ELEMENTS OF IMPERATIVE PROGRAMMING STYLE

Instructors: Crista Lopes

Objectives

- Level up on things that you may already know…
  - Machine model of imperative programs
  - Structured vs. unstructured control flow
  - Assignment
  - Variables and names
- …so to understand existing languages better

Imperative Programming Style

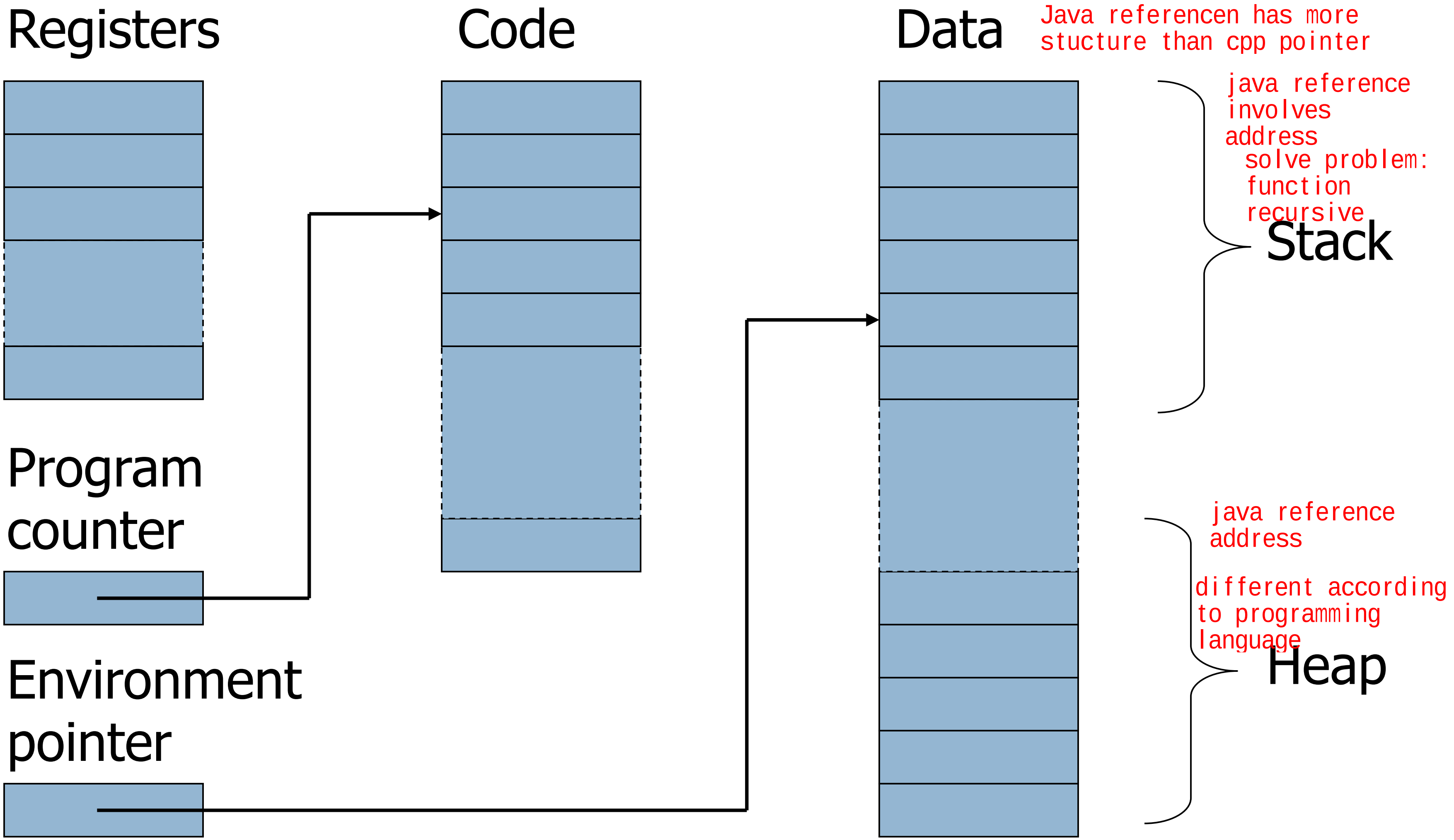
- Control-flow statements logic judgement , not necessary
  - Conditional and unconditional (GOTO) branches, loops
- Key operation: assignment
  - Side effect: updating state (i.e., memory) of the machine

