Nesne Yönelimli Programlama

Bellek Yönetimi

Pointerlar (İşaretçiler)

Pointer Kavramı ve Kullanımı

- **Pointer Tanımı**: Pointer, bir değişkenin bellekteki adresini tutan bir veri türüdür. Veri türünden bağımsız olarak bellekteki adresi işaret eder.
- **Özellikleri:** Hafıza yönetiminde esneklik sağlar ve daha verimli kod yazılmasını mümkün kılar.
- **Pointer Tanımlama**:

```
int x = 5;
int *ptr = &x; // x'in adresini ptr işaretçisi ile tutarız
```

- **Pointer ile İşlem Yapma**:
  - İşaretçiyi kullanarak, işaret ettiği değişkenin değerine ulaşabiliriz.

```
std::cout << "x'in değeri: " << *ptr << std::endl; // *ptr ile x'in değerini okuruz
```

- **Null Pointer**: Herhangi bir adresi işaret etmeyen işaretçilerdir.

```
int *ptr = nullptr;
```

Pointerda Kullanılan Semboller:

Sembol	Adı	Açıklama
`&`	Adres Operatörü	Bir değişkenin bellekteki adresini belirler.
`*`	Dolaylılık Operatörü	Adresteki değere erişim sağlar (pointer içeriği).

Bu semboller, işaretçilerle birlikte bellek adresleri üzerinde işlem yapmayı ve adreslenen verilere ulaşmayı mümkün kılar.

String Literalleri Nedir?

- **Tanım**: Kod içerisinde doğrudan tanımlanmış, çift tırnak içinde belirtilen karakter dizileri.

```
const char *str = "Merhaba"; // String literal bir karakter dizisidir
```

- **Özellikleri**: Değiştirilemezler; bellekte sabit bir konumda tutulurlar.

Void Pointer Nedir?

- **Void Pointer (Boş İşaretçi)**: Belirli bir veri türünü göstermeyen ve herhangi bir veri türüne atanabilen işaretçidir.
- C++'ta genellikle farklı veri tiplerini tek bir işaretçi ile işlemek için kullanılır.

Özellikleri

- **Tip Belirtilmez**: `void *ptr;` olarak tanımlanır ve int, float, double gibi türler kullanılmaz.
- **Dönüştürme Gerektirir**: `void` işaretçiyi kullanırken bir veri türüne dönüştürülmesi gerekir.

Örnek

```
1   int a = 5;
2   void *ptr = &a;
3   std::cout << *(int*)ptr; // Tip dönüşümü yapılarak erişim sağlanır
```

Geçersiz Pointer (Invalid Pointer)

Geçersiz pointer, başlangıç değeri verilmeden ya da bellekte geçerli bir adresi göstermeyen pointerlara denir. Bu pointerlar yanlış bellek adreslerini işaret edebilir ve kullanımları program çökmesine veya bellek hatalarına yol açabilir.

Geçersiz Pointer Örnekleri

1. **Başlangıç Değeri Verilmeyen Pointer**:

```
int *ptr;  // ptr herhangi bir adresi göstermiyor (geçersiz)
```

2. **Silinmiş Adresi Gösteren Pointer**:

```
int *ptr = new int(5);
delete ptr;  // ptr artık geçersiz
```

3. **Null Pointer Olmayan ve Tanımlanmamış Pointer**:

```
int *ptr = nullptr;
ptr++;  // Null pointer'ı artırmak geçersiz bir adrese götürür
```

Geçersiz pointerlardan kaçınmak için pointerlara `nullptr` atanarak veya bellekte uygun bir adres gösterildiğinden emin olunarak güvenli kod yazılabilir.

İşaretçiye İşaretçi(Pointer to Pointer)

- İşaretçiye işaretçi, başka bir işaretçinin adresini tutan işaretçidir.
- Bir işaretçinin bellekteki adresini işaret ederek katmanlı veri erişimine olanak sağlar.

Örnek

```
1   int x = 10;
2   int *p = &x;      // x'in adresini tutan işaretçi
3   int **pp = &p;    // p'nin adresini tutan işaretçiye işaretçi
```

Kullanım Alanları

- Çok boyutlu diziler
- Dinamik bellek yönetimi
- Gelişmiş veri yapılarına erişim

Erişim

- `**pp` ifadesi, işaret edilen verinin asıl değerini döndürür.

