CS 536 Announcements for Thursday, April 14, 2022

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

Parameter passing: terminology

R-value – value of an expression

L-value – value 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) { . . . }

In call to function/method/procedure
f(x + y, 7, true)
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 in C 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
  - actuals must be variable, not an arbitrary expression
    - used in Fortran IV & Ada (i.e., 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

```c
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**

```c
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; i.e., don't have a location

Note: type checker would catch this error.
- Function type includes param passing mode for each formal
- Type checker would ensure that each actual param passed by ref had an L-value

Example: **pass by value-result**

```c
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.)

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

<table>
<thead>
<tr>
<th></th>
<th>Pass by value (Java)</th>
<th>Pass by value (C++)</th>
<th>Pass by reference (C++)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dot</td>
<td>((2,3))</td>
<td>((1,2))</td>
<td>((1,2))</td>
</tr>
<tr>
<td>loc</td>
<td>((3,4))</td>
<td>((3,4))</td>
<td>((2,3))</td>
</tr>
</tbody>
</table>
Aliasing and parameter passing

How aliasing can happen

- via pointers (in pass-by-value) – aliasing of actuals and formals
  
  ```java
  doit(dot, loc);  // in Java
  ```

- when a global variable is **passed by reference**
  
  ```java
  int t = 0;
  void h(int x) {
    x = 7;
    t = 4;
  }
  void main() {
    h(t);
  }
  ```

- when a parameter is **passed by reference more than once**
  
  ```java
  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);
  }
  ```

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 is defined by the language (e.g., like Java)
3) order is implementation-dependent (e.g., like C/C++)
Code generation and parameter passing

Efficiency considerations (calls, accesses by callee, return)

Pass by value

- copy values into callee’s AR (slow)
- callee directly accesses AR locations (fast)

Pass by reference

- copy address into callee’s AR (fast)
- access in callee via indirection (slow)

Pass by value-result

- strictly slower than pass by value
  - need to know where to copy values back on return

Handling objects

```java
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;
    // ... initialize dot with position (1, 2)
    // ... initialize loc at (3, 4)
    doIt(dot, loc);
}
```

In Java, `loc` and `dot` hold the addresses of objects
- no overhead of copying entire object - just the address

In C++, `loc` and `dot` are objects in the stack
- use pointers to objects in heap for efficiency
Compare and contrast

Pass by value

- no aliasing - fewer unwanted side effects
- easier for static analysis (esp. optimization)
- called function (callee) is faster - no indirection
- but the call (& copying the values) may take time

Pass by reference

- more efficient when passing large objects
- can modify actuals

- const ref in C++ - pass by ref but not allowed to be modified - compiler checks & gives warning/error

Pass by value-result

- more efficient then pass by reference for small objects (no indirection)
- if no aliasing, can be implemented as pass by reference for large objects (so still efficient)
- BUT determining *if* there is aliasing (and *what* is aliased) is a challenging task (in general)