Imperative Programming Style

- Oldest and most popular paradigm
  - Fortran, Algol, C/C++, Java …
- Mirrors computer architecture
  - In a von Neumann machine, memory holds instructions and data

Simplified Machine Model

gabbish collection:
automatically clean memory



Registers

Code

Data

Java referencen has more
stucture than cpp pointer

java reference
involves
address

solve problem:
function
recursive

Stack

java reference
address

different according
to programming
language

Heap

Program counter

Environment pointer

Memory Management

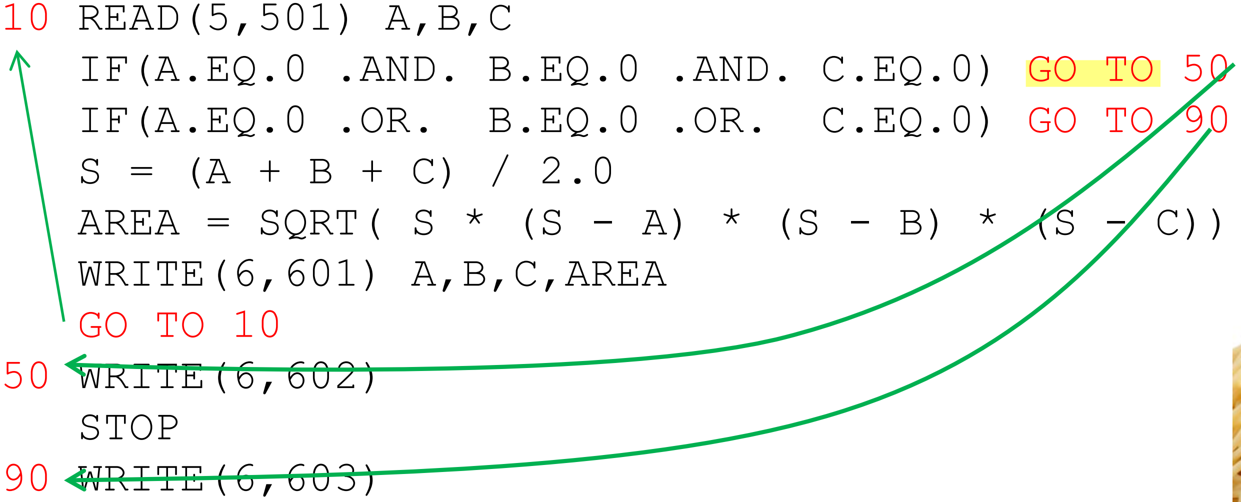
- Registers, Code segment, Program counter
  - Ignore registers (for our purposes) and details of instruction set
- Data segment
  - Stack contains data related to block entry/exit
  - Heap contains data of varying lifetime
  - Environment pointer points to current stack position
    - Block entry: add new activation record to stack
    - Block exit: remove most recent activation record

Control Flow

Control Flow

- Control flow in imperative languages is designed to be sequential
  - Instructions executed in order they are written
  - Some also support concurrent execution (Java)
- But with branching and looping instructions
  - **If** something is true do this **else** do that
  - **Case** x is value1 do this, x is value2 do that, x is value3 do that other thing
  - **While** something is true do this
  - Do this n times

Branching, originally (e.g. Fortran)



```
C AREA OF A TRIANGLE - HERON'S FORMULA
C INPUT - CARD READER UNIT 5, INTEGER INPUT, ONE BLANK CARD FOR END-OF-DAT
C OUTPUT - LINE PRINTER UNIT 6, REAL OUTPUT
C INPUT ERROR DISPAY ERROR MESSAGE ON OUTPUT
  501 FORMAT(3I5)
  601 FORMAT(4H A= ,I5,5H  B= ,I5,5H C= ,I5,8H AREA= ,F10.2,12HSQUARE UNIT
  602 FORMAT(10HNORMAL END)
  603 FORMAT(23HINPUT ERROR, ZERO VALUE)
      INTEGER A,B,C
   10 READ(5,501) A,B,C
      IF(A.EQ.0 .AND. B.EQ.0 .AND. C.EQ.0) GO TO 50
      IF(A.EQ.0 .OR.  B.EQ.0 .OR.  C.EQ.0) GO TO 90
      S = (A + B + C) / 2.0
      AREA = SQRT( S * (S - A) * (S - B) * (S - C))
      WRITE(6,601) A,B,C,AREA
      GO TO 10
   50 WRITE(6,602)
      STOP
   90 WRITE(6,603)
      STOP
      END
```

