FONKSİYONLAR

Fonksiyon

Yapısal Programlama mantığında çözülecek problem genel olarak fonksiyonların birbirini çağırmasıyla yapıldığı anlatılmıştı. Yani kodlaması yapılacak işi birbirini çağıran fonksiyonlar yazarak ana fonksiyondan bu fonksiyonları çağırarak yaparız.

Ayrıca yazılacak programda birbiriyle ilgili fonksiyonlar bir araya toplanarak ayrı bir dosyada tutulur. Bu durum *Modüler Programlama* başlığı altında anlatılmıştı.

Bu bölüme kadar gördüğümüz kodlarda çözümü oluşturan tüm tasarım parçaları ana fonksiyon olan `main()` fonksiyonu içerisine çözülmüştür. Bir bilgisayar programı, genel olarak, tek başına tüm görevleri yerine getiren tek bir main fonksiyonundan oluşmaz. Bu durumda ana problemin büyüklüğüne göre ana fonksiyon bloğumuz uzar, okunabilirliği azalır, müdahale etmek zorlaşır. Bu tip büyük programları kendi içerisinde, her biri özel bir işi çözmek için tasarlanmış parçacıklarından oluşturmak daha mantıklı olacaktır. Yani böl ve yönet (divide and conquer) tekniği uygulanır. Çünkü büyük bir problemin toptan çözülmesi, problemin parçalar halinde çözülmesinden her zaman daha yorucu ve zordur.

Yazılım geliştirmede problemi büyük ana parçalara ayırırız. Daha sonra bu büyük parçaları daha küçük parçalara ayırarak çözeriz. Buna yukarıdan aşağıya (top down) yaklaşım adı verilir. C++ dilinde fonksiyonlar;

- Fonksiyonlar, belirli bir görevi gerçekleştiren bir kod bloğudur.
- Parametre alabilir, girdilerle ya da girdi olmadan bir şeyler yapar ve ardından cevabı verebilir. Kısaca bilgiyi fonksiyona argümanlar ile aktarırız.
- Her fonksiyonun bir kimliği vardır. (*Değişken Kimliklendirme Kuralları* burada da geçerlidir.)
- Bir fonksiyon program içerisinde istenildiği kadar farklı yerlerden ulaşılarak çalıştırılabilir ya da çağrılabilir. Bir başka deyişle tekrar kullanılabilir.
- Bir fonksiyon, çağıran koda bir değer geri döndürebilir (return).
- Bir fonksiyon, bir başka fonksiyondan çağrılabilir.

C++ dilinde iki tip fonksiyon bulunmaktadır; başlık dosyalarında bize sunulan hazır fonksiyonlar ve programcının tanımlayacağı kullanıcı tanımlı fonksiyonlar (user defined function).

Hazır Fonksiyonlar

C++ geliştiricileri, programcıların kullanmaları için, önceden yazılmış olan hazır fonksiyonlar hazırlamışlardır. Hazır fonksiyonlar teknik olarak C++ dilinin parçası değillerdir. Yalnızca standart hale getirilmişlerdir.

Programcı, kullanmak istediği hazır fonksiyonları; hangi modülde ise o modüle ilişkin başlık (header) dosyasını `#include` ön işlemci yönergesi (preprocessor directive) ile kendi projesine dahil eder. Bunu Modüler Programlama başlığı altında incelemiştik. Bir modül olan bu başlık dosyalarında;

- Hazır fonksiyonların sadece prototipleri (prototype) bulunur.
- Fonksiyonların kendileri yani derlenmiş halleri her işletim sistemi için ayrı oluşturulmuş kütüphane (library) dosyaları içerisinde bulunur.
- Derleme işlemi sırasında bağlayıcı (linker) tarafından bu fonksiyonlar icra edilebilir dosyaya (executable file) dahil edilir.

Cmath Başlık Dosyası

Matematiksel iş ve işlemleri gerçekleştirmek için standart hale getirilmiş bir başlık dosyasıdır. Bu başlık `abs`, `sin`, `cos`, `floor`, `sqrt` gibi birçok matematiksel fonksiyonu barındırır.

Örnek olarak kenarları verilen bir üçgenin alanını hesaplayan bir program aşağıda verilmiştir.

```
#include <iostream>
#include <cmath>
using namespace std;
int main() {
    int kenar1,kenar2,kenar3;
    double alan,kenarlarToplamininYarisi;
    cout << "Üçgenin Kenarlarını Giriniz:";
    cin >> kenar1 >> kenar2 >> kenar3;
    if ((kenar1+kenar2 <= kenar3) ||
       (kenar2+kenar3 <= kenar1) ||
       (kenar1+kenar3 <= kenar2) ) {
        cout << "Böyle bir üçgen olamaz." << endl;
        return 1;
    }
    kenarlarToplamininYarisi=(kenar1+kenar2+kenar3)/2.0;
    alan=sqrt( kenarlarToplamininYarisi*
               (kenarlarToplamininYarisi-kenar1)*
               (kenarlarToplamininYarisi-kenar2)*
               (kenarlarToplamininYarisi-kenar3) );
    cout << "Üçgenin alanı: " << alan ;
}
```

Cstdlib Başlık Dosyası

Bu başlık içerisinde; birçok fonksiyonun yanında rastgele sayı üretme fonksiyonları `rand()` ve `srand()` bulunur.

Aşağıda bir zar için rastgele sayı üreten bir program verilmiştir.

