İSTİSNA YÖNETİMİ

Yapısal Programlamada Hata Yönetimi

Yapısal programlamada sorun olan, hata ayıklama kodu ile işlevsel kodun iç içe olması durumudur. Nesne yönelimli programlamanın getirdiği yenilikle bu sorun tarihe karışmıştır. Yapısal programlamada hatalı bir duruma yol açmamak ya da hatalı bir durumla karşılaşıldığında programdan çıkılır. Aşağıda örneği verilmiştir.

```
#include <stdio.h>
int faktoriyel(int sayi) {
  if (sayi<0) return -1;
  if(sayi <= 1) return 1;
  return sayi * faktoriyel(sayi - 1);
}
int main() {
  int okunan,deger,hata;
  do {
    printf("Bir Sayı Giriniz:");
    scanf("%d",&okunan);
    deger=faktoriyel(okunan);
    printf("%d nin faktöriyeli: %d\n", okunan, deger);
    if (deger==-1) {
       hata=100;
       break; // exit(100);
    }

  } while (okunan!=0);
  return hata;
}
/* Programın Çıktısı:
Bir Sayı Giriniz:-1
-1 nin faktöriyeli: -1

...Program finished with exit code 100
*/
```

İstisna Yönetimi

Nesne Yönelimli Programlamada ise istisna (exception), bir programın yürütülmesi sırasında ortaya çıkan beklenmeyen bir sorundur. İstisna, programın çalışması sırasında (run time) oluşur. Beklenmeyen durumlarda imal edilen istisna nesneleri (exception object) vardır. İstisna iki türlü ortaya çıkar;

- Eşzamanlı (synchronous): Giriş verilerindeki bir hata nedeniyle bir şeylerin ters gitmesi veya programın çalıştığı mevcut veri türünü işleme durumunda oluşan istisnalar (örneğin bir sayıyı sıfıra bölmek).
- Eşzamanlı olmayan (asynchronous): Disk arızası, klavye kesintileri vb. gibi yazılan programın kontrolü dışındaki istisnalar.

İstisnai durumları yönetmek için `throw`, `try` ve `catch` anahtar kelimeleri kullanılır;

- `try` talimatı: İçerisinde istisnai bir duruma düşüleceğimizi belirten bir kod bloğunu temsil eder. Bu bloğu bir veya daha fazla `catch` bloğu takip eder.
- `catch` talimatı: `try` bloğunda bir istisnai duruma düşüldüğünde yürütülecek bir kod bloğunu temsil eder. İstisnayı işlemek için kullanılan kod, `catch` bloğunun içine yazılır.

- Bir istisnai duruma `throw` anahtar sözcüğü kullanılarak da düşülebilir. Bir program bir `throw` ifadesiyle karşılaştığında hemen içinde bulunduğu `try` bloğu dışına çıkılır ve ilgili istisnayı işlemek için eşleşen bir `catch` bloğu bulmaya başlar.
- Bir `try` bloğu içinde birden fazla `throw` ifadesi bulunabilir.

```
try {
   /*
    ...
    İstisnai Durumun Ortaya Çıkabileceği Kod...
    ...
   */
   throw SomeExceptionType("İstisnai Durum Açıklaması");
   /*
    throw an exception: istisnai bir duruma düşme
   */
 } catch( ExceptionName e1 )  {
   /*
    catch bloğu try bloğundaki istisnayı yakalamak ve gerekli işlemi yapmak için kullanılır.
   */
 }
```

İstisna işlemeye niçin ihtiyaç duyarız?

- Hata işleme kodunun işlevsel koddan ayrılması için.
- Yöntemler yalnızca seçtikleri istisnaları işleyebilir. Bir yöntemde birden çok istisnai duruma düşülebilir. Ancak bunlardan bazılarını `catch` bloğunda işlemeyi seçebilir. Geri kalanlarını yöntemi çağıran yere bırakabilir.
- Hata Türlerinin Gruplanması: C++ dilinde hem temel tipler (int, float, string ...) hem de nesneler istisnai durum olarak belirlenebilir. Böylece bunların hiyerarşik olarak gruplaması yapılabilir.

