Imperative

Functional Haskell

Object-oriented C++, Java

Concurrent

Logic

Scripting

Programming is all about manipulating data.

Data always has some type.

primitive types

composite types
- structures
- records
- lists
- tuples
- ⋮

→ A type is a set of values and a set of operations that can be uniformly applied on those values.

e.g

bool {true, false}
{||, &&, ... }

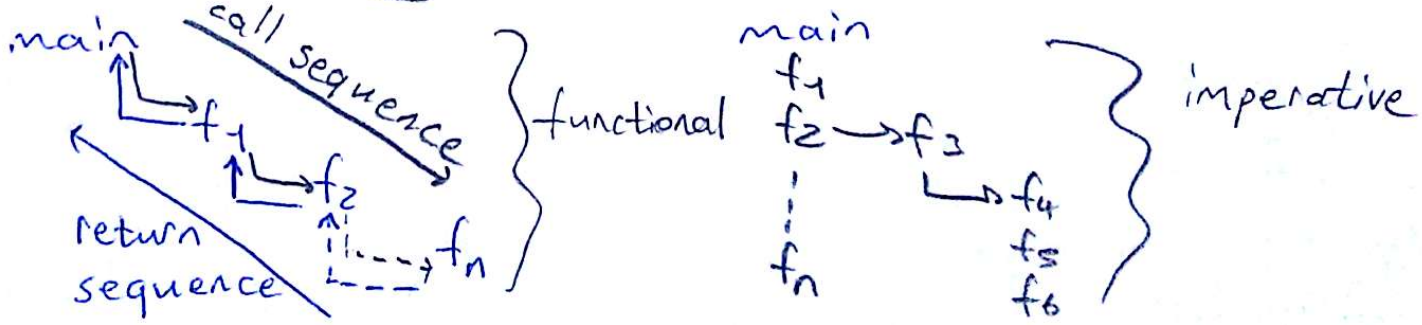
Expressions: Statements always returning a value.

strong vs weak typing

Dynamic vs static typing

— Functional Programming — (FP)

* The main operation in FP is the application of funcs to args.
* Problems are solved by the "main" (initial) func calling another func, which in turn calls another func and so on.
* The result of each func is returned to the caller which eventually produceses the final answer.



Scanned by CamScanner

Characteristics of FP

* No assignment statements (just definitions)

e.g
a = 4
a = 3 ✗

* No side effects
* No global state
* No pointers
* Type completeness
* Higher order funcs (funcs taking another func as input, or returning)
* Lazy evaluation

Higher Order Functions (HOFs)

HOFs are func that take other funcs as argument and/or return other funcs as return value.

This feature of FP allows functions to be glued together in a modular manner.

e.g Let's define a list as:

Tags

listof a = Nil | Cons a (listof a) → recursive def of a list

Nil → stands for any type

| → or

Nil → [ ]
listof a

Cons 1 (Nil) → [1]
listof a
listof a

Cons 2 (Cons 1 (Nil)) → [2, 1]
listof a

e.g
Cons 1 Cons 2 Cons 3 Nil
3rd
2nd
1st list
||
[1, 2, 3]

Scanned by CamScanner

Now let's def a func called "sum", that computes the sum of all list elements.

We will def "sum" by using a tech known as "pattern matching".

sum Nil = 0

sum (Cons x l) = x + sum l

(first elem) a list of a

sum (Cons 2 (Cons 1 (Nil))) # func call

x l

2 + sum (cons 1 (Nil)) } pattern matching

x l

2 + 1 + sum Nil

2 + 1 + 0

e.g sum [1,2,3,4] = 1 + sum [2,3,4]

e.g prod Nil = 1

prod (Cons x l) = x * prod l

mult x y = x * y

prod (Cons x l) = mult x (prod l)

e.g add x y = x + y

sum (Cons x l) = add x (sum l)

* foldr f e l

(higher-order) func input list

func to be applied value to be returned for empty list

Scanned by CamScanner

```
sum l = foldr add 0 l
prod l = foldr mult 1 l
```

* Again use pattern matching:

```
foldr f e Nil = e
foldr f e (Cons x l) = f x (foldr f e l)
```

def of foldr

e.g Define a func "count" which returns the number of elements in a given list.

```
count Nil = 0
count (Cons x l) = 1 + count l
```

now with foldr:

```
count l = foldr ? ? l
```

```
count l = foldr addOne 0 l
addOne x y = 1 + y
```

or

```
addOne _ y = 1 + y
```

↳ Don't care syntax

Lambda Funcs

Just in time func defs. (Def when we need them)

Nameless func

e.g

```
count l = foldr (\x y -> 1+y) 0 l
```

lambda func with 2 args

(\x y → 1+y): x y — args; 1+y — def

```
sum l = foldr (\x y -> x + y) 0 l
prod l = foldr (\x y -> x * y) 1 l
```

Scanned by CamScanner

e.g Def a func "max" returning the max elem. in a given list. Return -1 if list is empty.

max ~~l~~ = foldr (\x y → if x>y then x else y) -1 ~~l~~

partial func application

e.g

add x y = add x + y

add5 = add 5

add5 10 ⇒ the result is 15

incr = add 1

decr = add (-1)

Lazy Evaluation X Eager Evaluation

The ability to not evaluate something until it is abs. necessary.

Assume we have 2 programs (or funcs):

