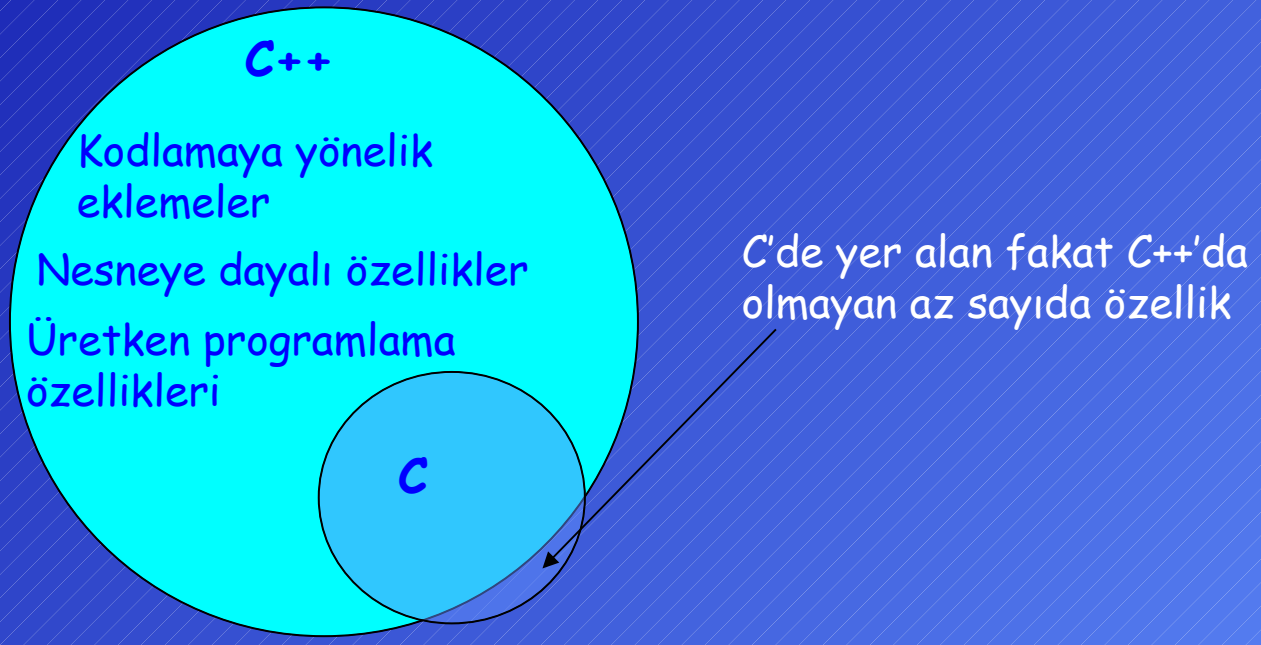
C++ : DAHA İYİ BİR C

C++ programlama dili, C diline bazı özelliklerin eklenmesiyle oluşturulmuştur. Yapılan eklemeler üç grupta toplanabilir:

1. Programcıların kodlama aşamasında kullanabilecekleri teknik özellikler. Bunlar programlama yöntemiyle ilgili değildir.
2. Nesneye dayalı programlama yöntemini destekleyen özellikler.
3. Üretken programlamayı (generic programming) destekleyen özellikler.

C++, C dilinin bir üst kümesi olduğundan bu dilin hemen hemen tüm özelliklerini içerir.



Nesneye dayalı programlama ile ilgili olmayan, ancak kodlama aşamasında C diline göre üstünlükler sağlayan eklentiler:

Tek Satırlık Açıklamalar

C'de kullanılan, /* */ sembolleri arasında yazılan ve birden fazla satıra yayılabilen açıklamalara ek olarak, C++'da sadece tek satır uzunluğunda olabilen ve // simgesiyle başlayan açıklamalar yazmak mümkündür. Örnek:

```
c = a + b ;    // Bu bir açıklamadır
```

Bildirimler (Declarations) ve Tanımlar (Definitions)

Bir **bildirim** yapıldığında, derleyiciye o program dosyasında kullanılacak olan verilerin ya da fonksiyonların sadece adı ve tipleri belirtilmiş olur.

