ŞABLONLAR

Şablon Nedir?

Şablonlar (template), farklı veri tipleriyle çalışan fonksiyonlar, sınıflar, algoritmaları tekrar tekrar yazmamak için bir programlama türü olan jenerik programlamanın (generic programming) temelini oluşturur. Yani jenerik programlamada veri tiplerine bağlı olmadan fonksiyon, sınıf ve algoritma oluşturmamızı sağlar.

Fonksiyon Şablonları

Fonksiyon şablonları (function template), aynı mantığı yeniden yazmadan bir fonksiyonun farklı veri türleri üzerinde çalışmasını sağlayan bir fonksiyon planı tanımlar. Bu plan aşağıdaki şekilde tanımlanır;

```
template <typename değişken-kimliği> foksiyon-bildirimi;
```

Burada `typename`, işlenecek bir tipe verilen bir kimliktir. Yani bir şablonun bildiriminde bir parametre tanıtır. Fonksiyon şablonuna aşağıdaki örnek verilebilir;

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

template <typename Degisken> void Degistir(Degisken& a, Degisken& b) {
   Degisken temp= a;
   a=b;
   b=temp;
}

int main () {
   int i = 10;
   int j = 20;
   Degistir(i, j); // Derleyici parametreleri int veri tipinde kabul etti
   cout << "Degiştir(i, j) sonrası i: " << i <<" , j: " << j << endl;

   double f1 = 15.0;
   double f2 = 25.5;
   Degistir(f1, f2); // Derleyici parametreleri double veri tipinde kabul etti
   cout << "Degiştir(f1, f2) sonrası f1: " << f1 <<" , f2: " << f2 << endl;

   string s1 = "Hello";
   string s2 = "World";
   Degistir(s1, s2); // Derleyici parametreleri string veri tipinde kabul etti
   cout << "Degiştir(s1, s2) sonrası s1: " << s1 <<" , s2:"  << s2 << endl;
}
/* Program Çıktısı:
Degiştir(i, j) sonrası i: 20 , j: 10
Degiştir(f1, f2) sonrası f1: 25.5 , f2: 15
Degiştir(s1, s2) sonrası s1: World , s2: Hello

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

Görüldüğü üzere her veri tipi için değiştir fonksiyonu yazmak yerine bir adet şablon fonksiyon ile bütün ihtiyacımız karşılanmıştır.

Çok Değişkenli Fonksiyon Şablonları

Değişken sayıda argüman alabilen yada çok değişkenli fonksiyon şablonları (variadic function template), herhangi bir değişken (sıfır veya daha fazla) sayıda argüman alabilen sınıf veya fonksiyon şablonlarıdır. C++ dilinde şablonlar, yalnızca bildirim sırasında belirtilmesi gereken sabit sayıda parametreye sahip olabilir. Ancak *Douglas Gregor* ve *Jaakko Järvi* tarafından ortaya koyulan değişkenli şablonlar (variadic template) bu sorunun üstesinden gelmeye yardımcı olur.

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

template <typename T> void yaz(const T& t) {
    cout << t << endl;
}

template <typename Ilk, typename... KalanParametreler>
    void yaz(const Ilk& ilk, const KalanParametreler&... kalan) {
       cout << ilk << ", ";
       yaz(kalan...); // özyinelemeli çağrı (recursive call)
}

int main() {
    yaz(1, 2, 3.14, "Merhaba ", "C++", "Öğreniyorum");
    yaz(10, 20, 30);
    yaz(1, 2, 3, 4, 5);
}
/* Program Çıktısı:
1, 2, 3.14, Merhaba , C++, Öğreniyorum
10, 20, 30
1, 2, 3, 4, 5

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

Sınıf Şablonları

Fonksiyonlara benzer şekilde, sınıf şablonları aynı zamanda herhangi bir veri türüyle çalışabilen sınıflar oluşturmak için bir plan tanımlar. Aşağıdaki şekilde tanımlanır;

```
template <veritipi değişken-kimliği[,veritipi değişken-kimliği-2]>
class şablon-sınıf-kimliği {
  private:
    değişken-kimliği örnek-değişkeni;
    [değişken-kimliği-2 örnek-değişkeni2;]
    ... .. ...
  public:
    değişken-kimliği fonksiyonKimligi(değişken-kimliği arg);
    ... .. ...
};
```

Burada, `sınıf-kimliği`, bir sınıf şablona konu olan yer tutucu sınıf kimliğidir. Virgülle ayrılmış bir liste kullanarak birden fazla genel veri tipi tanımlanabilir.

Bir şablon sınıftan bir nesne imal etme aşağıdaki gibi yapılır;

```
şablon-sınıf-kimliği <veritipi> object-sınıf-kimliği;
```

Aşağıda bir şablon sınıf örneği verilmiştir;

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

```
template <class veriTipi>
class Sayi {
   private:
    veriTipi num; // num kimlikli bir alan (field) tanımlandı

   public:
    Sayi(veriTipi n) : num(n) {}   // constructor

    veriTipi getNum() {
        return num;
    }
};

int main() {
    Sayi<int> intAlanliSinifNesnesi(5);
    Sayi<double> doubleAlanliSinifNesnesi(15.75);

    cout << "int Sayı<int>.getnum() = "
         << intAlanliSinifNesnesi.getNum() << endl;
    cout << "double Sayı<double>.getnum() = "
         << doubleAlanliSinifNesnesi.getNum() << endl;
}
```

Görüldüğü üzere hem `int` hem de `double` için ayrı ayrı sınıf yazmadan tek bir şablon sınıf ile tüm ihtiyacımız karşılanmıştır.

Şablon kullanmanın çeşitli üstünlükleri vardır;

- Yeniden Kullanılabilir Kod: Şablonlar, tüm veri tipleriyle çalışan genel kod yazmanıza olanak tanır ve böylece her gerekli tip için aynı kodu yazma ihtiyacını ortadan kaldırır. Bu da kod tekrarını ve büyümesini azaltarak geliştirme süresinden tasarruf sağlar.
- Azaltılmış Bakım: Bir şablon güncellendiğinde tüm örneklemelerdeki değişiklikleri otomatik olarak yapılmış olur. Bu, hata düzeltme, düzeltmeleri yapma ve tüm örneklemelerin faydalarını görme açısından üstünlük sağlar.
- Gelişmiş Performans: Şablon örneklemeleri derleme zamanında gerçekleşir ve çalışma zamanı hatalarını azaltır. Derleyici, kodu belirli veri tipleri için optimize eder.
- İyi Organize Edilmiş Kaynak Kod: Şablonlar algoritma mantığı veri türünden ayırdığından, geliştirme senaryosu açısından daha da iyi olan modüler kod oluşturmaya yardımcı olur. Bir kodun farklı uygulamalarını aramayı azaltmaya yardımcı olur.

