ERCİYES ÜNİVERSİTESİ MÜHENDİSLİK FAKÜLTESİ BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ PROGRAMMING LABORATORY NOTLARI

2. HAFTA NOTU (C Programlama)

Bu not dersin 2. haftasında yapacağınız uygulamaya hazırlanmanızı sağlayacak olup temel kavramları anlatan bilgileri içermektedir.

DERSİN ÖĞRETİM ÜYESİ

Prof. Dr. Bahriye AKAY

DENEYİ YAPTIRAN ÖĞRETİM ELEMANI

Arş. Gör. Mert Korkut ÇOBAN

2026
KAYSERİ

1-) DENEYDE KULLANILACAK KÜTÜPHANELER VE FONKSİYONLAR HAKKINDA KISA BİLGİ

1.1 stdio.h

stdio.h (Standard Input Output), kullanıcı ile etkileşim kurmak için kullanılan standart giriş/çıkış kütüphanesidir. Bu deneyde printf() ile kullanıcıya yönergeler ve sonuçlar yazdırılır; fgets() ile kullanıcıdan matematiksel ifade satır olarak alınır. fgets(), satırı güvenli şekilde okuduğu için konsol uygulamalarında sıklıkla tercih edilir.

1.2 ctype.h

ctype.h, karakter sınıflandırma fonksiyonlarını içerir. Bu deneyde isdigit() fonksiyonu ile bir karakterin rakam olup olmadığı kontrol edilerek ifadede geçen çok basamaklı sayılar karakter karakter okunur ve tamsayıya dönüştürülür.

1.3 stdlib.h

stdlib.h, dinamik bellek yönetimi için kullanılan standart kütüphanedir. Bu uygulamada ifade ağacı (expression tree) oluşturulurken her düğüm malloc() ile dinamik olarak ayrılır ve işlem tamamlandığında free() ile bellek serbest bırakılır.

1.4 string.h

string.h, karakter dizileri üzerinde işlem yapmayı sağlayan standart kütüphanedir. Bu uygulamada zorunlu olmamakla birlikte, geliştirme aşamasında strlen(), strcmp() gibi fonksiyonlarla girişin boş olup olmadığı veya çıkış komutlarının kontrolü yapılabilir.

2-) DENEYDE KULLANILACAK PROGRAMLAMA YAPILARI HAKKINDA KISA BİLGİ

2.1 Veri Türleri ve Bellek Yapısı

2.1.1 Makrolar ve Sabit Tanımları

C dilinde #define yönergesi ile makro sabitler tanımlanır. Makrolar derleme aşamasında metinsel yer değiştirme (text substitution) yapar; yani derleyici kodu işlerken makro adı yerine tanımlanan değeri yerleştirir. Bu nedenle çalışma zamanında ekstra bellek alanı kullanmaz.

Bu deneyde MAX_EXPR gibi sabitler, dizi boyutlarının tek bir yerden yönetilmesini sağlar ve kodun okunabilirliğini artırır.

2.1.2 Temel Veri Türleri

C dilinde her değişken belirli bir veri türü ile tanımlanır ve bu tür, değişkenin bellekte kaplayacağı alanı ve saklayabileceği veri tipini belirler. int veri türü tamsayı değerleri temsil eder ve sayısal hesaplamalarda kullanılır. char veri türü tek bir karakter saklamak için kullanılır ve karakter dizilerinin (string) temel yapı taşıdır. struct ise birden fazla veri türünü tek bir yapı altında birleştirerek daha karmaşık veri yapıları oluşturmaya imkân tanır.

Bu deneyde kullanıcıdan alınan matematiksel ifade bir char dizisi (expr[]) içinde saklanır. İfadeden okunan sayılar ve işlem sonuçları int türüyle tutulur. Operatörler (+, -, *, /) ve parantez karakterleri ((,)) char olarak temsil edilir. Ayrıca ifade ağacını (expression tree) kurmak için Node adlı bir struct tanımlanır; bu yapıda düğümün sayı mı operatör mü olduğu (is_number), sayısal değer (value) veya operatör karakteri (op) ve çocuk düğümlere giden bağlantılar (left, right) birlikte tutulur. Bu sayede ifade, bellekte düzenli ve anlamlı bir biçimde temsil edilebilir.

2.1.3 Diziler (Arrays)

Diziler, aynı veri türünden birden fazla değeri ardışık bellek alanlarında saklamaya yarayan veri yapılarıdır. Bir dizi elemanına indeks (index) üzerinden erişilir ve dizinin boyutu tanımlandığı anda belirlenir. Bu yapı, özellikle çok sayıda veriyi düzenli biçimde tutmak ve hızlı erişmek için kullanılır.

