil İşlemler:[#](#tekil-islemler)

**Adım:**

* **‘(ifade)++’**: solundaki sayısal ya da konum ifadesinin değerini, ifadeyi değerlendirdikten sonra, bir birim arttırır.
* **‘++(saf ifade)’**: sağındaki sayısal ya da konum saf ifadesinin değerini, ifadeyi değerlendirmeden önce, bir birim arttırır.
* **‘(ifade)--’**: solundaki sayısal ya da konum ifadesinin değerini, ifadeyi değerlendirdikten sonra, bir birim azaltır.
* **‘--(saf ifade)’**: sağındaki sayısal ya da konum saf ifadesinin değerini, ifadeyi değerlendirmeden önce, bir birim azaltır.

**Konum:**

* **‘\*(konum ifadesi)’**: Solunda verili konumun değerini döner.
* **‘&(değer ifadesi)’**: Solunda verili değerin konumunu döner.
* **‘(konum ifadesi)[x]**: Solunda verili konumun x’inci sıradaki değerini döner.

**Tür nicelik:**

* **‘@(ifade)’**: Solunda verili ifadenin bayt boyutunu döner.
* **‘$(ifade)’**: Solundaki verili ifadenin bayt sıralamasını döner.

**Tür:**

* **‘%(ifade)’**: Sağındaki ifadenin türünü döner. Üretim sürecinde anlamlı olsa da Örs durağan bir dil olduğu için; canlı anlamı yoktur.

Tanımlı yapıtaşları:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Temel İşlemler** | İsim | Tür | Makina Satırı Sayısı | Görev | İşleme durumu | Okuma Yeri | Eş Anlamlısı |
| **Aritmetik** |  |  |  |  |  |  |  |
| **+** | artı | yapıtaşı | 1 | aritmetik | canlı | orta |  |
| **-** | eksi | yapıtaşı | 1 | aritmetik | canlı | orta |  |
| \* | çarpı | yapıtaşı | 1 | aritmetik | canlı | orta |  |
| **/** | bölme | yapıtaşı | 1 | aritmetik | canlı | orta |  |
| **%** | kalma | yapıtaşı | 1 | aritmetik | canlı | orta |  |
| **Bit** |  |  |  |  |  |  |  |
| **&** | bit ve | yapıtaşı | 1 | bit | canlı | orta |  |
| **|** | bit veya | yapıtaşı | 1 | bit | canlı | orta |  |
| **^** | bit ters veya | yapıtaşı | 1 | bit | canlı | orta |  |
| **~** | bit tersleme | yapıtaşı | 1 | bit | canlı | orta |  |
| **<<** | sola kaydırma | yapıtaşı | 1 | bit | canlı | orta |  |
| **>>** | sağa kaydırma | yapıtaşı | 1 | bit | canlı | orta |  |
| **Mantık** |  |  |  |  |  |  |  |
| **&&** | ve | yapıtaşı | satır > 1 | mantık | canlı | orta | ve |
| **||** | veya | yapıtaşı | satır > 1 | mantık | canlı | orta | veya, ya\_da |
| **!** | değil | yapıtaşı | 1 | mantık | canlı | orta |  |
| **Karşılaştırma** |  |  |  |  |  |  |  |
| **<** | küçüktür | yapıtaşı | 1 | karşılaştırma | canlı | orta |  |
| **>** | büyüktür | yapıtaşı | 1 | karşılaştırma | canlı | orta |  |
| **<=** | küçük eşittir | yapıtaşı | 1 | karşılaştırma | canlı | orta |  |
| **>=** | büyük eşittir | yapıtaşı | 1 | karşılaştırma | canlı | orta |  |
| **==** | eşitlik | yapıtaşı | 1 | karşılaştırma | canlı | orta |  |
| **!=** | eşit değildir | yapıtaşı | 1 | karşılaştırma | canlı | orta |  |
| **Adım** |  |  |  |  |  |  |  |
| **(..)++** | arka artış | yapıtaşı, konum | satır > 1 | adım | canlı | son |  |
| **++(..)** | ön artış | yapıtaşı, konum | satır > 1 | adım | canlı | ön |  |
| **(..)--** | arka azalış | yapıtaşı, konum | satır > 1 | adım | canlı | son |  |
| **--(..)** | ön azalış | yapıtaşı, konum | satır > 1 | adım | canlı | ön |  |
| **Konum** |  |  |  |  |  |  |  |
| \* | derecelendirme | konum | 1 | konum | canlı | ön |  |
| **&** | konumlandırma | konum | 1 | konum | canlı | ön |  |
| **[]** | dizi erişim | konum | 1 | dizi, konum | canlı | son |  |
| **Nicelik** |  |  |  |  |  |  |  |
| **@** | boyut | tüm türler | 0 | nicelik | durağan | ön |  |
| **$** | sıralama | tüm türler | 0 | nicelik | durağan | ön |  |
| **Atama** |  |  |  |  |  |  |  |
| **=** | eşittir | tüm türler | satır > 1 | atama | canlı | orta |  |
| **+=** | artı eşittir | yapıtaşı | satır >= 2 | atama | canlı | orta |  |
| **-=** | eksi eşittir | yapıtaşı | satır >= 2 | atama | canlı | orta |  |
| **\*=** | çarp eşittir | yapıtaşı | satır >= 2 | atama | canlı | orta |  |
| **/=** | böl eşittir | yapıtaşı | satır >= 2 | atama | canlı | orta |  |
| **%=** | kal eşittir | yapıtaşı | satır >= 2 | atama | canlı | orta |  |
| **<<=** | sola kay eşittir | yapıtaşı | satır >= 2 | atama | canlı | orta |  |
| **>>=** | sağa kay eşittir | yapıtaşı | satır >= 2 | atama | canlı | orta |  |
| **||=** | veyala eşittir | yapıtaşı | satır >= 2 | atama | canlı | orta |  |
| **&&=** | vele eşittir | yapıtaşı | satır >= 2 | atama | canlı | orta |  |
| **~=** | bit tersle eşittir | yapıtaşı | satır >= 2 | atama | canlı | orta |  |
| **&=** | bit vele eşittir | yapıtaşı | satır >= 2 | atama | canlı | orta |  |
| **|=** | bit yadala eşittir | yapıtaşı | satır >= 2 | atama | canlı | orta |  |
| **^=** | bit ters yadala eşittir | yapıtaşı | satır >= 2 | atama | canlı | orta |  |
| **Tür Erişim** |  |  |  |  |  |  |  |
| **->** | tür alt konuma erişim | türler | 2 | tür erişim | canlı | orta |  |
| **.** | türü indirgeme | türler | 1 | tür erişim | canlı | orta |  |
| **Hafıza** |  |  |  |  |  |  |  |
| **<-** | geçirme | ~ | satır >= 2 | hafıza | canlı | orta |  |
| **<=>** | içerik değiştirme | ~ |  | hafıza | canlı | orta |  |
| **Arama** |  |  |  |  |  |  |  |
| **::** | arama | birimler | 0 | arama | durağan | orta |  |
| **Tür İfadesi** |  |  |  |  |  |  |  |
| **%** | tür alma | tüm türler | 0 | tür ifadesi | durağan | ön |  |

### Satırlar[#](#satirlar)

#### Satırlar:[#](#satirlar)

Yazılım dünyasında meşur bir tartışma konusu var. Gerçekten kılavuzun tamamında bu tartışmaya atıflar var ve değinmemiz gerektiğini düşünüyorum.

Kod kalitesini ya da miktarını **0x0A** ascii numarası ile anlamlandırılan yeni satır imgesi ile ölçebilir miyiz ölçemez miyiz ?

Böyle bir şeyin neden tartışılması gerektiği bir yana; gülünç bir şekilde alakalı isteği iş başvurusunda gördüm. İşveren ilanında “en az on bin satır kod yazmış olan.” diye şart koşmuş.

Burada kastedilen şey ne olabilir ki ? Eğer kastedilen şey "üretiğiniz yazılımlarınızda ne kadar **‘0x0A’** kodu var ?" ise zaten öyle saçma yazılım şirketinde çalışılmaz, o işverenle iş yapılmaz. Mesela jquery.min.js kütüphanesi tek metin satırlı. Örs ve benzeri dillerde hiç satırla yazılım üretmeniz fevkâlade mümkün ve anlamlı.

Diğer taraftan kastedilen **şey** eğer ki eğer bizim yazılım okurken algıladığımız **şey** ise gerçekten anlamlı ve derin doğaya sahip bir **şey** ile karşı karşıyayız ve bu **şey**'in ne olduğu hem Örs derleyicisi için hem de bu kılavuz için çok önemli.

Basitçe örnek vermem gerekirse: Ayak deyince aklınıza gelen ilk **şey** kendi ayağınız mı, masanızın ayağı mı, yoksa sandalyenin ayağı mı ? Tamam, dediniz ki "Kendi ayağım". Kaç tane ayağınız var ? İki. Peki ayak tanımını "Sizin ayakta durmanızı sağlayan uzuv." olarak tanımlayıp tekrar "Kaç ayağınız var ?" diye sorsam ? Elleriniz üzerinde, kafanız üzerinde de ayakta durmanız mümkün. Elleriyle ayakta durarak yürüyen engelli insanlar, kafalarının üzerinde ayakta durup raks eden dansçılar var.

"Ya ne alakası var yazılımla bunun yine delirdi Musa !", değil mi ? Deliliğe daha geliyoruz biraz sabredin. Size daha “işlemci kaç satır görüyor peki ?” diye sorup dumur edeceğim.

Diyelim ki ayak deyince aklınıza masanızın ayağı geldi. Yine yukarıdaki diyalektiği takip edelim. “Masanızın kaç ayağı var ?” diye sorsam ve ben yukarıdaki ayak tanımına uyarak cevap versem. Mesela benim masam kutu tasarımına sahip ve bacak gibi kesilmiş kerestelerle değil levhalarla ayakta duruyor ve 3 farklı levha parçası var ama tasarım kutu tasarımı olduğu için kutunun yan yüzü olarak görürsem 1 parça var.

Tekrar ayak uzvumuza geri dönelim. “Sizin ayakta durmanızı sağlayan uzva ayak denir.” tanımı ile topuklarımın üzerinde ayakta durduğum için topuklarımı da ayak sayabilir miyim ? Peki balerinler parmakları üzerinde ayakta duruyor o zaman Evet ve bunu ısrarla uzatıp iletişim kuramını kullanarak matematikle de kanıtlarım.

Şimdi bizim satır deyince yazılımda algıladığımız **şey** ne ? Bizim gördüğümüz mü ? Yani **0X0A** kodu mu ? Yoksa algıladığımız mı ? Yoksa Örs derleyicisinin gördüğü mü ? Yoksa derleyicinin algıladığı mı ? İşlemcinin çalıştırdığı mı ? Bizim Kasabın et doğradığı mı ? Yoksa hepsi mi ?

Artık internek kanallarından, gazetelere kod satırları haber olunca; yazılım dünyasında kullanılan satır kelimesinin hangi **şey**'e atıfta bulunduğunu belirlememiz ve terim haline getirmemiz gerekiyor.

Önerdiğim tanım: “Yazılım dilleri dünyasında satırlar sıralı ifadelerin anlamlandırılmış kümesidir.”

Örs ve diğer yazılım dilleri de satırları **sıralı ifadelerin kümesi** olarak algılar. Örnek verelim:

her değer i tam = 0; i < 10; i++:  
 her değer j tam = 0; j < 10; j++:  
 a := i + j;

Şimdi yapılan satır tanımıyla birlikte yine yukarıdaki diyalektiği yapmaya çalışalım. Yukarıdaki kod kesitinde kaç satır var ?

Gördüğümüz 3 satır var ama algıladığımız satırları saymaya başlayalım.

Her döngüsü var, sonra bedeni var, o bedende bir döngü daha var ve onun da bedeni var yani dört diyebilir miyiz ? Tutarlı olur mu ? En nihayetinde tutarlı olması lazım ki hem derleyici anlayabilsin hem de bilgisayar çalıştırabilsin.

Peki derleyici kaç satır görüyor ? Bu örnek için (3 + 1) + (3 + 1) satır. Peki ya işlemci kaç satır çalıştıracak ? Yukarıdaki örnek için (3 + 1) X (3 + 1). Yani nam-ı diğer hepimizin bildiği O(n^2). Özünde veri ağaçları olan satırların elamanları derleyici tarafından artarak algılanıyor diğer taraftan işlemci ise katlayarak algılıyor. Bu konuda yanlışım varsa düzeltilmek isterim. Zira ya benim liseyi tekrar okumam gerekiyor ya da biri birisinin alt türevi.