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main() {
    int rastgeleZar;
    rastgeleZar=1 + rand() %6;
    /*
     std::rand() fonksiyonunun üreteceği sayıyı kalan işleci ile
     6 sayısına bmleriz ki 0 ile 5 arasında bir sayı elde edelim.
     Buna da 1 ekler isek zarın rakamları ortaya çıkar.
    */
   cout << "Atılan Zar:" << rastgeleZar;
}
```

Fakat programın her çalışmasında `srand::rand()` aynı başlangıç değerini kullandığından aynı rastgele değerleri üretir. Bu nedenle aynı zar rakamı gelmiş olur. Bunu değiştirmek için `std::srand()` fonksiyonu kullanılır.

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main() {
    unsigned randBaslangicDegeri;
    cout << "std::rand() için başlangıç değeri olabilecek bir sayı giriniz:";
    cin >> randBaslangicDegeri;
    srand(randBaslangicDegeri);
    /*
     std::rand() fonksiyonunun üreteceği sayı her zaman aynı başlangıç değeriyle
     başladığından her program çalıştığında üterilecek ilk rastgele sayı
     ve sonrasındaki sayılar hep olur.
     Bu hesaplamanın bir başka başlangıçla başlaması üterilecek ilk rastgele sayı
     ve sonraski üretilecekleri değiştirir. Bunu değiştirmenin yolu std:srand()
     fonksiyonunu kullanmaktır.
```

```
    */
    int rastgeleZar;
    rastgeleZar=1 + rand() %6;
    cout << "Atılan Zar:" << rastgeleZar;
}
```

Rastgele sayı her seferinde kullanıcıdan alınan sayıya göre hesaplandığından her seferinde farklı bir zar rakamı üretilmiş olacaktır. Ancak başlangıç değerinin her seferinde kullanıcıdan alınması bir başka problemdir. Bunun üstesinden gelmek için bilgisayarın saatini sayı olarak veren `ctime` başlık dosyasındaki `time()` fonksiyonu `nullptr` parametresiyle kullanılabilir.

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
int main() {
    unsigned randBaslangicDegeri=time(nullptr);
    /*
     std:srand() fonksiyonu ile std::rand() fonksiyonu ilk değer verilir.
     Verilecek ilk değer std::time() fonksiyonuna nullptr verilerek sistem zamanından
     gelen değer kullanılır.
     Böylece her çalıştırmada farklı rastgele sayı üretilmesi sağlanacaktır.
    */
    srand(randBaslangicDegeri); // srand(time(nullptr)); olarak da yazılabilir.
    int rastgeleZar;
    rastgeleZar=1 + rand() %6;
    cout << "Atılan Zar:" << rastgeleZar;
}
```

Kullanıcı Tanımlı Fonksiyonlar

Programcılar kendi fonksiyonlarını oluşturarak kodlarını daha kullanışlı hale getirirler. Bu tür fonksiyonlara kullanıcı tanımlı fonksiyonlar (user defined function) denilir. Kullanıcı tanımlı fonksiyonu oluşturmak ve kullanmak için bu üç unsuru bilmemiz gerekir;

- Fonksiyon Bildirimi: Bir fonksiyonu çalıştıran koda, çağıran program adı verilir. Çağıran program kullanılacak fonksiyonu bilmeli. Bunun için çağıran program öncesinde fonksiyon bildirimi (function declaration) yapılmalı veya prototipi (function prototype) tanımlanmalıdır. Bildirim derleyiciye böyle bir fonksiyon var demenin yoludur.
- Fonksiyon Tanımı: Fonksiyon tanımı (function definition), bir fonksiyonun girdileri, yaptığı işlem ve döndüreceği değeri içerecek şekilde başlık ve gövdesinin kodlanmasından oluşur. Bildirimi yapılan fonksiyonun derleme yapılabilmesi için eksiksiz bir şekilde tamamlar. Bildirime benzer başlık ve tırnaklı parantez `{ }` kod bloğu ve arasında yer alan kodlardan oluşur. Bildirim yapılmadan da fonksiyon tanımlanabilir.
- Fonksiyon Çağırma: Fonksiyonu çağırmak (function call/invoke function) için fonksiyonun kimliğini ve ardından parantez içindeki argüman listesini yazmanız yeterlidir.

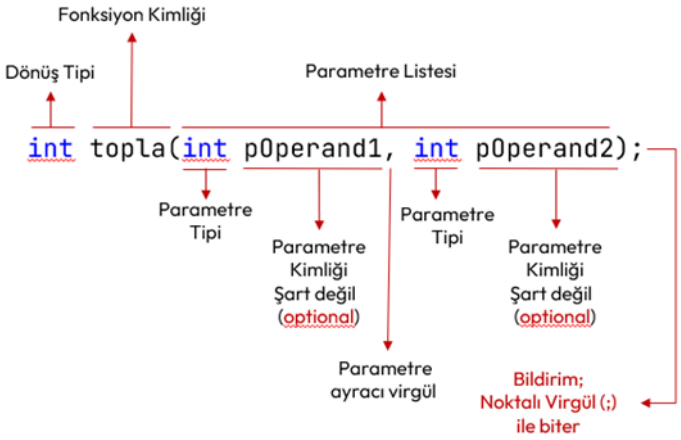
Fonksiyon Bildirimi Nasıl Yapılır?

Fonksiyon bildiriminde (function declaration); Dönüş tipi, işlevin döndürdüğü değerin veri tipidir. Bazı işlevler, herhangi bir değer döndürmeden istenen işlemleri gerçekleştirir. Bu durumda dönüş tipi `void` anahtar sözcüğüdür ve bu fonksiyonlar yordam (procedure) ya da alt rutin (subroutine) olarak adlandırılır.

```
/* Fonksiyon Bildirimleri */
int ikiSayiyiTopla(int,int); /* ikiSayiyiTopla kimlikli bir fonksiyon olduğu ve
                                bu fonksiyonun iki tamsayı parametre aldığı ve
                                bir tamsayı geri döndürdüğü derleyiciye bildiriliyor. */
void sayiyiKonsolaYaz(int);  /* sayiyiKonsolaYaz kimlikli bir fonksiyon olduğu ve
                                bu fonksiyonun bir tamsayı parametre aldığı ve
```