Aşağıda sıfıra bölme hatasını işleyen bir program örneği verilmiştir;

```
#include <iostream>
#include <stdexcept>
using namespace std;
int main() {
   cout << "Burası her zaman icra edilir." << endl;
   try { //try BLOĞU: İstisna olabilecek bir bok bloğu
      int bolunen = 10;
      int bolen = 0;
      int sonuc;
      if (bolen == 0) {
         throw runtime_error("Sıfıra Bölmeye İzin Verilmez!");
         //Buradan sonra yazılacak kodlar icra edilmez.
         cout << "Burası hiçbir zaman icra edilmez." << endl;
      }
      sonuc = bolunen / bolen;
      cout << "Bölme Sonucu: " << sonuc << endl;
   } catch (const exception& e) {
      cout << "İstisna: " << e.what() << endl;
   }
   cout << "Burası da her zaman icra edilir." << endl;
   return 0;
}
/* Programın Çıktısı:
Burası her zaman icra edilir.
İstisna: Sıfıra Bölmeye İzin Verilmez!
Burası da her zaman icra edilir.

...Program finished with exit code 0
*/
```

Yandaki örnekte `try` bloğunda izin verilmeyen bölme işleminde istisnai bir durum fark ediliyor ve istisna nesnesi imal ediliyor ve `throw` ile (istisna havuzuna) fırlatılıyor (throw an exception). `catch` bloğunda ise (havuzdan) istisna nesnesi yakalanarak (catch an exception) işleniyor. Örnekte standart `runtime_error` istisna nesnesi kullanılmıştır. Bu nesneler ön tanımlı nesnelerdir. `stdexcept` başlığında aşağıdaki istisnalar ön tanımlıdır; `runtime-error`, `logic_error`, `invalid_argument`, `range_error`, `out_of_range`, `overflow_error`, `underflow_error`.

Bir `try` bloğunda birden fazla istisna nesnesi (istisna havuzuna) fırlatılabilir; Aşağıdaki örnekte `try` bloğunda birden fazla istisnai nesnesi fırlatılmıştır.

```
#include <iostream>
using namespace std;
int main()
{
   try {
      throw 10;
      throw 'A';
      throw "Hata";
   }
   catch (char* excp) {
      cout << "İstisna: " << excp;
   }
   catch (...) { //default exception üç nokta ile belirtilir.
      cout << "Diğer Catchlerde işlenmeyen istisnalar burada işlenir.\n";
   }
   return 0;
}
/* Programın Çıktısı:
Diğer Catchlerde işlenmeyen istisnalar burada işlenir.

...Program finished with exit code
*/
```

Programda ilk karşılaşılan `throw` ile (istisna havuzuna) fırlatılan nesne tamsayı nesnesidir. Bu durumda `catch` bloklarına sırasıyla bakılır ilk (havuzdan) yakalanacak olan dizgi (string) nesneleridir. Havuzda dizgi nesnesi olmadığından bu `catch` bloğu icra edilmez. Bir sonraki `catch` bloğuna bakılır ve böyle devam edilir. Eğer havuzdan yakalanacak istisna nesnesi önceki `catch` bloklarında tanımlı değilse üç nokta ile belirtilen `catch` bloğu icra edilir.

Aşağıda Kişi sınıfta yaş özelliğine ilişkin istisna içeren bir kod örneği verilmiştir;

```
#include <iostream>
#include <stdexcept>
using namespace std;
class Kisi {
private:
    int yas;
public:
    Kisi() {
        yas=15;
    }
    int getYas() {
        return yas;
    }
    void setYas(int pYas) {
      if (pYas < 0)
        throw range_error("Yaş değeri sıfırdan küçük olamaz!");
      else if (pYas > 120)
        throw range_error("Çok Büyük Yaş Değeri: >120!");
      yas = pYas;
    }
```

```
};
int main() {
    Kisi ali;
    try {
        ali.setYas(12);
        cout <<ali.getYas()<<endl;
        ali.setYas(-1); // Bu talimatta istisna nesnesi oluşur.
        //Buradan sonrası icra edilmez!
        cout <<ali.getYas()<<endl;
        ali.setYas(130);
        cout <<ali.getYas()<<endl;
    }
    catch (const exception& istisna) {
        cout << "İstisnai Bir Durum Yakalandı!" << endl;
        cout << "Durum Açıklaması: " << istisna.what()  << endl;
    }
}
/* Programın Çıktısı:
12
İstisnai Bir Durum Yakalandı!
Durum Açıklaması: Geçersiz Yaş Değeri: <0!

...Program finished with exit code 0
*/
```

