



++ Eğitimi



Akademi: **DEPAR** AKADEMİ

<https://www.deparakademi.com.tr/>

Eğitmen: **Bülent Çobanoğlu** (Bülend Hoca)

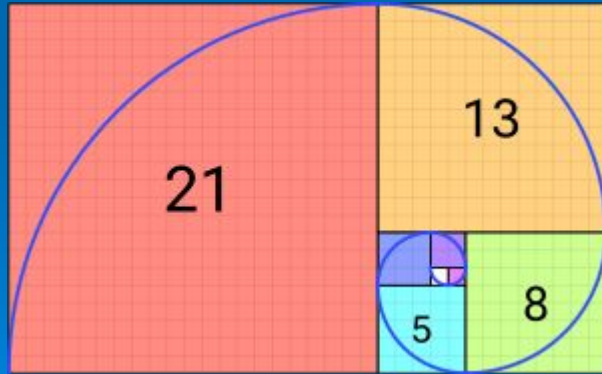
/ ++ Eğitimi-4

- –: Özyinelemeli Fibonacci Serisi
- –: C++ for each döngü yapısı
- –: enum sabiti
- --: Değişken ve Fonksiyonların Faaliyet Alanı
- --: static belirteci
- –: Yer belirteçleri
- –: Başlık (.H Uzantılı) Dosyaları (Header File) Oluşturma
- –: Hata Yakalama (Exception)
- --: C++ Template Fonksiyonları
- --: Fonksiyon Parametrelerinde Pointer Kullanımı

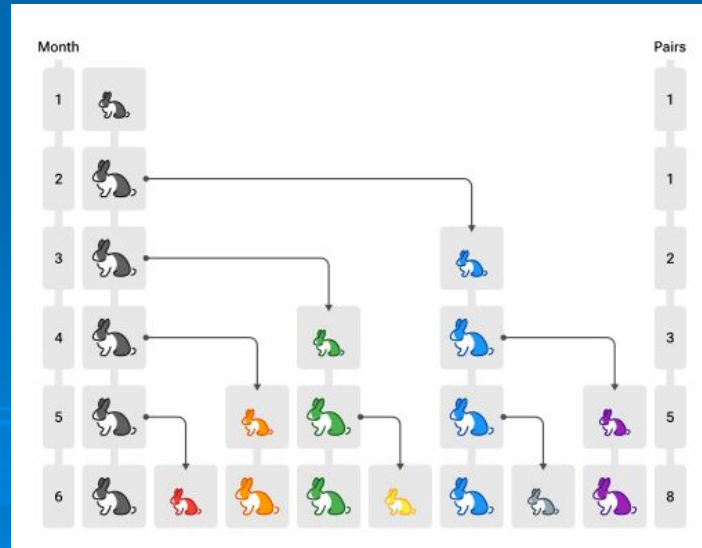
Fibonacci Serisi

The first 21 Fibonacci numbers F_n are:[1]

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}	F_{18}	F_{19}	F_{20}
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584	4181	6765



Fibonacci spirali



Fibonacci
doğuşu

Özyinelemeli Fibonacci Serisi

```
#include<stdio.h>
#define N 21
//fibonacci sayılarını üreten bir fonksiyon
int fib(int x) {
    if (x < 2)
        return x;
    return fib(x-1)+fib(x-2);
    //return x<2? x:fib(x-1)+fib(x-2);
}
```

```
//main fonksiyon
int main(){
    int fibo[N];
    int i;
    for (i; i<N; i++)
    {
        fibo[i] = fib(i);
        printf("%d ", fibo[i]);
    }
}
```

Programın Çıktısı:

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765

C++ for each döngü yapısı

C++ dilinde bir dizinin/listenin veya enum sabitinin bütün elemanlarına sırayla erişmek için klasik for döngüsü yerine for-each döngüsü yapısı kullanılabilir.

Kullanım Şekli;

```
for (veritipi degisken_Adı : dizi_Adı)
    //... işlemler;
```

şeklindedir.

Bu komut yapısı ile dizideki veya listedeki her eleman sırayla döngü içinde belirtilen değişkene aktarılır.

Not. foreach döngü yapısı C dilinde yoktur, C++ 11 sürümü ile birlikte C++ diline eklenmiştir.

