Algoritmalar ve Programlama II

Hafta 14
İstisna Yönetimi

İstisna Yönetimi Mantığı

- C programlama dili performans ve donanım kontrolü odaklıdır.
- Karmaşık hata yönetimi yapıları yerine **basit ve doğrudan** yapılar sunar.

- **Amaç:** Hataları erken tespit etmek, kontrol altına almak ve güvenli şekilde programdan çıkmak.

Kullanılan Temel Yapılar

İf-else ile Koşullu Kontrol

- En yaygın hata yönetimi yaklaşımıdır. Beklenen bir durum gerçekleşmezse, ***if*** bloğu çalıştırılır ve kullanıcıya bilgi verilir ya da programdan çıkılır.

```
if (sart) {
    // işlemler
} else {
    // hata yönetimi
}
```

➢ Dosya açılamadığında ,bellek ayrılamadığında, kullanıcıdan yanlış giriş alındığında kullanılır.

Kullanılan Temel Yapılar

return ile fonksiyondan çıkma

- Bir fonksiyon içerisinde hata oluştuğunda, ***return*** kullanılarak işlem kesilir ve hata kodu döndürülür.

```c
int dosyaAc() {
    FILE *fp = fopen("veri.txt", "r");
    if (fp == NULL) {
        return -1; // hata kodu
    }
    // işlem
    fclose(fp);
    return 0; // başarı
}
```

- Ana fonksiyon bu dönüş değerlerini kontrol eder.

Kullanılan Temel Yapılar

if (return !=0) ile fonksiyonun dönüşüne göre aksiyon alma

```
int sonuc = dosyaAc();
if (sonuc != 0) {
    printf("Dosya açılamadı. Program sonlandırılıyor.\n");
    return 1;
}
```

Örnek

```c
#include <stdio.h>
// Fonksiyon: Kullanıcıdan sayı alır ve pozitiflik durumunu kontrol eder
int kontrolEt(int sayi) {
    if (sayi > 0) {
        printf("Sayı pozitiftir.\n");
        return 0; // başarı
    } else if (sayi < 0) {
        printf("Sayı negatiftir.\n");
        return 0; // başarı
    } else {
        printf("Sayı sıfırdır.\n");
        return 0; // başarı
    }
}

int main() {
    int sayi;
    printf("Bir tam sayı giriniz: ");
    // Giriş kontrolü: scanf başarılı oldu mu?
    if (scanf("%d", &sayi) != 1) {
        printf("Hatalı giriş! Lütfen bir tam sayı girin.\n");
        return 1; // hata durumu
    }
    // Fonksiyonu çağır
    int sonuc = kontrolEt(sayi);
    if (sonuc != 0) {
        printf("İşlem sırasında hata oluştu.\n");
        return 1;
    }
    return 0; // program başarıyla tamamlandı
}
```

Scanf başarılı bir şekilde int okursa sayı değişkenine atama yapıp 1 döner. Bu yüzden kontrol ederiz

Kullanılan Temel Yapılar

errno

- errno, C programlarında **sistem çağrıları** veya **standart kütüphane fonksiyonları** (örneğin fopen, malloc) başarısız olduğunda **neden başarısız olduğunu belirtmek için** kullanılan özel bir **global tamsayı değişkenidir**.
- errno değişkeni <errno.h> başlığı ile tanımlıdır.
- Her hata türü için farklı bir **pozitif tamsayı hata kodu** atanır.
- Bir fonksiyon başarısız olduğunda errno otomatik olarak güncellenir.

Kullanılan Temel Yapılar

errno

- Bir fonksiyon hata döndürdüğünde, sadece başarısız olduğunu bilirsiniz. Ancak **neden başarısız olduğunu** öğrenmek için errno kontrol edilir.

strerror(errno) Fonksiyonu

- <string.h> başlığı altında bulunur.
- errno değişkenindeki hata kodunu **açıklayıcı bir hata mesajına** çevirir.