Şimdi malum soruya geldik. İşlemci kaç satır görüyor ?

Yukarıdaki tanıma göre görmüyor bile. İşlemcinin baktığı yerden hepsi aslında mimari biti genişliğinde dizi. Tıpkı Turingin tanımladığı gibi. İşletim sistemi yazdığınızı doğrusal olarak hafızaya geçirecek ve işlemciyi sizin kodunuzu koyduğu konuma yönlendirecek. Hatta yukarıdaki örnek, Örs, C, C++ vb dillerin iyileştiricileri tarafından da satır olarak görülmeyecek. Çünkü yukarıdaki döngünün hiçbir görevi yok. O yüzden işlemcinin calıştıracağı satır derken aslında örtülü bir varsayımda bulunup, derleyicinin iyileştirme seçeneklerinin kapalı oldunu düşünürek söylüyoruz.

Tam da buralarda satır tanımlamasının rengi bozulmaya başlıyor. Kendine verilen alanın dışına çıkıyor. İşte tam bu noktada satır tanımı görevini tamamlamış oluyor. Bu kılavuzda satır kelimesini duyunca aklınıza sadece yukarıdaki tanım gelsin. Bir matematikçinin, işlemci tasarımcısının, hatta ve hatta güvenlik ve iyileştirme kaygısı güden yazılımcının kaç satır algılayacağını bilmenin pek bir imkanı yok. Mesela önceki paragrafta iyileştirme aşaması için yukarıdaki örneğe döngü dedim. Kümelenmiş iki döngü demedim; "… döngünün hiçbir … " dedim. Çünkü iyileştirmenin özünde yan etkileri en aza indirmek var ve gereksiz koddan ziyade yan etki olabilir mi ? Mâna bu olunca burada tek satırlık bir döngü var orada ama iyileştirici algoritması iç döngünün içerisindeki etkisiz atamayı tespit edene kadar satır sayacak.

Bakınız konu o kadar derinleşmeye başladı ki altında ezilmeye başladım. Fark ettiyseniz mâna kelimesini kullandım. Anlam kelimesini kullansaydım aklınıza "Bizim anladığımız mı? Derleyicinin anladığı mı ?" diye sorular gelecekti. Örsün tasarım amaçlarından en güzeli belki de işte bu. Nihayet rüyâ görürken, sinirlenince konuştuğumuz dille bu harika konuları tartışabileceğiz.

#### Döngü Satırları[#](#dongu-satirlari)

Bilgisayar belleğinin aslında devasa bir dizi olduğu düşünülünce, bu dizinin ‘a’ noktasından ‘b’ noktasına gitmeyi istemek kadar temel bir istek olamaz. Tam olarak bu amacı yerine getirmek için Örs dili çeşitli döngü yöntemleri kullanmaktadır.

Döngülerin amacı verili kod dizisini koşul sağlanmayana kadar tekrar tekrar işletmektir. Bu yönüyle motorlara benzerler. En basitinden siz arabayı çalıştırdınız ve arabanın motoru dönmeye başlar ve öyle devam eder ta ki siz motoru kapayıncaya kadar ya da yakıt bitinceye kadar. Yani siz koşulu sağlamayana kadar o motor çalışmaya devam eder.

Burada dikkat edilmesi gereken husus Örs dilinin C dilinden daha fazla yönlendirmeler tanımlaması ve döngüler üzerinde daha fazla denetim sağlamasıdır. Yönlendirmeler konusuna ayrı bir başlık açacağız.

Her Döngüsü:[#](#term-Her-Dongusu)

iş ArgumanlarÖrneği  
 argumanSayisi tam,  
 \_argumanlar \*\*t8: tam  
{  
 // ilk döngü  
 //argümanları dizi erişimiyle yazdır.  
 her değer i tam = 0; i < argumanSayisi; i++:  
 stdio::printf("-> %s\n", \_argumanlar[i]);  
  
 //ikinci döngü:  
 //argümanları konumlarından yazdır.  
 her değer t \*\*t8 = \_argumanlar; \*t; t++:  
 stdio::printf("=> %s\n", \*t);  
}

Yukarıdaki örneğin ilk 4 satırını göz ardı edecek olursak, bu işlem basitçe, önce yazılım argümanlarını dizi erişimi ile daha sonra konum erişimiyle yazdırıyor.

‘Her’ sözcüğü ile başlayan satır içerisindeki ilk satırda değer ya da değerler tanımlanır. Sonrasındaki satırda koşul belirtilir, üçüncü satırda ise tanımlı değerler güncellenir ve her döngüsünün beden satırına geçilir.

Bu kısmı tabî ki de sadece bir söz dizimi. Hepimizin C dilinden eline yapışmış alışkanlık. Bu satırların anlam sıralaması farklı olsaydı yani ilkin güncelleyip, sonra koşulu değerlendiremezmiydik ? Tabi ki de değerlendirebiliriz ama mesele o değil. Örs dili makina diline dönüştüğünde, biz tasarımcılar olarak istiyoruz ki derleyicinin ürettiği kod C derleyicisinin ürettiği kod ile bire bir eşdeğer olsun. Yani her döngüsünü yukarıdaki gibi tanımladığınızda, üretilen makina kodu ile C derleyicisinin ürettiği makina kodu tıpa tıp aynı.

Diğer taraftan tam burada basit bir okuma sorunu ortaya çıkıyor. Özellikle internet, oyun, arayüz uygulamalarında çoğu zaman döngüyü koşullarla değil sayaçlarla yönlendiriyoruz ve 4 satırlık her döngüsü okurken göze batıyor. Bir diğer konu ise yukarıdaki söz dizimi makina dili üretirken de 5 satır kod üretiyor.

C tarzı döngü:

her ; ; güncellemeYapacakİşlem():  
{  
 işletilecekBeden();  
}  
/\*  
 Burası ise devam satırı.  
\*/

Şimdi bu döngü satırına basitçe bir ameliyat yapalım. Değer tanımlanması gereken kısımda değer tanımlanmamış. Yani boş satır var. Koşul satırı yine boş. Güncelleme satırında bir işlem çağırılmış ve bu üretilecek ilk makina dili satırı. Beden kısmında da dolu satır var ve bu da bizim ikinci makina dili satırımız.

Ama, fakat, lakin o yorumun olduğu yerde bir diğer satır daha var. Devam satırı. Makina kodunda devam ederken nereye devam edeceğini söylemen gerekiyor. Örs, C, C++ dillerinde devam satırını siz tabi ki de yazmıyorsunuz ama makina dilinde o satır var ve yukarıdaki döngünün makina dilinde, yani özünde, 3 satırı var.

Peki ama biz işlemci değiliz ki bize ne bundan ? Çok doğru ama biz bu kodu okuyacak olanız. 5 satır tanımlayıp, 3 satır ürettirmenin anlamı yok. Yani argümanlar örneğindeki gibi 4 satırlık döngü tanımladığınızda aslında makina 5 satır kod görecek. Bu örnekte ise tanım ve koşul kısmı boş satır ve derleyici hiçbir makina kodu üretmeyecek.

Peki öyleyse fazladan iki satırı tanımlamanın ne anlamı var ? Burada basit bir görsel soyutlama yapmamız ve her döngüsünün söz dizimini kıvraklaştırmamız gerekiyor.

Her döngüsü söz dizimi çeşitleri:[#](#term-Her-dongusu-soz-dizimi-cesitleri)

Sonsuz döngü:[#](#term-Sonsuz-dongu)

her:  
 herhangiİşlem();

Yukarıdaki örnekte görsel uygulamalarda, veri arz uygulamalarında bolca kullanılan bir sonsuz döngü örneği var. Sadece beden satırında bir takım işlemler çağırılmış. Derleyici 1 satır görecek ve 2 makina satırı üretecek.

Göze sade görünüyor. Derleyiciye sade görünüyor ve makina zaten sade görecek.

Tanımsız döngü:[#](#term-Tanimsiz-dongu)

her devamMı(); güncellemeYap():  
 bedenKısmı();

Fark ettiyseniz değer tanımlaması beklenen satır yok. Zaten iş yaparken çoğu zaman değerler önceki satırlarda tanımlanmış olacak. ‘her’ satırı içinde 2 satır tanımlı. Sırasıyla ilk kısım koşul satırı ve bu kısmın koşul olması bir kaide. İkinci satırda güncelleme kısmı var, sonrasında ise beden satırı ve gizli son satır. Derleyici Üç satır görecek ve dört satır üretecek.

Göze sade görünüyor. Derleyiciye sade görünüyor ve makina zaten sade görecek.

not:

şunu da eklemek isterim, ilk satırda değer tanımlanması sadece bekleniyor. İllaki değer tanımlayacaksınız diye bir kaide yok.

Koşullu döngü:[#](#term-Kosullu-dongu)

değer i tam = 0;  
her i < 10:  
{  
 stdio::printf("-> %d", i);  
 i++;  
 /\*i'yi güncellemek tamamen isteğe bağlı.\*/  
}

Yazılıma başladığınızda ilk göreceğiniz döngü örneği. Görüldüğü üzere ‘her’ sözcüğünden sonra koşul var ve bu Örs derleyicisi için şu anlama geliyor: ‘i’ değeri ondan küçükken döngü bedeni çalıştırılacak ve güncelleme döngü bedeni içerisinde halledilecek ya da halledilmeyecek.

C dilindeki ‘while’ döngüsü ile bire bir aynı kod Örs derleyicisi tarafından üretilecek.

Tüm döngüsü:[#](#term-Tum-dongusu)

değer i tam = 0;  
tüm i < 10:  
{  
 stdio::printf("-> %d", i);  
 i++;  
 /\*i'yi güncellemek tamamen isteğe bağlı.\*/  
}

ark ettiğiniz üzere yukarıdaki koşullu her döngüsü ile bire bir aynı. Ama ‘tüm’ öngü satırının çeşidi yok. C dilindeki ‘while’ döngüsüne karşılık gelen sade bir döngü satırı.

üm döngüsü için 2 satır istenecek ve sonuç olarak 3 satır makina kodu üretilecek.

Sonuç:[#](#term-Sonuc)

‘her’ sözcüğünü döngü satırı tanımlarken seçmemizin amacı türkçede akla yatkın olması. Türkçenin benzersiz matematiksel doğası var.

Mesela aşağıdaki satırı Türkçeye çevirmeye çalışalım:

her değer i tam = 10; i > 0; i--:  
{  
 birşeyler();  
}

‘i’ değerini 10’a eşitleyip; her bir ‘i’ ondan küçük iken; ‘i’ değerini azaltarak ‘birşeyler’ işlemini çalıştır.

Eğer burada söz dizimine odaklanmazsanız ilginç bir ayrıntıyı yakalayabilirsiniz. Türkçede diğer pek çok dilin aksine söz dizimi kuralı yok ve bu durum Türkçeye kıvraklık, ifade yeteneği ve güç katıyor. Yukarıdaki döngü satırını bu denli insan diline bire bir çevireceğiniz ve iyelik ekine kadar makinaya anlatabileceğiniz nadir dillerden bir tanesi Türkçe.

O yüzden ‘her’ sözcüğünü döngü satırlarını tanımlarken seçtim. Peki her sözcüğünü seçmeseydin ? Tabi ki de özünde hiç bir önemi yok. İsteserseniz kendi tasarladığınız dilde 124323 sayısını döngü satırları tanımlamak için bile kullanabilirsiniz.

Basitçe tekrar söylemem gerekirse: “Okuduğumuz kodun, derleyicinin okuduğu kodun ve üretilen kodun olabildiğince örtüşmesini sağlamak.”

Peki buradaki döngü satırlarının makina kodu üretimi konuları diğer yazılım dillerinde öyle değil mi ?

Bazen öyle, bazen değil ve özellikle üretilen kodun iyileştirilme aşamasında boş ya da gereksiz satırlar silinecek.

Burada bu tanımı yapmaktaki amaç gözümüzün gördüğü kod ile Örs’ün ürettiği kodun olabildiğince örtüşmesi ve bunun bir kaide olarak belirtilmesi. Yani diyelim ki Örs onuncu sürümünü yayınladı, bu kaideler değişmeyecek ve bizler nasıl kod üretiyor diye endişelenmeyeceğiz.

Altı çizilmesi gereken diğer konu ise üretilen kodun mümkün olduğunca C dili ile uyumlu olması. Konu döngüler olunca bu istek önemsiz durabilir; en nihayetinde işlem çağrıları dışında üretilen kodun diğer kütüphanelerle iletişimi pek olmuyor ama tepeden tırnağa C dili ile uyum sağlamak Örs’ün tasarım amaçlarından bir tanesi. Tasarımın merkezi kaidelerinden biri bu ve işletim sistemi kütüphaneleri ile rahat rahat iletişim kurabilmek için bu kaideyi tepeden tırnağa bozmayacağız.