Bildirimler veriler ya da fonksiyonlar için bellekte yer ayırmazlar.

Bir **tanımlama** satırı derlendiğinde ise veriler için bellekte yer ayrılır ve fonksiyonların gövdeleri belleğe yerleştirilir.

Bir veri ilk tanımlandığı yerden itibaren geçerli olur.

Verinin geçerliliği tanımlandığı bloktan çıkıldığında sona erer.

Bir verinin geçerli olduğu bloğa o verinin *yaşam alanı* (*scope*) denir.

```
extern int i;          // Bildirim: i bir tamsayıdır başka bir yerde tanımlanmıştır.
int i;                 // Tanımlama: tamsayı i'ye bellekte yer ayrılır.

struct ComplexT{   // Bildirim: ComplexT adında bir veri grubunun tipi bildiriliyor
   float re,im;
};
ComplexT c1, c2;              // Tanım: c1 ve c2 tanımlanıyor.

void fonk(int, float);        // Bildirim: fonksiyonun adı ve parametre tipleri.
                              // Bu fonksiyonun gövdesi fonksiyonun tanımı olacaktır.
```

C dilinde bildirimler ve tanımlar tüm yürütülebilir program satırlarından önce yapılmak zorundadır. C++'da ise bildirim ve tanımlamaları yürütülebilir program deyimlerinin yazıldığı yerlere, yani program kodunun içine yazılabilir.

```
int a=0;
for (int i=0; i < 100; i++){   // i for döngüsünün başında tanımlanmıştır
   a++;
   int p=12;                    // p değişkeni döngü içinde tanımlanıyor
   ...
}                               // p değişkeninin geçerliliği sona eriyor
```

Yaşam Alanı Belirtme (*Scope Resolution*) Operatörü (::)

Bir C++ programında iç içe yer alan bloklarda, aynı isimde veriler tanımlanabilir. Böyle bir durumda iç bloktaki (yerel) veri, aynı isme sahip olan dış bloktakini (global) örter (değeri bozulmaz).
Yaşam Alanı Belirtme (*Scope Resolution*) operatörü (::) kullanılarak örtülmüş olan verilere de erişmek mümkündür.

```
int y=0;          // Global y
int x=1;          // Global x
void f(){         // Yeni bir blok başlıyor
   int x=5;       // Yerel  x=5, global x'i örtüyor
   ::x++;         // Global x=2 oldu
   x++;           // Yerel x=6 oldu
   y++;           // Global y=1 oldu, örtülmediği için operatöre gerek yok.
}
```

C dilinde olduğu gibi *C++*'da da aynı simge birden fazla değişik anlamda kullanılmaktadır. İlerleyen bölümlerde (::) simgesinin de değişik anlamlarda kullanıldığı görülecektir.

İsim Uzayı (*Name Space*)

Program boyları büyüdükçe tüm verilere farklı isimler bulmak zorlaşmaktadır. Özellikle takım halinde yazılan programlarda isim çakışması problemi ile karşılaşılabilir.

Benzer bir durum nesne ya da fonksiyon arşivlerinin kullanımında da ortaya çıkmaktadır. Başkası tarafından yazılmış bir arşivi kullanırken yanlışlıkla kendi programınızın içindeki verilere de arşivdeki verilerle aynı isimleri verirseniz derleme hatası ile karşılaşırsınız.

Bu problemi gidermek C++'da isim uzayı kavramı geliştirilmiştir. Buna göre her programcı kendi tanımlamalarını kendine ait bir isim uzayı içinde yapar. Bir isim uzayındaki verinin ismi başka bir uzaydaki ile aynı olsa bile çakışma olmaz.

```
namespace programci1{          //programci1'in isim uzayı
    int tamsayi;               // programci1'e ait tamsayi
    void f(int);
    :                          // Diğer değişkenler
}                              // İsim uzayının sonu
namespace programci2{          // programci2'nin isim uzayı
    int tamsayi;               // programci2'ye ait tamsayi
    :                          // Diğer değişkenler
}                              // İsim uzayının sonu
```