Goto in C

```
# include <stdio.h>
int main(){
   float num,average,sum;
   int i,n;
   printf("Maximum no. of inputs: ");
   scanf("%d",&n);
   for(i=1;i<=n;++i){
       printf("Enter n%d: ",i);
       scanf("%f",&num);
       if(num<0.0)
         goto jump;
       sum=sum+num;
   }
jump:
  average=sum/(i-1);
  printf("Average: %.2f",average);
  return 0;
}
```

goto is dangerous

Structured Control Flow

- Program is structured if control flow is evident from syntactic (static) structure of program text
  - Hope: programmers can reason about dynamic execution of a program by just analysing program text
  - Eliminate complexity by creating language constructs for common control-flow patterns
    - Iteration, selection, procedures/functions

Historical Debate

- Dijkstra, “GO TO Statement Considered Harmful”
  - Letter to Editor, Comm. ACM, March 1968
  - Linked from the course website
- Knuth, “Structured Prog. with Go To Statements”
  - You can use goto, but do so in structured way …
- Continued discussion
  - Welch, “GOTO (Considered Harmful)^n, n is Odd”
- General questions
  - Do syntactic rules force good programming style?
  - Can they help?

Structured Programming

- Standard constructs that structure jumps

  if … then … else … end

  while … do … end

  for … { … }

  case …
- Group code in logical blocks
- Avoid explicit jumps (except function return)
- Cannot jump into the middle of a block or function body

Assignment

Assignment (you thought you knew)

```
x = 3
x = y+1
x = x+1
```

Informal:

“Set x to 3”
“Set x to the value of y plus 1”
“Add 1 to x”

Let’s look at some other examples

Assignment (you thought you knew)