Örnek

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

int main() {
    FILE *dosya = fopen("olmayan_dosya.txt", "r");  // Bilerek var olmayan bir dosya

    if (dosya == NULL) {  // Hata var mı kontrol et
        // errno otomatik olarak ayarlandı
        printf("Dosya açılamadı. errno: %d\n", errno);// Hata kodunu yazdır
        printf("Hata mesajı: %s\n", strerror(errno));// errno'yu açıklayıcı metne çevir
        return 1;  // Hata ile çık
    }
    // Eğer dosya açılmışsa, normal akış devam eder
    printf("Dosya başarıyla açıldı.\n");
    fclose(dosya);
    return 0;
}
```

```
Dosya açılamadı. errno: 2
Hata mesajı: No such file or directory
```

Kullanılan Temel Yapılar

setjmp, longjmp, jmp_buf

- Derin fonksiyon çağrıları zinciri içinde herhangi bir yerde hata olduğunda **doğrudan üst seviyeye geri dönmek** veya

- **Dallanma ve kurtarma (jump & recovery)** işlemleri yapmak gerektiğinde

- setjmp() ve longjmp() fonksiyonlarıyla **bir tür istisna yönetimi** (try-catch benzeri) yapılabilir.

Kullanılan Temel Yapılar

setjmp, longjmp, jmp_buf

`jmp_buf` : Atlamanın (jump) yapılabileceği konumu saklayan veri yapısı.

`setjmp()` : Geri dönülebilecek yeri kaydeder.

`longjmp()` : Daha önce `setjmp()` ile kayıt edilen yere atlar.

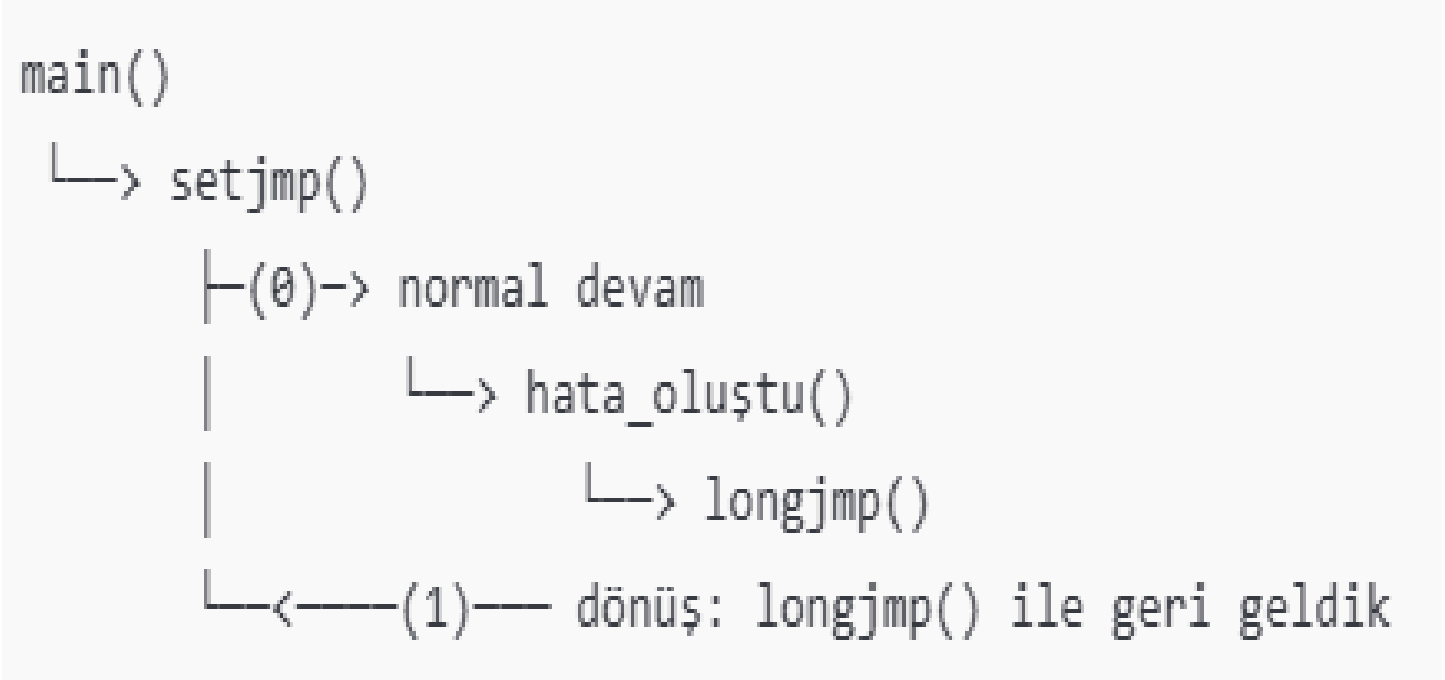
- Bu fonksiyonlar <setjmp.h> başlığı altında tanımlıdır.

Örnek

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

jmp_buf atlama_noktasi;

void hata_oluştu() {
    printf("Hata oluştu! Şimdi longjmp ile geri dönülüyor...\n");
    longjmp(atlama_noktasi, 1); // setjmp'e geri dön, dönüş değeri 1
}

int main() {
    if (setjmp(atlama_noktasi) == 0) {
        printf("setjmp ilk kez çağrıldı, şimdi hata fonksiyonunu çağırıyoruz.\n");
        hata_oluştu(); // longjmp çağrısı yapılacak
        printf("Bu satır çalışmaz çünkü longjmp ile zıplama yapıldı.\n");
    } else {
        printf("longjmp'den sonra geri dönüldü. Program kurtarıldı.\n");
    }

    return 0;
}
```