Akıllı Göstericiler

Akıllı göstericiler (smart pointer) sınıf şablonlarıdır. `std::unique_ptr`, dinamik olarak depolanan bir nesnenin ömrünü yöneten bir sınıf şablonudur. Dinamik nesne herhangi bir zamanda yalnızca bir `std::unique_ptr` örneğine aittir.

```
std::unique_ptr<int> ptr = std::make_unique<int>(20); /* Benzersiz bir göstericiye ait 20
değerine sahip dinamik bir int oluşturur */
```

Sadece `ptr` değişkeni dinamik olarak tahsis edilmiş bir tamsayıya (`int`) bir gösterici tutar. Bir nesneye sahip olan benzersiz bir gösterici kapsam dışına çıktığında, sahip olunan nesne silinir, yani nesne sınıf türündeyse onun yıkıcısı çağrılır ve o nesnenin belleği serbest bırakılır.

```
std::unique_ptr<int> ptr = std::make_unique<int>(59); /* Benzersiz bir göstericiye ait 59
değerine sahip dinamik bir int oluşturur */
std::unique_ptr<int[]> ptr2 = std::make_unique<int[]>(15); /* 15 adet int tutan diziye ait
benzersiz bir gösterici */
```

Akıllı göstericinin içeriğinin sahipliğini `std::move` kullanarak başka bir göstericiye aktarabilirsiniz, bu durum orijinal akıllı işaretçinin `nullptr`'yi işaret etmesine neden olur.

```
std::unique_ptr<int> ptr = std::make_unique<int>();
*ptr = 1; // gösterilen değer 1 yapıldı

std::unique_ptr<int> ptr2 = std::move(ptr);
int a = *ptr2; // 'a' is 1
int b = *ptr; // Hata! 'ptr' = 'nullptr'
```

`std::shared_ptr` sınıf şablonu ise bir nesnenin sahipliğini diğer paylaşılan göstericilerle paylaşır.

```
std::shared_ptr<Foo> firstShared = std::make_shared<Foo>(/*args*/);
```

Aynı nesneyi paylaşan birden fazla akıllı gösterici oluşturmak için, ilk paylaşılan gösterici takma adla adlandıran başka bir `shared_ptr` oluşturmamız gerekir. Bunu yapmanın 2 yolu vardır:

```
std::shared_ptr<Foo> secondShared(firstShared); // Birinci yol
std::shared_ptr<Foo> secondShared;
secondShared = firstShared; // 2. Yol. Atama ile
```

Akıllı gösterici tıpkı bilinen göstericiler gibi çalışır. Buda `*` işlecini (dereference operator) kullanabileceğiniz anlamına gelir. `->` işleci de aynı şekilde çalışır:

```
secondShared->test();
```

Ne yazık ki, `make_shared<>` kullanarak Dizileri tahsis etmenin doğrudan bir yolu yoktur. C++17 ile, `shared_ptr` dizi tipleri için özel destek kazandı. Artık `array-deleter`'ı açıkça belirtmek gerekmiyor ve paylaşılan işaretçi [] dizi dizin operatörü kullanılarak başvurudan kaldırılabilir:

```
std::shared_ptr<int[]> sh(new int[10]);
sh[0] = 42;
```

Paylaşılan göstericiler, sahip olduğu nesnenin bir alt nesnesine işaret edebilir;

```
struct Foo { int x; };
std::shared_ptr<Foo> p1 = std::make_shared<Foo>();
std::shared_ptr<int> p2(p1, &p1->x);
```

Standart Şablon Kütüphanesi

Standart şablon kütüphanesi (Standart Template Library), C++ dilinin en güçlü kütüphanesidir. Jenerik programlama (generic programming) ile birçok algoritma programcıya bu kütüphane ile sunulur.

Buraya kadar anlatılan şablon (template) kavramını anlamışsanız C++ dili Standart Şablon Kütüphanesi, vektörler, listeler, kuyruklar ve yığınlar gibi birçok popüler ve yaygın olarak kullanılan algoritmayı ve veri yapısını uygulayan şablonlarla genel amaçlı sınıflar ve fonksiyonlar sağlamak için güçlü bir şablon sınıfları kümesidir[19]. Bu küme dört başlıkta incelebilir;

- Konteynerler: Belirli bir türdeki nesne kümeleri yani koleksiyonlarını yönetmek için kullanılır. Deque, list, vector, map gibi çeşitli farklı konteyner türleri vardır. Konteynerler, tuttukları verilerden bağımsız dinamik veri yapılarıdır (dynamic data structure).
- Algoritmalar: Konteynerler üzerinde etki eder. Konteynerlerin içeriklerinin başlatılmasını, sıralanmasını, aranmasını ve dönüştürülmesini gerçekleştireceğiniz araçları sağlarlar.
- Yineleyiciler: Nesne koleksiyonlarının elemanları arasında gezinmek için kullanılır. Bu koleksiyonlar, kapsayıcılar veya kapsayıcıların alt kümeleri olabilir.
- Fonksiyon Nesneleri: Bir fonksiyon sanki bir fonksiyonmuş gibi çağrılabilen bir nesnedir. Normal fonksiyonlar gibi çağrılabilir. Bu yetenek, fonksiyon çağrı işleci (function call operator) olan `()`'in aşırı yüklenmesiyle elde edilir. Bu kelimelerin kısaltılmış hali olan functor olarak adlandırılırlar.

Aşağıda bir diziye benzeyen ancak büyümesi durumunda kendi depolama gereksinimlerini otomatik olarak karşılayan vektör konteynerini kullanan program örneği verilmiştir;

[19] https://www.tutorialspoint.com/cplusplus/cpp_stl_tutorial.htm sayfasından çoğu derlenmiştir.

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

int main() {
   vector<double> vec; // int değerler tutacak vector konteyneri
   int i;
   cout << "Vector Uzunluğu = " << vec.size() << endl;
   for(i = 0; i < 5; i++) { // 5 değer vektöre koyuluyor.
      vec.push_back(i*1.5);
   }
   cout << "Vektörün Son Uzunluğu = " << vec.size() << endl;

   for(i = 0; i < 5; i++) {// vektör elemanlarına erişiliyor
      cout << "vec[" << i << "] = " << vec[i] << endl;
   }

   // yineleyiciler üzerinden vektör elemanlarına erişim:
   vector<double>::iterator v = vec.begin();
   while( v != vec.end()) {
      cout  << *v << endl;
      v++;
   }
}
/*Program Çıktısı:
Vector Uzunluğu = 0
Vektörün Son Uzunluğu = 5
vec[0] = 0
vec[1] = 1.5
vec[2] = 3
vec[3] = 4.5
vec[4] = 6
0
1.5
3
4.5
6

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

Programı incelediğimizde `double` olan veri tipini değiştirme ihtiyacımız varsa sadece iki satırda değişiklik yapmamız yeterlidir. Program içinde yer alan;

- `push_back()` üye fonksiyonu vektörün sonuna değer ekler ve gerektiği gibi boyutunu genişletir.
- `size()` fonksiyonu vektörün boyutunu görüntüler.
- `begin()` fonksiyonu vektörün başlangıcını gösteren bir yineleyici döndürür.
- `end()` fonksiyonu vektörün sonunu gösteren bir yineleyici döndürür.