Program 1 : f
Program 2 : g
Input 1 : x
Input 2 : randy

Combine: g (f x) y

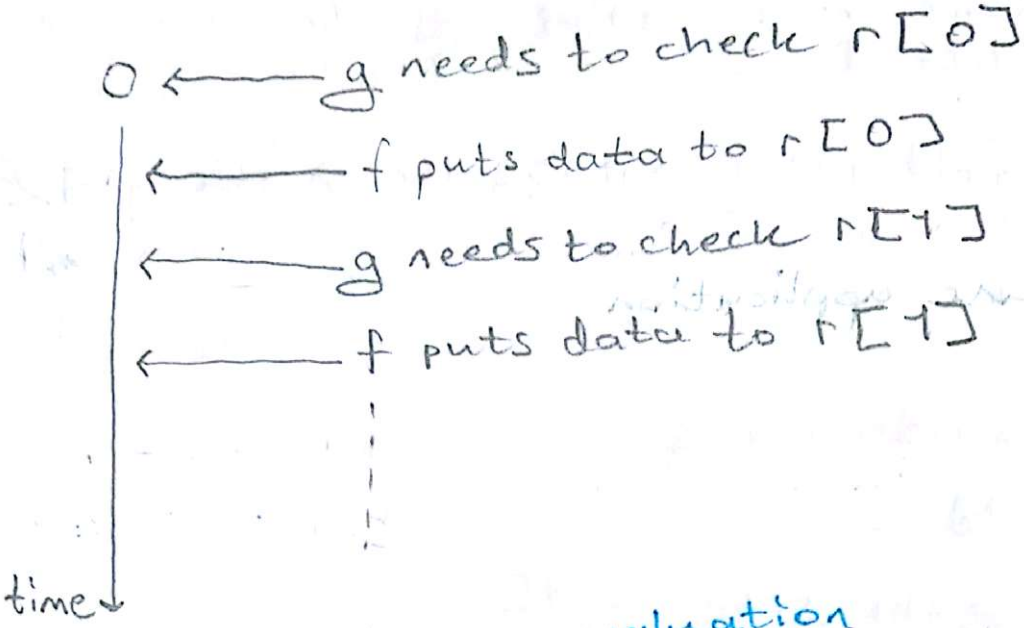
e.g

f is a program that reads all characters from a file and returns them in a list.

g is a program (func) that takes a list and a character and returns true if the list contains the char.

In lazy evaluation, the input is read by func f as much as it is required by func g.

Scanned by CamScanner

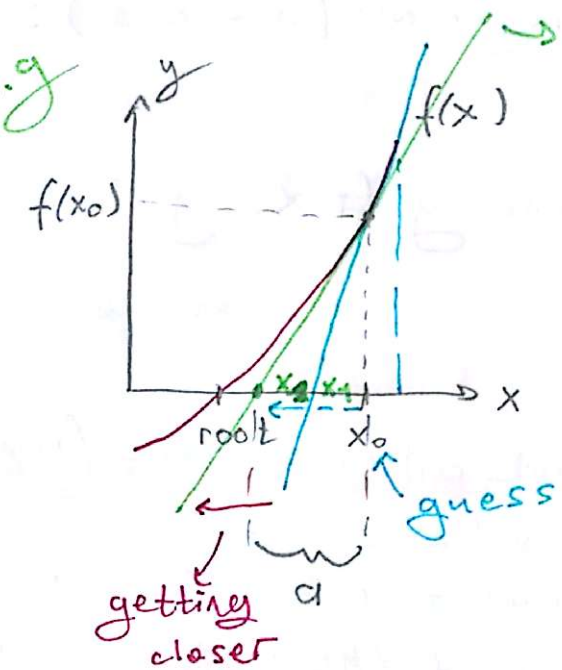


lazy evaluation

In this example the two programs run in a sychronized manner.

It is very different from C, C++, Java, ... in which "f" needs to finish before g can start.

e.g



Newton Ruphson method:

$$\frac{f(x_0)}{a} = f'(x_0) \rightarrow a = \frac{f(x_0)}{f'(x_0)}$$

$$\boxed{x_2 = x_0 - a = x_0 - \frac{f(x_0)}{f'(x_0)}}$$

update rule

in general,

$$\boxed{x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}}$$

stop when $|x_{i+1} - x_i| < \varepsilon$

ε → user defined

Scanned by CamScanner

e.g For square roots,

$f(x) = x^2 - a = 0 \rightarrow x^2 = a \Rightarrow x = \sqrt{a}$

$f'(x) = 2x$

$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = x_0 - \frac{x_0^2 - a}{2x_0} = \frac{x_0^2 + a}{2x_0} = \frac{1}{2}\left(x_0 + \frac{a}{x_0}\right)$

To generalize,

$x_{i+1} = \left(x_i + \frac{a}{x_i}\right)\frac{1}{2}$ Stop when $x_{i+1} = x_i = k$ ($\varepsilon = 0$ basically)

$k = \left(k + \frac{a}{k}\right)\frac{1}{2}$

$2k = k + \frac{a}{k} \Rightarrow k = \frac{a}{k} \rightarrow k^2 = a \Rightarrow k = \sqrt{a}$

(at that point we've reached the answer)

look the code examples (c and Hasskell)!

#pattern matching!

```
data Shape = Pointy | Circular Float
           | Rectangular(Float, Float)
```

new type (disjoint union)

```
area s =
  case s of
    Pointy          -> 0.0
    Circular r      -> pi * r * r
    Rectangular(h,w) -> h * w
```

#direct pattern matching!

```
area Pointy          = 0.0
area (Circular r) = pi * r * r
area (Rectangular(h,w)) = h * w
```

Scanned by CamScanner

Higher Order Funcs

"foldr"

$f(x), g(x)$
$g(f(x))$ } Let's define a func to simulate this compose.

$$\boxed{\text{compose } g\ f\ x = g(f(x))}$$

$$= g\ (f\ x)$$

In Haskell, there is a func called (.) that performs this

$$= g\ \$\ f\ x$$

↳ evaluate right side before passing its value to left side

3 + 4 infix notation
(.) 3 4 prefix notation

e.g

```
double x = x+x
square x = x*x
doubleSquare = double . square
doubleSquare x = compose double square x
                 ↓ necessary
               = (.) double square
               = (\x -> x+x) . (\x -> x*x)
```

e.g

compose g f x = g (f x)

type of compose ?

a (type of x)

a→b (type f returns: b)

b→c

type of f : a→b

↳ return types

$$\boxed{(b \to c) \to (a \to b) \to a \to c}$$ → return type of compose

type of compose ≡ $(b \to c) \to (a \to b) \to a \to c$

Scanned by CamScanner

Using Paranthesis or NOT

":: " indicates type

plus1 x y = x + y (x: a, y: a) → curried form

plus2 (x, y) = x + y → uncurried form

:: (a, a) → a

:: plus1 (type of plus1)

plus1 :: Num a ⇒ a → a → a

type constraint

a → (a → a)

plusFive = plus1 5

plusFive 10
→ 15

but ~~plusFive = plus2 (5, ?)~~

e.g plus2curried = curry plus2

$\boxed{\text{curry } f\ x\ y = f\ (x, y)}$

e.g plusuncurried = uncurry plus1

$\boxed{\text{uncurry } f\ (x, y) = f\ x\ y}$

NOTE: "curry" and "uncurry" zaten haskell'de var!

Function map

Takes a func and a list. Applies the func to each element of the list and returns the new list.

```
map f [] = []
map f (x:xs) = (f x) : (map f xs)
```

Scanned by CamScanner

map (\x -> x*x) [1..10]

-> [1, 4, 9, ..., 100]

Remark!

:t map (gives the type of map)

$\boxed{(a \to b) \to [a] \to [b]}$

Remark!