Kullanılan Temel Yapılar

setjmp, longjmp, jmp_buf Akış Diyagramı



Kullanılan Temel Yapılar

signal()

- signal() fonksiyonu, işletim sistemi tarafından gönderilen **sinyallere nasıl yanıt verileceğini** belirlemeye yarar.

```
#include <signal.h>

void (*signal(int signal_number, void (*handler)(int)))(int);
```

- Parametreler

`signal_number` : İzlenecek sinyal (örneğin `SIGINT`, `SIGFPE`, `SIGSEGV`, `SIGTERM`).

`handler` : Bu sinyal geldiğinde çalışacak fonksiyon (işleyici).

Kullanılan Temel Yapılar

signal()

Yaygın Sinyaller:

Sinyal	Açıklama	Tipik Neden
`SIGINT`	Program kesintisi	Ctrl+C
`SIGFPE`	Matematik hatası	0'a bölme
`SIGSEGV`	Bellek ihlali	Yanlış pointer kullanımı
`SIGTERM`	Program sonlandırma isteği	`kill` komutu
`SIGABRT`	Abort sinyali	`abort()` çağrısı

Kullanılan Temel Yapılar

exit()

- exit() fonksiyonu, programın çalışmasını **hemen sonlandırır** ve işletim sistemine bir çıkış kodu döndürür.

```
#include <stdlib.h>

void exit(int status);
```

`status = 0` : Başarıyla çıktı.

`status ≠ 0` : Hatalı çıkış.

Alternatif olarak `EXIT_SUCCESS` ve `EXIT_FAILURE` sabitleri de kullanılabilir.

Örnek

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

void ctrl_c_yakalandi(int sinyal) {
    printf("\nSIGINT yakalandı! (Ctrl+C bastınız)\n");
    printf("Program sonlandırılıyor...\n");
    exit(1);
}
int main() {
    // SIGINT sinyaline özel işleyici atıyoruz
    signal(SIGINT, ctrl_c_yakalandi);

    while (1) {
        printf("Program çalışıyor... Ctrl+C ile kesmeyi deneyin.\n");
        sleep(2);  // Her 2 saniyede bir mesaj
    }
    return 0;
}
```

Örnek

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

void sifira_bolme_handler(int sinyal) {
    printf("Hata: Sıfıra bölme tespit edildi! (SIGFPE)\n");
    exit(EXIT_FAILURE);
}
int main() {
    signal(SIGFPE, sifira_bolme_handler);  // Sıfıra bölme hatasını yakala

    int a = 5;
    int b = 0;
    int sonuc = a / b;  // SIGFPE sinyali üretir

    printf("Sonuç: %d\n", sonuc);
    return 0;
}
```

Kullanılan Temel Yapılar

signal()

- signal() ile hangi işlemler yapılmamalı?
  - signal() içindeki handler fonksiyon **yalnızca güvenli fonksiyonlar** kullanmalıdır. (Örn. printf() bazı sistemlerde güvenli olmayabilir)
  - Kritik dosya işlemleri veya malloc() gibi karmaşık işlevler sinyal içinde güvenli değildir.