Konteyner Şablonları

Konteynerler, vektörler, listeler, kümeler ve haritalar gibi veri veya nesne küme/koleksiyonlarını depolamak ve yönetmek için kullanılan veri yapılarıdır (data structures). Dinamik veri yapıları yapısal programlama sürecinde çokça geliştirilen ve artık standart hale gelen koleksiyonlardır. C++ dilinde koleksiyonlar, artık hazır bir kütüphanedir.

Ardışık konteynerler (sequential container), elemanları belirli bir doğrusal düzende depolarlar. Elemanları ve sırasını ve düzenlemesini yönetmek için işlevlere doğrudan erişim sağlarlar;

- Dizi (array): Bunlar sabit boyutlu eleman koleksiyonlarıdır.

- Vektör (vector): Gerektiğinde boyutunu değiştirebilen dinamik bir dizidir.
- İki Uçlu Kuyruk (deque): Her iki uçtan hızlı ekleme ve silmeye izin veren çift uçlu kuyruk.
- Liste (list): Çift yönlü yinelemeye izin veren çift bağlantılı liste.
- İleri Liste (forward list): Verimli ekleme ve silmeye izin veren ancak yalnızca bir yönde gezinmeye izin veren tek bağlantılı listedir.
- Dizgi (string): Diğer ardışık konteynerlere benzeyen dinamik bir karakter dizisidir.

Konteyner kütüphanesi, programcıların kuyruklar, listeler ve yığınlar gibi yaygın veri yapılarını kolayca uygulamasına olanak tanıyan sınıf şablonları ve algoritmaları içerir. Bu konteynerlerden hangisini ne zaman kullanacağımız çözülecek problemle ilgilidir [20].

İlişkili konteynerler (associated container), elemanları anahtarlara (key) göre hızlı bir şekilde geri almaya izin veren sıralı bir düzende depolar. Bu konteynerler, her bir elemanın benzersiz bir anahtarla ilişkilendirildiği bir anahtar-değer çifti yapısında çalışır. Bu anahtar, karşılık gelen değere hızlı erişim için kullanılır;

- Küme (set): Belirli bir düzende sıralanmış benzersiz elemanların bir koleksiyonudur.
- Harita (map): Anahtarların benzersiz olduğu anahtar-değer çiftlerinin bir koleksiyonudur.
- Çoklu Küme (multiset): Bir kümeye benzer, ancak yinelenen elemanların izin verir.
- Çoklu Harita (multimap): Bir haritaya benzer, ancak yinelenen anahtarlara izin verir.

Sıralanmamış ilişkisel konteynerler (unordered associated container), elemanları sırasız bir şekilde depolar ve anahtarlara (key) dayalı olarak verimli erişim, ekleme ve silmeye olanak tanır. Elemanların sıralı bir düzenini korumak yerine verileri düzenlemek için karma kullanırlar;

- Sıralanmamış Küme (unordered set): Belirli bir sırası olmayan, benzersiz elemanlardan oluşan bir koleksiyondur.
- Sıralanmamış Harita (unordered map): Belirli bir sırası olmayan, anahtar-değer çiftlerinden oluşan bir koleksiyondur.
- Sıralanmamış Çoklu Küme (unordered multiset): Belirli bir sırası olmayan, yinelenen elemanlara izin verir.
- Sıralanmamış Çoklu Harita (unordered multimap): Belirli bir sırası olmayan, yinelenen anahtarlara izin verir.

Konteyner adaptörleri (container adapter), mevcut konteynerler için farklı bir ara yüz sağlar;

- Yığın (stack): Son Giren İlk Çıkar (last in first out-LIFO) ilkesini izleyen bir veri yapısıdır.
- Kuyruk (queue): İlk Giren İlk Çıkar (first in first out-FIFO) ilkesini izleyen bir veri yapısıdır.
- Öncelikli Kuyruk (priority queue): Elemanların önceliğe göre kaldırıldığı özel bir kuyruk türüdür.

Algoritma Şablonları

Standart Şablon Kütüphanesi'ndeki algoritmalar (algotrithm), konteynerler üzerinde işlem yapmak için özel olarak tasarlanmış büyük bir fonksiyon kümesidir. Bu fonksiyonlar, konteynerlerin iç yapılarını bilmeye gerek kalmadan konteynerler arasında geçiş yapmak için kullanılan yineleyiciler (iterator) kullanılarak uygulanır.

Sıra değiştirmeyen algoritmalar;

- for_each: Bir aralıktaki her bir elemanı bulmak için bir fonksiyonda kullanılır.
- count: Bir aralıktaki değerin oluşum sayısını sayar.
- find(): Bir aralıktaki değerin ilk oluşumunu bulur.
- find_if: Bir yordamı karşılayan ilk elemanı bulur.
- find_if_not: Bir yordamı karşılamayan ilk elemanı bulur.
- equal: İki aralığın eşit olup olmadığını kontrol eder.

[20] https://docs.google.com/drawings/d/1c-qvy499kxaYXM70DM34rOnEBCzgQLopyNDEXdaK0eU/edit

– search: Bir dizi içinde bir alt diziyi arar.

Sıra değiştiren algoritmalar;

– copy(): Elemanları bir aralıktan diğerine kopyalar.
– copy_if: Bir yordamı karşılayan elemanları başka bir aralığa kopyalar.
– copy_n: Belirli sayıda elemanı bir aralıktan diğerine kopyalar.
– move: Elemanları bir aralıktan diğerine taşır.
– transform(): Bir aralığa bir fonksiyon uygular ve sonucu depolar.
– remove: Belirli bir değere sahip elemanları bir aralıktan kaldırır.
– remove_if: Bir yordamı karşılayan elemanları kaldırır.
– unique: Ardışık yinelenen elemanları kaldırır.
– reverse(): Bir aralıktaki elemanların sırasını tersine çevirir.
– swap(): Eelemanları değiştirir.

Sıralama algoritmaları;

– sort: Bir aralıktaki elemanları sıralar.
– stable_sort: Eşdeğer elemanların göreli sırasını koruyarak elemanları sıralar.
– partial_sort: Bir aralığın bir kısmını sıralar.
– nth_element: Aralığı, n'inci elemanın son konumunda olacak şekilde bölümlere ayırır.

Arama algoritmaları;

– binary_search: Sıralanmış bir aralıkta bir elemanın bulunup bulunmadığını kontrol eder.
– lower_bound: Sırayı korumak için bir elemanın eklenebileceği ilk konumu arar.
– upper_bound: Belirtilen değerden büyük olan ilk elemanın konumunu arar.
– binary_search: Sıralanmış bir aralıkta bir elemanın bulunup bulunmadığını kontrol eder.
– lower_bound: Sırayı korumak için bir elemanın eklenebileceği ilk konumu bulur.
– upper_bound: Belirtilen değerden büyük olan ilk elemanın konumunu arar.
– equal_range: Eşit elemanların aralığını döndürür.

Öbek (heap) algoritmaları;

– make_heap: Bir aralıktan bir öbek oluşturur.
– push_heap: Bir öbeğe bir eleman ekler.
– pop_heap: Bir öbekten en büyük eleman kaldırır.
– sort_heap: Bir öbekteki elemanları sıralar.