[1, 5, .. 10] = [1, 5, 9] ✓

[100, 99 .. 1] ✓

e.g (func filter)

- Takes a func and a list
- Returns a new list which only contains the element for which the func evaluates to true.

filter (\x -> mod x 2 == 0) [1..10]

-> [2, 4, 6, 8, 10]

- implementation

```
filter f [] = []
filter f (x:xs) = if f x == True
                  then x : (filter f xs)
                  else filter f xs
```

:t filter

$\boxed{(a \to Bool) \to [a] \to [a]}$

Scanned by CamScanner

List comprehension

$$\text{filter } f\ l = \underbrace{[x \mid x \leftarrow l, f\ x]}_{\text{list comprehension}}$$

$$[x \mid x \leftarrow l, f\ x == True]$$

Remark!
:set +s (enables interpreter to give time and byte)

last $ . . . . . (trick)

e.g quicksort [ ] = [ ]

quicksort (x:xs) = quicksort [y | y ← xs, y <= x]
++ [x]
++ quicksort [y | y ← xs, y > x]

practice : merge_sort

DO IT

Remark!
[1,2] : [ ]
→ [[1,2]]

e.g iterate

✓ It applies a func infinite number of times starting with an initial seed value and putting the results into an infinite list.

iterate (\x → 2*x) 1

[1, 2, 4, 8, 16, ...]

(iterate (\x → 2*x) 1)!!5 = 32

Scanned by CamScanner

implementation

```
iterate f s = s:(iterate (f s))
```

e.g

```
(map (\x -> x*x) (iterate (\x -> x+x) 1)) !! 0
```

9.3.17

they have to be capital (Remark! True | False)

```
data Unit = Warrior | Necromancer | Wizard
```

type — values

```
:t Warrior
-> Unit
```

```
beats Warior Necromancer = True
beats Necromancer Wizard = True
beats Wizard Warrior = True
```

pattern matching

```
beats Warior Wizard (func call) RUN-TIME ERROR
(şimdilik)
```

```
beats _ _ = False
```

Remark! Eğer üste yazılırsa, hep false döner.

e.g

Value constructor — Value

```
data Unit = Warrior Int | Necromancer Int |
            Wizard Int

beats (Warrior a) (Necromancer b) =
    if a >= 2*b then True else False

beats (Necromancer a) (Wizard b) =
    if a > b+3 then True else False
```

Scanned by CamScanner

beats (Wizard a)(Warrior b) = True

~~[illegible]~~
redundant

instead;

beats (Wizard _)(Warrior _) = True

beats _ _ = False

edition:

beats (Warrior a)(Necromancer b) =
  if a >= 2*b than (True, (Warrior a+1))
    else (Necromancer b+1)

edition:

data Unit a = Warrior a | Necromancer a | Wizard a

type — type constructor

type — value constructor

e.g

Unit Int
Unit String } types

beats (Warrior x) (Necromancer y) =
  if x >= y then True else False

e.g

beats (Warrior 2) (Necromancer 5)

:t beats

→ Num a => Unit a → Unit a → Bool

type constraint

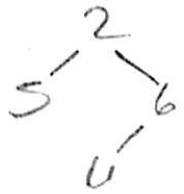
Scanned by CamScanner

Exercise

```
data Tree a = Branch a (Tree a) (Tree a) | Nil
              a      left     right
```

e.g

t1 →




```
t1 = Branch 2 (Branch 5 Nil Nil)
              (Branch 6 (Branch 4 Nil Nil) Nil)
```

In the interpreter:

```
t1
→ Error
```

To solve this problem;

```
data Tree a = Branch a (Tree a) (Tree a) | Nil
              deriving Show
```

```
getLeft (Branch _ l _) = l
        or
getLeft (Branch _ l _) = l
getRight (Branch _ _ r) = r
```

```
isEqual Nil Nil = True
isEqual (Branch x l1 r1) (Branch y l2 r2) =
        x == y && isEqual l1 l2 && isEqual r1 r2
isEqual _ _ = False
```

Scanned by CamScanner

~~isSubTree _ Nil = True~~

v1: isSubTree t1 t2 = isEqual t1 t2 ||
isSubTree (getLeft t1) t2 ||
isSubTree (getRight t1) t2

isSubTree Nil Nil = True
isSubTree Nil _ = False

v2: isSubTree t1@(Branch x l1 r1) t2
= isEqual t1 t2 || isSubTree l1 t2 ||
isSubTree r1 t2

t1@(Branch x l1 r1)
→ means t1 = (Branch x l1 r1)
↳ referring

Variables, Storage, and Commands

→ Each var. occupies some storage.
→ We update vars by commands

e.g

int a=3;

1) Allocates some memory (from where?)
2) Write the value of "3" into this memory
3) Create a binding between the name "a" and this memory.

a | 3 |

Scanned by CamScanner

e.g.

```
struct Date {
    int y,m,d;
};
Date today = {2017,3,13};
```

today

2017
3
13

e.g.

```
float v1[] = {2.0, 3.0, 4.0, 5.0}
```

v1

2.0
3.0
4.0
5.0

e.g.

```
float *V2 = new float [5];
```

address

?
?
?
?
?

V2 [address] (Sadece dynamicte değil, hepsinde olıyor.)