#### Koşullu Satırlar[#](#kosullu-satirlar)

Yazılımdaki yönlendirmeleri mantık kullanarak yapabileceğimiz, sıralı bağlanabilen satırlardır.

Koşullu satırların amacı, yönlendirme satırlarını mümkünse işlev dışı bırakıp, bu satırları derleyiciye makina dili ile örtülü şekilde ürettirmek, kodun okunabilirliğini arttırmak, hata ayıklamayı kolaylaştırmaktır.

Koşullu satırlar baş kısımlarında ‘eh’ türlü ya da ‘eh’ türüne çevrilebilen ifade bekler. Eğer ifade ‘eh’ yapıtaşına çevrilemiyorsa derleyici hata verir. Sayısal değerler eğer 0 dan farklı ise eksi ya da arto olmasının bir önemi olmaksızın tıpkı diğer sistem dilllerinde olduğu gibi "evet" yani 1 kabul edilirler; tam tersi durumda ise "hayır" yani 0 kabul edilir ve çevrilirler.

Konumlar da özlerinde sayısal değer olduklarından; eğer ki verili konum 0 değilse "evet" kabul edilir ve konum boş olduğunda ise "hayır" kabul edilirler.

Beden kısımlarında ise herhangi bir satırı kabul ederler.

‘eğer’ satırı yukarıdaki özelliklere uyar, tekil olarak kullanılabilirler ama diğer koşullu satırlardan farklı olarak arkasında koşul sağlanmadığında yönlendirebilecekleri ‘eğerki’ ya da ‘değilse’ satırlarını bekleyebilirler.

‘eğerki’ satırı yukarıdaki özelliklere uyar ama tekil olarak kullanılamazlar. ‘eğerki’ satırının aldığı koşul sağlanmadığında yönlendirebileceği ‘eğerki’ veya ‘değilse’ satırlarını beklerler. ‘eğerki’ satırı bedeni tekil satır değilse ve sonrasında ‘eğer’ satırı gelmişse derleyici hata verir ve süreç durdurulur.

‘değilse’ satırı yukarıdakilerin aksine değerlendirecekleri koşul beklemezler. Koşulların önceki ‘eğer’ veya ‘eğerki’ satırlarında değerlendirildiklerini varsayarlar. ‘değilse’ satırı bu doğası gereği tekil olamaz.

iş ÇiftMi a tam; tam:  
 eğer a % 2:  
 dön hayır;  
 değilse:  
 dön evet;

Yukarıdaki örnekte basitçe görüleceği gibi, eğer ‘a’ değerinin modu bir ise bedenindeki koda yönlenecek değilse; sıradaki ‘değilse’ satırına yönlecek.

Bu örnekte görüldüğü gibi davranışları C ve benzeri dillerdeki ‘if/else ..’ satırları ile bire bir örtüşür.

#### Yönlendirmeler[#](#yonlendirmeler)

##### İşlem yönlendirmeleri:[#](#islem-yonlendirmeleri)

###### ‘git’ satırı:[#](#git-satiri)

İşlemciyi isimlendirilmiş kesite yönlendirir. Genellikle döngüleri yönetilirken kullanılır. Her ne kadar namı kötü duyulsa da özellikle anlamlandırılacak metin ağaçlarını doğrusal alanda işlerken sıklıkla kullanılır.

‘git’ satırı arkasına işlem konumu bekler.

iş OnKereYaz  
{  
 i := 0;  
 #kesit1  
 stdio::printf("ilk kesit\n");  
 #kesit2  
 stdio::printf("ikinci kesit\n");  
 eğer i < 10:  
 {  
 eğer i % 2:  
 git kesit1;  
 değilse:  
 git kesit2;  
 }  
  
}

Bu örnekte sadece nasıl kullanılacağı gösterilmiştir ve ‘git’ satırının uygun kullanımına örnek değildir. Zira yukarıda ‘her’ döngüsü gerekmektedir. Konu ile alakalı sağlıklı örnekler metin işleme algoritmalarından verilebilir ama zaten o algoritmalar dolama gibi olduğundan amacın dışına çıkılacağı düşünülmüştür.

###### ‘dön’ satırı:[#](#don-satiri)

İşlem tanımlandığında belirlenen çıktı olgunlaşınca bu satır ile dönülür. Boş çıktısı olan işlemler için ifade kabul etmezken, aksi durumlarda ifade ister ve istenilen ifadenin konumu işletim yığınına konulur.

iş ÇiftMi a tam: tam =>  
 eğer a % 2:  
 dön hayır;  
 değilse:  
 dön evet;

Yukarıdaki örnekte görüldüğü gibi dönülecek ifadelerin türü ile iş çıktısı aynı olmalıdır. Aksi takdirde derleyici hata verecektir.

##### Döngü yönlendirmeleri:[#](#dongu-yonlendirmeleri)

###### ‘devam’ satırı:[#](#devam-satiri)

İstenilen durumlarda döngüyü sonraki devire atlatır. Tekildir.

###### ‘son’ satırı:[#](#son-satiri)

İstenilen durumlarda döngüyü sonlandırır. Yukarıda tanımlanan ‘git’ satırına muhalefeten tasarlanmıştır.

iş ÇiftseYazdır:  
 her a := 0; evet; a++:  
 {  
 eğer !(a % 2):  
 stdio::printf("Çift sayı");  
 eğerki a >= 10:  
 son;  
 değilse:  
 devam;  
 stdio::printf("a sayısı tekil olduğunda buraya ulaşılamayacak.");  
 }

Gayet okunabilir olan yukarıdaki sonsuz döngüde a çift sayı ise değeri yazdırılacak, a değeri 9’u aştığında ise ‘son’ yönergesiyle sonladırılacak, diğer koşulda ise döngüye diğer kesitlere uğranmadan devam edilecek.

##### Durum yönlendirmeleri:[#](#durum-yonlendirmeleri)

###### ‘tekrar’ satırı:[#](#tekrar-satiri)

İçinde bulunduğu durum satırını her bir ‘tekrar’ satırına uğranış için çalıştırır.

###### ‘geç’ satırı:[#](#gec-satiri)

İçinde bulunduğu seçim satırının ardılı olan seçim satırına atlamak için kullanılır.

iş HarfseYaz \_girdi \*t8  
{  
 i := 0;  
 durum \_girdi[i];  
 {  
 seçim ascii::Boş:  
 dön;  
 seçim  
 ascii::küçükler,  
 ascii::büyükler,  
 ascii::Alt\_Tire:  
 stdio::printf("%c", \_girdi[i]);  
 geç;  
 varsayılan:  
 i++;  
 tekrar;  
 }  
}

Yukarıdaki örnekte \_girdi ascii harfi dizisindeki okunabilir olanlar yazılacak, ve sonraki varsayılan satırında sırası arttırılıp durum ‘ascii::Boş’ sabitini görene kadar tekrar değerlendirilecek.

#### Durum ve Seçimler[#](#durum-ve-secimler)

Dijital tasarımın özü. Hadi şu “dijital” kelimesinden kurtulalım. Zira herkesin aklına fotoğraf makinesi geliyor. Sayısal tasarımın özü: Durumlar.

Durum satırlarının amacı sabit sayı ile ilişkilendirilmiş satırları çalıştırmaktır. **Düz mantıkla** şunu söyleyebiliriz ki derleyici kod kesitlerini diziye koyuyor ve işlemci ise erişilen dizideki kodu çalıştırıyor olarak **kabaca** anlaşılabilir. Özlerinde sıralı kod kesitleri oldukları için matematikteki, işlevsel dillerdeki benzer kavramlarla örtüşmezler.

Durum satırı baş kısmında tam sayı, özel olarak ‘t32’, yapıtaşını bekler ve eğer satırdaki ifade ‘t32’ değilse üretim aşamasında ‘t32’ türüne çevrilir. Özetle eğer siz durumunu seçmek için türü d64 olan bir ifade kullandığınızda bu siz çeviri yapsanız da yapmasanız da derleyici tarafından ‘t32’ türüymüşcesine algılanır ve çeviri yapmadığınız taktirde kendisi çevirir. Bu kısım özellikle ‘sayaç’ kavramı ile bağlantılıdır. Hatırlarsanaz sayaçlar da üye olarak sadece ‘t32’ sabitleri bekler.

Beden kısmında ise **seçim** satırları yer alır. Seçim satırına dahil olmayan satırlar varolduğu taktirde derleyici tarafından gereksiz görülecek ve silinecektir. Seçimler sabit sayılar, sabit sayı dizileri ve sayaç kümeleri beklerler. Değişken bir ifadenin seçim satırı olamaz. Eğer seçim satır başlarında verili ifadeler sabit değilse; derleyici hata verir ve üretimi durdurur. En nihayetinde üretilecek olan kod kesitini değişen sıraya koymanın ne imkanı ne de anlamı vardır.

Seçimler beden olarak seçim satırı hariç her herhangi bir satırı kabul ederler. Bu uygulamada şu anlama gelir: İki seçim sıralandığı takdirde sonraki gelen öncekinin devam kesiti değildir. Aşağıda örnekle tekrar belirteceğiz.

‘**varsayılan**’ satırı özel isimlendirilmiş bir seçim satırıdır. Seçmediğiniz tüm durumlar için varsayılan satırında seçilir. Adı üzerinde bizim ele almadığımız durumlarda yazılımın varsayılan davranışını belirler. Eğer durum satırlarınızda ‘varsayılan’ seçimi belirtmediğiğniz taktirde derleyici sizin varsayılan seçim için boş satır (yani: ‘{}’ simgesi ile belirtilen satır) seçtiğinizi algılar.

iş DurumÖrneği  
{  
 değer gelen tam = Sıfırİle10ArasındaRastgeleSayı();  
  
 durum gelen:  
 {  
 seçim 0:  
 stdio::printf("0 geldi.\n");  
 seçim 2:  
 stdio::printf("2 geldi ve sıradaki seçime geçildi.\n");  
 geç;  
 seçim 1:  
 stdio::printf("1 geldi.\n");  
 seçim 3, 4, 5, 6:  
 stdio::printf("3 <= durum <= 6 geldi.\n");  
 varsayılan:  
 stdio::printf("Seçilmeyen bir durum geldi.");  
 }  
  
 //durumSatırıSonu  
 stdio::printf("örnek tamamlandı.\n");  
}

Yukarıdaki örnekte gelen sayısına sıfır ve 10 dahil ama arasında bir değeri atandı. Bu dakidan sonra yapılacak olan şey sizin belirlediğiniz seçim satırlarını çalıştırmak. Seçim satırları yönlendirilmediği taktirde durum satırı sonuna geçecekler ve görevlerini tamamlayacaklar.

Altını çizerim bu C’deki 'switch/case' satırlarından oldukça farlı ki zaten isim değişikliğinin nedeni bu farklılık. Eğer özel olarak belirtilmezse C dilindeki seçim satırları sıradaki satıra geçiyor. Cidden çok sıkıntılı ve tehlikeli bir özellik. Bazı durumlarda gözden kaçan, veya koymayı unuttuğunuz tek bir yönergenin neden olduğu hataları ayıklamak saatler alabiliyor. Dahası Dahası bazı durumlarda hatayı aylarca fark etmiyorsunuz ve ummadığınız bir zamanda başınıza bela olabiliyor.

İkinci seçimde ise çok özel **'geç'** satırımız var. Gelen sayı 2 olduğunda ‘seçim 2’ satırı çalıştırılacak ama durum sonuna atlamak yerine ‘seçim 1’ satırına geçecek ve tabi ki ‘seçim 1’ satırı işi bitince durum sonuna atlayacak.

##### Durum döngüsü:[#](#durum-dongusu)

Özellikle metin işleme ya da oyun yazılımlarında çok karşılaşacağınız bir sorun. Belirlediğiniz duruma kadar döngü devam etsin istiyorsunuz. Sanal makinalarda genellikle sıfır durumuna kadar döngü bitsin istemiyorsunuz. Mesela:

iş BelgeYaz \_yol \*t8  
{  
 Belge := belge::bayt::Yeni(\_yol);  
 eğer Belge:  
 {  
 sıra := 0;  
 durum Belge.Dizi[sıra]:  
 {  
 seçim 0:  
 varsayılan:  
 stdio::printf("%c",  
 Belge.Dizi[sıra++]);  
 tekrar; //can alıcı kısmı.  
 }  
 }  
 değilse:  
 stdio::printf("%s belgesinin içeriği okunamadı.\n");  
}

Burada yapılan iş basitçe ilkin belge bayt olarak açıldıktan sonra bayt dizisine erişiliyor. Belge sonlarında sıfır olduğu için ‘seçim 0’ durum sonuna atlayıp devam edecek. Diğer tüm seçimler için ise, sıradaki harfi yazdırıp sonra durumu tekrar edecek. ‘tekrar’ sözcüğünün örs için anlamı bu.