```
                                geri değer döndürmediği derleyiciye bildiriliyor. */
int konsoldanTamsayiOku();   /* konsoldanTamsayiOku kimlikli bir fonksiyon olduğu ve
                                bu fonksiyonun bir parametre almadığı ve
                                bir tamsayı geri döndürdüğü derleyiciye bildiriliyor. */
int main() {
   sayiyiKonsolaYaz(13);
   int toplam= ikiSayiyiTopla (10,20);
   //...
}
```

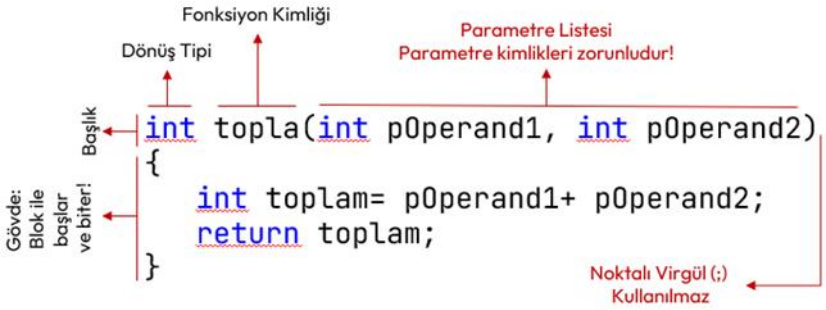
Fonksiyon adı, fonksiyonun benzersiz kimliğidir ve kimliklendirme, değişken kimliklendirme kuralları burada da geçerlidir. Bağımsız değişkenler parametre olarak adlandırılır. Bir fonksiyon çağrıldığı anda parametreye gerçek parametre ya da argüman adı verilir. Parametre listesi, bir işlevin parametrelerinin veri tipini, sırasını ve sayısını belirtir. Parametreler isteğe bağlıdır; yani bir fonksiyon hiçbir parametre içermeyebilir.



Şekil 18. Fonksiyon Bildirimi Örneği

Fonksiyona ilişkin bir bildirim yapıldığında bir yerlerde böyle bir fonksiyon olduğu derleyiciye iletilmiş olur. Bildirim sonrasında artık onu çağırabiliriz. Tanımı daha sonradan yapılabilir. Bildirim yapılmaz ise fonksiyon tanımlanmalıdır.

Fonksiyon Tanımı Nasıl Yapılır?



Şekil 19. Fonksiyon Tanım Örneği

Bir fonksiyon bir değer döndürebilir. Dönüş tipi, işlevin döndürdüğü değerin veri tipidir.

Fonksiyon adı, fonksiyonun benzersiz kimliğidir ve kimliklendirmede, değişken kimliklendirme kuralları geçerlidir. Eğer daha önce fonksiyon bildirimi (function declaration) yapılmış ise aynı kimlik kullanılır.

Bağımsız değişkenler parametre olarak adlandırılır. Parametre listesi, bir işlevin parametrelerinin veri tipini, sırasını ve sayısını belirtir. Parametreler isteğe bağlıdır; yani bir fonksiyon hiçbir parametre içermeyebilir. Bildirimden farklı olarak her bir parametreye kimlik verilmelidir.

Fonksiyon Gövdesi, fonksiyonun ne yaptığını tanımlayan talimatları içerir. Yapısal programlama yapıldığından ilk önce değişkenler tanımlanmalı daha sonra bu değişkenleri işleyecek talimatlar yazılmalıdır. Fonksiyonda istenilen sonuca ulaşmak için kullanacağı adımlara karar verilir yani algoritması belirlenir.

Dönüş tipine uygun olarak bir değer, return talimatı ile geri döndürülmelidir. Return talimatı çoğunlukla fonksiyonundaki son talimat olarak görünür.

Return talimatı, bir fonksiyonun yürütülmesini sonlandırır ve kontrolü çağrıldığı fonksiyona geri verir. Yani programın icrasına devam etmek için çağrıldığı yere dönülür. Bir fonksiyonun tanımında yer alan dönüş tipi (örnekte `int`) ile `return` talimatına ilişkin ifade sonucu (örnekte `toplam`) aynı veri tipinde olmalıdır.

Fonksiyonun çağrıldığı yerde, fonksiyonun döndürdüğü değerin tipi ile eşleşen bir değişkene dönüş değeri atanır.

Bildirimi yapılan bir fonksiyon, C++ derleyicisi tarafından dahil edilen başlık dosyalarını içerecek şekilde derlemeye konu olacak dosyaların tamamında yalnızca bir kez tanımlanabilir. Buna tek tanım kuralına (one definition rule) adı verilir.

```
int topla(int a,int b) {
    return a+b;
}
int topla(int x,int y) { // error: redefinition of ‘int topla(int, int)’
    return x+y;
}
```

Kullanıcı Tanımlı Fonksiyon Örnekleri

Aşağıda topla adlı bir kullanıcı tanımlı fonksiyona ilişkin program verilmiştir.

```
#include <iostream>
using namespace std;
int topla(int,int); // kullanıcı tanımlı "topla" kimlikli fonksiyon bildirimi
int main () {
   int sonuc;
   /* ÇAĞRI ORTAMI: main fonksiyonu içinde "topla" kimlikli
      Fonksiyonu çağrılacak. */
   topla(1,2); /* topla fonksiyonu çağrılıyor.
                  Ama geri döndürdüğü değer kullanılmıyor. */
   cout << "1. toplam:" <<  topla(2,3) << endl; /* topla fonksiyonu çağrılıyor.
                                                   Geri döndürdüğü değer printf
                                                   fonksiyonuna argüman olarak
                                                   giriyor. */
   sonuc= topla(3,4);  /* topla fonksiyonu çağrılıyor.
                          Geri döndürdüğü değer sonuc değişkenine atanıyor. */
   cout << "2. toplam:" << sonuc << endl;
   sonuc= topla(1,2)+topla(3,4); /* topla fonksiyonu iki kez çağrılıyor.
                                    Her bir çağrıdaki geri dönüş değeri
                                    gecici bir yerde saklanıp toplamı
                                    sonuc değişkenine atanıyor. */
   cout << "3. toplam:" << sonuc << endl;
   sonuc= topla(topla(1,2),3)+topla(4,5); /* ... */
   cout << "4. toplam: " << sonuc << endl;
}

// kullanıcı tanımlı topla fonksiyonu tanımı (definition)
int topla(int pOperand1,int pOperand2) //Fonksiyon başlığı
{ //Fonksiyon bloğu başlangıcı
   int toplam=pOperand1+ pOperand2;
   return pOperand1+ pOperand2;
} //Fonksiyon bloğu bitişi
```

```
/* Program çalıştırıldığında:
toplam: 5
toplam: 7
toplam: 10
toplam: 15

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

Aşağıda pozitif sayı girilmesini garanti ettikten sonra sayının 10 sayısına bölünebilirliğini bulan fonksiyon içeren program verilmiştir.

```
#include <iostream>
using namespace std;
int pozitifSayiOku(); // parametre almayan bir fonksiyon bildirimi.
int onaBolunurMu(int);
// int veri titpinde parametre alan bir başka fonksiyon bildirimi
int main () {
    int birPozitifSayi=pozitifSayiOku();
    if (onaBolunurMu(birPozitifSayi))
        cout << "Girilen Pozitif Sayı 10 ile bölünebilir.";
    else
        cout << "Girilen Pozitif Sayı 10 ile tam bölünmez.";
}
int pozitifSayiOku() {
    int okunan;
    do {
        cout << "Lütfen pozitif sayı giriniz: ";
        cin >> okunan;
        if (okunan<=0)
            cout << "HATA:Pozitif Sayı GİRMEDİNİZ!" << endl;
    } while (okunan <= 0);
    return okunan;
}
int onaBolunurMu(int p){
    return (p%10)==0;
}
```