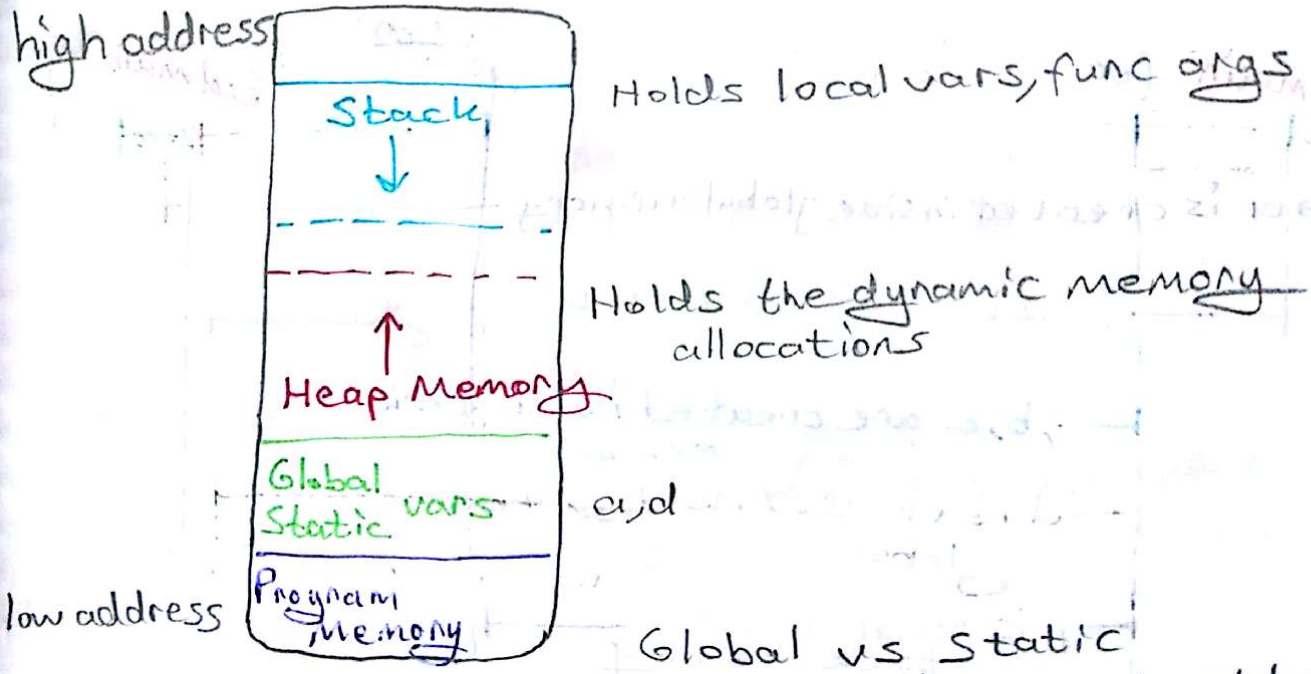
When some memory is allocated, the location of this memory depends on the location of the definition and whether pointers are involved or not.

Scanned by CamScanner

1) static (global) memory (The address of the memory does NOT change)
2) stack memory (can change)
3) heap memory (can change)

eg

```
int a;  → global
void foo(int b) {
    int c;  → local
    static int d;  → static
    int* p = new int[10];  → heap
    → local
}
int main() {
    int e;  → local
    foo(e);
    return 0;
}
```