Klasik for ile for-each karşılaştırması

For Each Döngü Yapısı	Klasik For Döngü Eş Değeri
<pre>for (int x : dizi){ cout << x << endl; }//foreach döngüsü sonu</pre>	<pre>for (int x=0; x <4; x++){ cout << dizi[x] << endl; }//klasik for döngüsü sonu</pre>

C++ for each döngü yapısı

```
#include <iostream>
```

```
int main()
```

```
{
```

```
    int fibo[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 };
```

```
    for (int i : fibo)        // fibo dizisinin her bir elemanı
```

```
    {                        // sırası ile i'ye atanır.
```

```
        std::cout << i << "\n";
```

```
    }
```

```
    return 0;
```

```
}
```

0
1
1
2
3
5
8
13
21
34
55
89

enum sabiti (Enumeration Constant)

- - Kullanıcının tanımladığı, sıralı semboller kümesi olarak tanımlayabileceğimiz enum sabiti, normalde mevcut olmayan, bize özel verileri sıralamak amacıyla kullanılır.
- - enum deyimi için; tam sayı tipindeki sembolik sabitler kümesidir de diyebiliriz. Kullanım şekli:

- **enum** sabit_ismi {A, B, C, D, ...};

enum sabiti ile birlikte parantez içerisindeki her bir sembolik sabit {A, B, C,..} sıfırdan başlayarak sırasıyla artan bir şekilde değer alır.

enum sabiti (Enumeration Constant)

enum sabit listesi ',' atomuyla ayrılan isimlerden oluşur. İlk enum sabitinin değeri sıfırdır. Örneğin:

- enum COLORS {Red, Green, Blue, Yellow};
- enum BOOL {False, True};

Bir enum sabitine '=' atomu ile bir değer verilebilir. Bu durumda sonrakiler onu izler.

- enum DAYS {**Monday = 1**, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};

enum sabiti



- Haftanın günlerinin rakamsal karşılığını aşağıdaki gibi veren programı enum ile C/C++ dillerinde kodlayınız.

```
Gunler...:  
1.gün Pazartesi  
2.gün Salı  
3.gün Çarşamba  
4.gün Perşembe  
5.gün Cuma  
6.gün Cumartesi  
7.gün Pazar
```

enum sabiti



- Aşağıdaki tanımlama da **f1** ve **f3**'ün değeri ne olur?

```
enum foo{  
    f1,  
    f2 = 3,  
    f3,  
    f4  
};
```

enum sabitine yeni değer atama

- Bir enum sabit tam sayı kümesinde yeni bir eleman tanımlanacak ise bu tanımlanacak yeni eleman, enum kümesinin elemanlarından birine eşit olmak zorundadır.
- Örneğin;

```
enum Rakam { f1=1, f2, f3, f4};
```

ise daha sonra program içerisinde Rakam kümesine

```
'enum Rakam f5 = f3;'
```

şeklinde yeni bir değişken eklenebilir fakat

'enum Rakam f5 = 7;' şeklinde bir değer ataması **derleme hatasına** neden olacaktır.

Scope (Kapsama Alanı) Nedir?

- - scope terimi; bir değişkenin erişilebilirliğini (faliyet alanını) ve ömrünü açıklar.
- - Bir değişken, programın tamamında ya da sadece belli bir alanında (fonksiyon içerisinde) faaliyet gösterebilir, hayatta kalabilir.
- - Faaliyet gösterdiği alan dikkate alınarak
- değişkenleri **genel (global)** ve **yerel (local)** değişkenler olmak üzere ikiye ayırabiliriz.

Scope (Kapsama Alanı) Nedir?

- - Faaliyet gösterdiği alan dikkate alınarak
 - değişkenleri/nesneleri;
 - - Genel (Global)
 - Global nesneler statik ömürlüdür.
 - - Yerel (Local),
 - Yerel nesneler dinamik ömürlüdür.
-
- olmak üzere ikiye ayırabiliriz.