Küme algoritmaları;

– set_union: İki kümenin birleşimini hesaplar.
– set_intersection: İki kümenin kesişimini hesaplar.
– set_difference: İki küme arasındaki farkı hesaplar.
– set_symmetric_difference: İki küme arasındaki simetrik farkı hesaplar.

Sayısal algoritmalar;

– accumulate: bir aralığın toplamını (veya diğer işlemleri) hesaplar.
– inner_product: İki aralığın iç çarpımını hesaplar.
– adjacent_difference: Bitişik elemanlar arasındaki farkları hesaplar.
– partial_sum: bir aralığın kısmi toplamlarını hesaplar.

Diğer algoritmalar;

– generate: Bir fonksiyon tarafından üretilen değerlerle bir aralığı doldurur.
– shuffle: Bir aralıktaki elemanları rastgele karıştırır.
– partition: Eelemanları bir öngörüye göre iki gruba ayırır.

Yineleme Şablonları

Standart Şablon Kütüphanesi'ndeki yineleyiciler (iterator), bir konteyner içindeki elemanlara gösterici görevi gören ve verilere erişmek ve bunları düzenlemek için tekdüze bir ara yüz sağlayan nesnelerdir. Algoritmalar ve konteynerler arasında bir köprü görevi görürler;

- Giriş Yineleyicileri (input iterator): Elemanlara salt okunur erişime izin verirler.
- Çıkış Yineleyicileri (output iterator): Elemanlara salt yazılır erişime izin verirler.
- İleri Yineleyiciler (forward iterator): Artırılabilen okuma ve yazmaya izin verirler.
- Çift Yönlü Yineleyiciler (bidirectional iterators): Artırılabilir ve azaltılabilirler.
- Rastgele Erişim Yineleyicileri (Random Access Iterator): Aritmetik işlemleri desteklerler ve elemanlara doğrudan erişebilirler.

Fonksiyon Nesneleri

Fonksiyon nesnesi (function object), fonksiyon çağrı işleci (functor) ile bir fonksiyonmuş gibi çağrılabilen bir nesnedir. Fonksiyon nesneleri, bir fonksiyon veya fonksiyon göstericisiymiş gibi ele alınabilen nesnelerdir. Aşağıda bir dizinin elemanlarını 1 artıran bir dönüşüm programı verilmiştir;

```
#include <bits/stdc++.h> //transform()
using namespace std;

int artirim(int x) { return (x+1); }

int main(){
	int dizi[] = {1, 2, 3, 4, 5};
	int adet = sizeof(dizi)/sizeof(dizi[0]);
	/*
	    Aşağıdaki dönüşüm fonksiyonu ile
	    dizi elemanları 1 artırılıyor
	*/
	transform(dizi, dizi+adet, dizi, artirim);
	for (int i=0; i<adet; i++)
		cout << dizi[i] <<" ";
	return 0;
}
/* Program Çıktısı:
2 3 4 5 6

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

Aşağıda bir fonksiyon nesnesi üzerinden aynı işlemi yapan bir program örneği verilmiştir;

```
#include <bits/stdc++.h> // transform
using namespace std;

class artirim// Functor
{
private:
	int sayi;
public:
	artirim(int n) : sayi(n) { }

	/* Aşağıda verilen işleç önyüklemesi ile
	arr_sayi kadar artırım işlemi yapılacaktır. */
	int operator () (int pMiktar) const {
		return sayi + pMiktar;
	}
};
```

```
int main() {
	int dizi[] = {1, 2, 3, 4, 5};
	int adet = sizeof(dizi)/sizeof(dizi[0]);
	int miktar = 5;
	transform(dizi, dizi+adet, dizi, artirim(miktar));
	for (int i=0; i<adet; i++)
		cout << dizi[i] << " ";
}
/*Program Çıktısı:
6 7 8 9 10

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

Aritmetik functor (aritmetic functor), aritmetik operatörler temel matematiksel işlemleri gerçekleştirmek için kullanılır. Toplama (+), Çıkarma (-), Çarpma (*), Bölme (/), Kalan (%), Negatif (−) işleç (operator) işlevlerini yerine getirirler.

Karşılaştırma fonksiyon nesneleri, özellikle konteynerlerde sıralama veya arama yapmak için değerleri karşılaştırmak amacıyla kullanılır. Küçüktür (<), Büyüktür (>), Küçüktür veya Eşittir (≤), Büyük veya Eşit (≥), Eşit (=), Eşit Değil (! =) işleç işlevlerini yerine getirirler.

Mantıksal fonksiyon nesneleri, Boole mantığını içeren senaryolar için mantıksal işlemleri gerçekleştirir. Mantıksal VE (&&), Mantıksal VEYA (||) ve Mantıksal DEĞİL (!) işleç işlevlerini yerine getirirler.

Bit düzeyi fonksiyon nesneleri, bit düzeyi işlemleri gerçekleştirir. Bit düzeyi VE (&), VEYA (|) ve ÖZEL VEYA (^) işleç işlevlerini yerine getirirler.

std::array

C dilinden bildiğimiz dizilere benzer ancak her türlü veri tipiyle oluşturulabilir. Bu sınıf şablonunun iki parametresi vardır; `class T` Dizi elemanlarının veri tipini belirtir, `std::size_t N` ise Dizide kaç eleman olduğunu belirtir.

Dizi tanımlama ve ilk değer verme (initialize);

T'nin sayılarla ifade edilen (scalar) veri tiplerini barındırması halinde aşağıdaki şekilde ilk değer verilebilir;

1. Dizilere ilk değer verme yöntemini kullanarak

```
std::array<int, 3> a { 0, 1, 2 };
// buna eşdeğer ifade:
std::array<int, 3> a = { 0, 1, 2 };
```

2. Kopya yapıcı (copy constructor) kullanarak

```
std::array<int, 3> a { 0, 1, 2 };
std::array<int, 3> a2(a);
// buna eşdeğer ifade
std::array<int, 3> a2 = a;
```

3. Tasıma yapıcısı (move constructor) kullanarak;

```
std::array<int, 3> a = std::array<int, 3>{ 0, 1, 2 };
```

T'nin sayılarla ifade edilmeyen (non scalar) veri tiplerini barındırması halinde aşağıdaki şekilde ilk değer verilebilir;

```
struct A { int values[3]; };
```

4. Tırnaklı parantez ile ilk değer verilebilir

```
std::array<A, 2> a{ 0, 1, 2, 3, 4, 5 };
// buna eşdeğer ifade:
```

```
std::array<A, 2> a = { 0, 1, 2, 3, 4, 5 };
```

5. Tırnaklı parantezleri alt elemanlarda kullanarak;

```
std::array<A, 2> a{ A{ 0, 1, 2 }, A{ 3, 4, 5 } };
// buna eşdeğer ifade:
std::array<A, 2> a = { A{ 0, 1, 2 }, A{ 3, 4, 5 } };
```