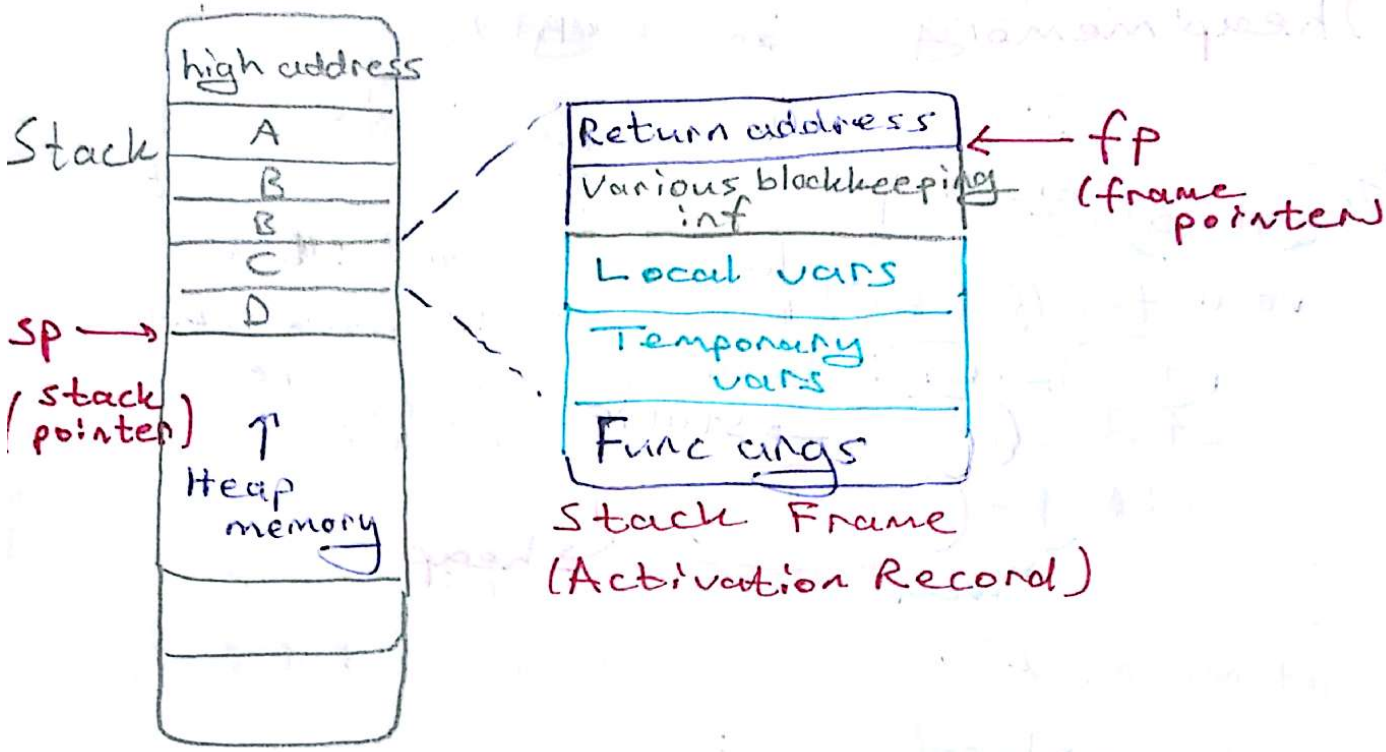


Global vs Static

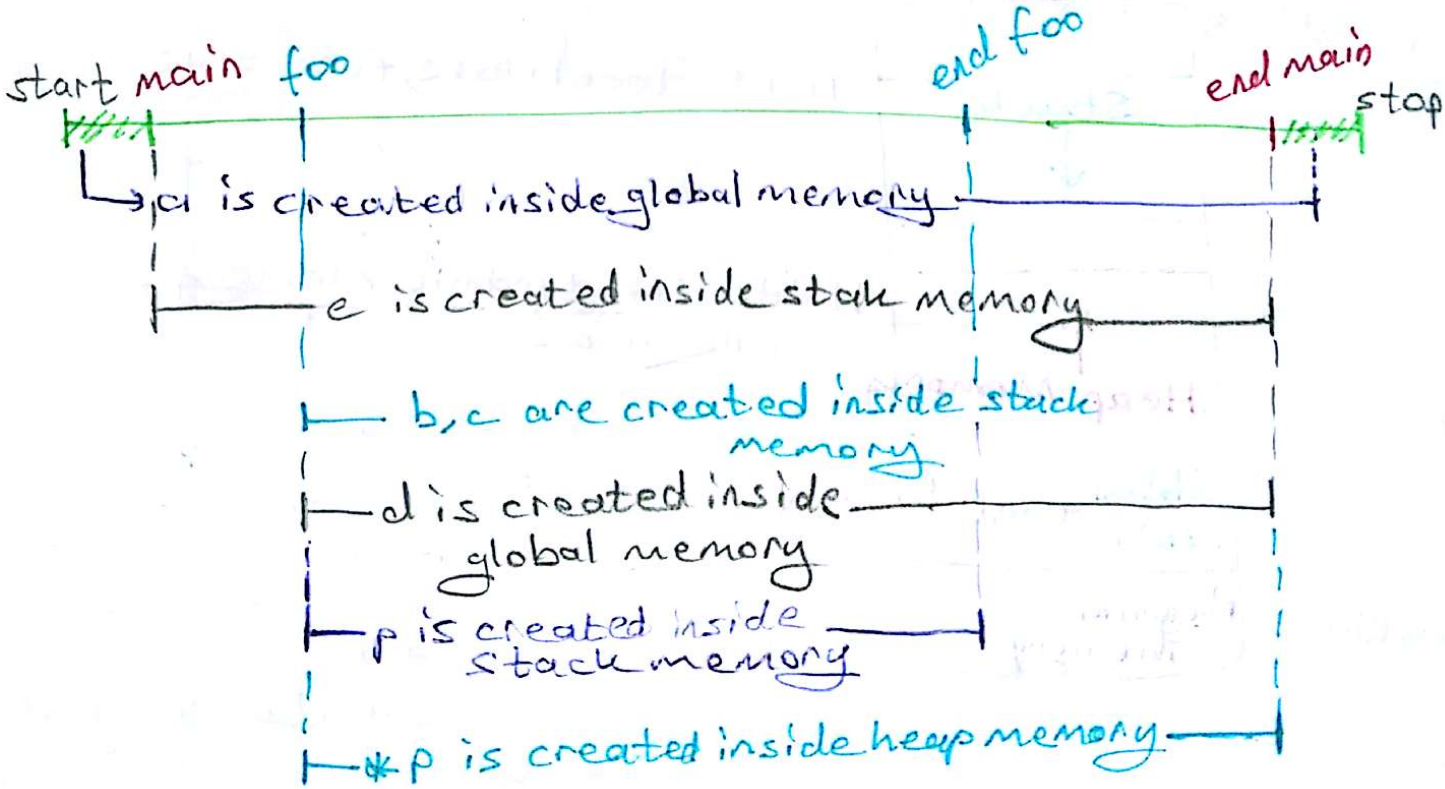
✓ ihtiyaç duyuldukunda allocate edilir
✓ address won't change

✓ address won't change

main	func A	func B	func C
A	B	if ... then B else C	D E



A B B C D →	A B B C E →	A B B C →	A B B →	A B →	A →



Scanned by CamScanner

✓ Each var occupies some storage.

static (global) storage

- Global vars
- Stat vars

address of vars does NOT change!!!

stack storage

- Local vars
- Func parameters
- Return values

addresses are DYNAMIC

heap storage
(not related with priority queue)

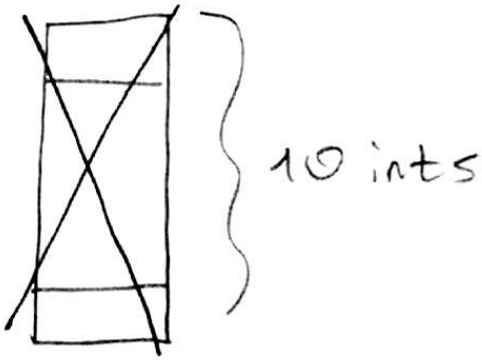
- used for dynamic memory allocatoins (e.g new, malloc)

Issues with Pointers

e.g.

```
int * p = new int [10]
:
int * q = p
:
delete[] p;
p = NULL;
```

q, ~~p~~ → dangling reference



q[0] = 5; ✗ ERROR

or

int *f = new int [10]; (f ve q aynı yere işaret edebilir!)

e.g

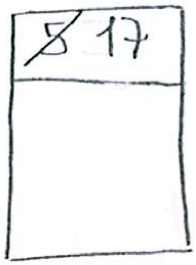
```
int * f( ) {
    int a=5;
    return &a;   // returning an address from the stack
}

int main( ) {
    int * p =f( );
    //*p = 10; //crashes
    g( );
    cout <<*p << endl //
}

void g( ) {
    int x=17;
}
```

dangling reference

does NOT crashes!!!
prints 17



Stack

e.g

```
int * f( ) {
    int a=new int [1];
    return a;
}

int main( ) {
    int * p=f( )
    *p=10;
    return p;
}
```

ENTIRELY CORRECT

HEAP (DYNAMIC ALLOCATION)

e.g

```
void f( ) {
    int * p= new int [8];
    return;
}
```

HEAP is destroyed, but the address of p still is occupied by p !!!

Scanned by CamScanner

e.g

```
void f(int n){
    int *p = new int [n];
    if (...){
      // delete [] p
      return;
    }
    delete [] p;
}
```

Garbage memory leak!!!

must be added!

e.g

```
void f (const char* filename){
    FILE * fp = fopen(filename, "r");

    fclose(fp) // Eğer close etmezsen, garbage memory leak olur!
}
```

Recall:

- ✓ Static (global)
- ✓ Stack
- ✓ Heap

} Destroyed when the program ends

- ✓ Persistent // Değerlerini korurlar program kapandıktan sonra bile!

Scanned by CamScanner

e.g

```
void f(const char* filename){
    FILE* fp = fopen(filename, "r");
    int *p = new int[5];
    ...
    p[0]=3;
    fprintf(fp, "3");
    exit(0);
}
```

persistent variable

heap variable

— SIDE EFFECTS —

Normally, expressions are evaluated to yield a value.

Some exprs also change some state.

e.g

```
int a= 3;
a;
if (a==3)
    // do something
```

returns a value

returns a value

e.g

```
if (a++==3)   //returns a value and changes a state
```

exprs like this have a side effect !!!

e.g

```
double b=5;
double f(){
    double a=3
    return b=a;        // side effect!
}
```

3 is returned

Scanned by CamScanner

e.g

```
enum Gender {male, female};
```

0 1

```
Gender g;
if (getchar(fp) == 'F')
    g = female;
else if (getchar(fp) == 'M')
    g = male;
```

file pointer (FILE*)

FMMF

pos

FMMF

pos

So, sideeffect!!!

Binding and Scope

Often, we use identifiers in our programs such as "n+1", "f(n)", "return x", ...

identifiers

Identifiers alone are not meaningful at all; they become meaningful when they are bound to a decleration.