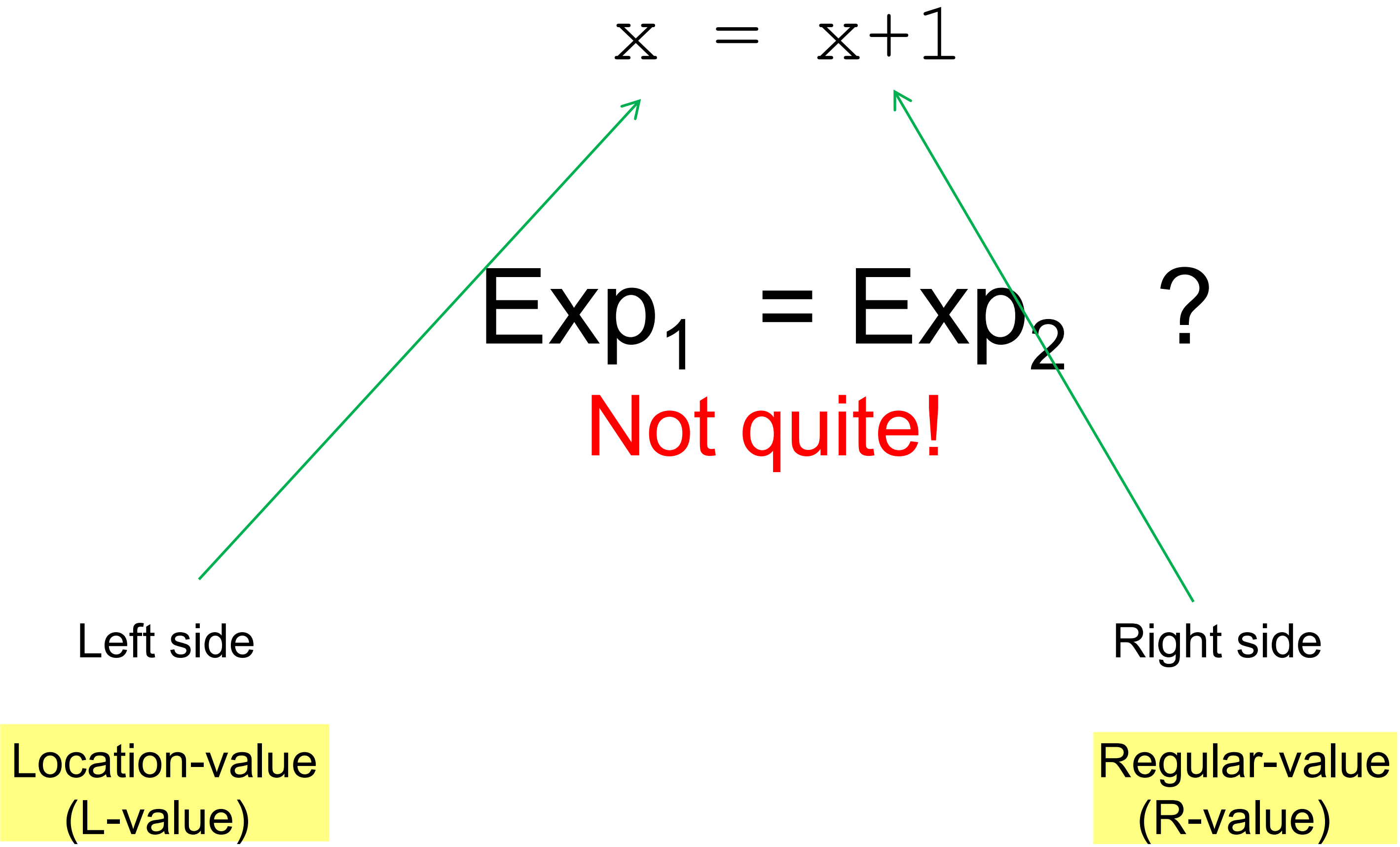
```
               i = (a>b) ? j : k
            m[i] = m[(a>b)? j : k]
m[(a>b) ? j : k] = m[i]
```

$$Exp_1 = Exp_2 \quad ?$$

Assume x is 5   x = x+1   means   5 = 6   ????

What ***exactly*** does assignment mean?

Assignment (you thought you knew)



```
x = x+1
```

$\mathrm{Exp}_1 = \mathrm{Exp}_2$?

Not quite!

Left side

Right side

Location-value
(L-value)

Regular-value
(R-value)

Assignment

- On the RHS of an assignment, use the variable's R-value; on the LHS, use its L-value
  - Example: x = x+1
  - Meaning: "get R-value of x, add 1, store the result into the L-value of x"
- An expression that does not have an L-value cannot appear on the LHS of an assignment
  - What expressions don't have l-values?
    - Examples: 1=x+1, x++ (why?)
    - What about a[1] = x+1, where a is an array? Why?

Locations and Values in Imperative Style

- When a name is used, it is bound to some memory location and becomes its identifier
  - Location could be in global, heap, or stack storage
- L-value: memory location (address)
- R-value: value stored at the memory location identified by l-value
- Assignment: A (target) = B (expression) destory
  - Destructive update: overwrites the memory location identified by A with a value of expression B
    - What if a variable appears on both sides of assignment?

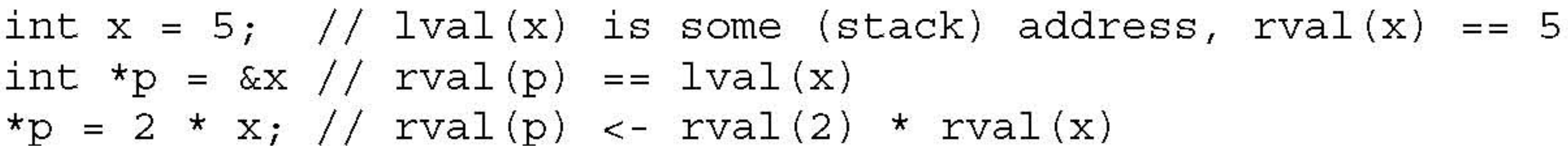
l-Values and r-Values (1)

point: l-value

- Any expression or assignment statement in an imperative language can be understood in terms of l-values and r-values of variables involved
  - In C, also helps with complex pointer dereferencing and pointer arithmetic
- Literal constants
  - Have r-values, but not l-values
- Variables
  - Have both r-values and l-values
  - Example: x=x*y means "compute rval(x)*rval(y) and store it in lval(x)"

l-Values and r-Values (2)

