local, non-local & global variables → *Scope
* static + Dynamic Scoping
* Referencing Env.
* Aliasing

clarify 1-page for next week

Sept-19

Scope (of a binding/name)

①

```
for (int i=0; i<n; i++) {
    |
    ≡
}
```

②

```
int g;
int gcd(int a, int b) {
    while (a != b) {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
    return a;
}// gcd()
int r;
int main() {
    int x, y;
    //Input x+y
    int g = gcd(x, y);
    //output g
    return 0;
}//main()
```

Scope: Region of program in which a name/binding is active/visible/accessible.

e.g. change ① to add

— printf("____", i);

- move i outside the for-loop.

② Add int z; at top.
int b between gcd() & main()

Terminology

- local variables - var defined + used in a pgm blk
- non-local variable - variable defined outside
- global variable - variable visible to the entire pgm

Static Scoping (aka lexical scoping)

When the scope of all names can be known by looking at the text of the program - i.e. compile time.

Dynamic Scoping

When the scope of a name can only be determined at run time, dictated by flow of execution of the program

Ex: static scoping

```
var N                                   ← Global variable.
function P1(A1) {
    var x                               ← x is local in P1
    function P2(A2) {                   ← x is non-local in P2 & P3
        function P3(A3){
            |
            =
        }//P3()
        =
    }//P2()
    function P4(A4) {                   ← x is non-local in P4
        function F1(A5){
            var x                       ← shadows/hides P1:x
            =
        }//F1()
        =
    }//P4()
    =                                   ← P1:x, P2(), P4() visible
}//P1()                                    P2(), P3(), F1() not visible.
```

Ex: Dynamic Scoping

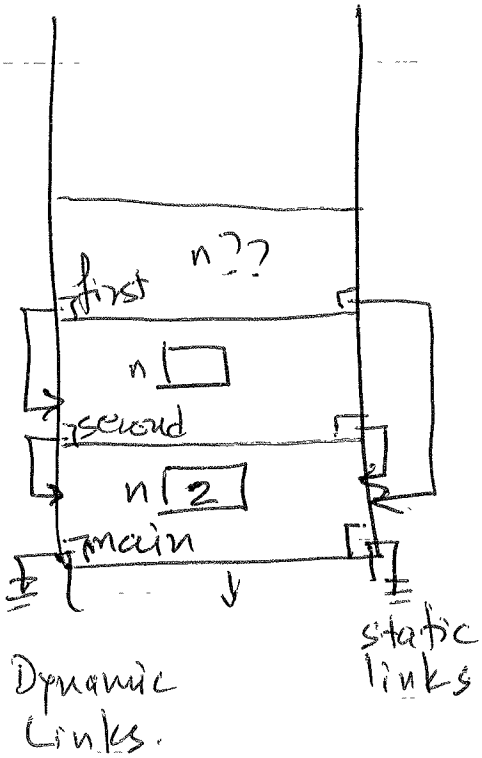
```
int n
function first(—) {
    n = 1                          <- non-local access
}//first()
function second(—){
    int n
    first(—)
}//second()

n = 2
if (read_int() > 0)
    second(—)
else first()
print(n)
```

Q: What is printed?
A: Depends on scoping rules.

Number Entered →	Negative	Positive
Static Scoping	1	1
Dynamic Scoping	1	2

Implementing Scope Rules



Trace program for each scoping rule.

Referencing Environment

Complete set of bindings at a given point in a program.

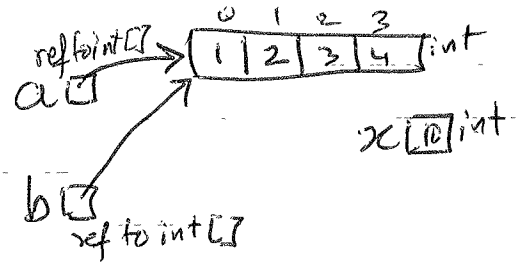
Can be determined by using stack frames and static/dynamic links.

But there are some other issues due to:

- aliasing
- overloading
- polymorphism
- first-class values
- etc.

Java

```
int [] a = {1,2,3,4};
int x = 10;
int [] b;
b = a;
b[0] = 10;
// print a
```



Q. What is printed?? A: 10, 2, 3, 4.

In Java, arrays and objects are reference variables

Above, b[0] is same as a[0].

Aliasing: when one or more names in a program (referencing environment) refer to the same object.

Above, a[] and b[] are aliases.

Aliasing is common in PLs where we have references/pointers.

e.g.

```
void f(int &a, int &b) {
}// f()
```

① f(x,x) → in f, a + b are aliases

② f(l[i], l[j]); // when i=j, a + b are aliases

Another example

C

```
int *n;
void f(int *a){
  |
  |
}//f()
int main(){
  | f(n);
  |
}
```

← In f() a + n are aliases