Pointer Aritmetiği

- Pointerlar üzerinde aritmetik işlemler yapılabilir. Bir işaretçinin değeri, işaret edilen veri türünün boyutuna göre artar veya azalır.
- **Örnek**:

```
int dizi[3] = {10, 20, 30};
int *ptr = dizi; // int *ptr = &dizi[0];
std::cout << *ptr << std::endl;      // İlk eleman (10)
std::cout << *(ptr + 1) << std::endl; // İkinci eleman (20)
```

- **Artırma ve Azaltma**:
  - `ptr++` : İşaretçinin işaret ettiği adresi bir sonraki veri bloğuna taşır.
  - `ptr--` : İşaretçinin işaret ettiği adresi bir önceki veri bloğuna taşır.

Diziler ve Pointerlar

- **Dizi ve Pointer İlişkisi**:
  - Bir dizinin adı, dizinin ilk elemanının adresini verir, yani `int arr[3];` tanımlandığında `arr` bir pointer gibi çalışır.
- **Pointer ile Dizi Elemanlarına Erişim**:

```
int arr[3] = {1, 2, 3};
int *ptr = arr;
for (int i = 0; i < 3; i++) {
    std::cout << *(ptr + i) << std::endl;
}
```

Fonksiyonlarda Pointer Kullanımı

- Fonksiyonlarda pointer kullanarak değişkenleri doğrudan bellekte güncelleyebiliriz.
- **Call by Value vs. Call by Reference (Pointer ile)**:
  - Normal değişkenler değer olarak gönderilir (call by value).
  - Pointer kullanarak değişkenlerin bellekteki adresini gönderip doğrudan değiştirebiliriz (call by reference).
- **Örnek**:

```
void guncelle(int *p) {
    *p = 20; // Bellekteki adres üzerinden değişkenin değeri güncellenir
}

int main() {
    int x = 10;
    guncelle(&x);
    std::cout << x << std::endl; // 20 olarak güncellenmiş x değeri
}
```

Referanslar ve İşaretçiler
1. Değer Bazında Çağrı

- **Tanım**: Fonksiyonlara doğrudan değişken değeri gönderilir. Değişiklikler yalnızca fonksiyon içinde etkilidir.
- **Özellik**: Orijinal değişkeni değiştirmez.

```
1   void fonksiyon(int x) { x = 5; }
```

2. İşaretçi Argümanıyla Referansa Göre Çağrı

- **Tanım**: Değişkenin adresi gönderilir, böylece orijinal değer değiştirilebilir.

```
1   void fonksiyon(int *x) { *x = 5; }
```

3. Referans Argümanı ile Referansa Göre Çağrı

- **Tanım**: Doğrudan değişkenin referansı gönderilir ve değişiklikler orijinal değeri etkiler.

```
1   void fonksiyon(int &x) { x = 5; }
```

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

// 1. Pass-by-Value (Değerle Çağrı)
int square1(int n) {
    cout << "square1() içindeki n1'in adresi: " << &n << "\n";
    n *= n;  // Sadece yerel n değişkeni etkilenir
    return n;
}

// 2. Pass-by-Reference with Pointer Arguments (İşaretçi ile Referans Çağrısı)
void square2(int* n) {
    cout << "square2() içindeki n2'nin adresi: " << n << "\n";
    *n *= *n;  // Orijinal n2 değişkeni etkilenir
}

// 3. Pass-by-Reference with Reference Arguments (Referans ile Çağrı)
void square3(int& n) {
    cout << "square3() içindeki n3'ün adresi: " << &n << "\n";
    n *= n;  // Orijinal n3 değişkeni etkilenir
}
```

```
void example() {
    // Değer ile Çağrı Örneği
    int n1 = 8;
    cout << "Ana fonksiyondaki n1'in adresi: " << &n1 << "\n";
    cout << "n1'in karesi: " << square1(n1) << "\n";
    cout << "n1'in değeri (değişmedi): " << n1 << "\n";

    // İşaretçi ile Referans Çağrısı Örneği
    int n2 = 8;
    cout << "Ana fonksiyondaki n2'nin adresi: " << &n2 << "\n";
    square2(&n2);
    cout << "n2'nin karesi: " << n2 << "\n";
    cout << "n2'nin değeri (değişti): " << n2 << "\n";