E.g

```
bool isOdd (int n) {
    return n % 2 == 1;
}
```

decleration

BINDING

identifier

e.g

```
int n = 5, m = 3;
bool isDivisible (int n) {
    return n % m == 0;
}
```

n = 3

Scanned by CamScanner

Environment

A set of bindings available at a certain time

Scope: It is a different way of looking at an environment.

The scope of a var. are the parts of the program where that variable is accessible.

In "C", scope only goes down!

Decleration : eg $\underbrace{\texttt{int f(int z);}}_{\text{decleration signature prototype}}$

vs

Definition

⇓

eg

```
int f(int z){
    //do something
}
```

} definition

Redefinitions cause ERRORS!

Soln:
```
#ifndef __FILE_NAME__
#define __FILE_NAME__
    :
#endif
```

Every header file must have this!

Defs'lar header'da olmamalı! (En iyisi)

$\underbrace{\texttt{extern int a;}}_{\text{var'ın declare edildiğini gösteriyor.}}$ (no storage used)

Scanned by CamScanner

Remark! g++ main.cpp test2.cpp test3.cpp

Remark! g++ -c test2.cpp

sadece compile et demek!
object dosyası oluşturur.

Remark! g++ main.cpp test2.o test3.o

hazır compile edilmiş
zaten

Binding & Scope (Summary)

- ☐ Binding: association between an identifier and its decleration
- ☐ Scope: range of the program in which the identifier can be used.
- ☐ Environment: a set of bindings

definition vs decleration

defines the meaning — just creates a name

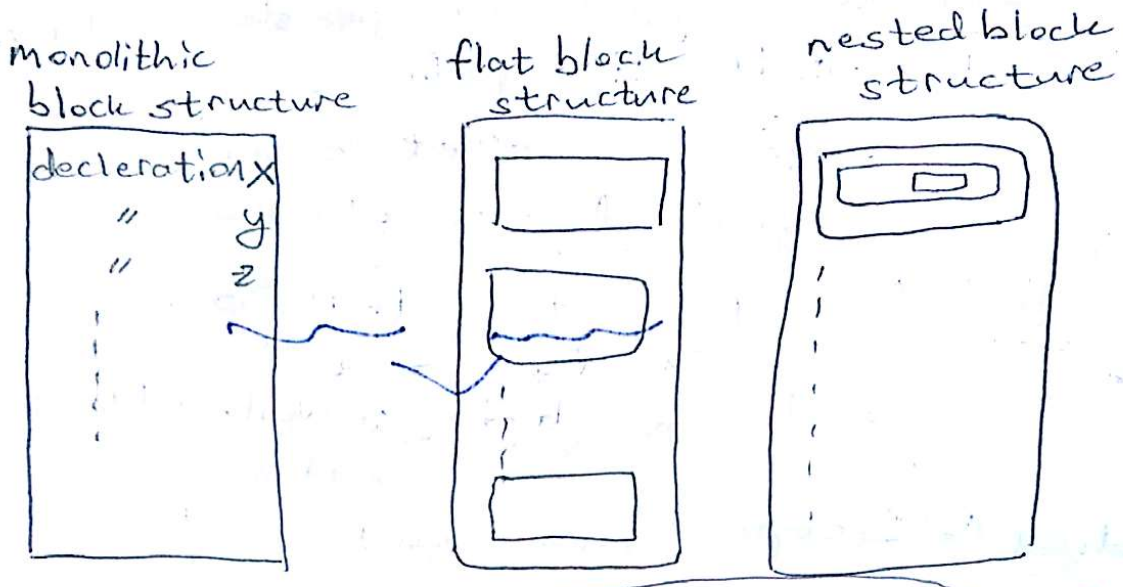
- ▷ Decleration go to header files
- ▷ Definitions go to source (.c, .cpp) files
- ▷ NEVER include source files!

~~#include "test.cpp"~~

Block Structure

```
if (...)        void f()        class A
{               {               {
  block           block           block
}               }               }
```

Scanned by CamScanner



e.g

```
myfunc = let  x=3
              y=2
              z=1
         in   let y=1
              in  x + let x=5
                      in  x+y+z
```

creates a BLOCK!

e.g

```
int x=3;
void f( ){
    int x=5;
    cout << x << endl;
    cout << :: x << endl;
}
```

5 3

e.g

```
int x=3; void f( ){
    int x=5;
    if( ) {
        int x=7;
        cout << x << endl;
        cout << :: x << endl;
    }
}
```

unaccessible

7 3

Dynamic vs Static Binding (Scoping)

- In static binding, the body of a func is executed in the environment of the func's def.
- In dynamic scoping, the body of a func is executed in the environment of the function call.

e.g Consider the following C-like program

```
int s=2;
int f(int x) {
    return s*x;
}

void p(int y)
    print(f(y));
}

void q(int z)
    int s=3;
    print(f(z));
}
```

Assume we call

p(5) // prints 10
q(5) // prints 10 (!)

→ Static scoping (e.g C)

p(5) // prints 10
q(5) // prints 15

→ Dynamic scoping (e.g Pearl)

⇒ SINAVDA ÇIKACAK!!!

Ch 1, 2, 3, 4 have been finished!

Abstraction & Parameter Passing

- Abstraction is the process of showing only what is necessary.
- There are differen degrees of abstraction

Scanned by CamScanner

- Just the process of using a high-level programming language is a means of abstraction.

e.g

In C	In Assembly

a = b + 9	mov b(%rip), %eax
	add $9, %eax
	mov %eax, a(%rip)

- Our real focus will be the abstraction that exists within a high-level language.

- **Function Abstraction**

  Funcs abstract over expressions and commands

- **Array Access Abstraction**

  ▷ Infact, the array access operator that exists in many languages is an abstraction itself.

e.g In C

```
int a[100];
a[24]=5
```

array access operator

or

```
*(a+24)=5
```

or

```
*(int *((unsigned char *)a + 24 * sizeof(int)))=5
```

Scanned by CamScanner

```
int b[50][75]
b[7][12] = 5
```

or

```
*((int*)(b+7)+12) = 5;
```

or

```
*(&(b[0][0]) + 7*75 + 12) = 5
```

Reminder (In C)

```
int **a;
c = new int *[50],
for (int i=0, i<50, i++)
    c[i] = new int [75]
c[7][12] = 5 // WORKS
*(&(c[0][0]) + 7*75+12) = 5 // NOT work!
```

NOT necessarily continuous memory blocks

- Generic Abstraction
- The ability to abstract over decleration/definitions

e.g C++

```
template <class T>
void swap(T & a, T & b){
    T tmp = a;
    a=b;
    b = tmp;
}
template <class T>
    class List{
        T data;
        List<T>* next;
        :
    }
```

Scanned by CamScanner

- Iterator Abstraction
- Iterators provide us a way to traverse a data structure without knowing its implementation details.

e.g

sort of dynamic array

std::vector →

std::list →


```
std::vector<int> v;
   ⋮
v[50] = 4;
```

Remark!

```
auto x=5 // int x=5
auto y=5.0f // float y=5.0
auto z=5.0 // double z=5.0
```

LOOK AT IMAGES!