6. Tamamıyla tırnaklı parantez kullanarak;

```
std::array<A, 2> a{{ { 0, 1, 2 }, { 3, 4, 5 } }};
// buna eşdeğer ifade:
std::array<A, 2> a = {{ { 0, 1, 2 }, { 3, 4, 5 } }};
```

7. Kopya yapıcı (copy constructor) kullanarak;

```
std::array<A, 2> a{ 1, 2, 3 };
std::array<A, 2> a2(a);
// buna eşdeğer ifade:
std::array<A, 2> a2 = a;
```

8. Taşıma yapıcısı (move constructor) kullanarak;

```
std::array<A, 2> a = std::array<A, 2>{ 0, 1, 2, 3, 4, 5 };
```

Dizi elemanlarına erişim;

1. `at(pos)` fonksiyonu ile: `pos` konumundaki elemana sınır denetimiyle bir referans döndürür. `pos`, konteynerin aralığında değilse, `std::out_of_range` türünde bir istisna atılır.

```
#include <array>
int main()
{
    std::array<int, 3> arr;
    arr.at(0) = 3; //0. Elemana 2 atandı
    arr.at(1) = 4; //1. Elemana 2 atandı
    arr.at(2) = 6; //2. Elemana 2 atandı
    int a = arr.at(0); // a değişkenine 0. Dizi elemanı atandı
    int b = arr.at(1); // a değişkenine 1. Dizi elemanı atandı
    int c = arr.at(2); // a değişkenine 2. Dizi elemanı atandı
}
```

2. `operator[pos]` işleci ile: `pos` konumundaki elemana sınır denetimi olmadan bir başvuru döndürür. `pos`, konteynerin aralığında değilse, bir çalışma zamanı hatası (segmentation violation) oluşabilir. Bu yöntem, klasik dizilere eşdeğer eleman erişimi sağlar ve bu nedenle `at(pos)`'tan daha verimlidir.

```
#include <array>
int main()
{
    std::array<int, 3> arr;
    arr[0] = 3;
    arr[1] = 4;
    arr[2] = 6;
    int a = arr[0];
    int b = arr[1];
    int c = arr[2];
}
```

3. `std::get<pos>` ile: `std::array` sınıfının üyesi olmayan bu fonksiyon, sınır denetimi olmaksızın derleme zamanı sabit konumu `pos`'taki elemana bir referans döndürür. `pos`, konteynerin aralığında değilse, bir çalışma zamanı (segmentation violation) hatası oluşabilir.

```
#include <array>
int main()
{
```

```
    std::array<int, 3> arr;
    std::get<0>(arr) = 3;
    std::get<1>(arr) = 4;
    std::get<2>(arr) = 6;
    int a = std::get<0>(arr);
    int b = std::get<1>(arr);
    int c = std::get<2>(arr);
}
```

4. `front()` fonksiyonu ile: Konteynerdeki ilk elemana bir referans döndürür. Boş bir konteynerde `front()`'u çağırmak tanımsızdır.

```
#include <array>
int main()
{
    std::array<int, 3> arr{ 2, 4, 6 };
    int a = arr.front();
}
```

5. `back()` fonksiyonu ile: Konteynerdeki son elemana bir referans döndürür. Boş bir konteynerde `back()`'u çağırmak tanımsızdır.

```
#include <array>
int main()
{
    std::array<int, 3> arr{ 2, 4, 6 };
    int a = arr.back();
}
```

6. `data()` fonksiyonu ile: Eleman depolaması olarak hizmet eden altta yatan diziye gösterici döndürür.

```
#include <iostream>
#include <array>
void yaz(const int* p, std::size_t size){
    for (std::size_t i = 0; i < size; ++i)
        std::cout << p[i] << ' ';
}
int main (){
    std::array<int, 3> a { 0, 1, 2 };
    yaz(a.data(), 3);
}
```

Dizi elemanları üzerinde yineleme (iteration);

```
#include <iostream>
#include <array>
int main() {
    std::array<int, 3> arr = { 1, 2, 3 };
    for (auto i : arr)
        std::cout << i << std::endl;
}
```

Dizi boyutunu kontrol etme;

```
#include <iostream>
#include <array>
int main() {
    std::array<int, 3> arr = { 1, 2, 3 };
    std::cout << arr.size() << std::endl;
}
```

Dizi elemanlarını tek seferde değiştirme;

```
#include <iostream>
```

```
#include <array>
int main() {
    std::array<int, 3> arr = { 1, 2, 3 };
    arr.fill(100);
}
```

std::vector

Bir vektör, otomatik olarak işlenen depolamaya sahip dinamik bir dizidir. Bir vektördeki elemanlara, bir dizideki elemanlar kadar verimli bir şekilde erişilebilir; avantajı, vektörlerin boyut olarak dinamik olarak değişebilmesidir. Depolama açısından vektör verileri (genellikle) dinamik olarak tahsis edilmiş belleğe yerleştirilir, bu nedenle bazı küçük ek yükler gerektirir; tersine C dizileri ve `std::array`, beyan edilen boyuta göre otomatik depolama kullanır ve bu nedenle herhangi bir ek yüke sahip değildir.

Vektör elemanlarına erişim;

Vektör elemanlarına `[]` işleci ve `at()` fonksiyonu ile erişebiliriz.

```
#include <iostream>
#include <vector>
int main() {
    std::vector<int> v{ 1, 2, 3 };
    // [] işleci:
    int a = v[1];
    v[1] = 4;
    // at() fonksiyonu:
    int b = v.at(2);
    v.at(2) = 5;
    //int c = v.at(3); // Hata 4. Eleman Yok!
    for (std::size_t i = 0; i < v.size(); ++i) {
        v[i] = 1;
    }
}
```

Ayrıca `front()` ve `back()` fonksiyonları ile de vektör elemanlarına erişilebilir;

```
std::vector<int> v{ 4, 5, 6 };
int a = v.front();
v.front() = 3;
int b = v.back(); // v.back() eşdeğerdir: v[v.size() - 1]
v.back() = 7;
```

Vektöre eleman ekleme `push_back()`, vektörden eleman çıkarma `pop_back()` ve boş olduğunu kontrol etme `empty()` fonksiyonları ile yapılabilir;

```
#include <iostream>
#include <vector>
int main ()
{
    std::vector<int> v;
    int toplam (0);
    for (int i=1;i<=10;i++) v.push_back(i); //vektör oluştutulup elemanlar ekleniyor
    while (!v.empty()) //vektörün boş olup olmadığı kontrol edilip ilerleniyor
    {
        toplam += v.back();
        v.pop_back(); //son elemanı vektörden çıkar
    }
    std::cout << "toplam: " << toplam << std::endl;
}
```

Vektörde `data()` yöntemi, `std::vector`'un elemanlarını dahili olarak depolamak için kullandığı ham belleğe bir gösterici döndürür. Bu gösterici, genellikle vektör verilerini C tarzı bir dizi bekleyen eski koda geçirirken kullanılır.

```
std::vector<int> v{ 1, 2, 3, 4 };
int* p = v.data();
*p = 4; // v[0]=4
++p;
*p = 3; // v[1]=3
p[1] = 2;
*(p + 2) = 1;
```

std::map, std::multimap ve std::pair

Haritalar, benzersiz anahtarlara sahip anahtar-değer çiftleri (key-value pair) içeren sıralı bir ilişkisel konteynerlerdir. Anahtarlar, `Compare()` karşılaştırma işlevi kullanılarak sıralanır. Arama, kaldırma ve ekleme işlemleri olan `search()`, `insert()`, `delete()` için logaritmik karmaşıklığa sahiptir.

