CS 536 Announcements for Wednesday, April 3, 2024

Last Time

- runtime environments
- runtime storage layout
- static vs stack allocation
- activation records
- what happens on function call, entry, return

Today

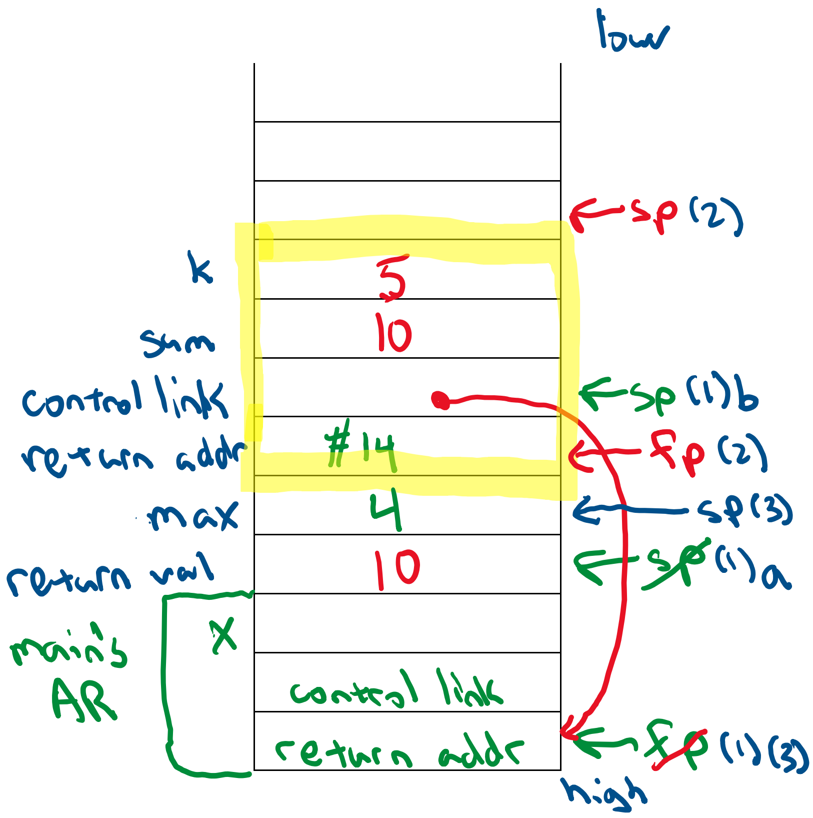
- parameter passing
- terminology
- different styles
    - what they mean
    - how they look on the stack
    - compare and contrast

Next Time

- runtime access to variables in different scopes

Example

```
#1  integer summation{integer max} [
#2     integer sum.
#3     integer k.
#4     sum = 0.
#5     k = 1.
#6     while k <= max [
#7        sum = sum + k.
#8        k++.
#9     ]
#10    return sum.
#11 ]
#12 void main{} [
#13    integer x.
#14    x = summation(4).
#15    write << x.
#16 ]
```

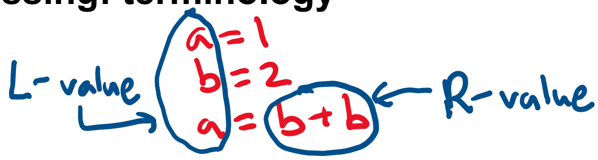


1 - caller (fctn call)
2 - callee (fctn entry)
3 - fctn exit

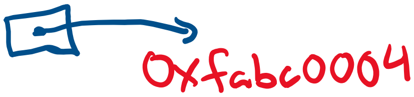
Parameter passing: terminology

R-value – value of an expression

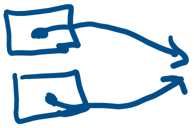
L-value – value with with a location



pointer – a variable whose value is a memory address



aliasing – when two (or more) variables hold the same address



In **definition** of function/method/procedure

```
    void f(int x, int y, bool b) { . . . }
```

formals, formal parameters, parameters

In **call** to function/method/procedure

```
    f(x + y, 7, true)
```

actuals, actual parameters, arguments

Types of parameter passing

pass by value

- when a procedure is called, the *values* of the actuals are copied into the formals

Java & C always use pass by value

C++ & Pascal can do this

pass by reference

- when a procedure is called, the *address* of the actuals are copied into the formals

C++ & Pascal can do this

C can simulate this by passing pointers

pass by value-result

- when a procedure is called, the values of actuals are passed
- when procedure is ready to return, final values of formals are copied back to the actuals

- actual must be variables (ie, have L-value), not an arbitrary expression
- used by Fortran IV & Ada (ie, not very modern)