Örnek programda görüldüğü üzere yapısal programa kapsamında bir programın iskeleti oluşmuştur. Ana program içinde problemin çözümü, çeşitli fonksiyonların birbirini çağırması ile yapılmıştır. Her bir fonksiyon içinde işlenecek veriye ilişkin değişkenler (en basit veri yapısı) tanımlanmış ve bu veriler ile bu veri yapılarını işleyen kontrol işlemleri birbirinden ayrılmıştır. Okunaklılık çok yüksek olan bir kod elde edilmiştir.

Bunların dışında hiçbir değer geri döndürmeyen fonksiyonlar da tanımlayabiliriz. Bunlara diğer dillerde prosedür (procedure) adı da verilir. Bu durumda değeri olmayan veri tipi olan `void` anahtar kelimesi kullanılır. Bu tür fonksiyonların da geri dönüş talimatı yalnızca `return;` şekilde yazılır.

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

void printMenu(); // geri değer döndürmeyen bir fonksiyon bildirimi.

int main() {
    int secim;
    do {
        printMenu();
        cin >> secim;
        switch(secim) {
            case 1:
                cout << "Verileri Sakladım." << endl;
                break;
```

```
            case 2:
                cout << "Verileri Okudum." << endl;
                break;
            case 0:
                cout << "Programdan Çıkılıyor..." << endl;
                break;
            default:
                cout << "Tekrar Seçim Yapınız!" << endl;
        }
    } while (secim!=0);
}

void printMenu() {
    cout << endl;
    cout << "1-Verileri Yaz." << endl;
    cout << "2-Verileri Oku." << endl;
    cout << "0-ÇIKIŞ." << endl;
    return;
}
```

Çağrı Kuralı

Çağrı kuralı (calling convention), bir fonksiyon çağrısıyla karşılaşıldığında argümanların fonksiyona hangi sıraya göre iletileceğini belirtir. İki olasılık vardır; Birincisi argümanlar soldan başlanarak sağa doğru fonksiyona geçirilir. İkincisi ise C dilinde kullanılır ve argümanlar sağdan başlanarak sola doğru fonksiyona geçirilir.

```
#include <iostream>
using namespace std;
int topla(int, int, int);
int main () {
    int toplam;
    toplam=topla(10,20,30);
    cout << "toplam:" << toplam << endl;
}
int topla(int x, int y, int z) {
    cout << "x:"<< x << " y:" << y  << " z:" << z<< endl;
    return x+y+z;
}
/* Program çalıştırıldığında:
x:10 y:20 z:30
toplam:60
*/
```

Yukarıdaki örnek incelendiğinde argümanların hangi sırada fonksiyona geçirildiğinin bir önemi yoktur. Ancak aşağıdaki verilen örnekteki durumu inceleyelim;

```
#include <iostream>
using namespace std;
int topla(int, int, int);
int main () {
    int a = 1,toplam;
    toplam = topla(a, ++a, a++);
    cout << "toplam:" << toplam << endl;
}
int topla(int x, int y, int z) {
    cout << "x:"<< x << " y:" << y  << " z:" << z<< endl;
    return x+y+z;
}
/* Program çalıştırıldığında:
x:3 y:3 z:1
```

```
toplam:7
*/
```

Bu kodun çıktısı `1 2 3` olarak beklenir ancak çağrı kuralından ötürü çıktı `3 3 1` olur. Bunun nedeni C++ dilinin çağrı kuralının sağdan sola olmasıdır. Bunu şöyle açıklayabiliriz;

1. Fonksiyona sağdan ilk argüman olarak a değişkeni değeri 1 geçirilir.
2. Ardından a++ ifadesi ile a değişkeni 2'ye yükseltilir.
3. Sonra ++a ifadesi ile a değişkeni 3'e yükseltilir.
4. Fonksiyona ikinci argüman olarak 3 geçirilir.
5. Fonksiyona son olarak a'nın son değeri olan 3 geçirilir.

Depolama Sınıfları

Depolama sınıfları (storage classes) bir değişkenin ömrünü (lifetime), görünürlüğünü (visibility), bellek konumunu (memory location) ve başlangıç değerini (initial value) belirlemek için kullanılır. Bu özellikler temel olarak bir programın çalışma süresi boyunca belirli bir değişkenin varlığını izlememize yardımcı olan kapsam (scope), görünürlüğü ve yaşam süresini içerir.

Yapısal programlama ve fonksiyon kavramının bilgisayarlarda kullanılmasıyla birlikte işlemciler de de değişiklikler olmuştur. *İşlemcinin Emri Alma, Çözme ve İcra Döngüsü* başlığı altında belirtilen işlemci kaydediciler dışında yeni kaydediciler de eklenmiştir;

– Verilerle icra edilen kod farklı bellek bölgesinde bulunur. Bu nedenle verileri işaret eden veri bellek adreslerini tutan kaydedicisi (data memory register).
– Yığın bellek kaydedicisi (stack register), fonksiyon çağrıları sırasında mevcut işlemci durumu ve yerel değişkenler gibi bazı işlemleri depolamak için kullanılan belleği gösteren (stack pointer) adresi tutan kaydedicidir.
– Bayrak kaydedicisi (flag register), işlemcinin durumunu veya sıfır (zero) bayrağı, taşıma (carry) bayrağı, işaret (sign) bayrağı, taşma (overflow) bayrağı, eşlik (parity) bayrağı, yardımcı taşıma (auxiliary carry) bayrağı ve kesme etkinleştirme (interrupt enable) bayrağı gibi çeşitli işlemlerin sonucunu tutmak için kullanılan özel bir kayıt türüdür.
– Genel amaçla kullanılan kaydediciler (general purpose registers).

Eklenen bu kaydediciler ile derlenen her bir program dört bellek bölgesinden (segment) oluşur;

1. Kod bellek ya da kaynak koda ithafla metin bellek (code segment / text segment) icra edilecek emirleri içeren bellek.
2. Programın işleyeceği verileri tutan veri belleği (data segment).
3. Fonksiyonların çağırmadan (call) önce mevcut işlemci durumu ve yerel değişkenler gibi bazı işlemleri depolamak için ve fonksiyondan dönülünce (return) işlemciyi ve çağrı ortamını eski hale getirmek için kullanılan yığın bellek (stack segment). Bu bellek bu nedenle son giren veri ilk çıkar (last in first out-LIFO) mantığıyla çalışır.
4. Programın icrası sırasında dinamik olarak eklenip çıkarılan veriler için hurdalık gibi kullanılan öbek bellek (heap segment). İleride *Dinamik Hafıza* başlığında anlatılacaktır.

Depolama sınıfları tanımlanan değişkenleri hangi bellek bölgesinde yer alacağını belirler. Beş tür depolama sınıfı vardır; `auto`, `extern`, `static`, `register` ve `mutable`. Mutable depolama sınıfı nesnelerle ilgili olduğundan *Mutable Depolama Sınıfı* başlığında anlatılacaktır.

Auto Depolama Sınıfı

Otomatik (auto) depolama sınıfı, bir fonksiyon veya blok içinde kimliklendirilmiş tüm değişkenler için ön tanımlı (default) depolama sınıfıdır. C++ dilinde bu depolama sınıfı için değişkenlerde hiçbir sıfat kullanılmaz. Otomatik değişkenlere; Yalnızca blok içinden erişilebilir, Değişkenin kapsamı içinde bulunduğu blok ile sınırlıdır ve yığın bellekte (stack segment) tutulur.

Otomatik değişkenlere ilişkin örnek aşağıda verilmiştir;