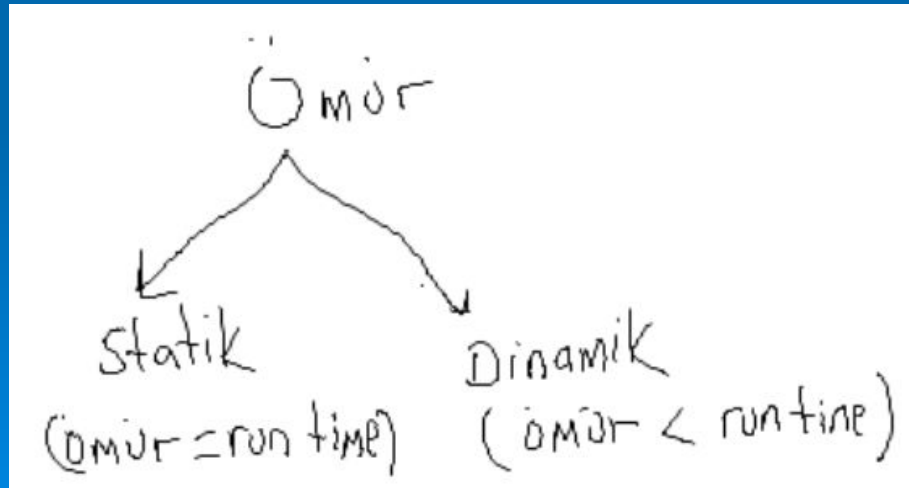
Global -Yerel değişkenlerin ömürleri

□ Genel (Global)

- Global nesneler statik ömürlüdür.

□ - Yerel (Local),

- Yerel nesneler dinamik ömürlüdür.



Scope (Kapsama Alanı)?

```
#include<iostream>
using namespace std;
string _var = "outer variable";

void func_var(){
    string _var = "inner variable";
    cout<<_var<<endl;
}

int main()
{
    func_var();
    cout<<_var << endl;
    return 0;
}
```

Bu programın
çıktısı ne olur?

Scope (Kapsama Alanı)?

```
#include<iostream>
using namespace std;
string _var = "outer variable";

void func_var(){
    string _var = "inner variable";
    cout<<_var<<endl;
}

int main()
{
    func_var();
    cout<<_var << endl;
    return 0;
}
```

inner variable

outer variable

Global Değişkenler

Özellikleri;

- Tüm fonksiyonların dışında tanımlanırlar,
- Program içindeki tüm fonksiyonlarda faaliyet gösterirler,
- Program çalıştırıldığında hafızada oluşturulurlar (yaşamları başlar), program çalıştığı sürece yaşamları/faaliyetleri devam eder. Program sonlandırıldığında yaşamları sona erer.

Global Değişkenler

```
#include<iostream>
using namespace std;
int x = 13; // x global değişken

void func_var(){
    int x = x*2;
    cout<< x <<endl;
}

int main()
{
    func_var();
    return 0;
}
```

Bu programın
çıktısı ne olur?

Global Değişkenler

```
#include<iostream>
using namespace std;
int x = 13; // x global değişken

void func_var(){
    int x = x*2;
    cout<< x <<endl;
}

int main()
{
    func_var();
    return 0;
}
```

0

6:13: warning: variable 'x' is uninitialized when used within its own initialization [-Wuninitialized]

Lokal Değişkenler

Özellikleri:

- Fonksiyon içinde tanımlanan değişkenler, sadece fonksiyon içerisinde faaliyet gösterirler, kullanılırlar.
- Tanımlandıkları fonksiyon dışından erişilmeleri mümkün değildir.
- Fonksiyon çağrıldığında hafızada oluşturulurlar (yaşamları başlar), fonksiyondan çıkıldığında ise hafızadan silinirler (yaşamları sona erer)
- Yerel değişkenler belleğin "stack" alanında yer alırlar..

Lokal Değişkenler

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

void func_var(){
    string x = "lokal";
    cout<< x <<endl;
}

int main()
{
    func_var();
    cout<< x << endl;
    return 0;
}
```

Bu programın
çıktısı ne olur?

Lokal Değişkenler

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

void func_var(){
    string x = "lokal";
    cout<< x <<endl;
}

int main()
{
    func_var();
    cout<< x << endl;
    return 0;
}
```

```
./var.cpp: In function 'int main()':
./var.cpp:12:11: error: 'x' was not declared in this scope
    cout<< x << endl;
           ^
```

static belirteci

static değişkenlerin özellikleri;