Doğal olarak sadece seçim satırlarında yer alabilir ve kümelenmiş durum satırlarında ‘tekrar’ satırı kullanıldığında en son durum satırına atıfta bulunur ve oraya atlar. Bunun dışındaki yerlerde tekrar yönerge satırı anlamsızdır ve hatalıdır. Derleyici uygun kod ile işlemi hatalı sonlandıracaktır.

Şu denilebilir ki “Bu çok tehlikeli bir özellik, ne gereği vardı, döngü kullanılarak da başarılabilirdi” vs. Hepsine "Evet haklısınız" diyorum ama yukarıda belirttiğim gibi bu biraz elinizdeki soruna yönelik bir çözüm. Belirttiğim gibi tekrar satırı metin çözümleme ve oyun tasarımı gibi alanlarda anlamlı ve üretken çözümler sunar.

Döngülerde özellikle her döngüsünde baş satırlar anlamlı olduğu için işlem gücüne mâl olur. Metin sorunları ise doğaları gereği hız isteyenler olduğundan her bir küçük işlemden bile olabildiğince kurtulup hızın sıkıp suyunu çıkarmamız gereken durumlarda Durum döngüleri anlamlı olur.

Tekrar belirtmek isterim gündelik durumlarda beklenmeyen, tehlikeli sonuçlara yol açabilir. En nihayetinde eğer döngü varsa bunu yönergelerle değil döngü satırları ile sağlamak hem okunabilirlik, hem sadelik ilkelerinin taleplerini sağlamak bakımından önemlidir.

Durum ve seçim satırları görevleri gereği C dilindeki ‘switch/state’, işlevsel dillerdeki ‘the guards’ kavramlarına benzer. Yukarıdaki farklılığı belirtmek adına kavramları doğrudan çeviri yöntemiyle isimlendirmek uygun görülmedi. Diğer dillerde ve Örs dilinde ifadenin durumuna göre seçim yapıldığı için konuşma dilindeki sözcüklerden ilham alındı.

### Hafıza[#](#hafiza)

#### Yeni:[#](#yeni)

İşletim sisteminden istenilen boyutta yeni hafıza konumu istemek için kullanılan ifadedir. Sadece ve sadece tür ifadesini kabul ederler ve diğer ifadelerin ise türünü alıp ona göre işlevlerini gerçekleştirirler. Mesela:

değer Sayılar \*tam = yeni(tam[20]);  
// 20 X 4 baytlık dizinin konumunu dönecek  
// ve bu konum sayılar değerinde tutulacak.x  
değer Matriks \*\*tam = yeni(%Sayılar[20]);  
// Sayılar değerinin türü bir konum olduğu için  
// (konum genişliği) x 20 baytlık bir konum dizisi dönecek ve matriks olacak.  
// ama farkettiğiniz üzere matriksin satırları boş.  
değer DoluMatriks \*\*tam = yeni(tam[20, 20]);  
// bu satırda ise yeni ifadesi 20x20'lik tam sayı matriksi oluşturdu.

Yeni ifadesinin C dilindeki karşılığı hepimizin bildiği malloc işlemidir. Mesela:

değer Sayılar \*tam = yeni(tam[20]);  
// yukarıdaki Örs satırı ile; aşağıdaki  
int\* Sayılar = malloc(20 \* sizedf(int));  
// C satırı özlerinde bire bir aynıdır.

#### Temiz:[#](#temiz)

Yeni ifadesinden sonra işletim sisteminin yazılıma tahsis edeceği hafızanın içeriği bilinemez. Bu yüzden genelde C dilinde yazarken hafıza ayrıldıktan sonra onu temizleriz. Tam bu noktada temiz ifadesi anlamlı oluyor. Mesela yukarıdaki örneği tekrar ele alalım:

değer Sayılar \*tam = temiz(tam[20]);

Bu örnekte ise temiz ifadesinin anlamı 20X4 bayt genişliğinde sıfırlanmış tam sayı dizisidir.

#### Yenile:[#](#yenile)

İşletim sistemi tarafından öncesinde ayrılmış kaynağı daraltmak ya da genişletmek için kullanılır. C dilindeki karşılığı realloc işlemidir.

Sil: ^^^ İşletim sisteminin bize verdiği kaynağın görevi sona erdikten sonra eğer ki bilgisayar hafızasının veri çöplüğüyle dolmasını istemiyorsak; kaynağı sahibine geri vermemiz gerekiyor.

‘Sil’ imgesi bir satırdır ve bir değer dönmez.

Silinen konum değerleri boşlanır. Yani konumun yeni sayısal değeri 0’dır.

**Mesela**:

değer Sayılar \*tam = temiz(tam[20]);  
her i := 0; i < @tam[20]; i++:  
 stdio::printf("0 mı : %d\n", (Sayılar[i] == 0));  
/\*  
 sıfır olduğunu gösterdik ve işimiz bitti,  
 şimdi kaynağı geri verme zamanı.  
\*/  
sil Sayılar;  
stdio::printf("Sayılar değerindeki konum : %p\n", Sayılar);  
/\*  
 Canlı hata verilmediyse arayüzde :  
 "Sayılar değerindeki konum : (nil)"  
 gibi bir çıktı alınacaktır.  
\*/

#### Boşalt:[#](#bosalt)

Görevi basitçe konuma gidip içeriğini sıfırlamaktır. İki farklı söz dizimi vardır. İlk söz diziminde; İlk değişkeni konumken, ikincisi boşaltırken o konumda ne kadar ilerleyeceği bilgisidir.

Boşalt imgesi bir satırdır.

**Mesela**:

değer Sayılar \*tam = yeni(tam[20]);  
boşalt Sayılar, 20;

Sayılar dizisindeki tüm tam sayı hücrelerini boşalt satırı sıfırlayacaktır.

İkinci söz dizisi ise tekildir. Konum alıp değerini sıfırlar.

**Mesela**:

değer Sayılar \*tam = yeni(tam[20]);  
boşalt &Sayılar[2];

Bu örnekte ise yukarıdakinin aksine sadece 2 nolu hücredeki tam sayı boşaltılmıştır.

Konumu ve türü bilinen herşey sıfırlanabilir. Burada dikkat edilmesi gereken türün boyutu ve canlılığıdır. Canlı ayrılan kaynaklarda eğer ki ne kadar kaynak ayrıldığı biliniyorsa sorun olmaz ama bilinmiyorsa ve işletim sistemi tarafından belirlenin alanın dışına taşılmışsa işletim sistemi hata verir ve yazılımı ‘SIGSEV’ koduyla sonlandırır.

### İşlemler[#](#islemler)

Eğer belirgin bir görevi yerine getiren satır kümesi var ise bu satır kümesine mantıken işlem denilebilir.

Örs dilinde işlemler sıralı satır kümesi olarak tanımlanır. Haskell, Ocaml vb. dillerin aksine Örs, C, C++ işlevsel değil; sıralı bir dildir. Tasarım, Örs işlemlerinin görevi ve işlevi olduğunu varsayar. Bu konuda yanlış anlaşılmamak gerçekten önemli. Zira yeni öğrenenler sıralı dillerdeki işlem kavramının matematikteki işlem kavramı olduğunu ister istemez düşünüyor. Oysa bilgisayara matematik işlemini tanıtmakla işlevi yerine getirmek bambaşka konular. Yani Örs bire bir lambda matematiği ile örtüşmez. Bunun tasarım nedenleri başlı başına ayrı bir konu ve burada bizi ilgilendiren kavram Örs ve diğer benzeri sistem dillerinde işlemlerin ele alınışının matematikteki işlem (fonksiyon) kavramı ile aynı olmadığıdır.

İşlemler Örs dilinde satır değildir ama beden olarak satır bekler.

**Sade işlemler:**[#](#term-Sade-islemler)

Girdisi ve çıktısı olmayan işlemlere sade işlemler Örs dilinde aşağıdaki gibi tanımlanır.

iş Merhaba:  
{  
 stdio::printf("Merhaba Dünya !");  
}

Örnekte görüldüğü üzere; işlemler tanımlanırken ‘iş’ sözcüğü kullanılarak başlanır. Sonrasında işin derleme birimindeki benzersiz adı verilir ve son olarak ise beden satırı tanımlanır.

Kısa yol olarak tek satırlık işlemler de aşağıdaki gibi tanımlanabilir.

iş Merhaba:  
stdio::printf("Merhaba Dünya !");  
  
/\*ya da\*/  
  
iş DöngülüMerhaba:  
 her değer i tam = 0; i < 10; i++:  
 stdio::printf("Merhaba Dünya !");

Bu örnekte sadece tek satırlı işler tanımlayabilme yeteneğindeki amaç okunurluğu sade ve kolay kılmaktır.

**Değişkenler:**[#](#term-Degiskenler)

İşlemlerin girdileri ve çıktıları olabilir.

Not

Örs veyâ diğer sıralı dillerdeki değişkenler kavramı matematikteki bağımlı ve bağımsız değişkenlerle karıştırılmamalıdır. Sıralı diller makinaya yakın olduğu için; üretilmiş makina kodu işletim sisteminin işlem yığınına alınacak ve bu aşamada matematikteki kavramlar yozlaşmaya uğruyor. O yüzden **değişkenler** kavramını kendi içerisinde ele almak daha sağlıklı.

iş Topla a tam, b tam; tam:  
{  
 dön a + b;  
}

Derleyiciye bu işlemin baş kısmıyla denilen şey basitçe: “a ve b tam sayı değişkenleri kabul edip, tam sayı dönen ‘Topla’ işini tanımla.”

Değişkenler tanımlı işin beden satırında değer gibi ele alınsalar da; hem derleyici tarafından yorumlanması hem de bilgisayarda işlenmesi bakımından özlerinde değerlerden farklılardır.

On altı farklı girdi değişkeni ve tek bir çıktı değişkeni tanımlanabilir.

Burada değişken sayısının üst sınırının 16 olmasının nedeni basitçe uygunluğudur. Bildiğim kadarı ile GNU C dilinde 255 farklı değişken tanımlanabiliyor ama Clang derleyicisi farklı ele alıyor olabilir. Diyelim sadece iki değişken kabul etseydi o zaman **türleri** kullanarak dilediğimiz kadar değişkeni tek seferde bile işlemek mümkün olurdu. Bu nedenden ötürü burada önemli olan değişken tanımlarında bir üst sınır olduğudur. Zira ben bu güne kadar 16 değişkeni olan bir işlem görmedim.

**Çıktılar:**

Yukarıdaki ‘Topla’ işlemine küçük bir değişiklik yapalım:

iş Topla a tam, b tam; dönüş tam:  
 dön a + b;

Derleyiciye emredilen yine değişmedi ama Örs dilinde çıktılara isimlendirdiğimiz sürece erişebiliriz. Bu **go** ve benzeri dillerin bizimle tanıştırdığı harika bir özellik. İşlemler içerisinde pek çok sefer döneceğimiz değeri tanımlamak zorunda kalıyoruz. Gerek bile yok. Çıktısı olan işlemlerde zaten dönülecek türün hafızası önceden ayrılıyor ve tekrar başka bir dönüş değeri tanımladığımızda iyileştirme aşamasında doğal olarak siliniyor çünkü gereksiz bir değer.

iş Topla a tam, b tam; dönüş tam:  
{  
 dönüş = a + b;  
}  
  
/\*ya da tek satırlık tanımla\*/  
  
iş Topla a tam, b tam; dönüş tam:  
 dönüş = a + b;

İşlem baş kısmı özet olarak yukarıdaki gibidir. Örs dilinde işlemler tanımlanırken ‘iş’ sözcüğü kullanılarak başlanır. Sonrasında işin birim hafızasındaki benzersiz adı verilir, eğer varsa değişkenler aralarına virgül konularak tanımlanır ve eğer varsa çıktı kısmı girdilerden ayırmak için çift nokta ile ayrılır ve son olarak ise satırı tanımlanır. Makina dili yapılanması C dilinin derlendiği işletim sistemleri için belirlediği standartlara uyar.