İsim uzaylarındaki verilere erişilmesi:

```
programci1::tamsayi =3;          //progamci1'in uzayındaki tamsayi
programci2::tamsayi=-345;        //progamci2'nin uzayındaki tamsayi
programci1::f(6);                //progamci1'in uzayındaki f fonksiyonu
```

using Bildirimi:

İsim uzaylarındaki verilere her erişimde uzayında isminin yazılması gereğini ortadan kaldırarak erişimi kolaylaştırmak için using bildirimi kullanılmaktadır.

```
using programci1::tamsayi;    // İsim uzayındaki bir değişken için geçerlidir
tamsayi = 3;                  // programci1'deki tamsayi
programci2::tamsayi = -345;   // Diğer tamsayi için isim uzayını belirtmek gerekli
programci1::f(6);             // f fonksiyonu için isim uzayını belirtmek gerekli
```

Bu bildirim istenirse tek bir veri yerine isim uzayının tamamı için geçerli yapılabilir. Bunun için using namespace sözcükleri birlikte kullanılır.

```
using namespace programci1;   // İsim uzayının tamamı için geçerli
tamsayi = 3;                  // programci1'deki tamsayi
f(6);                         // programci1'deki f
programci2::tamsayi = -345;   // Diğer isim uzayı için isim belirtmek
gerekli
```

Standart C++ Başlık (*Header*) Dosyaları

C++'ın ilk sürümlerinde başlık dosyalarında çoğunlukla '**.h**' uzantısı kullanılmıştır. Ancak değişik işletim sistemlerinde ve derleyicilerde farklı uzantılar da kullanıldığından (örneğin; .hpp, .H gibi) tam bir standart sağlanamamıştır.

Bu problemi ortadan kaldırmak için C++'nın son güncel standardında hem arşivlerdeki modellerin yapıları yeniden düzenlenmiş hem de başlık dosyalarındaki uzantı kaldırılmıştır.

Bu nedenle C++'nın standart başlık dosyalarında artık uzantı yoktur.

Örneğin; iostream, string, vector gibi.

Standart arşivler düzenlenirken tüm veriler **std** adındaki bir isim uzayı içinde tanımlanmıştır.

Bu arşivleri kullanırken aşağıda gösterildiği gibi isim uzayını dikkate almak gerekir:

```
#include <iostream>        // Başlık dosyası iostream programın başına eklendi
using namespace std;       // Arşivdeki veriler std uzayında olduğu için
```

Günümüzdeki bir çok C++ derleyicisi eski programlara destek vermek için hem daha eski sürümlerdeki arşivleri ve başlık dosyalarını hem de yeni arşivleri ve ilgili başlık dosyalarını içermektedir. Programcı isterse programının başına '.h' uzantılı eski başlık dosyalarını da ekleyebilir. Örneğin,

```
#include <iostream.h> // Eski başlık dosyası iostream.h programın başına eklendi
```

C++ derleyicilerine C dilinden gelen başlık dosyaları da yeni standarda uydurulmuş ve uzantıları kaldırılmıştır. Bu dosyaların C'den geldiğini belirtmek için isimlerinin başına '**c**' harfi eklenmiştir.

C'de	C++'da
include<stdio.h>	#include<cstdio>
include<stdlib.h>	#include<cstdlib>

Giriş/Çıkış İşlemleri

C++'da C'den farklı olarak giriş/çıkış fonksiyonlarının yerine giriş/çıkış nesneleri kullanılır.

cin : Standart giriş birimi. Çoğunlukla tuş takımı işlemlerinde kullanılır.
cout : Standart çıkış birimi. Çoğunlukla ekran işlemlerinde kullanılır.

cout nesnesi üzerinde ekrana bir veri yazmak için gönderme operatörü '**<<**' kullanılır.

Bkz. Örnek: o21.cpp

Giriş işlemlerinde ise cin nesnesi ve giriş operatörü '**>>**' kullanılmaktadır.