Bu deneyde kullanıcıdan alınan ifade char expr[MAX_EXPR] dizisinde saklanır. İfade üzerinde dolaşmak için pointer kullanmadan, pos adlı bir indeks değişkeni ile expr[pos] üzerinden karakter karakter ilerlenir. Böylece hem çok basamaklı sayıları ardışık rakamlardan oluşturarak okumak, hem de operatör ve parantez karakterlerini sırayla kontrol etmek mümkün olur. Diziler, parsing sürecinin temel veri taşıyıcısı olarak programın daha düzenli ve anlaşılır çalışmasını sağlar.

2.1.4 Dinamik Bellek ve İşaretçiler (Ağaç Düğümleri)

C dilinde dinamik bellek yönetimi, çalışma zamanında ihtiyaç duyulan kadar bellek ayırıp daha sonra serbest bırakmayı sağlar. Bu amaçla malloc() fonksiyonu ile heap bölgesinden bellek ayrılır; iş bitiminde free() ile bu bellek geri verilir. Dinamik bellek, özellikle boyutu önceden kesin bilinmeyen veya çalışma sırasında büyüyüp küçülen veri yapılarında kullanılır.

Bu deneyde ifade ağacı (expression tree) oluşturulurken her düğüm için malloc() ile yeni bir Node alanı ayrılır. Düğümler birbirine left ve right işaretçileriyle bağlanarak ikili ağaç yapısı kurulur. Programın sonunda free_tree() fonksiyonu ile ağaç üzerindeki tüm düğümler recursive olarak dolaşılıp free() edilerek bellek sızıntısı (memory leak) oluşması engellenir. Bu yaklaşım, parantezli ifadelerin ağaç yapısında esnek ve doğru temsil edilmesini sağlar.

2.1.5 İfade Ağacı (Expression Tree)

İfade ağacı (expression tree), matematiksel bir ifadeyi ikili ağaç yapısında temsil eden bir veri yapısıdır. Bu ağaçta yaprak düğümler sayıları, iç düğümler ise operatörleri temsil eder. İfadede yer alan öncelik ilişkileri ve parantez yapısı, ağacın hiyerarşisine doğal olarak yansır.

Bu deneyde parsing işlemi sırasında her sayı için bir sayı düğümü, her operatör için ise iki alt ifadeyi birleştiren bir operatör düğümü oluşturulur. Örneğin * ve / gibi yüksek öncelikli işlemler, + ve - işlemlerinden önce ağaçta daha “alt seviyelerde” gruplanır. Parantezli yapılar ise ayrı bir alt ağaç olarak kurulup dış ifadeye bağlanır. Bu sayede ifade ağacı, hem ifadenin doğru şekilde modellenmesini sağlar hem de ağaç üzerinde yapılacak dolaşım ve hesaplama işlemlerini kolaylaştırır.

2.1.6 Özyineleme (Recursion) ve Recursive Descent Parsing

Özyineleme (recursion), bir fonksiyonun kendisini doğrudan veya dolaylı şekilde çağırarak problemi daha küçük alt problemlere bölmesi yaklaşımıdır. Bu yöntem, özellikle iç içe yapılar içeren problemlerde (parantezli ifadeler, ağaç dolaşımı gibi) oldukça etkilidir. Recursive descent parsing ise dilbilgisi (grammar) kurallarını fonksiyonlara bölerek ifadeyi adım adım ayrıştıran özyinelemeli bir parsing tekniğidir.

Bu deneyde parantezli matematiksel ifadeler expr → term → factor mantığıyla ayrıştırılır. factor seviyesinde bir sayı okunabilir veya (görülürse tekrar expr çağrılarak parantezin içi ayrı bir alt ifade olarak parse edilir ve) ile bu recursion seviyesi tamamlanır. Bu yaklaşım, iç içe parantezlerin doğal biçimde çözülmesini sağlar. Ayrıca oluşturulan expression tree hem parsing sırasında recursion ile kurulabilir, hem de sonucun hesaplanması için eval() fonksiyonu ile ağaç recursive olarak değerlendirilir.

2.2 Kontrol Yapıları

2.2.1 Döngüler

Döngüler, belirli bir koşul sağlandığı sürece kod bloklarının tekrar çalıştırılmasını sağlar. while döngüsü koşul doğru olduğu sürece çalışır ve genellikle tekrar sayısı önceden kesin olmayan durumlarda tercih edilir. Döngüler, veri üzerinde sıralı biçimde dolaşmayı ve adım adım işlem yapmayı kolaylaştırır.

