C++ Veri Yapıları

Kapsamlı Kullanım Rehberi

C++ Programming Guide

20 Eylül 2025

İçindekiler

1 Giriş

Bu rehber, C++ programlama dilindeki temel veri yapılarının kullanımını detaylı şekilde açıklamaktadır. Standard Template Library (STL) bünyesindeki veri yapıları olan `stack`, `queue`, `vector`, `list`, `deque`, `set` ve `map` yapılarının kullanım örnekleri, performans özellikleri ve uygulama senaryoları ele alınmıştır.

2 Stack (Yığın)

2.1 Temel Kavram

Stack, **LIFO** (Last In, First Out) prensibiyle çalışan bir veri yapısıdır. Son eklenen eleman ilk çıkarılır. Tabak yığını gibi düşünülebilir.

2.2 Tanımlama ve Başlatma

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

stack<int> s;                      // Integer stack
stack<string> strStack;            // String stack
stack<double> dblStack;            // Double stack
stack<pair<int, int>> pairStack; // Pair stack
```

Listing 1: Stack Tanımlama

2.3 Temel Operasyonlar

```
// Eleman ekleme
s.push(10);
s.push(20);
s.push(30);

// Eleman karma ve eri im
if (!s.empty()) {
    int top = s.top();  // En stteki elemana eri im (30)
    s.pop();            // En stteki eleman kar
}

// Boyut kontrol
cout << "Boyut: " << s.size() << endl;
cout << "Bo mu? " << (s.empty() ? "Evet" : "Hay r") << endl;
```

Listing 2: Stack Temel İşlemler

2.4 Pratik Uygulama: Parantez Kontrolü

```
bool isValidParentheses(string str) {
    stack<char> s;

    for (char c : str) {
```

```
        if (c == '(' || c == '{' || c == '[') {
            s.push(c);
        }
        else if (c == ')' || c == '}' || c == ']') {
            if (s.empty()) return false;

            char top = s.top();
            s.pop();

            if ((c == ')' && top != '(') ||
                (c == '}' && top != '{') ||
                (c == ']' && top != '[')) {
                return false;
            }
        }
    }

    return s.empty();
}
```

Listing 3: Geçerli Parantez Kontrolü

2.5 Kullanım Alanları

- Geri alma (undo) işlemleri
- Fonksiyon çağrı yönetimi
- Matematiksel ifade değerlendirme
- Depth-First Search (DFS) algoritması
- Compiler tasarımında syntax analizi

3 Queue (Kuyruk)

3.1 Temel Kavram

Queue, **FIFO** (First In, First Out) prensibiyle çalışır. İlk eklenen eleman ilk çıkarılır. Gerçek hayattaki kuyruk sistemi gibidir.

3.2 Tanımlama ve Başlatma

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

queue<int> q;
queue<string> strQueue;
queue<pair<int, string>> taskQueue;
```

Listing 4: Queue Tanımlama

3.3 Temel Operasyonlar

```
// Eleman ekleme (arkadan)
q.push(10);
q.push(20);
q.push(30);

// Eleman eri imi ve   karma
if (!q.empty()) {
    int front = q.front(); //  lk  elemana eri im (10)
    int back = q.back();   // Son elemana eri im (30)
    q.pop();               //  lk  eleman    kar
}

// Boyut kontrol
cout << "Boyut: " << q.size() << endl;
cout << "Bo  mu? " << (q.empty() ? "Evet" : "Hay r") << endl;
```

Listing 5: Queue Temel İşlemler

3.4 Priority Queue (Öncelikli Kuyruk)

```
#include <queue>

// Max heap (b  y  k elemanlar  ncelikli )
priority_queue<int> maxHeap;

// Min heap (k    k elemanlar  ncelikli )
priority_queue<int, vector<int>, greater<int>> minHeap;

// Custom comparator
struct Task {
    int priority;
    string name;

    bool operator<(const Task& other) const {
        return priority < other.priority; // Y ksek  ncelik   nce
    }
};

priority_queue<Task> taskQueue;
```

Listing 6: Priority Queue Kullanımı

3.5 Pratik Uygulama: BFS Algoritması

```
void BFS(vector<vector<int>>& graph, int start) {
    vector<bool> visited(graph.size(), false);
    queue<int> q;

    q.push(start);
    visited[start] = true;

    cout << "BFS Gezinme: ";
    while (!q.empty()) {
```

```
        int current = q.front();
        q.pop();

        cout << current << " ";

        for (int neighbor : graph[current]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
    cout << endl;
}
```

Listing 7: Breadth-First Search

4 Vector (Dinamik Dizi)

4.1 Temel Kavram

Vector, dinamik boyutlu dizi yapısıdır. Çalışma zamanında boyutu değiştirilebilir ve sürekli bellek alanında tutulur.