Bkz. Örnek: o22.cpp

Mantık Değişkenleri

C dilinde, doğru (*true*), yanlış (*false*) gibi mantıksal değerleri tutmak için özel değişkenler yoktur. Bu işlemler için de tamsayılar kullanılır. Sıfır değeri 'yanlış'a karşı düşürülürken, sıfırdan farklı her değer de 'doğru' olarak kabul edilir.
Bu özellik C++'da da sürdürülmekle beraber programların anlaşılırlılığını arttırmak amacıyla mantıksal değişkenler için yeni bir tip (**bool**) yaratılmıştır. Ayrıca mantıksal işlemlerde kullanılmak üzere true ve false özel sözcükleri de derleyiciye eklenmiştir.

```
bool buyuk_mu;        // Mantıksal değişken
buyuk_mu = false;     // Mantıksal değer ataması
int a,b;
.................
buyuk_mu = a > b;     // Mantıksal işlem
if (buyuk_mu)  ....   // Mantıksal değerlendirme
```

Sabitler

Standart C dilinde sabitler tanımlamak için önişlemcinin #define direktifi kullanılmaktadır. Örnek olarak aşağıda PI sabitinin tanımlanması gösterilmiştir.

```
#define  PI        3.14
```

C++'da ise sabitlerin tanımlanması için çok daha etkili bir yol geliştirilmiştir. Bunun için **const** sözcüğü kullanılmaktadır.

```
const int i = 100;          // i sabittir ve değeri 100'dür.
.....
i = 5;                      // Derleme hatası!  Çünkü i sabit olarak tanımlanmıştır.
```

const sözcüğü veri tipinin önünde (solunda) yer alacağı gibi arkasında (sağında) da yer alabilir. Programın çalışması açısından ikisi arasında hiç bir fark yoktur.

```
int const i = 100;          // const int i = 100;   ile aynı
```

Dinamik Bellek Yönetimi

C++'da dinamik bellek işlemlerini gerçekleştirmek için C'ye göre kullanımı daha kolay olan ve daha çok yeteneğe sahip operatörler tanımlanmıştır. Bu operatörlerden **new** bellekten yer almak için, **delete** ise alınan yerleri geri vermek için kullanılır.

```
int *ip;            // Tamsayılara işaret eden işaretçi tanımlanıyor
ip = new int;       // İşaretçiye bir tamsayılık yer alınıyor.
......
delete ip;          // Alınan yer geri veriliyor.
```

Aşağıdaki örnekte ip işaretçisinin işaret ettiği yere başlangıç değeri olarak 5 yerleştirilmiştir:

```
ip = new int(5);   // İşaretçiye bir tamsayılık yer alınıyor, başlangıç değeri 5
```

Aynı tipten birden fazla eleman için bellekte yer ayrılacaksa, yani bir **dizi** yaratılacaksa new[eleman sayısı] operatörü kullanılır. Bu şekilde alınan yerleri geri vermek için de delete [] operatörü kullanılır.

```
int *ipd = new int[10];                  // 10 elemanlı dizi için yer ayrılıyor
for (int k=0; k<10; k++) ipd[k]= 0;      // dizinin elemanları sıfırlanıyor
delete [] ipd;                           // Ayrılan yerler serbest bırakılıyor.
```

Fonksiyon Bildirimleri ve Tanımları

C++ dilinde hata olasılığını azaltmak için fonksiyon çağrılarında tip kontrolleri katı bir biçimde yapılmaktadır.

Bir fonksiyon çağrısı yapılan dosyada, mutlaka o fonksiyonun bildirimi ya da tanımı olmalıdır. Bir fonksiyonun bildirimine o fonksiyonun **prototipi** denir.

```
char basari(int, int, int);                    // basari adli fonksiyonun prototipi (bildirimi)
```

Fonksiyonun tanımında parametrelerin gerçek adları yer alacaktır:

```
char basari (int vize1, int vize2, int final)       // basari adli fonksiyonun tanımı
{
    char sonuc;
        .....          // Fonksiyonun gövdesi, sonuc uygun şekilde hesaplanacak
    return sonuc;
}
```

Bir fonksiyon parametre almayacaksa parametre listesi (parantezin içi) boş bırakılır. C dilinde olduğu gibi buraya void yazmaya gerek yoktur.

```
int print (void);     /* C'deki yöntem */
int print ();         // C++'daki yöntem
```

inline Fonksiyonlar (Makrolar)

C dilinde makroları tanımlamak için önişlemcinin #define direktifi kullanılmaktadır. C++'da ise makrolar normal fonksiyonlar gibi yazılabilmektedir.