#### İskelet:[#](#iskelet)

**Hazırlanıyor.**

Beden kısmına ise her biri sıralı ifadelerin kümesi olan satırların olduğu sıralı küme satırı kabul eder.

İşlemler konusunu irdelemeyi **türler** ve **birimler** konularını ele alana kadar burada bırakmak zorundayız.

### Türler[#](#turler)

#### Yapıtaşları[#](#yapitaslari)

Örs dilinde hafızada konumlandırabileceğiz en küçük anlamlı parçalardır.

Konumlandırabileceğiz diyoruz zira türleri üyelerine ayrıştırınca öyle bir noktaya erişiyoruz ki daha fazla parçalamak yazılımda verim, hız, bütünlük gibi sorunlara yol açıyor.

Bu noktada yazılım dilleri tüm tanımlanabilir türlerin temel alacağı yapıtaşları belirler.

Bunu yaparlarken genelde güttükleri kaygı uyumluluk üzerinedir. Mesela işlem çağrılarını birbirine giren parçalar olarak hayal edin. Eğer parçalar belli bir ölçüye uymuyorlarsa beraber çalışmaları zorlaşır ve hatta imkansızlaşır. Bu noktada işlemciler tarafından belirlenen mimarilere uymak ve tür temelinde genel uyumu yakalamak önemlidir. Bu mânada; hayalen dilediğiniz bit genişliğine sahip yapıtaşlarını hem makina dili hem de yazılım dili seviyesinde tanımlamak tamamen mümkündür. Sistem dilleri için sorun bu bit genişliğinin işlemci mimarisi ile uyumlu olup olmadığıdır. Zira hedefi makina kaynaklarına sağlıklı erişim olan dillerin; mimari uyumsuzluğu yüzünden hız ve verim sorunlarına katlanmasını beklemek biraz anlamsızdır.

Yukarıdaki uyum sorunu cidden önemli çünkü aynı zamanda yazılımımızı dış dünyaya bu yapıtaşlarının ele alanışı üzerinden belirliyoruz. Eğer derleyici tarafından belirlenen genişlikler yaygın kullanıma uymuyorsa, iki iletişim kaynağının arasına görevi çevirmenlik yapmak olan bir yazılımı eklemek zorunda kalıyoruz.

Bu yüzden Örs yapıtaşları konusunda C’dili ile bire bir uyumluluk siyasetini güder. C dilinde bile derleyiciden derleyiciye değişen ve hatta sürümler arası farklar olan bir konuyu daha da karmaşıklaştırmak anlamsız ve gereksizdir.

En nihayetinde gerçek hayatta yazılımdaki görevleri bir araya getirerek türler oluşturmaktır ki kendileri de aynı zamanda türdür. Türlerden farkı en küçük anlamlı konumladırılabilir parçalar olmasıdır. Onun dışında türlerle aynı davranışları sergilerler.

Aşağıda Örs dilinde tanımlı yapıtaşlarına ve C’deki denklerine ulaşabilirsiniz.

Tanımlı yapıtaşları:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Yapıtaşları** | Bit genişliği | Alt sınır | Ust sınır | İşaretli mi | C’deki karşılığı | Biçimlendirme |
| **Tam Sayılar** |  |  |  |  |  |  |
| **t8** | 8 bit | -127 | 128 | evet | char | %c |
| **t16** | 16 bit | -32,768 | 32,767 | evet | short | %hd |
| **t32** | 32 bit | -2,147,483,648 | 2,147,483,647 | evet | int | %d |
| **t64** | 64 bit | -2^63 | 2^63 | evet | int64\_t | %lld |
| **t128** | 128 bit | -2^127 | 2^127 | evet | \_\_int128\_t | %lld\* |
| **tam** | 32 bit | -2,147,483,648 | 2,147,483,647 | evet | int | %d |
| **Doğallar** |  |  |  |  |  |  |
| **d8** | 8 bit | 0 | 255 | hayır | uint8\_t | %c |
| **d16** | 16 bit | 0 | 65,535 | hayır | uint16\_t | %hu |
| **d32** | 16 bit | 0 | 4,294,967,295 | hayır | uint32\_t | %u |
| **d64** | 64 bit | 0 | 2^64 | hayır | uint64\_t | %llu |
| **d128** | 128 bit | 0 | 2^128 | hayır | \_\_uint128\_t |  |
| **doğal** | 32 bit | 0 | 4,294,967,295 | evet | uint32\_t | %u |
| **Ondalıklar** |  |  |  |  |  |  |
| **o32** | 32 bit | ~ | ~ | ~ | float | %f |
| **o64** | 64 bit | ~ | ~ | ~ | double | %lf |
| **o128** | 128 bit | ~ | ~ | ~ | ~ | %Lf |
| **ondalık** | 64 bit | ~ | ~ | ~ | double | %lf |
| **Özel** |  |  |  |  |  |  |
| mimari | işlemci | ~ | ~ | hayır | size\_t | %lu |
| şey | işlemci | ~ | ~ | ~ | void\* | %p |

#### Sayaç[#](#sayac)

Örs dilinde konumlandırılamayan türlerdendir.

Konumları alınamaz çünkü aslında kaynak kodunu anlamlı kılmak adına kümelendirilmiş sabit 32 bit genişliğinde tam sayılar **kümesidir.**

Tüm yazılım dillerinde farklı söz dizimleriyle bulunan bir kavramdır.

sayaç Rakam  
{  
 Sıfır = 0;  
 Bir;  
 İki;  
 Üç;  
 Dört = 4;  
 Beş;  
 Altı;  
 Yedi;  
 Sekiz;  
 Dokuz;  
}

Yukarıdaki örnekte C dilindeki ‘enum’ kavramından farklı değil. Atamalardan sonra diğer üyeler birer arttırılarak değerlenecek. C dilinden farkı sayaçların üye olarak sayaç üyelerini kabul eden kümeleri de kabul etmesidir. Çünkü sayaçlar yazılımda ‘durum’ kavramının tamamlayıcılarıdır ve bazen farklı durumlar için aynı seçim satırını kullanabiliriz. Tam bu noktada birden çok seçim satırını tekrar tekrar yazmak sadece kalabalık oluşturuyor ve okumayı güçleştiriyor. Bu noktada yardıma sayaç kümeleri yetişiyor. Mesela yukarıdaki örneği tekrar yazalım:

sayaç Rakam  
{  
 Sıfır = 0;  
 Bir;  
 İki;  
 Üç;  
 Dört = 4;  
 Beş;  
 Altı;  
 Yedi;  
 Sekiz;  
 Dokuz;  
  
 tekler(Bir, Üç, Beş, Yedi, Dokuz);  
 çiftler(Sıfır, İki, Dört, Altı, Sekiz);  
}

Burada tek ve çift sayılar tekrar kümelenmiş ki kullanımlarında kolaylık olsun. Mesela:

iş RakamTekMiÇiftMi a tam  
{  
 durum a  
 {  
 seçim Rakam::tekler:  
 stdio::printf("Tek rakam: %d\n", a);  
 seçim Rakam::çiftler:  
 stdio::printf("Çift rakam: %d\n", a);  
 varsayılan:  
 stdio::printf("Gelen sayı %d bir rakam değil !\n", a);  
 }  
}

Yukarıdaki örnekte okuma kolaylığını görebiliyorsunuz. Sayaç kümeleri burada kodu daha sade ve okunabilir kılmış ki gerçek hayatta bu durumu bolca kullanıyoruz.

#### Türler[#](#turler)

İşlemler nasıl bilgisayar dünyasının kalbinin attığı yerse; türler de bu dünyanın iskeleti ve tamamlayanı.

Örs bize tür konusunda geniş bir kavram yelpazesi sunuyor. Kalıplar, yalınlar, ortaklar, künyeler ve daha nicesi ama, fakat, ve lâkin; Tüm diğer kavramlar bu yazıda ele alınacak tür kavramının süslenmiş; yani hazır yapılandırılmış hali.

Türler özlerinde sıralıdır türler kümesidir. Kümesidir zira kendi kendisinin üyesi olamaz. Kümesidir çünkü tanım tekrarı yapılamaz. Sıralıdır çünkü tüm sistem dillerinde olduğu gibi üyelerin sırası bit genişliğini ve bit sıralamasını belirler.

Örs dilinde türler nesne tabanlı dillerle benzeşmez. C’deki ‘struct’ kavramının genişletilmiş halidir. Tıpkı C dilindeki bir tür tanımladığınızda nesne dosyasında nasıl yapılanacağı hayalinizde somutlaşıyorsa Örs’de de öyledir. Bu yönüyle Örs’deki tür soyutlaması makineye yakındır.

tür Metin  
{  
 boyut t32;  
 hacim t32;  
 \_harfler \*t8;  
}

Her ne kadar Örs nesne tabanlı yazılım tarıznı desteklemese ve de yapısalcılığı öne cıkarsa da Örs’de türlere işlem bağlamak mümkün.

//yukarıdaki örneğin devamı  
öz \*Metin  
iş Yaz Belge \*stdio::FILE  
{  
 stdio::fprintf("metin: %s\n", öz->Metin);  
}

#### Kalıplar[#](#kaliplar)

Gerçek tür olmayan, yani sanal bir kavram olan kalıplar tür kavramındaki tasarım eksiğini kapamayı amaçlar.

Mesela, C dilinde en basit yapı türlerinden olan zincirler için her bir farklı türdeki zincir elemanı için yeniden tür tanımlamak zorunda kalıyoruz.

Dahası C++ ve Rust gibi dillerde elimizde var olan ve yapısı, boyutu, sıralaması belirgin olan türler taçlandırılıyor ve türün boyutu ya da sıralaması değişebiliyor. Bu konuda biz yazılımcıların bu dillerin tanıştırdığı harika özelliği suistimal etmemizin de payı tabi ki yadsınamaz ama sistem yazılımcıları açımızdan çok ciddi bir belirsizlik ve bir takım hata ayıklamalar samanlıkta iğne aramaya dönüşüyor.

Bu iki soruna karşılık Örs dili kalıp kavramını öne sürdü. Böylece, ‘kalıp’ kavramı vasat türler kavram bütünlüğünü, yapısal katılığını korurken kalıplar aracılığıyla hem tanım tekrarının önüne geçebilmek hem de biz sistem yazılımcılarının haklı takıntılarına çözüm üretmeyi amaçlıyor.

kalıp Dal x  
{  
 Öz x;  
 Önceki \*Dal'x;  
 Sonraki \*Dal'x;  
}  
  
kalıp Zincir x  
{  
 boyut t32;  
 Baş \*Dal'x;  
 Son \*Dal'x;  
}

Yukarıdaki örnek Örs’ün zincir kütüphanesinden alıntı. Basitçe anlaşılacağı üzere kalıp ‘zincir’ ne olduğu bilinmeyen ‘x’ türü ile taçlandırılana kadar derleyici hafızasında özel bir sanal tanım olarak yer alıyor. Genel yapısı belirgin olsa da boyutu belli olmadığı için gerçek bir tür değil ve tamamen sanal.

Kalıplar herhangi değişken ya da değerin türünü belirlerken anlam kazanıyor. Mesela yukarıdaki zinciri tam türüyle taçlandıralım.

iş ZincirÖrneği  
{  
 değer rakamlar Zincir'tam;  
}

Buyrun artık elimizde canlı, tam türüyle taçlandırılmış zincir türü var.

##### Kalıp işlemler:[#](#kalip-islemler)

Nam-ı diğer sanal işlemler. Malûmunüz üzere kalıplar sanal olduğu için onlara bağlanan işlemler de sanal olur. Yukarıdaki örneği devam ettirelim:

öz Zincir'x  
sanal iş Ekle Veri x: x  
{  
 Dal := temiz(%\*öz.Baş);  
 Dal->Öz = Veri;  
 eğer öz.boyut:  
 {  
 Dal->Önceki = öz.Son;  
 öz.Son->Sonraki = Dal;  
 öz.Son = Dal;  
 }  
 değilse  
 {  
 öz.Son = Dal;  
 öz.Baş = Dal;  
 }  
 öz.boyut++;  
 dön Dal->Öz;  
}  
  
  
iş ZincirÖrneği  
{  
 değer rakamlar Zincir'tam;  
  
 her değer i tam = 0; i < 10; i++:  
 rakamlar.Ekle(i);  
}

##### Taçlar ve Donatımlar:[#](#taclar-ve-donatimlar)

Aşağıda çok kullandığımız çizelge veri türü var.

kalıp kök x,y  
{  
 Sıradaki \*kök'x,y;  
 hash d32;  
 ad x;  
 öz y;  
}  
  
kalıp k x,y  
{  
 boyut tam;  
 hacim doğal;  
 yığın dizi::k'\*kök'x,y;  
 Nesneler \*\*kök'x,y;  
}  
  
iş DonatımÖrneği  
{  
 değer sözlük k'metin,tam;  
}

Bu örneğin içeriği değişebilir ve konumuz dışında ama fark ettiğiniz üzere x ve y olası türleri sanki işlemlerdeki değişken kavramı gibi davranıyor. Göze öyle görünse de gerçekte tabi ki de kesinkes öyle değil. Burada bu kavram çatışmasının önüne geçmek için kalıpların kabul ettiği olası tür simgelerini ‘taç’ olarak adlandırmayı ve kavramlandırmayı seçtik.