```
#include <iostream>
```

```
using namespace std;
int main() {
    int a = 32;
    int b=20;
    /* a ve b değişkenleri bu fonksiyon bloğu ve iç bloklarda geçerlidir.
       Yani faaliyet alanları (scope) bu fonksiyon bloğu ile sınırlıdır.*/
    if (a==32){
        int b=10; // Ana fonksiyon bloğunda tanımlı b değişkenine artık ulaşılamaz.
        int c=50;
        /* if bloğunda ve tanımlanabilecek iç bloklarda geçerlidir. */
        a=16;// Bu blokta a da geçerlidir.
        cout << "a:" << a << " b:" << b << " c:" << c << endl;
             // Çıktı: a:16, b:10 c:50
   }
   /*if bloğu dışında sadece ana fonksiyon bloğunda
     tanımlı değişkenlere ulaşılabilir. */
   cout << "a:" << a << " b:" << b; // Çıktı: a:16, b:20
}
```

Static Depolama Sınıfı

Statik (static) depolama sınıfı yaygın olarak kullanılan statik değişkenleri kimliklendirmek için kullanılır. Statik değişkenler, kapsam (scope) dışına çıktıktan sonra bile değerlerini koruma özelliğine sahiptir! Statik değişkenler;

- Kapsamları yani geçerli olduğu yer, tanımlandıkları fonksiyonla ya da blokla sınırlıdır.
- Kapsamlarındaki son kullanımlarının değerini korur.
- Program genelinde geçerli olan evrensel (global) statik değişkenlere programın herhangi bir yerinden erişilebilir.
- Veri bellekte (data segment) yer alır. Derleme sırasında bu bellek hep sıfırla doldurulduğundan ilk değerler (initial value) hep sıfırdır.

Statik (static) olarak tanımlanan değişken, program başladığında veri bellekte oluşturulup program bitene kadar aynı bellek bölgesini kullanan değişkendir.

```
#include <iostream>
using namespace std;
void func() {
    int sayac = 0;
    for (sayac = 1; sayac < 10; sayac++)
    {
        static int statikOlan = 5;
        int statikOlmayan = 10;
        statikOlan++;
        statikOlmayan++;
        cout << "sayaç: " << sayac << ", staticOlan: " << statikOlan << endl ;
        cout << "sayaç: " << sayac << ", statikOlmayan: " << statikOlmayan << endl;
   }
   //Burada statikOlan değişkene erişilemez.
}
int main() {
    func();
}
/* Program çalıştırıldığında:
Sayaç: 1, staticOlan: 6
sayaç: 1, statikOlmayan: 11
sayaç: 2, staticOlan: 7
sayaç: 2, statikOlmayan: 11
sayaç: 3, staticOlan: 8
sayaç: 3, statikOlmayan: 11
sayaç: 4, staticOlan: 9
```

```
sayaç: 4, statikOlmayan: 11

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

Programı çalıştığında `statikOlan` değişken yalnızca `for` bloğu içinde geçerli olduğu ve her kullanımda değerini koruduğu görülmektedir.

Harici Depolama Sınıfı

Harici (extern) depolama sınıfı bize basitçe değişkenin kullanıldığı blokta değil başka bir yerde tanımlandığını söyler. Harici değişkenler;

- Bu değişkenlere değerler tanımlandığı yerde değil farklı bir blokta atanır ve üzerinde değişiklik yapılabilir.
- Bir global değişken, `extern` anahtar sözcüğünü kimliklendirme önüne yerleştirerek harici de yapılabilir.
- Bu değişkenler de veri bellekte (data segment) yer alır. Derleme sırasında bu bellek hep sıfırla doldurulduğundan ilk değerler hep sıfırdır.

Bir harici (extern) değişkene bildirim (declaration) yapıldığında bir başka dosyada tanımlanmış (definition) yapıldığı kabul edilir. Bunun için iki farklı kaynak kod dosyası gereklidir. Birinci kaynak kod dosyası (`extern.c`) aşağıda verilmiştir.

```
//EXTERN.CPP
int externDegisken; //Diğer dosyalarda kullanılacak değişken tanımlandı.
void funcExtern() //Diğer dosyalarda kullanılacak fonksiyon tanımlandı.
{
    externDegisken++;
}
```

Bu dosyadaki değişken ve fonksiyonları kullanan bir başka kaynak kod (`main.cpp`) aşağıda verilmiştir;

```
#include <iostream>
using namespace std;
extern void funcExtern(); // bir başka dosyada tanımlandı. Buada bildirim yapıldı.
int main() {
   extern int externDegisken; // bir başka dosyada tanımlandı. Burada bildirim yapıldı.
   funcExtern();
   cout << "extern: " << externDegisken << endl;
   externDegisken = 2;
   cout << "extern: " << externDegisken << endl;
}
```

Bu iki dosyayı birlikte derlemek için `g++ extern.cpp main.cpp -o main.exe` komutu ile derleyiciye iki farklı dosya parametre olarak verilir. Çevrimiçi derleyicimizde ise bir projeye birden çok kaynak kod işaretli butona basılarak eklenebilir ve derleme yapılabilir.