Fonksiyonların gövdeleri bir kez derlenir ve makine dilindeki kod belleğe yerleştirilir. Bu fonksiyon her çağırıldığında çağıran programdan fonksiyonun koduna gidilir ve bu kod yürütüldükten sonra yine çağıran programa dönülür.

Makrolarda ise, çağrının yapıldığı her yere makronun kodu eklenir. Yani her makro çağrısı programın boyunun uzamasına neden olur. Burada başka bir koda gitme, geri gelme ve parametre aktarımı olmadığından makro çağrıları fonksiyonlara göre daha hızlı gerçekleşirler.

Özelikle hızlı çalışmasını istediğimiz ve kodu kısa olan program parçalarını makro olarak oluşturmak doğru olacaktır.

```
inline int max (int i1,int i2){        // adı max olan, iki tamsayı parametre alan
    return(i1>i2) ? i1 : i2;           // ve bir tamsayı değer üreten makro
}
```

Makroların kullanımları normal fonksiyon çağrıları gibidir.

```
int j, k, l ;          // Üç adet tamsayı değişken
.....                  // k ve l üzerinde bazı işlemler yapılıyor ve değerler atanıyor
j = max( k, l )        // max adlı makro çağırılıyor
```

Fonksiyon Parametrelerinin Başlangıç Değerleri

Örnek olarak aşağıda üç adet parametre alan bir fonksiyon tanımlanmıştır. Birinci parametrenin (char c) başlangıç değeri yoktur. İkinci parametrenin (int i1) başlangıç değeri 0, üçüncü parametrenin (int i2) başlangıç değeri ise 1'dir.

```
void f(char c, int i1 = 0, int i2 = 1)   // i1 ve i2'nin başlangıç değerleri var
{ ....}                                  // Fonksiyonun gövdesi, önemli değil
```

Bu fonksiyon aşağıdaki gibi üç farklı şekilde çağrılabilir:

```
f('A',4,6);          // c='A', i1=4, i2=6
f('B',3);            // c='B', i1=3, i2=1
f('C');              // c='C', i1=0, i2=1
```

Bir fonksiyon çağırılırken parametrelere karşı düşecek olan argümanlar soldan sağa doğru verilmelidir.

```
f('C', ,7);    //  HATA! İkinci parametre eksik bırakılmış, üçüncü verilmiş
```

Fonksiyonlar yazılırken de parametrelere başlangıç değerleri sağdan başlanarak ve parametre atlamadan verilmelidir.

```
void f(char c = 'A', int i1, int i2 = 1)      // HATA! Aradaki parametre atlanmış.
```

Fonksiyon İsimlerinin Yüklenmesi (*Overloading of Function Names*)

C++ derleyicileri fonksiyonları birbirinden ayırırken sadece isimlerini değil parametre listelerini de dikkate alırlar.
Buna göre iki farklı fonksiyon, parametrelerinin tipleri, sayıları veya sıraları farklı olmak koşulu ile aynı isme sahip olabilirler.
Bir fonksiyonun ismi ve parametre listesi o fonksiyonun **imzasını** (*signature*) oluştururlar.

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

struct ComplexT{                    // Karmaşık sayıları için bir yapı
   float re,im;                     // reel ve sanal kısımlar
};

void print (float deger){           // reel sayılar için
    cout <<" deger = " << deger << endl;
}

void print (ComplexT c){            // Karmaşık sayılar için
    cout <<"reel= " << c.re << " , sanal= " << c.im << endl;
}

void print (float deger,char c){    // reel sayılar ve char için
    cout << "deger= " << deger << " , c= " << c << endl;
}
```

```
int main()
{
    ComplexT z;

    z.re=0.5;
    z.im=1.2;
    print(z);
    print(4.2);
    print(2.5,'A');
    return 0;
}
```

Bkz. Örnek: o23.cpp

Referans

C++'da ampersan (&) simgesini kullanarak değişkenler için referanslar tanımlamak mümkündür. Bir değişken için referans tanımlanması aynı bellek gözüne ikinci bir isim vermek demektir.