4.2 Tanımlama ve Başlatma

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

vector<int> v;                    // Bo   vector
vector<int> v2(10);               // 10 elemanl , t  m   0
vector<int> v3(10, 5);            // 10 elemanl , t  m   5
vector<int> v4 = {1, 2, 3, 4, 5}; // Initializer list ile
vector<int> v5(v4);               // Copy constructor
```

Listing 8: Vector Tanımlama

4.3 Temel Operasyonlar

```
// Eleman ekleme
v.push_back(10);                    // Sona ekle
v.insert(v.begin() + 2, 15);        // 2. index'e ekle
v.insert(v.end(), {20, 25, 30});    // Sona birden fazla ekle

// Eleman eri imi
cout << v[0] << endl;               // Index ile eri im
cout << v.at(0) << endl;            // G venli eri im (boundary check)
cout << v.front() << endl;          //  lk  eleman
cout << v.back() << endl;           // Son eleman

// Eleman silme
v.pop_back();                       // Son eleman  sil
v.erase(v.begin() + 2);             // 2. index'teki eleman  sil
```

```
v.erase(v.begin() + 1, v.begin() + 4); // Range silme
v.clear();                             // T m elemanlar sil

// Boyut ve kapasite
cout << "Size: " << v.size() << endl;
cout << "Capacity: " << v.capacity() << endl;
cout << "Empty: " << v.empty() << endl;
```

Listing 9: Vector Temel İşlemler

4.4 İterator Kullanımı

```
vector<int> v = {1, 2, 3, 4, 5};

// Range-based for loop (C++11)
for (int x : v) {
    cout << x << " ";
}
cout << endl;

// Iterator ile
for (auto it = v.begin(); it != v.end(); ++it) {
    cout << *it << " ";
}
cout << endl;

// Reverse iterator
for (auto it = v.rbegin(); it != v.rend(); ++it) {
    cout << *it << " ";
}
cout << endl;

// STL algoritmalar ile
sort(v.begin(), v.end());                    // S ralama
reverse(v.begin(), v.end());                 // Ters evirme
auto it = find(v.begin(), v.end(), 3);       // Arama
```

Listing 10: Vector İterator Örnekleri

5 List (Çift Bağlı Liste)

5.1 Temel Kavram

List, çift bağlı liste implementasyonudur. Elemanlar bellekte ardışık değildir, ancak başa/araya ekleme/silme işlemleri verimlidir.

5.2 Tanımlama ve Temel Operasyonlar

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

list<int> l;

```

```
// Eleman ekleme
l.push_front(10);                    // Ba a ekle
l.push_back(20);                     // Sona ekle
l.insert(++l.begin(), 15);           //  kinci  pozisyona ekle

// Eleman silme
l.pop_front();                       // Ba taki eleman  sil
l.pop_back();                        // Sondaki eleman  sil
l.remove(15);                        // De er 15 olan t m elemanlar
    sil

// Liste operasyonlar
l.sort();                            // S rala
l.reverse();                         // Ters  evir
l.unique();                          // Ard    k duplike'leri kald r

// Merge i lemi
list<int> l2 = {1, 3, 5};
list<int> l3 = {2, 4, 6};
l2.merge(l3);  // l2 ve l3'   birle tir (s ral  olmal lar)
```

Listing 11: List Kullanımı

6 Deque (Double-ended Queue)

6.1 Temel Kavram

Deque, hem başından hem sonundan hızlı ekleme/silme imkanı veren yapıdır. Vector ve list'in avantajlarını birleştirir.

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

deque<int> dq;

// Eleman ekleme
dq.push_back(10);                    // Sona ekle
dq.push_front(5);                    // Ba a ekle

// Eleman silme
dq.pop_back();                       // Sondan sil
dq.pop_front();                      // Ba tan sil

// Eri im (vector gibi)
cout << dq[0] << endl;               // Index ile eri im
cout << dq.at(0) << endl;            // G venli eri im
cout << dq.front() << endl;          //  lk  eleman
cout << dq.back() << endl;           // Son eleman
```

Listing 12: Deque Kullanımı

7 Set ve Map

7.1 Set (Küme)

```cpp
#include <set>
#include <unordered_set>
using namespace std;

set<int> s;                          // S ral  , benzersiz elemanlar
multiset<int> ms;                    // S ral  , tekrarl  elemanlar
unordered_set<int> us;               // Hash tabanl , s ras z

// Operasyonlar
s.insert(10);
s.insert(5);
s.insert(15);
s.insert(10);  // Eklenmez (zaten var)

// Arama
if (s.find(10) != s.end()) {
    cout << "10 bulundu!" << endl;
}

// Count (set'te 0 veya 1, multiset'te fazla olabilir)
cout << "10'un say s  : " << s.count(10) << endl;

// Range-based iteration (s ral  )
for (int x : s) {
    cout << x << " ";  // 5 10 15
}
```

Listing 13: Set Kullanımı

7.2 Map (Harita)

```cpp
#include <map>
#include <unordered_map>
using namespace std;

map<string, int> m;                  // S ral  key-value
unordered_map<string, int> um;       // Hash tabanl

// Eleman ekleme
m["Ali"] = 25;
m["Ay e"] = 30;
m.insert({"Mehmet", 35});
m.insert(make_pair("Fatma", 28));