Herhangi bir kalıp ikiden fazla türle taçlandırılamaz ve o şekilde tanımlanamaz. İkiden fazla tacın aslında gereksiz olduğu bir yana Örsün tasarım ilkeleriyle de tutarlıdır. Yani Örs mantığına göre siz kalıp için ikiden fazla taç tanımlamışsanız sizin kalıbınızda bir sorun vardır.

Mesela diyelim ki yukarıdaki ‘kök’ kalıbı 3 taç kabul ediyor.

kalıp kök x,y  
{  
 Sıradaki \*kök'x,y;  
 hash z;  
 ad x;  
 öz y;  
}

Bu örnekte bu kalıp belli ki çizelge algoritması tarafından kullanılacak ve ‘hash’ üyesinin bit genişliği algoritma için önemli zira diyelim ki d64 türünde olsaydı farklı sabit ya da algoritma kullanmanız gerebilirdi. Evet değişken doğalı sözlük algoritmaları var ama ‘k’ veri türü o değişken doğalı algoritmayla uyum sağlayamaz. Yani ? Yani kalıp tanımında sorun var.

Örs’te kalıp da dahil tüm diğer kavramları olabildiğince doğrusal düşünmek istiyoruz. İkili tacı sakın küçümsemeyin. O iki tacın içine yüz tane farklı iç içe geçmiş donatılmış tür yerleştirebilirsiniz ama eğer veri türlerini bu şekilde saçaklandırma huyunuzun Örs’ün kendi tasarım kaidelerini çiğneyip desteklemesini beklemeyiniz.

Donatım kavramı ise başlık altındaki misalde görüldüğü üzere kalıpların türe dönüştüğünde aldığı olası türlere atıfta bulunuyor. Malûm örnek için metin ve tam sayı sözlüğü değeri ‘k’ kalıbı ‘metin’ ve ‘tam’ türleri ile donatılmış.

Peki bu iki muğlak kavramı sanal işlemler için de kullanabilir miyiz ? Buradaki uygulamasını kabul etmediğimiz takdirde, elbette. Farkı ise sanal işlemlerde tıpkı işlemler gibi 16’dan fazla değişken almazlar ve bu değişkenlerin muğlak türlerine hitaben ‘taç’; türü belirlenmiş, yani çağrılmış, hallerine hitaben ‘donatım’ kavramlarını bire bir olmamak şartıyla fevkâlade kullanabiliriz.

#### Yalın tür tanımları[#](#yalin-tur-tanimlari)

Tekil gizli üyeleri olan türlerdir.

Bir yönüyle her hangi bir türü yeniden adlandırma görevini üstlenirken; sıfatlarıyla güçlü bir kavrama evrilir. ————– Basitçe:

tür TamSayı tam;  
  
sayı TamSayı  
iş Yazdır  
{  
 stdio::printf("sayı: %d", sayı);  
}

Görüldüğü üzere varsayılan yapıtaşı ‘tamsayı’ olarak tekrar tanımlanmış ve kendisine ait işlemler bağlanmış. Bu yönüyle yalın türlerin görevi; aynı veri türlerini kullanan ama tamamen farklı algoritmaları kullanan yazılımlarda kavramları belirgin yapmak ve yaygın türlere görevi olduğu ortamın amacına uygun görevler tanımlamaktır;

tür Sayılar dizi::k'tam;  
  
sayılar Sayılar  
iş Yazdır  
{  
 her i := 0; i < sayılar.boyut; i++;  
 stdio::printf(  
 "%d'inci sayı = %d",  
 i, sayılar.Öğe[i]);  
}

Bir diğer önemli konu ise kalıpları donatırlar. Kalıplara, tasarımları gereği, gerçek işlemler bağlanamaz. Yalın türler aracılığıyla donatıldıklarında kalıpların değer satırlıyla tekrar tekrar tanımlanmasının önüne geçer ve kalıpların anlam bulduğu ortamda ilişkili işlemler ve algoritmalar bağlanabilir hale gelir.

#### Ortak[#](#ortak)

Tekil hafızanın çoğul değerler için kullanıldığı özel türlerdir.

Ortak olarak isimlendirilmesinin nedeni özlerinde konumlar beklemeleridir. Zira yazılımda konumu ‘şey’ değerine saklar ve sonrasında bu konumun aslında ne konumu olduğunu bilmeden tehlikeli sularda yüzerek işlemlerimizi yapabilirdik. Ortak konumun farklı türler için ortak kullanılan bir konum olarak düşünüldüğünü varsayarak bir çerçeve etrafında konumu yorumlar. Bu bakımdan ‘şey’ türünün kavram olarak daraltılmış halidir.

C dilindeki ‘union’ kavramının Örs dilindeki dengidir.

ortak çıktı  
{  
 sayı t64;  
 harfler t8[8];  
}  
  
iş Örnek  
{  
 değer i çıktı;  
 i.sayı = 0x0\_41\_42\_43\_44\_45\_46\_47;  
 stdio::printf(  
 "Konumlar:\n&i.sayı = %p\n&i.harfler\n"  
 "metni = '%s'\n",  
 &i.sayı, i.harfler, i.harfler);  
 //benim işlemcinin mimarisinde i.harfler GFEDCBA olarak görünüyor  
  
 stdio::printf(  
 "i = %x\n"  
 "harfler[0] = %x\n"  
 "harfler[1] = %x\n"  
 "harfler[2] = %x\n"  
 "harfler[3] = %x\n"  
 "harfler[4] = %x\n"  
 "harfler[5] = %x\n"  
 "harfler[6] = %x\n"  
 "harfler[7] = %x\n",  
 i,  
 i.harfler[0],  
 i.harfler[1],  
 i.harfler[2],  
 i.harfler[3],  
 i.harfler[4],  
 i.harfler[5],  
 i.harfler[6],  
 i.harfler[7]);  
}

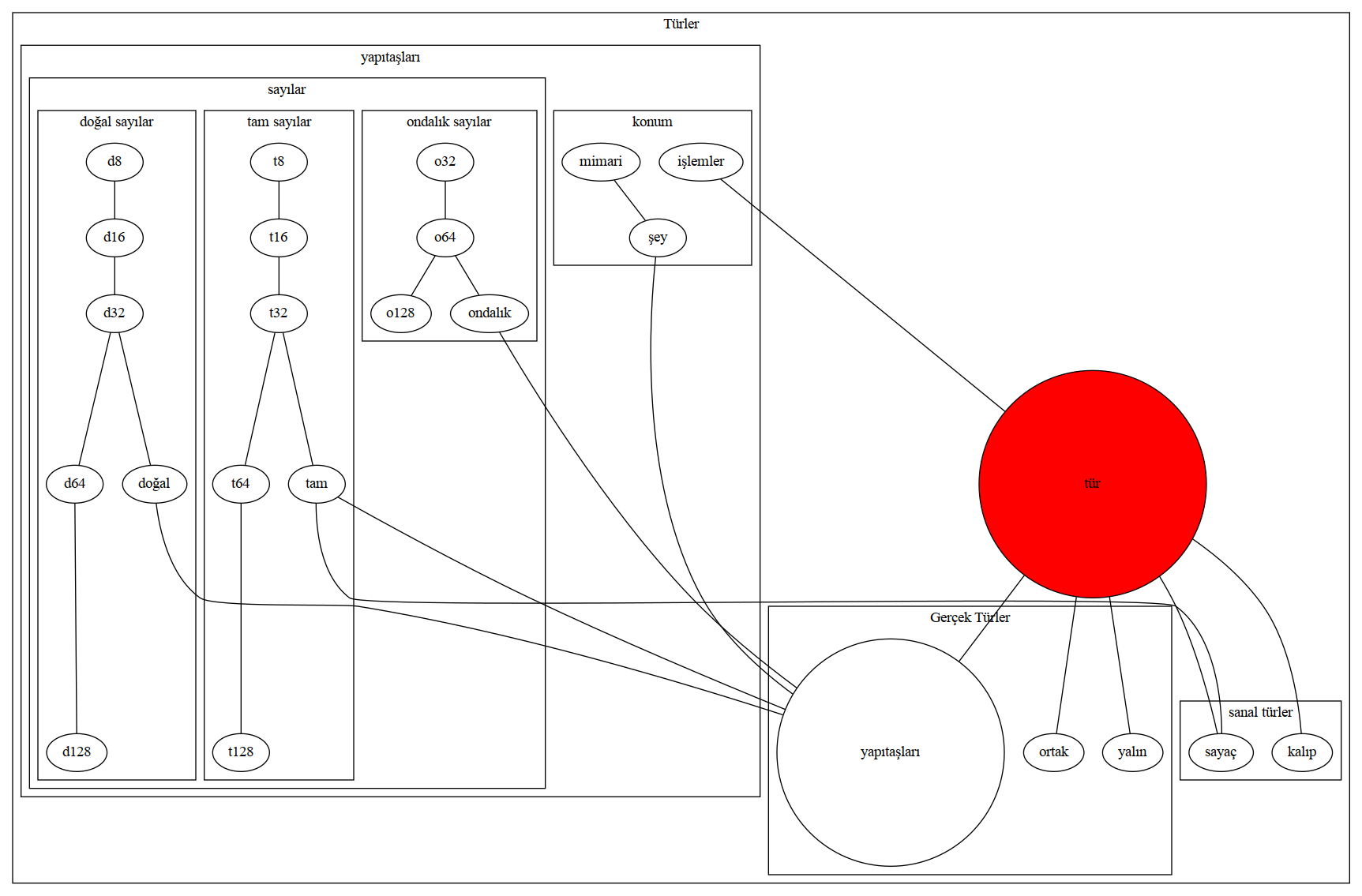
Bu örneğin anlamsız olduğu düşünülebilir ki gerçekten de öyle. Tek amacı sayı ve harfler adlı üyelerin aynı konumu paylaştığını göstermek ama zaten ortakların yaygın kullanımı da bu şekilde olmaz. Genelde künyelendirilerek kullanılırlar. Mesela:

ortak çıktı  
{  
 sayı t64;  
 harfler \*t8;  
 url \*t8;  
 Metin \*metin;  
}  
  
sayaç seçimler  
{  
 Sayı = 1;  
 Harfler;  
 Url;  
 Metin;  
}  
  
tür imge  
{  
 künye t32;  
 içerik çıktı;  
}

Veri iskeletinden de anlaşılacağı üzere basitçe metin çözümleme türü tanımlanmış. Bu iskeleti derleyicilerden, dil sunucularına, veri tabanlarından, oyunlara her yerde kullanacaksınız. En az zincirler konusu kadar yaygın bir iskelet.

Buraya kadar işlemler ve işlevlerle ilgilendik ama tüm bunlar anlamlı yapılandırılmış veri olmadan tamamen anlamsız kalıyor.

Bu bölümde Örs dilinin türlere yaklaşımını ele alacağız.



### Birimler[#](#birimler)

Örs dilinde çözümlenen söz dizisinin ilk ürünü olarak kabaca tanımlanabilir.

Uyarı

Örs’teki birim özelliğini lütfen, sakın ki sakın nesne tabanlı dillerdeki özelliklerle karıştırmayınız.

Kabaca diyorum çünkü bu konu göründüğünden daha derin. C’de her bir ‘.c’ belgesi genelde ‘.o’ belgesine çevrilir.

Not

Yazının devamında;

‘.o’ uzantılı belgeye **nesne** belgesi,

‘.a’ uzantılı belgeye **dahili** kütüphane,

‘.so’ uzantılı belgeye ise **harici** kütüphane

diye atıfta bulunacağız.

Nesne belgelerinin içeriğinde basitçe makina koduna yorumlanmış işlemleriniz ve yazılımınızın hafıza iskeleti bulunur. Durağan değerlerin hafızada nasıl canlanacağı bilgisi bu belgededir. İşlemler ise yalındır. Yani dahili ve harici kütüphanelerle nasıl iletişime geçeceği bilgisini bulundurmaz.