Scanned by CamScanner

Parameter Passing

```
void f(int a) {
    :
}
int b;
f(b); // func call
```

a → formal parameter

b → actuall parameter

relationship

Copy Mechanizm

- Copy-in — pass by value
- Copy-out — pass by result
- Copy in-out — pass by value-result

Address Mechanizm

- Reference
- Constant Reference method

Copy-in

```
void f(int a) {
    a=5;
}
int b=3;
f(b);
```

Copy-in

b[3] copy-in

b[3] a[3] f(b)

b[3] a[5] a=5

b[3] ~~a[5]~~ return

Copy-out

b[3]

b[3] a[?] f(b)

b[3] a[5] a=5

copy-out

b[5] ~~a[5]~~ return

e.g (In C) void f(out int a); // decleration

f (out b); // func call

// usage of copy-out in C.

NOTE: In "copy-out", the func must initialize the parameters value!

Scanned by CamScanner

e.g

(In C#) void g(int a, out int b, ref int c)

// decleration

Reference Mechanism

The formal parameter becomes just the other name for the actual parameter / they become aliases of other)

b [3]

a, b [3] f(b)

a, b [5] a=5 //not possible if a was a constant reference

~~a~~, b [5] return

e.g (In C++)

```
void f(int & a){
    a=5;
}
int b=3;
f(b);
```

```
void f(const int & a){
    a=5; // COMPILE ERROR
}
```

(In C#)

```
void f(ref int a){
    a=5;
}
int b;
f(ref b);
```

NOTE:
Ref is a abstraction over direct usage of pointers.

Scanned by CamScanner

▷ (In C) There is no reference mechanism

```
void f (int *a){
    *a=5;
}
int b;
f (&b)
```

int e=5;

NOTE: 99%

1) const int *a;=&e; *a=3 //ERROR a=&f // OK

2) int const *b;=&e;

3) int *const c;=&e; *a=3 // OK a=&f // ERROR

① and ② are identical!

4) const int *const d=&e;

// *a=3; // error //a=&f // ERROR

In C++	In C (using pointers)
b [3]	b [3]
a, b [3] f(b)	b [3] a [b]
a, b [5] a=5	b [5] a [b] *a=5;
~~a~~, b [5] return	b [5] a ~~[b]~~ return
a,b has the same address	a,b has the different address.

Scanned by CamScanner

Example

```
int x=1, y=2;
void f(int a, int b){
    x += a+b;
    a++;
    b = a/2;
}
int main(){
    f(x,y);
    print x,y;
    return 0;
}
```

x	y	a	b
1	2	1	2
4	2	1	2
4	2	2	2
4	2	2	1
2	1	2	1

Output under reference mode; 5 2

" " copy in-out; 2 1

copy-in : 4 2

copy-out; ERROR

x	y	a	b
1	2	1	2
4	2	1	2
4	2	2	2
4	2	2	1
4	2	2	1

Output under copy-out (Assume we insert a=b=0;) : 1 0

x	y	a	b
1	2	?	?
1	2	0	0
1	2	1	0
1	2	1	0
1	0	1	0

Scanned by CamScanner

Evaluation Orders

** 1) Eager Evaluation

Evaluate exprs before passing it as a parameter.

* 2) Lazy Eval.

Only evaluate the expr just before it is used for the first time. (remember you evaluated it or not)

3) Normal-order Eval.

Leave all usages unevaluated until they are used.

e.g Given the following program.

```
f x y = if x<12
        then x*x+y*y+x
        else x+x*x
```

Q: How many addition and multiplication oprs will be performed for each evaluation order type for f (3*12+7) (24+16*3)

Eager eval: 3 3

Lazy eval: 2 2

Normal-order: 5 5

— — — Church-Rosser Property — — —

In a PL without side-effects, using different eval. types for exprs that can be evaluated does NOT change the result of the program!

! ! ! (i.e a/0 cannot be evaluate

Scanned by CamScanner

e.g (In C/C++)

```
#define DEG2RAD(x)(3.1415*x/180)
```

uses NORMAL-ORDER eval.

Attention for normal order eval!

insert insert

```
int x = 90;
int xRad2 = DEG2RAD(x) + DEG2RAD(x)
```

Abstraction

↓

Encapsulation : Bringing together related concepts
e.g writing a class

↓

Abstract Data types : Hiding the actual data definition

e.g module Vector where #encapsulation

data Vector = Vector Float Float Float

type — data constructor — add

(Vector x y z) `add` (Vector a b c)
= Vector x+a y+b z+c

(Vector x y z) `sub` (Vector a b c) = Vector x-a y-b z-c

s `mul` (Vector a b c) = Vector s*a s*b s*c

magnitude (Vector a b c) = sqrt $ a*a+b*b+c*c

norm v@(Vector a b c) = let m = magnitude v
in (Vector a/m b/m c/m)

Scanned by CamScanner

neg (Vector a b c) = Vector (-a) (-b) (-c)

back to the beginning:

module Vector where deriving Show

(ie this makes the Vector type an instance of the show typeclass) we can imagine this as an interface or

default imp.

print etmek istediğimizde string şeklinde print eder

– We need to make our "Vector" as an instance of the "Show" type class.

```
Instance Show Vector where
    show (Vector a b c) = "(" ++ show a ++ ","
                          ++ show b ++ "," ++ show c
                          ++ ")"
```

overloading

not recursion (Float version of show)

eg

> Vector 3 4 5

(3, 4, 5)

module Vector (Vector, vec, sub, mag)

where

public

data constructor vector

(bu kısım yazılmazsa her şey public)

vec a b c = Vector a b c

Scanned by CamScanner

interface

module Vector (Vector (Vector)) where

implementation

data Vector = Vector Float Float Float

details of data def.

OR

module Vector (Vector, vec, ...) where

implementation

data Vector = Vector Float Float Float

vec x y z = Vector x y z

constructor

Vector x y z 'add' Vector a b c

= vector (x+a) (y+b) (z+c)

-- Inside a module we can use private funcs!