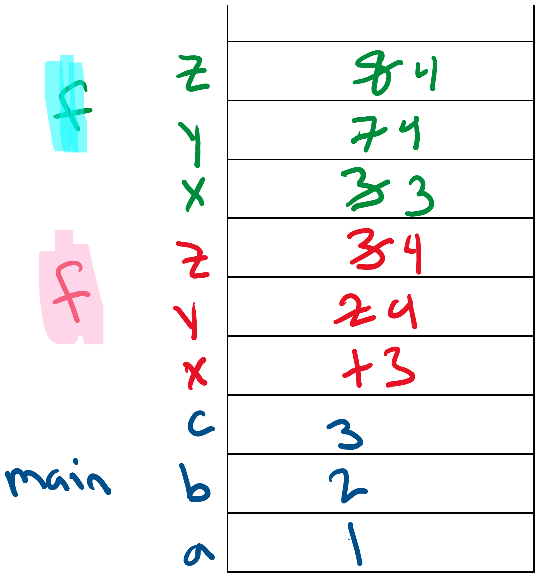
pass by name

- (conceptually) each time a procedure is called, the body of the procedure (the callee) is rewritten with the actual text of the actual parameters
- like macros in C/C++, but conceptually the rewriting occurs at runtime

- used in Algol
- hard to understand/debug

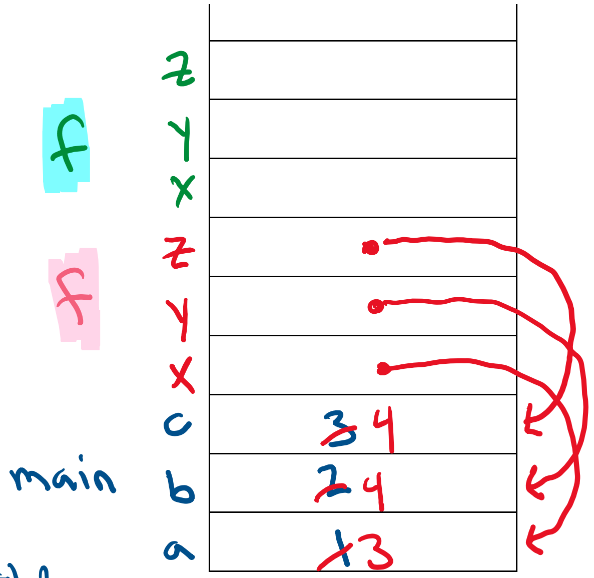
Example: pass by value

```
void f(int x, int y, int z) {
    x = 3;
    y = 4;
    z = y;
}

void main() {
    int a = 1, b = 2, c = 3;
    f(a, b, c);
    f(a+b, 7, 8);
}
```



Example: pass by reference

```
void f(int x, int y, int z) {
    x = 3;
    y = 4;
    z = y;
}

void main() {
    int a = 1, b = 2, c = 3;
    f(a, b, c);
    f(a+b, 7, 8);
}
```



error; actuals have R-values but don't have L-values, ie, don't have a location

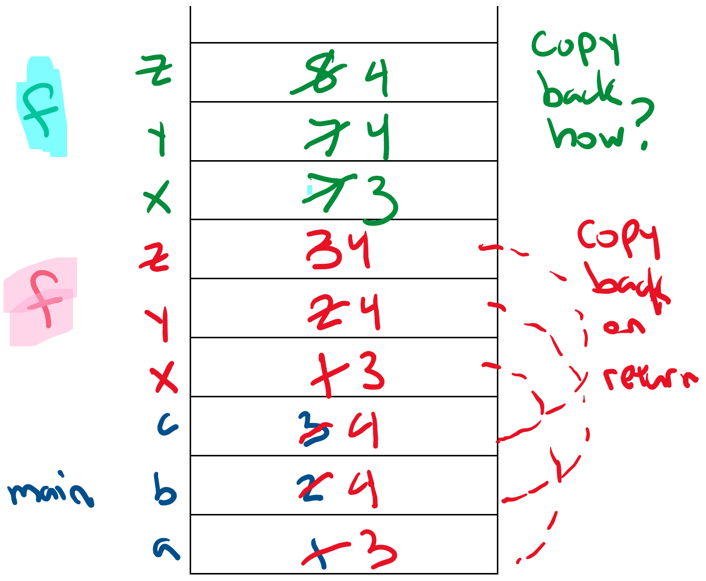
Note: typechecker would catch this error

- function type includes param passing mode for each formal
- typechecker would ensure that each actual param passed by reference has an L-value

Example: pass by value-result

```
void f(int x, int y, int z) {
    x = 3;
    y = 4;
    z = y;
}

void main() {
    int a = 1, b = 2, c = 3;
    f(a, b, c);
    f(a+b, 7, 8);
}
```

error - just like for pass by reference (caught by typechecker)