```
int i = 5;
int &j = i;          // j, i'ye bir referanstır.  j ve i aynı adrese sahipler.
j++;                 // i = 6 oldu
```

Fonksiyonlara Parametrelerin Referans Olarak Gönderilmesi

C'de eğer parametre olarak gönderilen bir verinin orijinal değerinin fonksiyon içinde değişmesi isteniyorsa bu durumda o verinin **adresinin** parametre olarak fonksiyona gönderilmesi gerekir.

```
/* C'deki Çözüm: İşaretçiler */
void hesap(int *j) {
    *j = *j**j/2;                /* Okuması ve anlaması zor */
}
int main()
{
    int i=5;
    hesap(&i);                   /* i'nin adresi gönderiliyor */
    return 0;
}
```

Örnekte kullanılan & simgesi adres operatörüdür (referans değildir).

Aynı fonksiyonun C++'da referanslar kullanılarak yazılması

```
// C++'da referanslar ile çözüm
void hesap(int &j)   // j gelecek olan veriye bir referanstır, aynı adrese sahiptir
{ j = j*j/2; }       // Fonksiyonun gövdesi, j normal bir değişken gibi

int main( )
{
    int i=5;
    hesap(i);
    return 0;
}
```

Büyük boyutlu veriler bir fonksiyona parametre olarak aktarılırken değerleri yerine adreslerinin aktarılması tercih edilir.
Çünkü değer olarak aktarılması durumunda büyük boyutlu verinin tamamı yığına kopyalanacaktır.
Bu da hem bellekte fazla yer harcanmasına hem de programın yavaşlamasına neden olacaktır.
Eğer bir veri içeriğinin değişmesi için değil, sadece aktarım zamanını kısaltmak için referans olarak aktarılacaksa fonksiyondaki referans parametresi sabit (**const**) olarak tanımlanmalıdır.

```
#include <iostream>                    // Giriş/çıkış için
#include <cstring>                     // Karakter katarları için
using namespace std;

struct Kisi{                           // Kişileri tanımlamak için bir yapı
    char isim [40];                    // İsim alanı 40 sekizli
    int sicil_no;                      // Sicilno alanı 4 sekizli
};                                     // Toplam 44 sekizli

void goster (const Kisi &k)            // parametre sabit referans
{
    cout << "İsim: " << k.isim << endl;    // isim ekrana
    cout << "No: " << k.sicil_no << endl;  // sicil_no ekrana
}

int main(){
    Kisi ahmet;                        // Kisi tipinden bir veri ahmet
    strcpy(ahmet.isim,"Ahmet Bilir");  // isim = "Ahmet Bilir"
    ahmet.sicil_no=324;                // sicil_no= 324
    goster(ahmet);                     // Fonksiyon çağırılıyor.
    return 0;
}
```

Fonksiyonlardan Verilerin Referans Olarak Döndürülmesi

C++'da ise gerekli olduğu durumlarda geri döndürülen veri bir referans olarak tanımlanabilir ve bu durumda değer aktarımı adresler ile yapılır.
Bu tür aktarım iki yarar sağlar:

- Büyük boyutlu veriler yığına kopyalanmamış olur
- Geri döndürülen adrese çağıran program içinde atama yapılabilir

```
// max fonksiyonu dizinin başlangıç adresini ve uzunluğunu alıyor
// en büyük elemana bir referansı geri döndürüyor
int&  max(int a[], int uzunluk)
{
     int i=0;                                  // i en büyük elemanın indisi olacak
     for (int j= 0 ; j<uzunluk ; j++)          // En büyük eleman aranıyor
              if (a[j] > a[i])  i = j;
     return a[i];                              // Eleman geri döndürülüyor (referans)
}

int main()
{
    int dizi[] = {12, -54 , 0 , 123, 63};  // Bir dizi tanımlanıyor, değerler atanıyor
    max(dizi,5) = 0;                       // En buyuk eleman bulundu ve sıfırlandı
      :
    return 0;
}
```

Bkz. Örnek: o24.cpp

Büyük boyutlu verileri geri döndürmek için de referanslar kullanılır.

Böyle durumlarda geri döndürülen verinin ana programda değiştirilmemesi istenebilir. Bunu sağlamak için geriye sabit (const) referanslar döndürülür.