- Pointer variables
  - Their r-values are l-values of another variable
    - Intuition: the value of a pointer is an address
- Overriding r-value and l-value computation in C
  - &x always returns l-value of x
  - *p always return r-value of p
    - If p is a pointer, this is an l-value of another variable



```
int x = 5;  // lval(x) is some (stack) address, rval(x) == 5
int *p = &x // rval(p) == lval(x)
*p = 2 * x; // rval(p) <- rval(2) * rval(x)
```

What are the values of
p and x at this point?

Copy vs. Reference Semantics

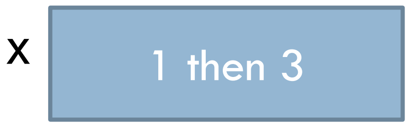
- Copy semantics: expression is evaluated to a value, which is copied to the target
  - Used by imperative languages
- Reference semantics: expression is evaluated to an object, whose pointer is copied to the target
  - Used by object-oriented languages location

Copy vs. Reference Semantics

In Java/C/C++:

```
x = 1;
x = 3;
```

Copy semantics

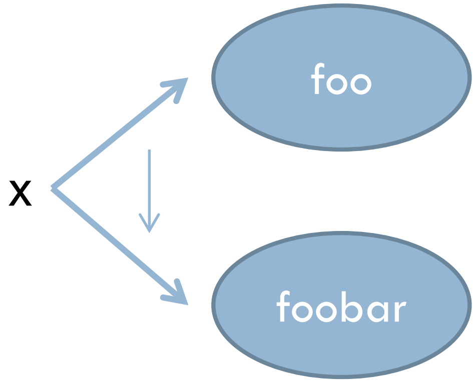


Overwrites the r-value of x from int 1 to int 3

In Java/C++/Python/Ruby:

```
x = new Foo;
x = new FooBar;
```

Reference semantics

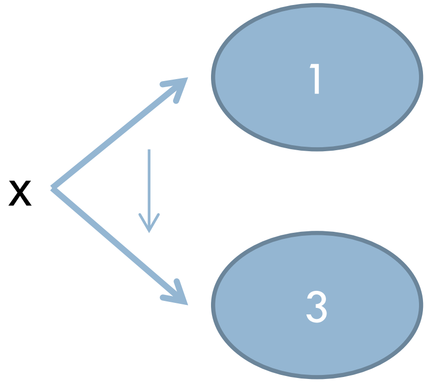


Overwrites the r-value of x too, but that value is a “pointer”

In Python/Ruby:

```
x = 1;
x = 3;
```

Reference semantics



Overwrites the r-value of x too, but that value is a “pointer”

Typed Variable Declarations

- Typed variable declarations restrict the values that a variable may assume during program execution
  - Built-in types (int, char …) or user-defined
  - Initialization: Java integers to 0. What about C?
- Variable size
  - How much space needed to hold values of this variable?
    - C on a 32-bit machine: sizeof(char) = 1 byte, sizeof(short) = 2 bytes, sizeof(int) = 4 bytes, sizeof(char*) = 4 bytes (why?)
    - What about this user-defined datatype:

Variables vs. names

- Variables: pieces of memory that hold values of a certain type; bound to names
- **Names don’t have types; values do**
- Python, Perl, Ruby:

```
x = 1
x = “hello”
```

Assignment vs. Construction

Assignment vs. Construction

Unconstrained use of assignment

```
int x;
x = 3
```

```
mylst = []
for n in range(10):
   mylst[n] = n
```

More imperative

Constrained use of assignment (construction of objects only)

```
int x = 3;
```

```
mylst = [n for n in range(10)]
```

Less imperative

The Problems With Stateful Code

- Harder to trace than stateless code
- Does not play well with concurrency