- `std::map` veya `std::multimap`'ten herhangi birini kullanmak için `<map>` başlık dosyası projeye dahil edilmelidir.
- `std::map` ve `std::multimap` her ikisi de elemanlarının anahtarların artan sırasına göre sıralanmış halde tutar.
- `std::multimap` durumunda, aynı anahtarın değerleri için sıralama yapılmaz.
- `std::map` ve `std::multimap` arasındaki temel fark, `std::map`'in aynı anahtar için yinelenen değerlere izin vermemesidir.

Çiftler aşağıdaki gibi oluşturulup karşılaştırılabilir;

```
std::pair<int, int> p1 = std::make_pair(1, 2);
std::pair<int, int> p2 = std::make_pair(2, 2);
if (p1 == p2)
    std::cout << "eşit";
else
    std::cout << "eşit değil!"
```

Çiftler bir konteyner içindeyken de küçük işleci ile sıralanabilir;

```
#include <iostream>
#include <utility>
#include <vector>
#include <algorithm>
#include <string>
int main()
{
    std::vector<std::pair<int, std::string>> v = { {2, "Ali"}, {2, "Mustafa"}, {1, "İlhan"} };
    std::sort(v.begin(), v.end());
    for(const auto& p: v) {
        std::cout << "(" << p.first << "," << p.second << ") ";
    }
}
/*Program Çıktısı:
(1,İlhan) (2,Ali) (2,Mustafa)
*/
```

Haritalarda elemanlar anahtar değer ikilisidir ve aşağıdaki şekilde yineleyiciler ile erişilir;

```
#include <iostream>
#include <map>
int main()
{
    std::map < int, std::string > siralama {
                                                std::make_pair(2, "ahmet"),
```

```
                                        std::make_pair(1, "ali") };

    siralama[1]="mustafa";
    std::cout << siralama.at(2) << std::endl; // ahmet
    auto it = siralama.begin();
    std::cout << it->first << " : " << it->second << std::endl; // "1 : mustafa"
    it++;
    std::cout << it->first << " : " << it->second << std::endl; // "2 : ahmet"

    std::map < std::string , int> notlar {
                                        std::make_pair("ahmet",100),
                                        std::make_pair("ali",90),
                                        std::make_pair("mustafa",85)
                                        };
    notlar["mustafa"]=95;
    std::cout << notlar.at("ali") << std::endl; // 90
    auto it2 = notlar.rbegin(); //tersten
    std::cout << it2->first << " : " << it2->second << std::endl; // "mustafa: 95"
    it2++;
    std::cout << it2->first << " : " << it2->second << std::endl; // "ali: 90"
}
```

Aşağıdaki örnekte ise `std::multimap` örneği de verilmiştir;

```
#include <iostream>
#include <map>
int main()
{
    std::multimap < int, std::string > mmp {
                                        std::make_pair(2, "ahmet"),
                                        std::make_pair(1, "ali"),
                                        std::make_pair(2, "mustafa")

    };
    auto it = mmp.begin();
    std::cout << it->first << " : " << it->second << std::endl; //"1 : ali"
    it++;
    std::cout << it->first << " : " << it->second << std::endl; //"2 : ahmet"
    it++;
    std::cout << it->first << " : " << it->second << std::endl; //"2 : mustafa"

    std::map < int, std::string > mp {
                                    std::make_pair(2, "ahmet"),
                                    std::make_pair(1, "ali"),
                                    std::make_pair(2, "mustafa")
                                    // Aynı anathardan var!
    };
    auto it2 = mp.rbegin(); //tersten
    std::cout << it2->first << " : " << it2->second << std::endl; //"2 : ahmet"
    it2++;
    std::cout << it2->first << " : " << it2->second << std::endl; //"1 : ali"
}
```

Haritaya anahtar değerler aşağıdaki şekilde eklenebilir;

```
std::map< std::string, size_t > meyvesayisi;
meyvesayisi.insert({"üzüm", 20});
meyvesayisi.insert(make_pair("portakal", 30));
meyvesayisi.insert(pair<std::string, size_t>("muz", 40));
meyvesayisi.insert(map<std::string, size_t>::value_type("çilek", 50));
```

`insert()` fonksiyonu bir yineleyici ve bir bool değerden oluşan bir çift döndürür. Döndürülen değer true ise haritaya çift eklenmiştir. Değilse eklenmeye çalışılan çift zaten haritada vardır.

```
auto success = meyvesayisi.insert({"üzüm", 20});
if (!success.second) { // zaten üzüm var
    success.first->second += 20; // üzüm müktarı değiştirlebilir
}
meyvesayisi["karpuz"]=10; //haridata olmayan karpuz eklenmişir.
meyvesayisi.insert({{"şeftali", 1}, {"kayısı", 1}, {"limon", 1}, {"mango", 7}});
```

Konteynerlere başlatma listeleriyle de eleman eklenebilir;

```
std::map< std::string, size_t > meyvelistesi{ {"zeytin", 0}, {"elma", 0}, {"kavun", 0}};
meyvesayisi.insert(meyvelistesi.begin(), meyvelistesi.end());
```

Haritaya eklenecek çiftler yukarıdaki örneklerde olduğu gibi `make_pair()` ve `emplace()` yöntemleriyle hazırlanabilir;

```
meyvesayisi.emplace("Harnup", 200);
```

Haritalarda bir anahtarın ilk oluşumunun yineleyicisini almak için `find()` fonksiyonu kullanılabilir. Anahtar mevcut değilse `end()` döndürür;

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
auto it = mmp.find(6);
if(it!=mmp.end())
    std::cout << it->first << ", " << it->second << std::endl; // 6, 5
else
    std::cout << "Aranan Anahtar Bulunamadı!" << std::endl;
it = mmp.find(66);
if(it!=mmp.end())
    std::cout << it->first << ", " << it->second << std::endl;
else
    std::cout << "Aranan anahtar bulunamadı!" << std::endl;
```

Haritalarda bir anahtarın mevcut olduğunu bulmanın bir başka yolu da `count()` fonksiyonunu kullanmaktır. Bu fonksiyon, belirli bir anahtarla ilişkili değer sayısını sayar;

```
std::map< int , int > mp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
if(mp.count(3) > 0)
    std::cout << "Anahtar bulundu!" << std::endl;
else
    std::cout << "Anahtar Bulunamadı!" << std::endl;
```

Haritaları tanımlama ve ilk değer verme;