```
// max fonksiyonu dizinin başlangıç adresini ve uzunluğunu alıyor
// en büyük elemana bir sabit referansı geri döndürüyor
const int&  max(int a[], int uzunluk)
{
    int i=0;                                  // i en büyük elemanın indisi olacak
    for (int j= 0 ; j<uzunluk ; j++)          // En büyük eleman aranıyor
            if (a[j] > a[i])  i = j;
    return a[i];                              // Eleman geri döndürülüyor (referans)
}
```

Bu şekilde yazılan fonksiyon atama deyimlerinin sadece sağında yer alabilir.

```
int main()
{
   int buyuk;                                 // En büyük elemanı tutacak değişken
   int dizi[] = {12, -54 , 0 , 123, 63};      // Bir dizi tanımlanıyor, değerler atanıyor
   buyuk = max(dizi,5);                       // En buyuk eleman buyuk'e atandı
   return 0;
}
```

Yerel değişkenler referans ile döndürülemez!

Yerel değişkenler sadece tanımlandıkları bloğun içinde geçerlidirler. Benzer şekilde bir fonksiyonun içinde tanımlanan yerel değişkenler de fonksiyondan çıkıldıktan sonra bellekten kaldırılırlar.

```
int&  f( )                    // Referans ile veri geri döndürülecek
{
      int i;                  // Yerel değişken. Bir çok sistemde yığında yaratılır
      :
      return  i;              // Hata! i'nin adresi artık geçerli değildir.
}
```

Doğru olan yerel değişkenlerin değer olarak döndürülmeleridir.

```
int  f( )                     // Değer döndürülecek
{
      int i;                  // Yerel değişken. Bir çok sistemde yığında yaratılır
      :
      return  i;              // Doğru, çünkü değer döndürülüyor.
}
```

Operatörlere Yeni İşlevlerin Yüklenmesi (*Operator Overloading*)

C++'da, + , - , *, !, ++ gibi operatörlere ilişkin fonksiyonlar yazılarak bu operatörlere yeni işlevler yüklenebilir.

- Sadece C++'da zaten var olan operatörler için fonksiyon yazılabilir. Örneğin C++'da var olmayan '^' operatörü için üs almak üzere bir fonksiyon yazılamaz.
- Operatörlerin orijinalindeki operand sayısı değiştirilemez. Örneğin '+' operatörü için mutlaka iki operandlı (parametreli) bir fonksiyon, '!' operatörü için tek operandlı bir fonksiyon yazmak gerekir.
- Operatörlerin orijinal öncelikleri değiştirilemez. Örneğin yeni bir işlev yüklense bile '*' operatörü her zaman '+' operatöründen daha önceliklidir.

Bir operatöre işlev yükleyen fonksiyonların adı operator özel sözcüğünü içerir. Örneğin '+' operatörüne işlev yükleyecek olan fonksiyonun adı operator+ ve '*' operatörüne işlev yükleyecek olan fonksiyonun adı da operator* olmalıdır.

```
struct ComplexT{                        // Karmaşık sayıları tanımlamak için bir yapı
        float re,im;                    // Reel ve sanal kısımlar
};

// + operatörünün karmaşık sayıları toplamak için yüklenmesi
ComplexT operator+ (const ComplexT &v1, const ComplexT &v2)
{
    ComplexT sonuc;                     // sonucun yazılacağı yerel değişken
    sonuc.re=v1.re+v2.re;               // reel kısımlar toplanıyor
    sonuc.im=v1.im+v2.im;               // sanal kısımlar toplanıyor
    return sonuc;                       // sonuç çağıran programa döndürülüyor
}

int main()
{
    ComplexT c1,c2,c3;                  // Üç adet karmaşık sayı tanımlanıyor
    c1.re= 0.5;                         // Sayıların alanları dolduruluyor
    c1.im= -1;
    c2.re= 1.5;
    c2.im= 0.5;
    c3= c1 + c2;                        // Yukarıdaki operator+ fonksiyonu
   return 0;                            // c3= operator+(c1,c2);  aynı işi yapar
}
```

Bkz. Örnek: o25.cpp