    // Referans ile Çağrı Örneği
    int n3 = 8;
    cout << "Ana fonksiyondaki n3'ün adresi: " << &n3 << "\n";
    square3(n3);
    cout << "n3'ün karesi: " << n3 << "\n";
    cout << "n3'ün değeri (değişti): " << n3 << "\n";
}

// Ana Fonksiyon
int main() {
    example();
}
```

Açıklamalar

- **Değer ile Çağrı (`square1`)**: Fonksiyona gönderilen değişken kopyalanır; fonksiyon içinde yapılan değişiklikler orijinal değişkeni etkilemez.
- **İşaretçi ile Referans Çağrısı (`square2`)**: Orijinal değişkenin adresi geçilir, fonksiyon içindeki değişiklikler orijinal değişkeni etkiler.
- **Referans ile Çağrı (`square3`)**: Değişkenin kendisi geçilir, bu yüzden değişiklikler doğrudan orijinal değişkeni etkiler.

C++'da Bellek Yönetimi

Bellek yönetimi, genel sistem performansını iyileştirmek için bellek alanını programlara atayarak bilgisayar belleğini yönetme sürecidir. C++ programlama dilinde bellek yönetimi, geliştiricilerin dinamik bellek ayırma ve serbest bırakma işlemlerini kontrol etmesine olanak tanır.

C dilinde, belleği çalışma zamanında dinamik olarak tahsis etmek için `malloc()` veya `calloc()` fonksiyonları kullanılır. Dinamik olarak tahsis edilen belleğin tahsisini kaldırmak için `free()` fonksiyonu kullanılır. C++ ise bu fonksiyonları desteklerken, aynı zamanda bellek ayırma ve serbest bırakma işlemleri için `new` ve `delete` gibi tekil operatörleri de tanımlar.

Bellek Yönetimi Türleri

1. **Statik Bellek Yönetimi**
   - Derleme zamanında bellek ayırma.
   - Global ve statik değişkenler için kullanılır.
   - Bellek alanı programın yaşam süresince ayrılmış kalır.
2. **Dinamik Bellek Yönetimi**
   - Çalışma zamanında bellek ayırma.
   - Kullanıcının ihtiyacına göre bellek alanı ayırma ve serbest bırakma.
   - `new` ve `delete` operatörleri kullanılır.

C Dilinde Bellek Yönetimi

1. `malloc()`

- Dinamik bellek ayırmak için kullanılır.
- Belleği ayırır, ancak bu belleği başlatmaz.

```
1   int* ptr = (int*)malloc(sizeof(int)); // 1 int için bellek ayır
```

2. `calloc()`

- Dinamik bellek ayırmak için kullanılır.
- Belleği ayırır ve başlatır (tüm baytları sıfıra ayarlar).

```
1   int* arr = (int*)calloc(5, sizeof(int)); // 5 elemanlı dizi için bellek ayır
```

3. `free()`

- Daha önce tahsis edilen belleği serbest bırakmak için kullanılır.

```
1   free(ptr); // Tek değişken için bellek serbest bırak
2   free(arr); // Dizi için bellek serbest bırak
```

C++ Dilinde Bellek Yönetimi

1. `new`

- Dinamik bellek ayırmak için kullanılır.
- Belleği ayırır ve bu belleği varsayılan bir değerle başlatır.

```
int* ptr = new int; // 1 int için bellek ayır
```

2. `new[]`

- Dizi için dinamik bellek ayırmak için kullanılır.

```
int* arr = new int[5]; // 5 elemanlı dizi için bellek ayır
```

3. `delete`

- Daha önce `new` ile tahsis edilen belleği serbest bırakmak için kullanılır.

```
delete ptr; // Tek değişken için bellek serbest bırak
```

4. `delete[]`

- Daha önce `new[]` ile tahsis edilen dizi belleğini serbest bırakmak için kullanılır.