```
std::multimap < int, std::string > mmp { std::make_pair(2, "C Lang"),
                                         std::make_pair(1, "CPP Lang"),
                                         std::make_pair(2, "Java Lang") };

std::map < int, std::string > mp { std::make_pair(2, "C lang"),
                                   std::make_pair(1, "CPP Lang"),
                                   std::make_pair(2, "Java Lang") // Eklenmeyecek!
                                 };
// Yineleyici kullanarak;
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {6, 8}, {3, 4}, {6, 7} };
auto it = mmp.begin();
std::advance(it,3); //baştan üçüncü çifte git {6, 5}
std::map< int, int > mp(it, mmp.end()); // {6, 5}, {8, 9}, {6, 8}, {3, 4}, {6, 7}

//std::pair dizisi kullanarak;
std::pair< int, int > arr[10];
arr[0] = {1, 3};
arr[1] = {1, 5};
```

```
arr[2] = {2, 5};
arr[3] = {0, 1};
std::map< int, int > mp(arr,arr+4); //{0 , 1}, {1, 3}, {2, 5}

//std::pair vektörü kullanarak
std::vector< std::pair<int, int> > v{ {1, 5}, {5, 1}, {3, 6}, {3, 2} };
std::multimap< int, int > mp(v.begin(), v.end()); // {1, 5}, {3, 6}, {3, 2}, {5, 1}
```

Haritaların boş olup olmamasına bağlı olarak true veya false döndüren bir üye `empty()` fonksiyonuna sahiptir. Ayrıca sahip olunan eleman sayısı da `size()` üye fonksiyonuna da sahiptir.

```
std::map<std::string , int> rank {{"facebook.com", 1} ,{"google.com", 2}, {"youtube.com", 3}};
if(!rank.empty()){
    std::cout << "Hatitadadaki eleman sayısı: " << rank.size() << std::endl;
}
else{
    std::cout << "Harita Boş!" << std::endl;
}
```

Bir de karma harita (hash map) vardır. Düzenlenmemiş harita (unordered map) olarak da adlandırılan bu haritalar, normal bir haritaya benzer şekilde anahtar-değer çiftlerini depolar. Ancak elemanları anahtara göre sıralamaz. Bunun yerine, anahtar için bir karma (hash) değer, ihtiyaç duyulan anahtar-değer çiftlerine hızlı bir şekilde erişmek için kullanılır;

```
#include <string>
#include <unordered_map>
std::unordered_map<std::string, size_t> meyveSayisi;
```

Haritadaki elemanlar aşağıdaki şekilde silinebilir;

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
mmp.clear(); //hepsini sil

auto it = mmp.begin();
std::advance(it,3); // baştan üçüncü elemana git {6, 5}
mmp.erase(it); // {1, 2}, {3, 4}, {8, 9}, {3, 4}, {6, 7}
```

Belli anahtarlara sahip elemanlar `erase()` üye fonksiyonu ile silinebilir;

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
mmp.erase(6); // {1, 2}, {3, 4}, {3, 4}, {8, 9}
```

std::list ve std::forwardlist

Konteynerin herhangi bir yerinden öğelerin hızlı bir şekilde eklenmesini ve kaldırılmasını istiyorsak `std::forward_list` bir konteynerdir. Hızlı rastgele erişim desteklenmez. Bu konteyner, `std::list` ile karşılaştırıldığında, çift yönlü yineleme gerekmediğinde daha fazla alan tasarrufu sağlar.

```
#include <forward_list>
#include <list>
#include <string>
#include <iostream>
template<typename T> std::ostream& operator<<(std::ostream& s, const std::forward_list<T>& v)
{
    s << "Forward List";
    s.put('[');
    char comma[3] = {'\0', ' ', '\0'};
    for (const auto& e : v) {
        s << comma << e;
        comma[0] = ',';
    }
    return s << ']';
```

```
}
template<typename T> std::ostream& operator<<(std::ostream& s, const std::list<T>& v) {
    s << "List";
    s.put('[');
    char comma[3] = {'\0', ' ', '\0'};
    for (const auto& e : v) {
        s << comma << e;
        comma[0] = ',';
    }
    return s << ']';
}
int main() {
std::forward_list<std::string> words1 {"İlhan", "Ahmet", "Mehmet", "Mustafa", "Abdullah"};
std::cout << "İsimler1: " << words1 << '\n';

std::list<std::string> words2 {"İlhan", "Ahmet", "Mehmet", "Mustafa", "Abdullah"};
std::cout << "İsimler2: " << words2 << '\n';
}
/* Program Çıktısı:
İsimler1: Forward List[İlhan, Ahmet, Mehmet, Mustafa, Abdullah]
İsimler2: List[İlhan, Ahmet, Mehmet, Mustafa, Abdullah]
*/
```

std::set ve std::multiset

Küme (`std::set`), elemanları sıralanmış ve benzersiz olan bir tür konteynerdir. Ancak `std::multiset` birden fazla eleman aynı elemana sahip olabilir. Lamda ifadeleri C++ 14 ile hayatımıza girmiştir.

```
#include <iostream>
#include <set>
#include <stdlib.h>
struct custom_compare final
{
    bool operator() (const std::string& left, const std::string& right) const
    {
        int nLeft = atoi(left.c_str());
        int nRight = atoi(right.c_str());
        return nLeft < nRight;
    }
};
int main ()
{
    std::set<std::string> sut({"1", "2", "5", "23", "6", "290"});
    std::cout << "Ön tanımlı sıralama std::set<std::string> :" << std::endl;
    for (auto &&data: sut)
        std::cout << data << ", ";
    std::set<std::string, custom_compare> sut_custom(
                                          {"1", "2", "5", "23", "6", "290"},
                                          custom_compare{});
    std::cout << std::endl << "Özel Sıralama :" << std::endl;
    for (auto &&data : sut_custom)
        std::cout << data << ", ";
    auto compare_via_lamda = [](auto &&lhs, auto &&rhs){ return lhs > rhs; };
    using set_via_lamda = std::set<std::string, decltype(compare_via_lamda)>;
    set_via_lamda sut_reverse_via_lamda({"1", "2", "5", "23", "6", "290"},
                                          compare_via_lamda);
    std::cout << std::endl << "Lamda Sıralaması :" << std::endl;
    for (auto &&data : sut_reverse_via_lamda)
        std::cout << data << ", ";
    return 0;
}
```

```
/*g++ main.cpp -std=C++14 ile derlenen Program Çıktısı:
Ön tanımlı sıralama std::set<std::string> :
1, 2, 23, 290, 5, 6,
Özel Sıralama :
1, 2, 5, 6, 23, 290,
Lamda Sıralaması :
6, 5, 290, 23, 2, 1,
*/
```

std::optional

`std:optional` olarak tanımlananlar isteğe bağlı/belki tipleri olarak da bilinir. İçeriği mevcut olabilen veya olmayabilen bir veri tipi temsil etmek için kullanılır.

C++17'den önce, `nullptr` değerine sahip göstericilere sahip olmak genellikle bir değerin yokluğunu temsil ediyordu. Bu, dinamik olarak tahsis edilmiş ve zaten göstericiler tarafından yönetilen büyük nesneler için iyi bir çözümdür. Ancak, bu çözüm nadiren göstericiler tarafından dinamik olarak tahsis edilen veya yönetilen int gibi küçük veya ilkel türler için iyi çalışmaz. `std::optional` bu yaygın soruna uygulanabilir bir çözüm sağlar.