- static değişken, programın çalışması boyunca bellekten kaybolmaz. Dolayısıyla static belirteci yerel değişkenlerin ömrünü uzatır.
- Tanımlandıkları fonksiyondan çıkıldığında yerel (local) değişkenlerin hafızadan atılmalarına mukabil static değişkenler hafızaki yerlerini ve değerlerini korurlar,
- Fonksiyon tekrar çağrıldığında bir önceki çağrılmadaki bellek adresi ve içindeki değerler kullanılır.

static belirteci

Tür	Faaliyet Alanı	Ömrü
global değişken	Tüm modüller (dosya)	Programın çalışma süresince
<code>static</code> global değişken	Sadece tanımlandığı modül	Programın çalışma süresince
yerel değişken	Kod bloğu	Kod bloğunun çalışma süresince
<code>static</code> yerel değişken	Kod bloğu	Programın çalışma süresince

NOT

`static` yerel değişkenler, programcı tarafından başlangıç değeri atandıktan sonra kullanılırlar (derleme zamanında işlem görürler).

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  int git () {
4      static int a=9; //static yerel deęişken
5      a+=5;
6      return a;
7  }
8  int main() {
9      int b;
10     b=git();//git() fonk. çağrıldı
11     printf("\nb ...:d",b);
12     b=git();
13     printf("\nb ...:d",b);
14     return 0;
15 }
16 //Not. Farkı görmek için 2. çalıştırma da

```

Get URL

options

compilation

execution

b ...:14

b ...:19

Bülent Foca

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  int git () {
4      int a=9; // yerel deęişken
5      a+=5;
6      return a;
7  }
8  int main() {
9      int b;
10     b=git();//git() fonk. çağrıldı
11     printf("\nb ...:d",b);
12     b=git();
13     printf("\nb ...:d",b);
14     return 0;
15 }

```

Get URL

options

compilation

execution

b ...:14

b ...:14

extern belirteci

Teoride global bir değişken, varsayılan (default) olarak tüm modüllerde faaliyet gösterebilir. Fakat global bir “a” değişkeninin kendi modülünde değilde başka bir modülde tanımlı olduğunu bildirmek için “**extern**” belirteci kullanılır.

Kullanım şekli;

```
extern int a; /* a değişkeni başka bir modülde tanımlanmış*/
```

C/C++ Yer belirteçleri

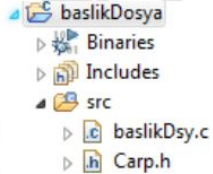
Storage Class	Keyword	Lifetime	Visibility	Initial Value
Automatic	auto	Function Block	Local	Garbage
External	extern	Whole Program	Global	Zero
Static	static	Whole Program	Local	Zero
Register	register	Function Block	Local	Garbage

static const int (a = 10, b = 20);

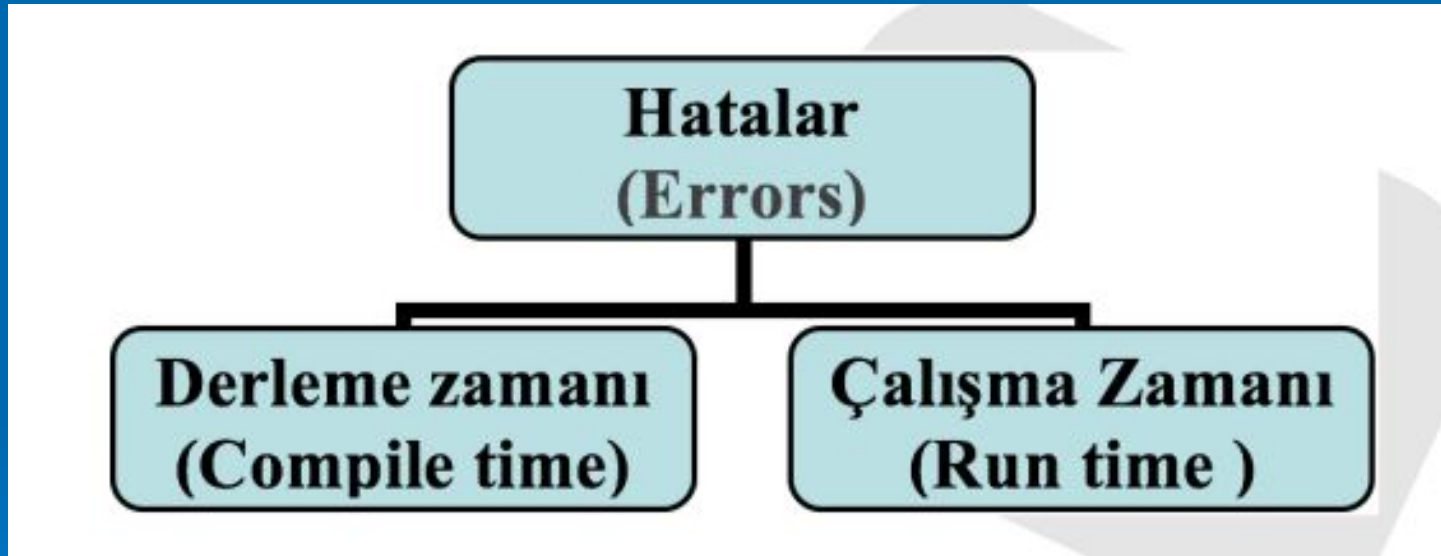
↑ yer belirteçleri; tür belirteçleri; tür