```
delete[] arr; // Dizi için bellek serbest bırak
```

C ve C++ Arasındaki Farklar

- **Fonksiyonlar vs. Operatörler**: C, `malloc()`, `calloc()`, ve `free()` gibi fonksiyonlar kullanırken, C++ `new`, `new[]`, `delete`, ve `delete[]` gibi operatörler kullanır.
- **Tip Güvenliği**: C++'da `new` operatörü, uygun türde bir işaretçi dönerken, `malloc()` ve `calloc()` fonksiyonları her zaman `void*` döner, bu nedenle tür dönüşümü yapılması gerekir.
- **Kapsamlı Başlatma**: `new` operatörü, tahsis edilen belleği varsayılan değeriyle başlatırken, `malloc()` belleği başlatmaz.

Dinamik Bellek Yönetimi

1. Bellek Ayırma

- C++'da dinamik bellek ayırma işlemi `new` operatörü ile yapılır.

```
int* ptr = new int; // Tek bir int için bellek ayırma
int* arr = new int[10]; // 10 elemanlı bir dizi için bellek ayırma
```

2. Bellek Serbest Bırakma

- Bellek serbest bırakma işlemi `delete` ve `delete[]` operatörleri ile yapılır.

```
delete ptr; // Tek bir int için belleği serbest bırakma
delete[] arr; // Dizi için belleği serbest bırakma
```

Bellek Yönetimi Örnekleri

Örnek 1: Tek Değişken için Bellek Ayırma

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

int main() {
    int* num = new int; // Bellek ayır
    *num = 42; // Değer ata
    cout << "Değer: " << *num << endl;
    delete num; // Belleği serbest bırak
    return 0;
}
```