Kullanılan Temel Yapılar

#define ile Makro Tanımlama

- Makrolar, C dilinin **ön işlemci direktifleri** kullanılarak tanımlanır ve derleyici tarafından kod derlenmeden önce işlenir.

```
#define MAKRONUN_ADI değeri

#define PI 3.14159
#define HATA_MESAJI "Hata oluştu!"
```

- Bu tanımlar sayesinde, kodda PI veya HATA_MESAJI yazıldığında, derleme öncesi bu değerlerle **yerine koyma** (substitution) yapılır.

Örnek

```c
#include <stdio.h>

#define PI 3.14159
#define KARE(x) ((x) * (x))
#define HATA_MESAJI "Geçersiz yarıçap! Lütfen pozitif bir değer giriniz."

int main() {
    double yaricap;

    printf("Dairenin yarıçapını giriniz: ");
    scanf("%lf", &yaricap);

    if (yaricap <= 0) {
        printf("%s\n", HATA_MESAJI);
        return 1;
    }

    double alan = PI * KARE(yaricap);
    printf("Yarıçapı %.2f olan dairenin alanı: %.2f\n", yaricap, alan);

    return 0;
}
```

Kullanılan Temel Yapılar

goto Deyimi

- goto, programın **belirli bir etiketli satıra atlamasını** sağlar. Genellikle önerilmez ancak istisna yönetimi gibi durumlarda yapay try-catch yapısı oluşturmak için kullanılabilir.

```
goto etiket;

...

etiket:
    // Burada işlem yapılır
```

Örnek

```c
int main() {
    int a = 5, b = 0, sonuc;

    if (b == 0) {
        goto hata;
    }

    sonuc = a / b;
    printf("Sonuç: %d\n", sonuc);
    return 0;

hata:
    printf("Hata: Sıfıra bölme yapamazsınız.\n");
    return 1;
}
```

Kullanılan Temel Yapılar

Try-catch Makroları (Sahte Yapılar)

- C dili, C++'ın try-catch-throw yapısını **doğrudan desteklemez**. Ancak, #define- goto- setjmp gibi yapılarla benzer davranışlar taklit edilebilir.

```
#include <stdio.h>
#include <setjmp.h>
jmp_buf hata_noktasi;
#define TRY if (setjmp(hata_noktasi) == 0)
#define CATCH else
#define THROW longjmp(hata_noktasi, 1)
void test_fonksiyon() {
    int x = 0;
    if (x == 0) {
        THROW;  // hata fırlat
    }
    printf("Bu satır çalışmaz.\n");
}
int main() {
    TRY {
        printf("TRY bloğu başladı\n");
        test_fonksiyon();
        printf("TRY bloğu bitti\n");
    }
    CATCH {
        printf("Hata yakalandı: sıfıra bölme ya da başka bir sorun.\n");
    }
    return 0;
}
```

Kaynaklar

- Doç. Dr. Caner ÖZCAN, KBÜ Yazılım Mühendisliği www.canerozcan.net
- Doç. Dr. Fahri Vatansever, “Algoritma Geliştirme ve Programlamaya Giriş”, Seçkin Yayıncılık, 12. Baskı, 2015.
- Kaan Aslan, “A’dan Z’ye C Klavuzu 8. Basım”, Pusula Yayıncılık, 2002.
- Paul J. Deitel, “C How to Program”, Harvey Deitel.
- “A book on C”, All Kelley, İra Pohl

Dinlediğiniz için teşekkürler