Bu deneyde döngüler iki farklı amaçla kullanılır. İlk olarak, girdi ifadesi üzerinde karakter karakter ilerlerken ardışık rakamları okuyup çok basamaklı sayılar oluşturmak için while yapısından yararlanılır. İkinci olarak, parse_expr ve parse_term fonksiyonlarında operatör zincirleri (örneğin ardışık * ve / ya da + ve -) tüketilirken döngü ile aynı seviyedeki işlemler sırayla işlenir. Böylece ifade, doğru öncelik kurallarıyla ağaç yapısına dönüştürülebilir.

2.2.2 Koşul Yapıları

Koşul yapıları, programın farklı durumlara göre farklı davranışlar sergilemesini sağlar. if-else yapısı belirli bir koşulun doğru olup olmadığını kontrol ederek buna göre işlem yapılmasına imkân tanır. Bu sayede program, girişteki karakter veya durum değiştikçe uygun işlemi seçebilir.

Bu deneyde koşul yapıları, sıradaki karakterin sayı mı, operatör mü yoksa parantez mi olduğunu ayırt etmek için kullanılır. Örneğin isdigit() ile rakam kontrolü yapılarak sayı okunur; (görüldüğünde parantez içi ifade için recursion başlatılır; + - * / operatörlerine göre yeni operatör düğümleri oluşturulur. Ayrıca main fonksiyonunda kullanıcı q girerse programın sonlandırılması yine koşul yapıları ile kontrol edilir

3-)KOD ÖRNEKLERİ

3.1 Dinamik Bellek ve İşaretçiler

Örnek: Dinamik dizi ile ortalama hesaplama

```
#include <stdio.h>     /* printf, scanf */
#include <stdlib.h>    /* malloc, free */
int main(void) {
  int n, i;                                    /* n: eleman sayısı, i: döngü sayacı */
  int *arr;                                    /* dinamik olarak ayrılacak dizi işaretçisi */
  long sum = 0;                                /* toplam (taşmayı azaltmak için long) */
  printf("Kac sayi gireceksiniz? ");           /* kullanıcıdan eleman sayısını iste */
  scanf("%d", &n);                             /* n değerini oku */

  arr = (int*)malloc(n * sizeof(int));         /* n adet int için bellek ayır */
  if (arr == NULL) {                           /* ayırma başarısız mı? */
    printf("Bellek ayirma basarisiz!\n");
    return 1;                                  /* programı hata ile bitir */  }

  for (i = 0; i < n; i++) {                    /* 0'dan n-1'e kadar oku */
    printf("%d. sayi: ", i + 1);               /* kullanıcıyı yönlendir */
    scanf("%d", &arr[i]);                      /* i. elemanı diziye yaz */
    sum += arr[i];                             /* toplama ekle */ }
  printf("Ortalama = %.2f\n", (double)sum / n); /* ortalamayı yazdır */

  free(arr);                                   /* ayrılan belleği geri ver */
  return 0; }                                  /* başarılı bitiş */
```

- **Amaç:** Çalışma zamanında kullanıcıdan alınan n kadar sayı için malloc ile dizi ayırmak, ortalama hesaplamak ve free ile belleği temizlemek.
- **Önemli satırlar:**
  - malloc(n * sizeof(int)) → dinamik bellek ayırma
  - if (arr == NULL) → bellek ayırma kontrolü
  - free(arr) → bellek sızıntısını önleme

3.2 Özyineleme (Recursion)

Örnek: Dizide maksimum elemanı recursive bulma

```
#include <stdio.h>          /* printf */
/* a[] dizisinin ilk n elemanı içindeki maksimumu döndürür */
int max_recursive(int a[], int n) {
  if (n == 1)                          /* temel durum: tek eleman kaldıysa */
    return a[0];                       /* maksimum odur */
```

```
  /* alt problemin cevabı: ilk n-1 elemanın maksimumu */
  int m = max_recursive(a, n - 1);

  /* son eleman ile alt problemin sonucunu karşılaştır */
  return (a[n - 1] > m) ? a[n - 1] : m;
}
int main(void) {
  int arr[] = {7, 2, 9, 4, 5};                    /* örnek dizi */
  int n = 5;                                      /* eleman sayısı */

  printf("Maksimum = %d\n", max_recursive(arr, n));   /* çıktı: 9 */
  return 0;
}
```

- **Amaç:** Recursion'un temel mantığını göstermek: temel durum + kendini çağırma + sonuçları birleştirme.
- **Önemli satırlar:**
  - if (n == 1) → recursion'un duracağı temel koşul
  - max_recursive(a, n - 1) → problemi küçültüp kendini çağırma
  - return (a[n-1] > m) ? ... → alt sonuçla birleştirip final sonucu üretme