Örnek 2: Dizi için Bellek Ayırma

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

int main() {
    int* array = new int[5]; // Dizi için bellek ayır
    for (int i = 0; i < 5; ++i) {
        array[i] = i * 10; // Değer ata
    }
    for (int i = 0; i < 5; ++i) {
        cout << array[i] << " "; // Değerleri yazdır
    }
    delete[] array; // Belleği serbest bırak
    return 0;
}
```

Bellek Sızıntıları

- Bellek sızıntıları, dinamik olarak ayrılan bellek alanlarının serbest bırakılmaması durumunda ortaya çıkar.
- Bellek sızıntılarını önlemek için her `new` işlemi sonrası uygun bir `delete` işlemi yapılmalıdır.

Garbage Collector (Çöp Toplayıcı)

Garbage Collector (GC), programlama dillerinde otomatik bellek yönetimi sağlayan bir mekanizmadır. GC, programın çalışması sırasında kullanılmayan veya erişilemeyen bellek alanlarını tespit ederek bu belleği serbest bırakır. Bu işlem, bellek sızıntılarını önlemeye ve genel sistem performansını artırmaya yardımcı olur.

Bellek Yönetiminde Zorluklar

- **Dinamik Bellek Tahsisi**: Geliştiriciler, dinamik olarak bellek tahsis ederken dikkatli olmalıdır.
- **Bellek Sızıntıları**: Kullanılmayan bellek alanlarının serbest bırakılmaması.
- **Büyük Veri Yapıları**: Karmaşık veri yapılarının yönetimi zor olabilir.

Çöp Toplayıcının Amaçları

1. **Bellek Yönetimini Otomatikleştirmek**: Geliştiricilerin bellek yönetimiyle uğraşmalarına gerek kalmadan uygulama geliştirmelerine olanak tanır.
2. **Bellek Sızıntılarını Önlemek**: Kullanılmayan nesneleri otomatik olarak serbest bırakarak belleğin etkin kullanımını sağlar.
3. **Performansı Artırmak**: Bellek yönetimini optimize ederek uygulama performansını artırır.

Çöp Toplayıcı Çeşitleri

1. **Referans Sayımı (Reference Counting)**
   - Her nesne için bir referans sayısı tutar.
   - Referans sayısı sıfıra düştüğünde bellek serbest bırakılır.
   - **Avantaj**: Hızlı serbest bırakma.
   - **Dezavantaj**: Çember referansları (circular references) sorun yaratabilir.

2. **Köklerden Temizleme (Root Scanning)**
   - Uygulama çalışırken erişilebilen nesneleri tarar.
   - Erişilemeyen nesneleri tespit edip serbest bırakır.
   - **Avantaj**: Çember referanslarını yönetebilir.
   - **Dezavantaj**: Performans maliyeti yüksektir.

3. **Generational Garbage Collection**
   - Nesneleri "genç" ve "yaşlı" olarak sınıflandırır.
   - Genç nesneler daha sık temizlenir; yaşlı nesneler daha az sıklıkla temizlenir.
   - **Avantaj**: Genellikle daha iyi performans sağlar.
   - **Dezavantaj**: Uygulama karmaşıklaşabilir.

Çöp Toplayıcının Çalışma Prensibi

1. **Nesne Takibi**: Uygulama çalışırken hangi nesnelerin aktif olduğunu takip eder.
2. **Çöp Toplama Zamanlaması**: Belirli zaman aralıklarında veya bellek baskısı altında devreye girer.
3. **Temizleme**: Kullanılmayan nesneleri tespit ederek bellek alanını serbest bırakır.

Çöp Toplayıcının Avantajları

- **Kullanıcı Dostu**: Geliştiricilerin bellek yönetimiyle daha az uğraşmasını sağlar.
- **Bellek Sızıntılarını Azaltır**: Kullanılmayan belleğin otomatik olarak serbest bırakılmasını sağlar.
- **Geliştirilmiş Performans**: Uygulama performansını artırarak daha verimli bellek kullanımı sağlar.

Çöp Toplayıcının Dezavantajları

- **Performans Maliyeti**: Çöp toplama işlemi, uygulamanın genel performansını etkileyebilir.
- **Belirsizlik**: Nesnelerin ne zaman serbest bırakılacağı belirsizdir, bu da bellek kullanımını tahmin etmeyi zorlaştırabilir.
- **Hafıza Kullanımı**: Dinamik bellek kullanımı nedeniyle bellek baskısı yaratabilir.

Sonuç

Garbage Collector, modern programlama dillerinde bellek yönetimini otomatikleştiren önemli bir bileşendir. Çöp toplama mekanizmaları, bellek sızıntılarını önlemeye ve uygulama performansını artırmaya yardımcı olurken, belirli dezavantajlar da taşıyabilir. Geliştiricilerin bu mekanizmaları anlaması, daha etkili ve verimli uygulamalar geliştirmelerine olanak tanır.

C++'da Bellek Yönetimi Örnekleri

Örnek 1: Dinamik Dizi Oluşturma ve Kullanma

Bu örnekte, kullanıcıdan bir dizi boyutu alarak dinamik bir dizi oluşturacağız, kullanıcıdan bu dizinin elemanlarını girmesini isteyeceğiz ve ardından bu elemanların ortalamasını hesaplayacağız.

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

int main() {
    int size;
    cout << "Dizi boyutunu girin: ";
    cin >> size;

    // Dinamik dizi oluşturma
    int* arr = new int[size];

    // Kullanıcıdan dizi elemanlarını alma
    for (int i = 0; i < size; i++) {
        cout << "Dizi elemanı " << i + 1 << ": ";
        cin >> arr[i];
    }
```

```
    // Elemanların toplamını ve ortalamasını hesaplama
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += arr[i];
    }
    double average = static_cast<double>(sum) / size;

    cout << "Dizi elemanlarının ortalaması: " << average << endl;

    // Belleği serbest bırakma
    delete[] arr;

    return 0;
}
```

Örnek 2: İki Boyutlu Dinamik Dizi

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

int main() {
    int rows, cols;
    cout << "Matris satır sayısını girin: ";
    cin >> rows;
    cout << "Matris sütun sayısını girin: ";
    cin >> cols;

    // İki boyutlu dinamik dizi oluşturma
    int** matrix = new int*[rows];
    for (int i = 0; i < rows; i++) {
        matrix[i] = new int[cols];
    }

    // Kullanıcıdan matris elemanlarını alma
    cout << "Matris elemanlarını girin:" << endl;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            cout << "Eleman [" << i << "][" << j << "]: ";
            cin >> matrix[i][j];
        }
    }
```

```
    // Transpozeyi hesaplama ve yazdırma
    cout << "Transpoze matris:" << endl;
    for (int j = 0; j < cols; j++) {
        for (int i = 0; i < rows; i++) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }

    // Belleği serbest bırakma
    for (int i = 0; i < rows; i++) {
        delete[] matrix[i]; // Her satır için bellek serbest bırak
    }
    delete[] matrix; // Satır dizisi için bellek serbest bırak

    return 0;
}
```