Yukarıdaki kod örneğindeki gibi her zaman standart istisna nesneleri kullanmak zorunda değiliz. Bu durumda `exception` başlığını projemize dahil ederek kullanarak kendi istisna nesnemizi de oluşturabiliriz.

```
#include <iostream>
#include <exception>
using namespace std;

struct yasMaxException: public exception {
   const char* what () const throw () {
      return "Çok Büyük Yaş Değeri: >120!";
   }
};
struct yasMinException: public exception {
   const char* what () const throw () {
      return "Geçersiz Yaş Değeri: <0!";
   }
};

class Kisi {
private:
    int yas;
public:
    Kisi() {
        yas=15;
    }
    int getYas() {
        return yas;
    }
    void setYas(int pYas) {
      if (pYas < 0)
        throw yasMinException();
      else if (pYas > 120)
        throw yasMaxException();
      yas = pYas;
    }
```

```
};
int main() {
    Kisi ali;
    try {
        ali.setYas(12);
        cout <<ali.getYas()<<endl;
        ali.setYas(-1); // Bu talimatta istisna nesnesi oluşur.
        //Buradan sonrası icra edilmez!
        cout <<ali.getYas()<<endl;
        ali.setYas(130);
        cout <<ali.getYas()<<endl;
    }
    catch (const exception& istisna) {
        cout << "İstisnai Bir Durum Yakalandı!" << endl;
        cout << "Durum Açıklaması: " << istisna.what()  << endl;
    }
}
/* Programın Çıktısı:
12
İstisnai Bir Durum Yakalandı!
Durum Açıklaması: Geçersiz Yaş Değeri: <0!

...Program finished with exit code 0
*/
```

noexcept İşleci

İşlenen değerlendirmesinin bir istisna nesnesi imal edip etmeyeceğini belirleyen tekli işleç `noexcept` işlecidir. Çağırılan yöntem ya da fonksiyonların gövdelerinin incelenmediğini, dolayısıyla `noexcept` işlecinin yanlış sonuçlar verebileceğini unutulmamalıdır;

```
#include <iostream>
#include <stdexcept>
using namespace std;
void fonksiyon() { throw std::runtime_error("oops"); }
void bosfonksiyon() {}
struct yapi {};
int main() {
    cout << noexcept(fonksiyon()) << endl; // 0
    cout << noexcept(bosfonksiyon()) << endl; // 0
    cout << noexcept(1 + 1) << endl; // 1
    cout << noexcept(yapi()) << endl; // 1
}
```

Bu işleç, bir fonksiyon bildirilirken, fonksiyonun gövdesinde bir istisna nesnesi imal edip etmeyeceğini belirtmek için de kullanılır;

```
void f1() { throw std::runtime_error("oops"); }
void f2() noexcept(false) { throw std::runtime_error("oops"); }
void f3() {}
void f4() noexcept {}
void f5() noexcept(true) {}
void f6() noexcept {
    try {
        f1();
    } catch (const std::runtime_error&) {}
}
```

Bu örnekte, `f4`, `f5` ve `f6` fonksiyonlarının istisna imal etmeyeceğini beyan ettik. (Bir istisna `f6` fonksiyonunun yürütülmesi sırasında atılabilse de yakalanır ve fonksiyondan dışarı yayılmasına izin verilmez.) `f2` fonksiyonunun bir istisnayı imal edebileceğini beyan ettik. `f1` ve `f3` fonksiyonlarının

istisnaları imal edeceğini dolaylı olarak beyan ettik. `noexcept` belirteci atlandığında, `noexcept(false)` ile eşdeğerdir.