Vector x y z 'mul' s = Vector (x*s) (y*s) (z*s)

s 'mul' v @ (Vector x y z)

= v 'mul' s

:info Num

Remark: + operators

(+) func e.g. it (+)

Scanned by CamScanner

instance Num Vector where

(Vector x y z) `(+)` (Vector a b c)

infix notation

= Vector (x+a) (y+b) (z+c)

`(-)`

`(*)`

signum (Vector a b c)

= error "Signum is NOT defined"

↳ to give an error

keywordi typesystem

— Type Systems —

Design choices about how a PL deals with types.

- ✓ monomorphic vs polymorphic
- ✓ static vs dynamic
- ✓ strong vs weak
- ✓ type equivalence
- ✓ type relationships (inheritance, etc)

— MONOMORPHIC Type System —

Funcs operate on a single type only. Need to define multiple func with different names to operate on multiple types (e.g C)

Scanned by CamScanner

```
void swapInt (int *a, int *b) {
    int c = *a;
    *a = *b;
    *b = c;
} //We need swapFloat, swapDouble etc to support other types.
```

@@ Polymorphic TS @@@@@

- Ad-hoc Poly. (overloading)
- Parametric Poly. (templates)
- Inclusion Poly. (Base Class Derived Class relationships)

→ C++ supports all 3 of them.

Ad-hoc Polymorphism (Overloading)

Funcs with the same name can be applied on different types to obtain different results.

e.g

```
struct Complex {
    float real;
    float imag;
}
Complex add (const Complex & c1, ... Complex c2) {
    Complex result;
    result.real = c1.real + c2.real;
    result.imag = c1.imag + c2.imag;
    return result;
}
```

Scanned by CamScanner

```
string add (string s1, string s2){
    string result = s1 + s2;
    return result;
}
```

- C++ supports this but parameter types must be different. If only the return type is different it does NOT work!

Parametric Polymorphism

The same func work on multiple types uniformly.

eg (Haskell) swap (a,b) = (b,a)

```
> swap(3,4)
(4,3)
> swap (4,'a')
('a',4)
```

eg (C++)

```
template <class T>
void swap (T&a, T&b){
    T tmp = a;
    a = b
    b = tmp;
}

int a=3, b=4;
swap(a,b);
float x= 3.0, y= 4.0;
swap(x,y);
```

Scanned by CamScanner

```
swap(3,4); //COMPILE ERROR
const int a=3;
  "   "   b=4;
swap(a,b);
```

SAME

Summary:

Type Systems

Polymorphism: Whether func can operate on multiple types

✓ ad-hoc (overloading)

```
void f(int)
void f(string)
```

1+ definitions

✓ generic (template)

```
templat <class T>
void f(T);
```

only 1 def.

✓ inclusion (subtyping)

- Based on relationship with types funcs defined for base types can be used for derived types.

eg

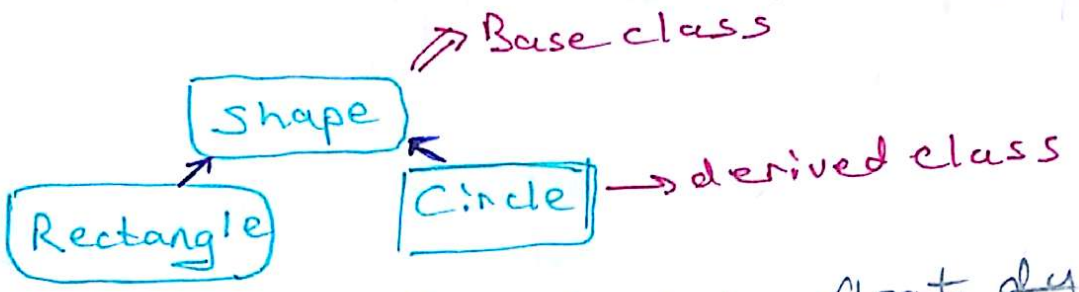
```
class Shape{
  public:
    float x,y; //position
    Shape(): x(0), y(0) { }

};
class Circle: public Shape{
  public:
    float r; //radius
    Circle(): r(0) { }

};
```

Scanned by CamScanner

```
class Rectangle : public Shape {
    public:
        float w,h; // size
        Rectangle(): w(0), h(0) { }
};
```




```
void move(Shape& s, float dx, float dy) {
    s.x += dx;
    s.y += dy;
}

Shape s;
Circle c;
Rectangle r;
move(s, 3, 4);
move(c, 5, 6);
move(r, 2, 1);
```

Inclusion polymorphism

Static vs Dynamic Typing

✓ In static typing, types of all vars are fixed at compile time.

✗ In dynamic typing, types of all vars can change during run-time.

Scanned by CamScanner

eg

```
void f(int a){
}
```

→ static

add x y = x+y

Num a => a → a → a

Python, has dynamic typing !

Dynamic is FLEXIBLE but prone to MORE ERRORS

Static is LESS FLEXIBLE, but MORE SECURE

→ run-time sırasında check yok (her şey belli) => daha hızlı

Strong vs Weak typing

✓ Flexibility of a language to make IMPLICIT TYPE CONVERSIONS

eg (Javascript)

```
var a = "2"+1; // a = "21"
var b = "2"-1; // b = 1
```

} Weak typing

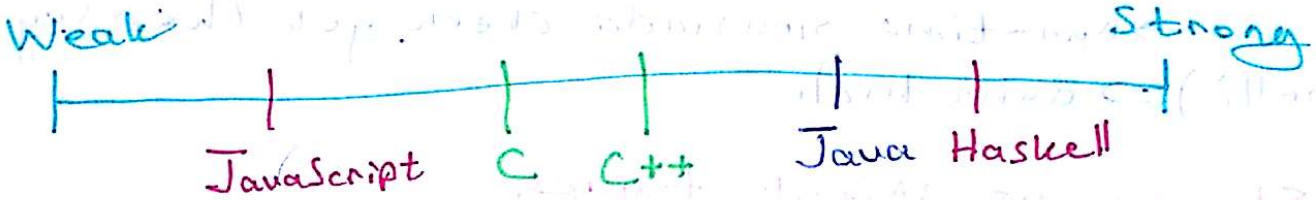
eg

```
class A {
  public:
    A (int x): y(x) { }
    int y;

};
A a = 3;   // type converted
           // ok
```

Scanned by CamScanner

```
class A {
    public:
explicit A(int x); y(x) { }
    int y;
}
A a=3; // ✗ ERROR
A a(3); ✓
A a2=A(3) ✓
```



✓ Strong typing:
- ▸ Very rigid
- ▸ More secure

✓ Weak typing:
- ▸ Very flexible
- ▸ Less secure

Scanned by CamScanner