// Eri im
cout << m["Ali"] << endl;            // 25

// G venli eri im
if (m.find("Ali") != m.end()) {
    cout << "Ali'nin ya   : " << m["Ali"] << endl;
}

// Iterasyon
for (auto& pair : m) {
    cout << pair.first << ": " << pair.second << endl;
}
```

```cpp
// Key kontrol
if (m.count("Ali") > 0) {
    cout << "Ali mevcut" << endl;
}
```

Listing 14: Map Kullanımı

8 Performans Karşılaştırması

Tablo 1: Veri Yapıları Performans Karşılaştırması

İşlem	**Vector**	**List**	**Deque**	**Set**	**Map**	**unordered_map**
Rastgele Erişim	$O(1)$	$O(n)$	$O(1)$	-	-	-
Başa Ekleme	$O(n)$	$O(1)$	$O(1)$	-	-	-
Sona Ekleme	$O(1)^*$	$O(1)$	$O(1)$	-	-	-
Araya Ekleme	$O(n)$	$O(1)$	$O(n)$	-	-	-
Arama	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)^{**}$
Insert/Find	-	-	-	$O(\log n)$	$O(\log n)$	$O(1)^{**}$
Silme	$O(n)$	$O(1)$	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)^{**}$

** Amortized constant time*
*** Average case, worst case $O(n)$*

9 Veri Yapısı Seçim Rehberi

9.1 Vector Kullan Eğer:

- Rastgele erişim gerekiyorsa
- Çok fazla iterasyon yapacaksan
- Bellek kullanımı kritikse
- Cache performansı önemliyse
- Matematiksel hesaplamalar yapacaksan

9.2 List Kullan Eğer:

- Sık sık başa/araya ekleme/silme yapacaksan
- Iterator'lar geçersiz olmamalı
- Büyük objelerle çalışıyorsan
- Splice operasyonları gerekiyorsa

9.3 Deque Kullan Eğer:

- Hem baştan hem sondan işlem yapacaksan
- Queue ve stack işlemlerini birlikte yapacaksan
- Rastgele erişim + başa ekleme gerekiyorsa

9.4 Set/Map Kullan Eğer:

- Benzersiz elemanlar gerekiyorsa
- Hızlı arama yapacaksan
- Sıralı veri gerekiyorsa
- Key-value eşleştirmesi yapacaksan

10 İleri Seviye İpuçları

10.1 Bellek Optimizasyonu

```
// Vector kapasitesini  nceden  ayarla
vector<int> v;
v.reserve(1000);  // 1000 elemanl k yer ay r

// Belle i serbest b rak
vector<int>().swap(v);  // v'nin belle ini temizle

// Shrink to fit (C++11)
v.shrink_to_fit();  // Fazla kapasiteyi serbest b rak

// Move semantics kullan (C++11)
vector<int> v2 = std::move(v1);  // v1'den v2'ye ta
```

Listing 15: Bellek Yönetimi İpuçları

10.2 Exception Safety

```
#include <cassert>
#include <stdexcept>

// Assert kullan
assert(!s.empty() && "Stack bo !");

// Exception handling
try {
    cout << v.at(100);  // S n r d       eri im
} catch (const std::out_of_range& e) {
    cout << "Hata: " << e.what() << endl;
}

// RAII prensibi
class SafeVector {
```

```
    vector<int>* vec;
public:
    SafeVector() : vec(new vector<int>) {}
    ~SafeVector() { delete vec; }
    vector<int>& get() { return *vec; }
};
```

Listing 16: Güvenli Kod Yazma

10.3 STL Algoritmaları ile Kombinasyon

```
#include <algorithm>
#include <numeric>

vector<int> v = {5, 2, 8, 1, 9};

// S ralama
sort(v.begin(), v.end());                    // Artan s ra
sort(v.begin(), v.end(), greater<int>());    // Azalan s ra

// Arama
auto it = find(v.begin(), v.end(), 5);
bool found = (it != v.end());

// Ko ullu arama
auto it2 = find_if(v.begin(), v.end(),
                   [](int x) { return x > 5; });

// Sayma
int count = count_if(v.begin(), v.end(),
                     [](int x) { return x % 2 == 0; });

// Transform
transform(v.begin(), v.end(), v.begin(),
          [](int x) { return x * 2; });

// Toplam
int sum = accumulate(v.begin(), v.end(), 0);
```

Listing 17: STL Algoritmaları

11 Sonuç

Bu rehberde C++ STL'nin en önemli veri yapıları detaylı şekilde incelenmiştir. Her veri yapısının kendine özgü avantajları ve kullanım alanları vardır. Doğru veri yapısını seçmek, programın performansını ve okunabilirliğini önemli ölçüde etkiler.

Temel ilkeler:

- Probleminizin gereksinimlerini analiz edin
- Performans özelliklerini göz önünde bulundurun
- Kod okunabilirliğini ihmal etmeyin
- STL algoritmaları ile entegrasyonu düşünün

- Exception safety'yi unutmayın

Sürekli pratik yaparak bu veri yapılarının kullanımında ustalaşabilir ve daha verimli C++ programları yazabilirsiniz.