Effect: same as pass by reference unless we have aliasing

Parameter passing example

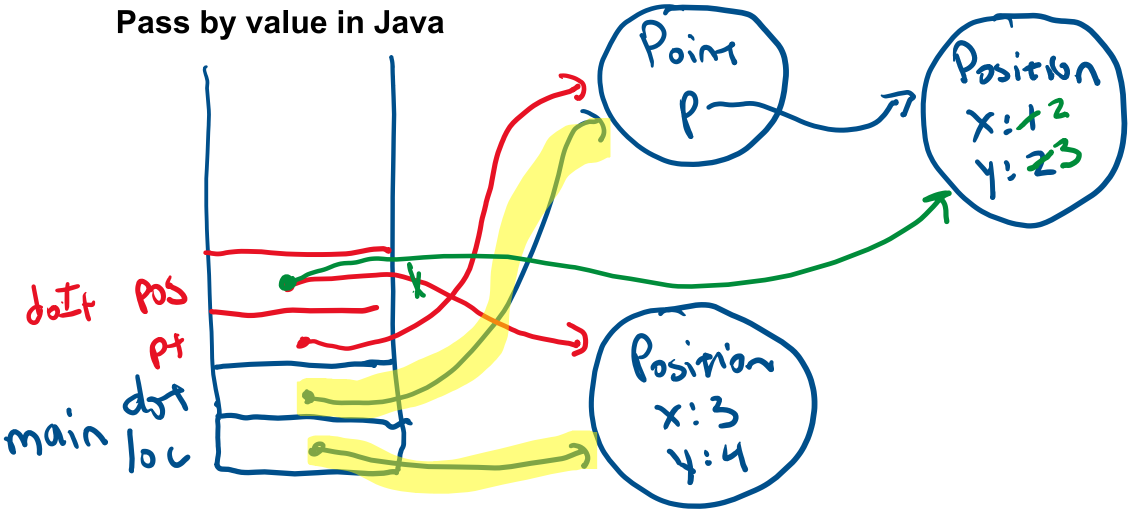
```
class Point {
     Position p;
     ...
}
class Position {
     int x, y;
     ...
}

void doIt(Point pt, Position pos) {
    pos = pt.p;
    pos.x++;
    pos.y++;
}

void main() {
    Position loc;
    Point dot;
    // code to initialize Point dot with position (1, 2)
    // code to initialize Position loc at (3, 4)
    doIt(dot, loc);
}
```

In Java, `loc` & `dot` are *references* to objects (in the heap)

In C++, `loc` & `dot` *are* objects (in the AR of `main`)

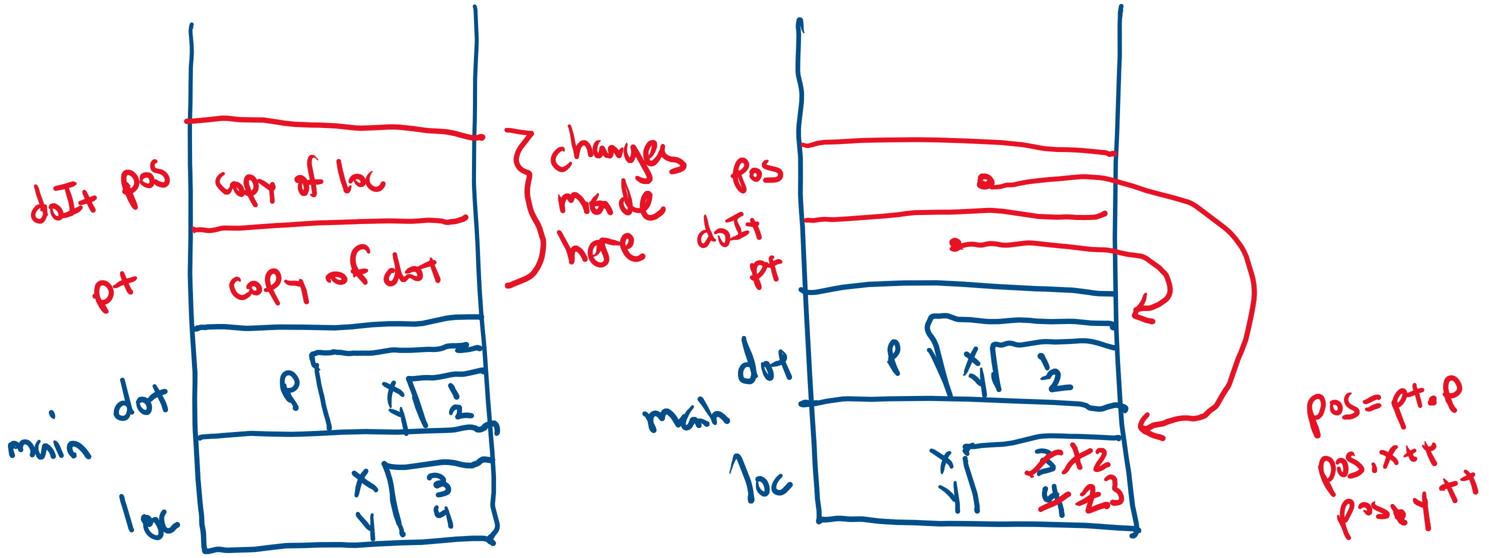
Parameter passing example (cont.)