Eğer diğer kütüphanelerle sade yöntem olan işlem çağrıları ile iletişim kursaydık ve bu işlem çağrılarının küresel değişken değerler kullanmadığını varsaysaydık bu konu tüm dillerde basmakalıp ele alınırdı ama ne güzel ki gerçek öyle değil.

Nesne belgeleri yazdırılıp üretim süreci tamamlanınca isteğe göre bağlama süreci başlar. İsteğe bağlı diyorum çünkü ürünler illa ki bağlanmak zorunda değil. Çünkü bağlama sürecinin yazılım dilleri ile alakası bile tartışılabilir. Yaygın kullanım isimleriyle ‘.a’ ve ‘.so’ uzantılı belgelerinin biri dahili bağlanmışken diğerinde sadece harici atıflara nasıl canlı ya da yazılım hafızaya alınırken bağlanacağı bilgisi vardır. Sistem dilleri genelde bu sorunun işletim sistemine ait olduğu varsayımı ile hareket ederler ki pek çok zaman sistem dillerini derletmek içim ‘cmake’ gibi yazılımları kullanmamızın ana nedenlerinden biridir. Bir takım kütüphanelere atıfta bulunacağız ama bu belgelerin nerede olduğunu bilmiyoruz sorununa çözüm üretirler.

Diğer taraftan örs dili türlerinin neden değişken türlü işlemler tanımlamamasının kökünde yatan nedenlerden birisi de budur. Örste değişken türlü sanal işlemler tanımlayabiliyorken; değişken türlü işlemler tanımlanamıyor. Şimdi bunun her bir tür başlatılıp işlem çağrıldığında yeniden işlemin derlenmesi sorunu bir tarafa aynı zamanda bağlama sürecinde de sıkıntı yaşatıyor. Burada bu özelliğe sahip diller iyi ya da kötü dillerdir diye bir argüman savunmuyoruz. Bu bambaşka bir tartışma konusu. Birimlerle alakası bu işlemlerin nasıl bağlanacağı ve konumlarının nasıl belirleneceğidir. Örnek vermem gerikirse; diyelim ki C++ mantığında derleme yapan bir dil ile; bir yerde ‘dizi<tam>’ türünde, başka bir yerde ise ‘dizi<hesaplar>’ diye iki farklı tür tanımladınız. Eğer kalıp taçları konum olsaydı neyin konumu olmaksızın tür aynı olacaktı ama değil. Peki yorumlanan türleri aynı olmayan dizi kalıbının işlemleri ne olacaktı ? Tüm bu işlemleri ya her çağrıldıklarında yeniden derleyecektiniz ya da değeri birinci dereceden konummuş gibi hayal edip yığın çizelgesi oluşturacaktınız. Üçüncü çözümde ise hiç derlemeyip sanal işlemmiş gibi ele alacaktınız ama o kavram zaten Örs için tanımlı ve geriye elinizde 2 çözüm kalıyor. Peki bu işlem atfı yığın çizelgesini nasıl bağlattıracaktınız ? C++ dilinin namının şaibeli olmasının temel nedenlerinden bir tanesi bu çok ciddi sorun bu arada. Bu ciddi soruna ‘Sanal İşlemler’ konusunda tekrar değineceğiz. Birimlerin nasıl ele alındığı ve nasıl algılanması gerektiği ile derinden alakalı temel bir sorun olduğu için hatırlatma ihtiyacı duyduk.

Birim kavramının amaçlarından bir diğeri ise isim çakışmasını ve dolaşmasını engellemektir. Mesela elinizde ekleme yapan birden fazla birim olduğunu düşünün. Farzedelim ki bunlardan biri ‘dizi’ diğeri ‘çizelge’ birimi olsun. Bu iki birimde doğal algoritmalarının bir parçası olarak ‘ekle’ işlemi tanımlayacaktı. Eğer Örs bu iki birimi tek birim olarak ele alsaydı; bu işlemlerin isimleri çakıştığı için hata vermek zorunda kalacaktı. Birim kavramı işlevde bu dolaşmanın da önüne geçer.

Altı çizilmesi gereken bir diğer konu Örs’teki birimler kavramı kesinlikle nesne tabanlı dillerle karşılaştırılmamalıdır. Örste birimden kasıt ‘CU’ yani üretim süreci kavramı olan ‘compilation unit’ diye ingilizce tabir edilen kavrama ve bu kavramın üretilmiş haline, yani ‘.o’ belgeleri ve alâkalı türevlerine atıfta bulunur. Bizim istediğimiz şey basitçe; bu ürünlerin nasıl bir araya getirilip kütüphanelere dönüşeceğinin belirgin olmasıdır. Bu mânada birim kavramının amacı derlemenin tarama, çözümleme, üretim ve bağlama süreçlerinin olabildiğince örtüşmesi ve bu yazılımlar hayata geçtiğinde birbirleri ile iletişim kurarken tasarlanma sürecindeki kavram kümesi ile ortak kavram alanında kesişmesidir. Zira sistem dilleri için yukarıda özetle anlatıldığı gibi nesne belgelerinin üretiminden sonra yazılımınız özünde bir soyutlama olan yazılım dilinden epey ayrışıp; gerçek hayattaki görev ve ilişkileri ile örtüşmeye başlamaktadır. Kavramların olabildiğince örtüşük olmasını istiyoruz ki on binlerce kütüphanenin birbirine dolanırcasına bağlandığı ve tüm bu süreci denetlemek için ayrı yazılımların kullanıldığı cengâmelere saplanma olasılığımızı düşürebilelim.

Çünkü hoşumuza gitse de gitmese de üretilen birimler birbirleri ile ilişkilerine göre dallanıp, derecelenmeye ve ağaçlanmaya başlıyor.

Daha ilginç olanı ise kaynağınızda dosya yaratmışsınız ve bu dosya genelde yukarıda bahsettiğimiz ilişki ağacının bir parçası oluyor ama pek çok yazılım dili bu dosyaları birim olarak kabul etmiyor.

Birimler tüm bu yukarıdaki bahsettiğimiz dallanma sorununu, derecelenmeyi, yazılım dili seviyesinde çözmeye yarayan, içeriğinde derlenebilir öğeleri küme olarak tutan ve izdüşümü nesne belgeleri olan kavramlardandır. Bazı dillerden farkı sizin kaynak kodu ağacınızla örtüşük olmasıdır.

Uyarı

Tekrar uyarmak gerekiyor. Aşağıdaki örnek basitliği için seçilmiştir. Lütfen birim özelliğininin nesne tabanlı tarzla ne bire bir örtüştüğünü ne de öyle bir gayesi olduğunu biliniz.

dahili merkez::c::stdio;  
  
birim işlemler  
{  
 birim katlı  
 {  
 iş Çarp a tam, b tam: tam =>  
 dön a \* b;  
 }  
 iş Topla a tam, b tam: tam => dön a + b;  
 iş Topla a tam, b tam: tam => dön a - b;  
}  
  
iş SonuçYaz: tam  
{  
 stdio::printf(  
 "işlemler::Topla = %d\n"  
 "işlemler::katlı::Çarp = %d\n,  
 işlemler::Topla(11, 22),  
 işlemler::katlı::Çarp(11, 22));  
}

Yapılan şey şu örnekte diğer dillerden farklı değil. İşlemi tanımladık ve giriş işleminden çağırdık. Fark edeceğiniz üzere tanımlı birimlerden işlem çağırabilmek için birimin içeriklerine erişmek gerikiyor ve bunu ‘::’ simgesini kullanarak yapabiliyoruz.

‘SonuçYaz’ işlemi fark ettiğiniz üzere anlamlandırdığınız ağaçta en üst seviyede. Bu bağlamda elimizde üretilmesi gereken 3 farklı birim var. Bağlama sürecinde çalıştırma belgesi, dahili veya harici kütüphene üretilecek olduğunda yapılacak olan şey basitçe ‘işlemler::katlı’ birimi işlemlere eklenecek ve bağlanacak; ‘işlemler’ birimi de üst birime eklenecek ve toplanacak.

Bir diğer örnek varsayılım ki elimizde aşağıdaki gibi her bir işlemin ayrı belgede ele alındığı; gayet yaygın olan bir kaynak ağacı var:

birim\_örneği  
├── işlem  
| ├── katlı  
| | └── çarpma.ors  
| ├── çıkarma.ors  
| └── toplama.ors  
└── sonuç\_yaz.ors

Bu dosya ağacının oluşturacağı birim ağacı ile bir üst örnekte oluşacak olan birim ağaçları bire bir aynı. Tek fark kaynak kodunuzu düzenleme tercihiniz ki o size kalmış. Ama Örs derleyicisi eğer dosya açıp içine kaynak kodu koymuşsanız bunu birim olarak kabul eder ve yukarıdaki bağlama sürecine tabi tutar.

Tam burada birimlerin nasıl ele alındığı iyice belirginleşmeye başladı.

**Peki ya Örs dili dahili veya harici kütüphaneleri nasıl görüyor ve ele alıyor ?**

Yukarıdaki kaynak ağacını tekrar ele alalım.

birim\_örneği  
├── özelleştirme.uzn  
├── işlem  
| ├── katlı  
| | └── çarpma.ors  
| ├── çıkarma.ors  
| └── toplama.ors  
└── sonuç\_yaz.ors

Sadece aşağıdaki uzengi satırını dosya kökündeki; ‘özelleştirme.uzn’ belgesine yazmak yeterli olacaktır.

özelleştirme:  
 ad: "birim\_örneği",  
 çıktı: birim\_örneği,  
 ürün\_türü: çalıştırma;

Hatırlatma:  
Ürün türünü belirlemek için;  
'nesne' : yalın nesne belgesi,  
'çalıştırma' : çalıştırılabilir tetik belgesini,  
'makina' : makina dili,  
'dahili' : dahili kütüphane,  
'harici' : harici kütüphane  
seçenekleri; ilgili çıktıları üretir.

### Yine Yeniden İşlemler:[#](#yine-yeniden-islemler)

Tür ve birimler konusunu ele aldığımıza göre işlemleri irdelemeye devam edebiliriz.

C dilinde en çok yaşanan sıkıntılardan birisi, önce türü tanımlayıp sonrasında o türle ilişkili işlemler tanımlanırken özellikle isimlendirme ve türün kendisini tekrar tekrar tanımlama konusunda yaşanan köle işidir.

Not

C’de bu tarz kıymıkların olmasının nedenlerinden birisi o dönemdeki bilgisayarların kaynak yetersizliğinden ileri gelir. Mesela C’de bolca ‘strlen, fcntl’ vb bir takım okunması, ezbenlenmesi anlanması bile zor olan işlemler ve kütüphaneler var ve bunların amacı sözlük algoritmasının işini kolaylaştırmaktan öte değil.

Yani özet olarak C’yi eleştirirken biraz dönemin koşullarını da göz önününde bulundurmak gerekiyor.

Mesela dediniz ki:

struct dizi  
{  
 int hacim;  
 int boyut;  
 int\* Icerik;  
}

Bu türe işlemler tanımlayacaksınız ama gördüğünüz gibi bu işlemler sadece **‘int\*’** dizisi için düzgün çalışacak ve bu da yetmezmiş gibi aşağıdaki gibi çirkinlikleri gözden çıkaracaksınız.

void kütüphaneAdı\_dizi\_Topla(dizi\* Dizi, int toplanacakSayı)  
{  
 for(int i = 0; i < Dizi->boyut; i++)  
 Dizi->Icerik[i] += toplanacakSayı;  
}

Şimdi beden kısmını boşverelim: İsim aslında içinde doğrusal bir ağaçlandırma yapıyor. ‘kütüphaneAdı’ + ‘işlemi anlam kümesine alan türün adı’ + ‘işlemin görevinin adı’ gibi bir kalıba uyuyor. C’deki bu eksiklik çeşitli doğrusal kalıplarla aşılabilir. Daha ciddi sorun hem bu isim kalıplarının hem de onların bedenlerinin farklı yapıtaşları ve türler için tekrar edebiliyor olması.

Mesela yukarıdaki diziyi, tam sayı dizisi değil de ondalık dizisi olarak yazsaydık ve bunların her bir elemanını bir ondalık ile toplayacak olsaydık yine aynı şeyleri tekrar edecektik.

C’de bu sorunu öngeçişler ile fevkalede aşabiliriz ama bu sefer de derleyiciye türü tekrar tekar tanımlayacaktık. Dahası her bir öngeçiş işlemi çağrısı bizim kütüphanemizin boyutunu daha da şişirecekti ki bazen çok basit yazılımlar için ağaçlanan öngeçişler ürünü devasâ boyutlara ulaştırtabiliyor.