```
#include <iostream>
#include <optional>
#include <string>
struct Hayvan {
    std::string adi;
};

struct Person {
    std::string adi;
    std::optional<Hayvan> evcilhayvan;
};
int main() {
    Person person;
    person.adi = "Ali";
    if (person.evcilhayvan) {
        std::cout << person.adi << " Adlı Kişinin " << person.evcilhayvan->adi
                  << "Evcil Hayvanı Var" << std::endl;
    }
    else {
        std::cout << person.adi << " adlı kişinin evcil hayvanı yok!" << std::endl;
    }
}
/* g++ -std=c++17 main.cpp olarak derlenen Program Çıktısı:
Ali adlı kişinin evcil hayvanı yok!
*/
```

Fonksiyondan geri dönüş değeri olarak da `std::optional` kullanılabilir;

```
std::optional<float> divide(float a, float b) {
    if (b!=0.f) return a/b;
    return {};
}
```

`std:optional` olarak tanımlı bir değeri işlemek için `value_or` kullanılır;

```
void print_name( std::ostream& os, std::optional<std::string> const& name ) {
    std::cout "Name is: " << name.value_or("<name missing>") << '\n';
}
```

std::function ve std::bind

Herhangi bir fonksiyona kılıf çekmek için kullanılır. C tipi fonksiyon göstericisi kullanmak yerine C++ dilinde `std::function` kullanılır.

```
#include <iostream>
#include <functional>
std::function<void(int , const std::string&)> myFuncObj;
void theFunc(int i, const std::string& s) {
    std::cout << s << ": " << i << std::endl;
}
int main(int argc, char *argv[]) {
    myFuncObj = theFunc;
    myFuncObj(10, "hello world");
}
```

Argümanlarla bir fonksiyonu geri çağırmamız gereken bir durumda `std::bind` ile kullanılan `std::function` aşağıda gösterildiği gibi çok güçlü bir tasarım yapısı verir.

```
#include <iostream>
#include <functional>
class Ocak {
public:
    std::function<void(int, const std::string&)> YumurtaKaynadi = nullptr;
    void YumurtaKaynat() {
        if (YumurtaKaynadi) {
            YumurtaKaynadi(10, "Yumurta Pişti");
        }
    }
};
class Kisi {
public:
    Kisi() {
        auto yumurtaPisinceHaberVer = std::bind(&Kisi::eventHandler, this,
                                                std::placeholders::_1,
                                                std::placeholders::_2);
        ocakNesnesi.YumurtaKaynadi = yumurtaPisinceHaberVer;
    }
    void eventHandler(int i, const std::string& s) {
        std::cout << i << " dakikada "<< s <<std::endl;
    }
    void OcaktaYumurtaKaynat() {
        ocakNesnesi.YumurtaKaynat();
    }
    Ocak ocakNesnesi;
};
int main(int argc, char *argv[]) {
    Kisi ali;
    ali.OcaktaYumurtaKaynat();
}
/*Program Çıktısı:
10 dakikada Yumurta Pişti
*/
```

Aşağıda `std:function` ile çeşitli fonksiyonların çağrılmasını gösteren bir program verilmiştir;

```
#include <iostream>
#include <functional>
#include <iostream>
#include <vector>
using std::cout;
using std::endl;
```

```
using namespace std::placeholders;
double c_function(int x, float y, double z) {
    double res = x + y + z;
    std::cout << "c_function çağrıldı: "
    << x << "+" << y << "+" << z
    << "=" << res
    << std::endl;
    return res;
}
struct yapi
{
    double yapi_function(int x, float y, double z) {
        double res = x + y + z;
        std::cout << "yapi::yapi_function çağrıldı: "
        << x << "+" << y << "+" << z
        << "=" << res
        << std::endl;
        return res;
    }
    double farkli_yapi_function(int x, double z, float y, long xx) {
        double res = x + y + z + xx;
        std::cout << "farkli_yapi_function çağrıldı: "
        << x << "+" << y << "+" << z << "+" << xx
        << "=" << res
        << std::endl;
        return res;
    }
    double operator()(int x, float y, double z) {
        double res = x + y + z;
        std::cout << "yapi::operator() çağrıldı: "
        << x << "+" << y << "+" << z
        << "=" << res
        << std::endl;
        return res;
    }
};
int main(void) {
    using function_type = std::function<double(int, float, double)>;
    yapi ornekyapi;
    std::vector<function_type> bindings;

    function_type var1 = c_function;
    bindings.push_back(var1);
    function_type var2 = std::bind(&yapi::yapi_function, ornekyapi, _1, _2, _3);
    bindings.push_back(var2);
    function_type var3 = std::bind(&yapi::farkli_yapi_function, ornekyapi, _1, _3, _2, 11l);
    bindings.push_back(var3);
    function_type var4 = ornekyapi; //operator()
    bindings.push_back(var4);

    function_type var5 = [](int x, float y, double z)
    {
        double res = x + y + z;
        std::cout << "lamda çağrıldı: "
        << x << "+" << y << "+" << z
        << "=" << res
        << std::endl;
        return res;
    };
    bindings.push_back(var5);
```

```
    std::cout << "Vektöre saklana fonksiyonlar test ediliyor: x = 1, y = 2, z = 3"
    << std::endl;
    for (auto f : bindings)
        f(1, 2, 3);
}
```

std::tupple

Farklı tiplerdeki değerler de dahil olmak üzere herhangi bir sayıda değeri tek bir dönüş nesnesine toplamak için `std::tuple` kullanılır.

```
std::tuple<int, int, int, int> foo(int a, int b) { // or auto (C++14)
    return std::make_tuple(a + b, a - b, a * b, a / b);
           /*dört farklı int değeri geri döndürülüyor */
}
```

C++ 2017 uyarlamasından sonra tırnaklı parantez olan başlatma listesi kullanılır;

```
std::tuple<int, int, int, int> foo(int a, int b) {
    return {a + b, a - b, a * b, a / b};
}
```

Döndürülen `tuple`'dan değerleri almak zahmetlidir ve `std::get` şablon fonksiyonunun kullanımını gerektirir:

```
auto mrvs = foo(5, 12);
auto add = std::get<0>(mrvs);
auto sub = std::get<1>(mrvs);
auto mul = std::get<2>(mrvs);
auto div = std::get<3>(mrvs);
```

Eğer tipler fonksiyon dönmeden önce bildirilebiliyorsa, o zaman `std::tie` bir `tuple`'ı mevcut değişkenlere açmak için kullanılabilir:

```
int add, sub, mul, div;
std::tie(add, sub, mul, div) = foo(5, 12);
```

`std::tie` ile döndürülen değerlerden birine ihtiyaç duyulmuyorsa `std::ignore` kullanılabilir:

```
std::tie(add, sub, std::ignore, div) = foo(5, 12);
```

Yapılandırılmış bağlamalar `std::tie`'dan kaçınmak için kullanılabilir:

```
auto [add, sub, mul, div] = foo(5,12);
```