CS342: Organization of Programming Languages

Topic 17: Parameter Passing Semantics

- Overview
- Call by Value
- Multile Value Return
- Call by Reference
- Call by Value-Result
- The Sneaky Loop-Hole
- This and That
- In and Out Parameters
- Call by Name

Overview

- How do parameter values get passed to a function/method/subroutine?
- How to values get returned to the caller?
- Which variables and data structures get modified?

Call by Value

- "Call by Value" causes *copies* of the values of the arguments to be transmitted from the caller to the callee.
- The data may be passed on a call stack, in registers, or by some other mechanism.

```
void f(int k) { k = 7; }
void g(int n) { int i = n; f(i); }
```

- In the example above, the value of *i* is not changed by the call to *f*.

Multiple Value Return

- If many results are requrired, must use some way to get them back.
- One way is to use global variables for 2nd, 3rd etc results.
- One way is to have language support for returning multiple values:

```
scanInteger(String s,  int start):
            (int value, int nextStart)
{
   ...
}
```

This is supported by sl values and *call-with-values* in Scheme

```
(define swap (lambda (a b) (values b a))
(call-with-values (lambda () (swap 3 4)) /)  ; gives 4/3
```

- Another way is to use "call by reference" parameters.

Call by Reference

- Pass the address of where the data value is stored.
- Allows value to be read and written.

```
void f(int k, int h) { k = 7; print(h); }
void g(int n) { int i = n; f(i,i); }
```

- Causes *i* in the above program to become 7.
- Some languages use this as the default/only parameter passing mechanism.
- Can give surprising results due to *aliasing*, when two parameters reference the same thing. The function *f* prints 7 above.
- Leads to awkward semantics when values are passed, e.g. *f(2)* or *f(i+j)*.
- Can be specified in C++ using the `&` syntax

```
void f(int & n, int &m) { n = m;  }
```

- Can be simulated in C using pointers

```
void f(int *n, int *m) { *n = *m; }
```

Call by Value-Result

- All the "benefits" of call by reference, but cleaner semantics.
- At function invocation, the values of the arguments are copied into the parameters.
- At the function return, the new values of the parameters are copied back into the given arguments, if applicable.
- This is sometimes used with inter-language interfaces between languages with call by value and call by reference.

```
int i;
void f(int k) { k = 7; print(i); }
void g(int n) { i = 8; f(i); }
```

- The call to *f* will print 8, and the value of *i* afterward will be 7.
- This is particularly helpful (compared to call by reference) when using lexically nested functions, and the inner functions can see the outer functions' variables.

The Sneaky Loop-Hole

- Some languages use call by value, but some of the "values" are pointers.
- Compare:

```
struct s { int i, j; } sv = { 1, 3};

void f(struct s a, struct s *b) { ... }
```

  The first argument is a structure passed by value, the second is a pointer to a structure passed by value.

- Languages allow this primarily for efficient passing of big objects.
- A feature/trap is that the objects can be updated by the callee.
- Java claims to hide pointers from the user and also to have call by value.
  What do you think?

This and That

- Methods in OO languages have an implicit reference parameter, often referred to as "this".
- Binary operations are now artificially asymmetric.

```
// Do you like this ...

Complex operator +=(Complex x) {
    return Complex(real + x.real, imag + x.imag);
}

// or that ... ?

Complex* operator +(Complex* that) {
    return new Complex(this->real + that->real,
                       this->imag + that->imag);
}
```

In and Out Parameters

- Some languages, like Ada and CORBA IDL, allow parameters to be declared as “in”, “out”, or “in out”.
- The “in” parameters are normally *call by value*.
- The “out” parameters are nomrally *call by result*.
- The “in out” parameters are normally *call by value/result*.

Call by Name

- Default in Algol 60. Jensen's device:

```
real procedure SIGMA(x, i, n);
   value n;  real x; integer i, n;
begin
   real s; s := 0;
   for i := 1 step 1 until n do s := s + x;
   SIGMA := s;
end

SIGMA(a(i), i, 10)
SIGMA(a(i) * b(i), i, 10);
```

- Elegant idea, but inefficient and surprising.
- Idea: parameters are implicitly converted to nullary closures and called at the points they are used.
- Not the same as usual lazy evaluation because the parameter expressions are re-evaluated at each use.
- If a variable is passed, it is equivalent to call by reference.
- If a constant is passed, it is equivalent to call by value.