Kaydedici Depolama Sınıfı

Kaydedici (register) depolama, otomatik değişkenlerle aynı işlevselliğe sahiptir. Tek fark, derleyicinin, eğer boş bir işlemci kaydedicisi mevcutsa değişken olarak o kaydedicinin kullanılmasıdır. Genel amaçlı kaydediciler bu amaçla kullanılır. Bu da bu değişkenlerin, bellekte saklanan değişkenlerden çok daha hızlı olmasını sağlar.

Eğer boş bir CPU kaydedicisi mevcut değilse, değişken yalnızca bellekte saklanır. `register` anahtar sözcüğüyle tanımlanan değişken nitelendirilir. `register` Anahtar sözcüğü ile sadece bloklar içinde tanımlanan yerel değişkenler (local variable) nitelendirilebilir, evrensel değişkenler (global variable) bu anahtar sözcük ile kullanılamaz.

Aşağıda kaydedici (register) depolama sınıfın ilişkin örnek verilmiştir. Kuradaki sayma ve toplama işlemleri uygun olan işlemci kaydedicilerinde yapılır.

```
#include <iostream>
using namespace std;
int main() {
    register int k, sum;
    for(k = 1, sum = 0; k < 1000; sum += k, k++);
        //döngü bloğu olmayan for talimatı örneği
        cout << sum << endl ; //Çıktı:499500
}
```

Öncelikle C++ standardı tek bir değişkenle birden fazla depolama sınıfının kullanılmasını yasaklar. Aynı değişkene hem `static` hem `extern` niteleyicisi kullanılamaz. Tablodaki çöp (garbage) kavramı tahsis edilen bellek içeriği o anda ne ise değişken o değeri alır anlamındadır. Depolama sınıfları aşağıdaki tabloda özetlenmiştir;

Depolama sıfatı	Yer aldığı bellek bölgesi (segment)	Başlangıç Değeri	Kapsamı (scope)	Ömür (lifetime)
Sıfat kullanılmaz!	Yığın Bellek (Stack Segment)	Çöp	Bulunduğu blok ve bu blok içindeki bloklar	Bulunduğu blok sonunda biter
`extern`	Veri Bellek (Data Segment)	Sıfır	Evrensel (tüm dosyalarda)	Program sonunda biter
`static`	Veri Bellek (Data Segment)	Sıfır	İçinde bulunduğu blok	Program sonunda biter
`register`	İşlemci kaydedicileri (CPU Registers)	Çöp	Bulunduğu blok ve bu blok içindeki bloklar.	Bulunduğu blok sonunda biter

Tablo 18. Depolama Sınıfları Özet Tablosu

Evrensel ve Yerel Değişken

C++ dilinde bir kaynak koda ait dosya içinde bir değişkene her yerden erişilmek istenirse evrensel değişken (global variable) olarak tanımlanır. Bu değişkenler bir fonksiyon bloğu içinde tanımlanmazlar. Tanımlandıkları yerden sonra kaynak kodun her yerinde kullanılabilirler. Bu nedenle genellikle dosyanın başında tanımlama yapılır.

```
#include <iostream>
using namespace std;
int evrenselDegisken=10; /* evrenselDeğişken bu noktadan sonra
                            her yerde kullanılabilir. */
void fonksiyon();
int main() {
    int yerelDegisken=20; /* Bu yerel değişken sadece main bloğu
                             içinde kullanılabilir.*/
    cout << "evrensel:" << evrenselDegisken << ", yerel: "
         << yerelDegisken << endl;
    evrenselDegisken++; //evrensel değişken değiştiriliyor.
    yerelDegisken--; //yerel değişken değiştiriliyor.
    cout << "evrensel:" << evrenselDegisken << ", yerel:"
         << yerelDegisken << endl;
    fonksiyon();
}
void fonksiyon() {
   int fonksiyonYerelDegiskeni=30;
   evrenselDegisken++;
   cout << "evrensel:"<< evrenselDegisken << ", fonsiyonYerel:"
        << fonksiyonYerelDegiskeni << endl;
   if (fonksiyonYerelDegiskeni==30) {
       int evrenselDegisken=100; //Artık evrensel değişkene bu blokta ulaşılamaz.
       cout << "evrensel:"<< evrenselDegisken << ", fonsiyonYerel:"
            << fonksiyonYerelDegiskeni << endl;
   }
}
```

```
/* Program Çıktısı:
evrensel:10, yerel:20
evrensel:11, yerel:19
evrensel:12, fonsiyonYerel:30
evrensel:100, fonsiyonYerel:30

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

Bu program incelendiğinde if bloğu içindeki değişken her ne kadar `evrenselDegişken` olarak kimliklendirilse de yerel değişkendir ve sadece bu if bloğu içinde geçerlidir. Yani tanımlamada mantıksal hata (logical error) yapılmıştır.

Yerel değişkenler (local variable) ise bir fonksiyon bloğu ya da bir kod bloğu içinde tanımlanan değişkenlerdir. Bu değişkenlere evrensel değişkenler ile aynı kimlik verilirse yukarıdaki örnekteki gibi evrensel değişkene artık ulaşılamaz. Böyle bir durumda evrensel değişkene ulaşmak için iki tane iki nokta üst üste karakteri olan kapsam çözümleme işleci (scope resolution operator) kullanılır.