Pass by value in Java



Pass by value in C++

Pass by reference in C++



What are the (*x*,*y*) coordinates of `dot` and `loc` after the call to `doIt`?

	Pass by value (Java)	Pass by value (C++)	Pass by reference (C++)
`dot`	(2,3)	(1,2)	(1,2)
`loc`	(3,4)	(3,4)	(2,3)

Aliasing and parameter passing

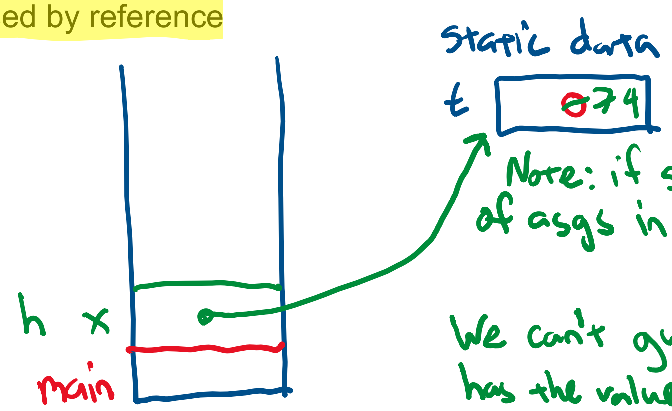
How aliasing can happen

- via pointers (in pass by value) – aliasing of actuals and formals – not really of interest here

```
      doIt(dot, loc);   // in Java
```

- when a global variable is passed by reference

```
    int t = 0;

    void h(int x) {
        x = 7;
        t = 4;
    }

    void main() {
        h(t);
    }
```

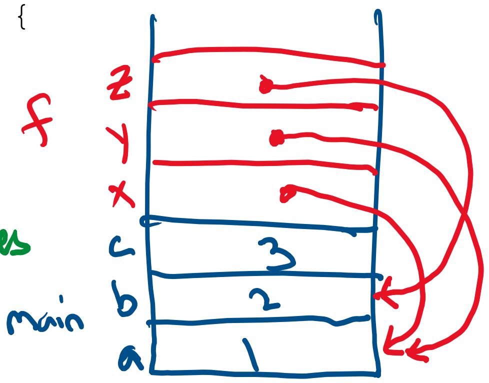


Note: if switch order of asgs in h: t=4;
x=7;
We can't guarantee t has the value 4 at end of h

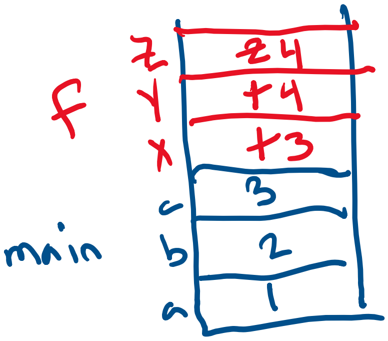
- when a parameter is passed by reference more than once

```
    void f(int x, int y, int z) {
        x = 3;
        y = 4;
        z = y;
    }

    void main() {
        int a = 1, b = 2, c = 3;
        f(a, a, b);
    }
```

x&y are aliases



What happens in pass by value-result?



When returning from f, in what order are values copied back to actuals?

Options for handling this

1) compile error
2) order defined by language (eg like Java)
3) order is implementation dependent (eg like C/C++)

Code generation and parameter passing

Efficiency considerations (calls, accesses by callee, return)

Pass by value

- copy values into callee's AR
- callee directly accesses AR locations

Pass by reference

- copy addresses into callee's AR
- access in callee via indirection

Pass by value-result

Handling objects

```
class Point {                    class Position {
    Position p;                      int x, y;
    ...                              ...
}                                }

void doIt(Point pt, Position pos) {
    pos = pt.p;
    pos.x++;
    pos.y++;
}

void main() {
    Position loc;
    Point dot;
    // ... initialize dot with position (1, 2)
    // ... initialize loc at (3, 4)
    doIt(dot, loc);
}
```

In Java, `loc` and `dot` hold the addresses of objects

In C++, `loc` and `dot` are objects in the stack

Compare and contrast

Pass by value

- no aliasing
- easier for static analysis
- called function (callee) is faster

Pass by reference

- more efficient when passing large objects
- can modify actuals

Pass by value-result

- more efficient than pass by refence for small objects
- if no aliasing, can be implemented as pass by reference for large objects