Diğer taraftan bu konuyu çözmüş yazılım dilleri yok mu ? Tabi ki de var. Özellikle nesne tabanlı yazılım tarzının geliştirilme nedenlerinden bir tanesi bu yukarıda irdelediğimiz sorun ama sistem dillerinde nesne tabanlı tarz aşılması zor yapısal sorunlara yol açıyor ve daha içinden çıkılamazı hata ayıklamayı zorlaştırıyor. Bunlar yetmezmiş gibi işlemlerin cağrı yöntemini değiştire de biliyorlar; ki sistem sürücüleri ile iletişime geçerken yeniden köprüler yazmanız, yazılımınızla sürücü arasında haberleşme sağlayacak arayüz geliştirmeniz gerekiyor. Nerede kaldı şimdi tasarım yapmanın keyfi ? Nerede bizim işimizi kolaylaştıran araçlar ? Yani her bir özelliği kullandığımda başımı nerede ağrıtacak öngöremiyorsam o özelliği kullanmanın ne anlamı var ? Bu bir kabus.

Konu göründüğünden derin olduğu için Örs’ün bu konuda tüm dertlere devâ büyülü çözümler bulduğunu ileri sürmek hayalperestlik olur ama C iletişimini zorlaştırmadan, çağrı yöntemini değiştirmeden, türleri yozlaştırmadan da yukarıdaki sorunları ele almak mümkün.

Şimdi konuyu derinleştirebiliriz:

#### Tür İşlemleri:[#](#tur-islemleri)

Yukarıdaki örnekte gösterildiği gibi tanımlı türlerin anlam kümelerinde yer alan işlemlerdir. C işlem çağrıları ile tam tutarlılığı vardır ve bu kaideyi bozmamak adına ilk değişkenleri pek çok yazılım dilinde olduğu gibi türün kendisidir. Mesela yukarıdaki C örneğini Örs’e çevirelim:

tür dizi  
{  
 hacim tam;  
 boyut tam;  
 İçerik \*tam;  
}  
  
Dizi dizi  
iş Yenile hacim tam  
{  
 Dizi.boyut = 0;  
 Dizi.hacim = hacim;  
 Dizi.İçerik = temiz(tam[boyut]);  
}  
  
Dizi \*dizi  
iş Topla toplanacakSayı tam  
{  
 her i:=0; i < Dizi->boyut; i++:  
 Dizi->İçerik[i] += toplanacakSayı;  
}  
  
// çağrı örneği:  
iş Giriş argümanSayısı tam, \_argümanlar \*\*t8  
{  
 değer tamDizisi dizi;  
 dizi.Yenile(20);  
 (&dizi).Topla(5);  
}

Baş kısmınında tanımlanan ilk değişken türün kendisi oluyor. Bu söz dizimi tanımlı işlemi türe bağlıyor. Fark edeceğiniz üzere türün derecesi var. Evet Örs dilinde tanımlı tür işlemlerinde ilk değişkenin tür kısmı önemli. Burada işlem bizden tanımlı türle yapılandırılmış 2 dereceli değer istiyorsa buna riayet edilmeli çünkü her ne kadar ilk değişken türe bağlıyor olsa da onun dışındaki davranışları herhangi bir diğer değişken ile **aynı**.

Nasıl türlere işlem bağlayabiliyorsak yalın türlere de aynı şekilde işlem bağlamak mümkün.

#### Sanal İşlemler:[#](#sanal-islemler)

Canlı işlem çağrısı yapmayan, türü illa ki belirli olmayan ve üretileceği zaman türü anlamlandırılan işlemlerdir. C’deki işlevsel dengi öngeçiş işlemleridir denilebilir ama özlerinde farklıdırlar. Öngeçiş işlemleri çözümleme sürecinde anlamlandırılmışken sanal işlemler üretim sürecine kadar anlamlı değillerdir. Sanal çağrılarının yapıldığı satıra gelince türleri belirlenir ve çağrıldığı yere yamanırlar.

İşlem çağırıları sanal olduğu için doğalarında hızlılardır ama derleme ürününün boyutunu artırırlar. Kullanım amaçlarından en önemlisi tekrar eden girdi çıktı işlemleridir. Bu işlemler özlerinde aynı şeyler olduklarından bunları mümkün olduğunca hızlı ve türlerini önemsemeden çağırmak isteriz. Bu örnek bir kaide olmamakla birlikte sanal işlemlerin uygulamadaki tasarım nedenlerinden bir tanesidir.

#### Sanal Tür İşlemleri:[#](#sanal-tur-islemleri)

Türlerle ilişkili bazı işler var ki bunların gerçek işlem olmasının anlamı bile yok. Mesela türlerin başlatılması, yenilenmesi, temizlenmesi gibi türler arası tekrar eden işlemler için canlı işlem çağrısı yapmayan, ilk türü hariç diğer türleri illa ki belirli olmayan ve üretileceği zaman anlamlandırılan işlemlere bu özel durumda ihtiyaç vardır. Mesela yukarıdaki örneği yeniden ele alalım:

tür dizi  
{  
 hacim tam;  
 boyut tam;  
 İçerik \*tam;  
}  
  
Dizi dizi  
sanal iş  
Yapılandır hacim tam  
{  
 Dizi.boyut = 0;  
 Dizi.hacim = hacim;  
 Dizi.İçerik = temiz(tam[hacim]);  
}  
  
Dizi \*dizi  
iş Topla toplanacakSayı tam  
{  
 her i:=0; i < Dizi->boyut; i++:  
 Dizi->İçerik[i] += toplanacakSayı;  
}  
  
// çağrı örneği:  
iş Giriş argümanSayısı tam, \_argümanlar \*\*t8  
{  
 değer tamDizisi dizi;  
 dizi.Yenile(20);  
 (&tamDizisi).Topla(5);  
}

Burada **‘Yapılandır’** işlemini sanal yaptık zira bu iş türü tekrar yapılandırmaktan başka bir görevi üstlenmiyor. Eğer ki sanal iş tanımlama özelliği olmasaydı **‘Giriş’** işleminde değeri oluşturduktan, fark ederseniz başlatma fiilini kullanmadım, sonraki satırlarda oluşturulan değeri yapılandıracaktı. Sanal türlerin yerinde kullanımına daha iyi örnek olamaz. Zira hızın bir önemi yok çünkü oluşturulan değeri zaten yapılandıracağız, ürün boyutundaki artışın bir önemi yok yapılandırmak zorunada olduğumuz için türün boyutu artacak. Özet olarak hız ve hafıza arasındaki ters orantı tahteravallisine binmeden bu sorunu okunabilir bir şekilde hallettik ki sanal türlerin uygulamadaki tasarım amaçlarından en önemlisi budur. Mesela şunu da yapabilir miydik ?

/\*yukarıdaki örneği tekrar ele alalım\*/  
...  
sanal iş  
Dizi hacim tam : dönüş dizi  
{  
 dönüş.boyut = 0;  
 dönüş.hacim = hacim;  
 dönüş.İçerik = temiz(tam[hacim]);  
}  
...  
iş Giriş argümanSayısı tam, \_argümanlar \*\*t8  
{  
 tamDizisi := Dizi(20);  
 (&tamDizisi).Topla(5);  
}

Bu örnekte tanımlı **‘Dizi’** işi sanal ama türe bağlı değil ve bize yapılandırılmış birinci dereceden dizi değeri dönüyor. Tahteravallide değiliz, temiz ve okunabilir kod.

#### Yalın Tür Sanal İşlemleri:[#](#yalin-tur-sanal-islemleri)

Yalın türlerin amaçlarından en önemlisi onlara tür bağlayabilmektir. En basitinden posix kütüphanesindeki işlemlerin çoğu **‘int’**, Örs dengi **‘tam’**, yapıtaşıyla işlem yapar ama bu yapıtaşlarının kendi görev alanlarında anlam kümeleri vardır. Bazen bu tam sayı hata anlamı taşırken, bazen açılan belge anlamını ve başka başka anlamları taşır.

Yalın türlerde sanal işlemler ile tüm bu anlam kargaşasının önüne geçebilir ve aynı yapıtaşlarının kullandığı gibi görünen ama özlerinde farklı olan işler tasarlayıp kavramlar arası çizgiyi belirginleştirebiliriz. Bu kavram sınırları önemli mesela elinizde **‘rgba’** renk türü vardır ve bu tür aynı anda hem resimlerde, hem oyunlarda farklı kavram alanlarında kullanılabilir. Tür aynı ama görevleri farklı.

Mesela:

tür belge tam;  
  
öz belge  
sanal iş  
Aç \_konum \*t8, izinler tam, mod tam : tam  
{  
 öz = sys::open(\_konum,  
 izinler,  
 mod);  
 eğer öz < 0:  
 {  
 durum error::no:  
 {  
 seçim error::code::Access:  
 stdio::printf("Erişim hatası\n");  
 varsayılan:  
 stdio::printf("bilinmeyen hata : %d.\n", öz);  
 }  
 }  
 dön öz;  
}

Yukarıda meşhur **‘open’** işleminin döneceği **‘tam’** türü belge olarak yalın halde tanımlanmış ve kavram alanında olası hatayı ele alan yeni bir işlem tasarlanmış. Bu görevi belge türündeki değerin oluşturulduğu yerde de yerine getirebilirdik ama burada aynı tür için olası tekrar eden sıralı satırlar kümesi var. Bir diğer deyişle sanal işlem kullanımı için mükemmel bir örnek.

##### Kalıp Sanal işlemler:[#](#kalip-sanal-islemler)

Nasıl gerçek olmayan, özlerinde sanal türler olan kalıplar var ise bunlara bağlanabilecek sanal işler de vardır. Kalıpların türü belirgin olmadığı için işlem tanımlayamayız ama zaten kalıplar uygulamadaki güçlerini buradan alıyor.

Yukarıdaki örnekte eğer ki **‘tam’** dizisi değil de ondalık dizi ya da istenilen türde bir dizi tanımlayıp işlemleri kullanmak isteseydik bunu yapamayacak olduğumuz gibi tekrar tanımlamak zorunda kalırdık. Tekrar aynı şeyi bu kadar temel bir algoritma ve tür için yazmanın anlamı yok. Örneği kalıpları kullanarak yeniden ele alalım

kalıp dizi x  
{  
 hacim tam;  
 boyut tam;  
 İçerik \*x;  
}  
  
Dizi dizi'x  
sanal iş  
Yapılandır hacim tam  
{  
 Dizi.hacim = hacim;  
 Dizi.boyut = 0;  
 Dizi.İçerik = temiz(%\*Dizi.Öğe[hacim]);  
 //% işlemiyle tür alıyoruz ki tür belirlendiğinde hesabı tutulabilsin.  
}  
  
Dizi dizi'x  
sanal iş  
Topla toplanacakSayı x  
{  
 her i:=0; i < Dizi->boyut; i++:  
 Dizi->İçerik[i] += toplanacakSayı;  
}  
  
// çağrı örneği:  
iş Giriş argümanSayısı tam, \_argümanlar \*\*t8  
{  
 değer tamDizisi dizi'ondalık;  
 dizi.Yapılandır(20);  
 tamDizisi.Topla(5.0);  
}

Özellikle bu konunun **‘dizi’** kalıbıyla anlatmayı seçtim zira bu kalıp o kadar temel ve o kadar çok kullanılan bir kalıp ki C’de ne kadar çok dizilerle ilgili aynı işlemi yazdım sayısını unuttum.

## Acemi Ocağı[#](#acemi-ocagi)

## Algoritmalar[#](#algoritmalar)

## Unix Ortamı[#](#unix-ortami)

## Tasarım[#](#tasarim)

Uyarı

Daha hala hazırlık aşamasındayız. Bu illaki kötü bir şey olacak diye bir kaide yok. Eğer yapıcı eleştirilerinizi iletirseniz hazırlık aşamasında katkıda bulunabilir ve Örs dilinin gelişimine kendi isteklerinizi de ekleyebilirsiniz.

İletişim için [musakalayci@gmail.com](mailto:musakalayci%40gmail.com) adresine yazarsanız, konusunu belirtirseniz eleştirilerinizi mümkün olduğunca dikkate alabilirim.

Hatta önerileriniz varsa iletişim kanallarını açık tutarız ve beraber sorunlara çözümler geliştirebiliriz.

Uslu çocuk olursak şirinleri bile görebiliriz.

* [Arama Sayfası](search.html)

By musakalayci  
© Copyright 2022, musakalayci.