```
#include<iostream>
using namespace std;
int ogrenciSayisi = 5; // evrensel değişken
void ogrenciSayisiniYaz() {
    cout<< ogrenciSayisi << endl; // evrensel değişken konsola yazılıyor.
}
int main() {
   int ogrenciSayisi=10; // yerel değişkene evrensel değişkenin kimliği veriliyor
   ogrenciSayisiniYaz();
   ogrenciSayisi++; // yerel değişken değiştiriliyor.
   ::ogrenciSayisi++; // evrensel değişken değiştiriliyor.
   ogrenciSayisiniYaz();
}
```

Fonksiyon Çağırma Süreci

Bir fonksiyon çağrıldığında (call function/invoke function) nelerin yapıldığı aşağıdaki programdan anlaşılabilir;

```
#include <iostream>
using namespace std;
int topla(int, int); /* İki tamsayıyı toplayan fonksiyon bildirimi/prototipi */
int main () {
    /* main() fonksiyonu içinde geçerli yerel (local) değişkenler*/
    int a = 10;
    int b = 20;
    int toplam = 0;
    cout << "çağrı öncesi a:" << a << ", b:" << b << ", toplam:" << toplam << endl;
    toplam = topla(a, b);
    cout << "çağrı sonrası a:"<< a << ", b:" <<b << ", toplam:" << toplam << endl;
}
int topla(int x, int y) { /* iki tamsayıyı toplayan fonksiyon tanımı*/
    /* Topla fonksiyonu yerel değişkenler */
    int temp= x+y;
    x=100; y=200;
    /*
      Burada değiştirilenler parametre değişkenleridir.
      main() içindeki yerel değişkenler değildir.
    */
    cout << "çağrı içinde x:" << x << ", y:" << y  << endl;
    return temp;
}
```

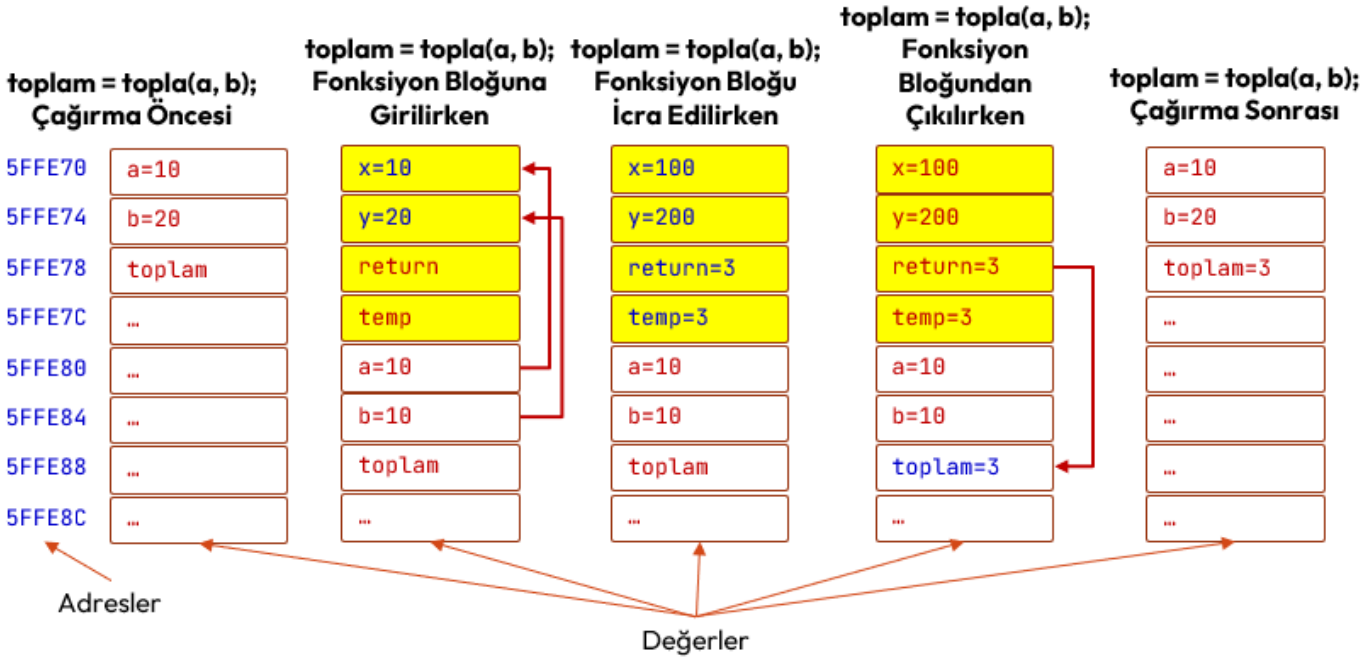
```
/* Program çalıştırıldığında:
çağrı öncesi a:10, b:20 toplam:0
çağrı içinde x:100, y:200
çağrı sonrası a:10, b:20 toplam:30

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

Yukarıdaki kod incelendiğinde;

- İcra sırası `topla(a,b)` fonksiyonuna geldiğinde fonksiyon çağrılmadan önce main fonksiyonu yerel değişkenleri a, b ve toplam değişkenleri yığın (stack) belleğe itilir (push).
- `topla(a,b)` fonksiyonu çağrıldığı anda fonksiyon bloğuna başlamadan parametrelere sırasıyla a ve b değişkenlerinin değerleri kopyalanarak fonksiyona geçirilir.
- `topla(a,b)` fonksiyonu icra edilirken fonksiyon yerelindeki `temp` ve parametre değişkenleri `x`, ve `y` istenildiği gibi değiştirilebilir. Geri dönüş değeri de belirlenir.
- `toplam=topla(a,b)` fonksiyon bloğundan çıkılırken geri dönüş değeri çağrı ortamındaki toplam değişkenine atanır ve fonksiyon için yığın (stack) belleğe itilen değerler geri çekilir.

Aşağıda örneği verilen programda fonksiyon çağırma sürecindeki yığın bellek (stack segment) değişimi gösterilmektedir.



Şekil 20. Fonksiyon çağırma sürecinde yığın bellek

Özyinelemeli Fonksiyonlar

Özyineleme (recursion), bir fonksiyonun kendi çağrısını yapma tekniğidir. Bu teknik, karmaşık sorunları çözülmesi daha kolay basit sorunlara ayırmanın bir yolunu sağlar. Özyineleme, yineleme (iteration), döngüler (loop) veya tekrar yerine geçebilecek çok güçlü bir tekniktir. Özyinelemeli (recursive) algoritmalarda, tekrarlar fonksiyonun kendi kendisini kopyalayarak çağırması ile elde edilir. Bu kopyalar işlerini bitirdikçe kaybolur. Bir problemi özyineleme ile çözmek için problem iki ana parçaya ayrılır.