↑ değişken listesi

Başlık (Header) Dosyası (.h File Oluşturma

Carp.h dosyamız	Açıklama
<pre>int Carp(int a, int b) { return a * b; }</pre>	<p>Not: Kaynak dosyamız (baslikDsy.c veya baslikDsy.cpp) ile başlık dosyamız (Carp.h) aynı klasör içerisinde olmalıdır.</p>  <p>Şekil 3-4</p>
C Dili Kodlaması (baslikDsy.c)	C++ Dili Kodlaması (baslikDsy.cpp)
<pre>#include <stdio.h> #include <stdlib.h> #include "Carp.h" //bizim kütüphanemiz int main() { printf("3 * 4= %d", Carp(3,4)); return 0; }</pre>	<pre>#include <iostream> #include "Carp.h" //bizim kütüphanemiz using namespace std; int main () { cout << "3*4= " << Carp(3, 4); return 0; }</pre>

Hatalar



syntax / semantics
errors

exception errors

En maliyetli yazılım hataları: runtime error



Ariane 5

Software error



Integer overflow

https://www.youtube.com/watch?v=PK_yguLapqA

Overflow(Taşma)

```
0111 1111 --> +127
0000 0001
-----
1000 0000 --> -128
```

Her veri tipinin bir sınırı var, bu sistemde pozitif sınır yanlışlıkla aşılsa kendimizi negatif yüksek sayılarda buluruz. Buna programlamada üstten **taşma (overflow)** denilmektedir. Taşma alttan da (underflow) olabilir. Örneğin -128'den 1 çıkartırsak +127 elde ederiz.

Overflow(T

```
1  #include<iostream>
2  #include<cstdlib>
3  #include <climits>
4  using namespace std;
5  int main()
6  {
7      int a = INT_MAX;
8      int b = a + 1;
9      cout<<"a..:"<< a<<endl;
10     cout<<"b..:"<< b<<endl;
11     return 0;
12 }
```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

a..:2147483647

b..:-2147483648

İstisna Yakalama (Exception Handling)

Tek bir try bloğu içerisinde çoklu fırlatma (throw) ve yakalama (catch) işlemleri gerçekleştirilebilir.

C++ dilinde hata yakalama işlemi **üç anahtar sözcük** ile yönetilir; **try**, **throw** ve **catch**.

İstisna tespit etme-try: Olası hata fırlatma/üretme ihtimali olan kodun yazıldığı bloktur. Dolayısıyla istisnalar, hatalar burada tespit edilir.

İstisna fırlatma-throw: Hataya neden olacak kodu çözüm yerine fırlatır. Olası bir hatada **try** ile **catch** blokları arasındaki iletişim **throw** deyimi ile sağlanır.

Çok sık karşılanmasa da **throw** deyimi; **try-catch** blokları haricinde tek başına da kullanılabilir. Normalde programın hata oluşturma ihtimali var ya da yok ama siz yine de kullanıcıya bazı özel durumlar için uyarı mesajı göndermek istiyorsanız **throw** deyimini kullanabilirsiniz.