3.3 Ağaç Yapısı (Tree)

Örnek: Basit ikili ağaç kurma ve inorder dolaşma

```
#include <stdio.h>                /* printf */
#include <stdlib.h>               /* malloc, free */

/* Ağaç düğümü: isim (char) + iki çocuk bağlantısı */
typedef struct Node {
  char name;                      /* düğüm etiketi (örnek amaçlı) */
  struct Node *left;              /* sol çocuk */
  struct Node *right;             /* sağ çocuk */
} Node;

/* Yeni bir düğüm oluşturur ve başlangıçta çocukları NULL yapar */
Node* create(char ch) {
  Node *n = (Node*)malloc(sizeof(Node));      /* düğüm için bellek ayır */
  n->name = ch;                               /* etiketi yaz */
  n->left = NULL;                             /* sol çocuk yok */
  n->right = NULL;                            /* sağ çocuk yok */
  return n; }                                 /* düğüm adresini döndür */

/* Inorder dolaşım: sol -> kök -> sağ */
void inorder(Node *root) {
  if (root == NULL) return;                   /* temel durum: boş dal */
  inorder(root->left);                        /* önce sol alt ağaç */
```

```
  printf("%c ", root->name);              /* sonra kök */
  inorder(root->right);                   /* sonra sağ alt ağaç */
}

/* Ağaçtaki tüm düğümleri recursive olarak serbest bırakır */
void free_tree(Node *root) {
  if (root == NULL) return;               /* temel durum */
  free_tree(root->left);                  /* sol altı serbest bırak */
  free_tree(root->right);                 /* sağı serbest bırak */
  free(root);                             /* en son kendisini free et */
}

int main(void) {
  /* Basit bir ağaç kur:
        A
       / \
      B   C
     /
    D
  */
  Node *root = create('A');               /* kök düğüm */
  root->left = create('B');               /* A'nın solu */
  root->right = create('C');              /* A'nın sağı */
  root->left->left = create('D');         /* B'nin solu */

  printf("Inorder: ");
  inorder(root);                          /* çıktı: D B A C */
  printf("\n");

  free_tree(root);                        /* tüm düğümleri temizle */
  return 0;
}
```

- **Amaç:** İkili ağaç düğüm yapısını göstermek, düğümleri birbirine bağlayarak ağaç kurmak ve inorder dolaşım mantığını görmek.
- **Önemli satırlar:**
  - typedef struct Node { ... left, right ... } → ağaç düğüm yapısı
  - root->left = ... / root->right = ... → ağaç bağlantıları
  - inorder(root) → recursion ile ağaç dolaşımı
  - free_tree(root) → ağaçtaki tüm düğümleri güvenli serbest bırakma

Ders Katkıları

Bu uygulama kapsamında kullanılan yapılar, öğrencinin aşağıdaki teknik kazanımları edinmesini sağlar:

1. Algoritmik Düşünme

- İç içe parantezli ifadeyi adım adım ayrıştırma (parsing)
- Gramer temelli yaklaşım ile çözüm geliştirme (expr–term–factor)
- Operatör önceliğini (*/ önce, +- sonra) doğru yönetme stratejisi kurma

2. Özyineleme (Recursion) Yetkinliği

- Fonksiyonların kendini çağırdığı recursive yapıyı kurma ve takip etme
- Parantez açıldığında yeni alt ifade seviyesine geçme, kapandığında geri dönme mantığını kavrama

3. Veri Yapıları Bilinci: İfade Ağacı (Expression Tree)

- Matematiksel ifadeyi ağaç veri yapısı ile modelleme
- Düğüm kavramı (sayı düğümü / operatör düğümü) ve ikili ağaç bağlantıları (left/right)
- Ağaç üzerinde dolaşımın (inorder vb.) ifade gösterimindeki rolünü öğrenme

4. Dinamik Bellek Yönetimi

- malloc() ile düğüm oluşturma ve bellek ayırma
- free_tree() ile ağaç düğümlerini recursive serbest bırakma

5. Modüler Programlama

- Ayrıştırmayı ve değerlendirmeyi fonksiyonlara ayırma (parse_expr, parse_term, parse_factor, eval)
- Global indeks (pos) ile parser durum yönetimini görme (kolaylık/riski)
- Kod okunabilirliğini artıran isimlendirme ve yorum kullanımı

6. Girdi–Çıktı Kontrolü

- fgets() ile satır bazlı ifade okuma
- printf() ile kullanıcı yönlendirme, ağaç/sonuç yazdırma
- Konsol tabanlı döngü ile sürekli giriş alma ve q ile çıkış akışı kurma