1. Cevabı kesin olarak bildiğimiz temel durum (base case)
2. Cevabı bilinmeyen ancak cevabı yine problemin kendisi kullanılarak bulunabilecek durum.

Faktöriyel örneğini inceleyelim; Matematikte, negatif olmayan bir tam sayı olan n'nin faktöriyeli, $n!$ ile gösterilir ve n 'den küçük veya ona eşit olan tüm pozitif tam sayıların çarpımıdır.

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$$

Negatif olmayan bir tam sayı olan n'nin faktöriyeli aynı zamanda n'nin bir sonraki daha küçük faktöriyelle çarpımına eşittir. Yani problemin tanımı yine kendisini içerir:

$$n! = n \cdot (n-1)!$$

Örneğin;

$$5! = 5 \cdot 4! = 5 \cdot 4 \cdot 3! = 5 \cdot 4 \cdot 3 \cdot 2! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$$

Aşağıda faktöriyel için girilen sayıdan sonra tüm sayıların çarpımıyla hesaplayan bir fonksiyon yazılmıştır.

```
#include <iostream>
using namespace std;
unsigned int faktoriyel(unsigned int n) { //Fonksiyon tanımı yapıldı. Bildirim yapılmadı.
    int sonuc=1,indis;
    for (indis=n;indis>0;indis--)
        sonuc=sonuc*indis;
    return sonuc;
}
int main() {
    unsigned int sayi,sonuc; //Pozitif tamsayı yanımı yapılmış değişkenler.
    cout << "Pozitif Bir Sayi Giriniz:";
    cin >> sayi;
    sonuc=faktoriyel(sayi);
    cout << sayi << "!=" << sonuc;
}
```

Sıfır sayısı da pozitif bir sayıdır ama sonucum sıfır çıkmaması için 0 sayısının faktöriyeli 1 kabul edilir. Bu aslında faktöriyel problemi için temel durumdur (base case) ve boş ürün (emty product) kuralı olarak bilinir. Kısaca faktöriyel fonksiyonu aşağıdaki şekilde gösterilir;

$$n \in \mathbb{N} \; olmak \; üzere \; f(n) = n! = \begin{cases} 1, & n = 0 \\ n \cdot f(n-1), & n > 0 \end{cases}$$

Özyinelemeli (recursive) olarak ise bu fonksiyon aşağıdaki şekilde kodlanabilir;

```
unsigned int faktoriyel(unsigned int n)
{
    if (n==0)
        return 1;
    else
        return n*faktoriyel(n-1);
}
```

Girilen taban ve üs ile kuvvet hesaplama fonksiyonu özyinelemeli olarak aşağıda verilmiştir;

```
#include <iostream>
using namespace std;
int Kuvvet(int taban, int us) {
    if (taban == 0) { return 0; } //Taban 0 ise 0 dön
    if (us == 0) { return 1; } // Üs 0 ise 1 dön
    return taban* Kuvvet(taban, us - 1);
}
int main(){
    int taban,us;
    cout << "Tabanı Giriniz:";
    cin >> taban;
    cout << "Üssü Giriniz:";
    cin >> us;
    cout << taban <<" uzeri "<< us <<" = " << Kuvvet(taban,us);
}
/*
Tabanı Giriniz:3
Üssü Giriniz:7
```

```
3 uzeri 7 = 2187

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

Satır İçi Fonksiyonlar

Fonksiyon Çağırma Süreci başlığında anlatıldığı üzere fonksiyonlar çağrılırken sürekli yığın belleğe (stack segment) it-çek yapılır. Bazı durumlarda bu işlem performans gereği istenmez. Bu durumda fonksiyonlar, satır içi fonksiyon (inline function) olarak tanımlanır. 2017 C sürümü ile gelen bu özellik, fonksiyona inline sıfatı (inline specifier) kullanılır.

```
#include <iostream>
using namespace std;
inline int topla (int op1, int op2) {
    int toplam= op1+op2;
    return toplam;
}
int main() {
    int x=10, y=15, sonuc;
    sonuc = topla(x,y);
    cout << x << "+" << y << "=" << sonuc;
}
```

Bu kod aşağıdaki şekle çevrilmiş gibi derlenir;

```
#include <stdio.h>
int main() {
    int x=10, y=15, sonuc;
    {
        int toplam= op1+op2;
        sonuc = toplam;
    }
    cout << x << "+" << y << "=" << sonuc;
}
```

Bir fonksiyon aşağıda belirtilen durumlarda satır içi fonksiyon olarak derlenmez;

- Bir döngü içeriyorsa.
- Statik değişkenler içeriyorsa.
- Özyinelemeli ise.
- Dönüş türü `void` haricinde olup return ifadesi işlev gövdesinde mevcut değilse.
- Switch veya `goto` talimatı içeriyorsa.

Bu tip fonksiyon tanımlamadaki amacımız, fonksiyon çağırma yükünüm azaltılması için işlemci kaydedicilerinin daha çok kullanılmasıdır. Bu da icra hızını yükseltir.

Varsayılan Argümanlar

Varsayılan argüman (default argument), bir fonksiyon bildiriminde bir parametre için sağlanan ve çağıran fonksiyon bu parametreler için bir değer sağlamazsa derleyici tarafından otomatik olarak atanan bir değerdir. Bu parametreye çağrı sırasında bir değer geçirilirse, varsayılan değer dikkate alınmaz.

```
#include <iostream>
using namespace std;
void yaz(int a = 10) { //a parametresi için varsayılan argüman 10 dur.
    cout << a << endl;
}
int main() {
    yaz(); // 10
    yaz(200); //200
```

```
}
```

Varsayılan argüman bildirimde farklı, tanımda farklı verilemez;

```
#include <iostream>
using namespace std;
void yaz(int a = 10); //a parametresi için varsayılan argüman 10 dur.
int main() {
    yaz(); // 10
    yaz(200); //200
}
void yaz(int a = 20); // Derleme hatası olur. 10 olarak verilmelidir.
{
    cout << a << endl;
}
```

Birden fazla parametreye varsayılan argüman atanacak ise sağdan sola doğru hepsine atanmalıdır;

```
void yaz(int x, int y = 20); // Geçerli Bildirim.
void yaz(int x = 10, int y); /* Derleme hatası olur. GEÇERSİZ Bildirim:
                                sağdaki y parametresine varsayılan
                                argüman atanmadı */
```