İstisna yakalama-catch: **try** bloğundan fırlatılan istisna **catch** bloğu ile yakalanır. Hatanın, sisteme zarar vermeden (*programı kilitlemeden*) sonlandırılması sağlanır.

İstisna Yakalama (Exception Handling)

```
#include <iostream>
using namespace std;
int main()
{
    double pay, payda, sonuc;
    try {
        cout << "say1-1..:";
        cin >> pay;
        cout << "say1-2..:";
        cin >> payda;
        if (payda == 0)
            throw (int) payda; // 0 fırlatıldı

        sonuc = pay / payda;
        cout << "Sonuç.." << sonuc;
    }

    catch (int) // yakalandı
    {
        cerr << "Payda 0 olamaz!!"; //cout<<"Payda 0 olamaz!!";
    }
    return 0;
}
```

Exception handling mekanizması

```
#include <iostream>
using namespace std;
int main() {
    try
    {
        // Normal program kodu burada yer alır
        // Koşula bağlı bir fırlatma yapılır
        throw "hata mesajı";
    }
    catch (char *e) {
        /*try bloğundaki kod; bir bir çalışma
        zamanı hatası üretecek olursa yapılacak
        işlemler buradaya yazılır..*/
        cerr<<"Hata yakalandı!"<< e <<endl;
    }
    return 0;
}
```

Taşma (Overflow) Hatasını Yakalayalım

```
#include<iostream>
#include<cstdlib>
#include <climits>
using namespace std;
int main()
{
    try{
        short a = SHRT_MAX; //a=32767
        short b = a + 1;
        cout<<"a..:"<< a <<endl;
        if (b<0)
            throw int(b);

        cout<<"b..:"<< b <<endl;
    }
    catch(int b){
        cout<<"b..:"<< ~b+1 <<endl;
    }
    return 0;
}
```

□ C++ Template Fonksiyonları

Şablon(template) fonksiyon tanımlanmasında '**template**' , `<typename>` ve `<class>` anahtar kelimeleri kullanılır. Şablon (jenerik) fonksiyon ile tip bağımlılığı ortadan kalktığından kod satırı azalır.

□ C++ Template Fonksiyonları

```
#include <iostream>
using namespace std;
template<typename T>
T add(T a,T b) {
    return a+b;
}

int main() {
    cout << add(3, 7) << endl;
    cout << add(3.5, 7.2) << endl;
}
```



TASK

:: operatörünün işlevi nedir?

Fonksiyon Parametrelerinde Pointer Kullanımı

Fonksiyonlara parametre aktarımı iki şekilde yapılır;

- Verinin doğrudan değerinin aktarılması ki buna **değeri ile aktarma (pass by Value - byVal)**,
-
- Verinin adresinin aktarılması ki buna **referansı ile aktarma (pass by reference)**

Doğal olarak verinin **adresinin aktarımında işaretçi(pointer)** kullanılmaktadır.

Fonksiyon Parametrelerinde Pointer Kullanımı

Değeri ile aktarma

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  //Yerdeğiştirme fonksiyonu
4  void swap(int s1, int s2) {
5      int bos;
6      bos = s1; // s1, bos değ. aktarıldı
7      s1 = s2;  // s2, s1'e aktarıldı
8      s2 = bos; // bos, s2'e aktarıldı
9  }
10 //Ana program
11 int main(){
12     int a = 9, b = 13;
13     swap (a, b);
14     printf ("a= %d\nb= %d", a,b);
15     return 0;
16 }
17
```

C:\WINDOWS\SYSTEM32\cmd.exe

a= 9
b= 13

Fonksiyon Parametrelerinde Pointer Kullanımı

Adresi ile aktarma

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  //Yerdeğiştirme fonksiyonu
4  void swap(int *s1, int *s2) {
5      int bos;
6      bos = *s1; // s1, bos değ. aktarıldı
7      *s1 = *s2; // s2, s1'e aktarıldı
8      *s2 = bos; // bos, s2'e aktarıldı
9  }
10 //Ana program
11 int main(){
12     int a = 9, b = 13;
13     swap (&a, &b);
14     printf ("a= %d\nb= %d", a,b);
15     return 0;
16 }
17
```

C:\WINDOWS\SYSTEM32\cmd.exe

a= 13

b= 9