Programming Languages: The first law of computer science: Every problem is solved by yet another indirection

Professor Keith Sullivan

Parameter Passing Mode

- Passing information from the call site to the function.
- Parameter Passing Mode tells us how the information is communicated from the call site to the function.

Example

```
int foo1_global;
void foo1() { foo1_global = foo1_global * 2; }

int foo2(int in) { return in * 2; }

void foo3(int& inout) { inout *= 2;}

int main() {
  int x=1, y=2, z=3;

  foo1_global = x;
  foo1();
  cout << foo1_global << endl;

  cout << foo2(y) << endl;

  foo3(z);
  cout << z << endl;
```

Pass by Value

- Function receives a copies of the arguments
- Function cannot modify the originals, and copies go out of scope when function returns
- Arguments are one-way communication from call site to function
  - Can the function communicate back?
- C/C++ use pass by value by default
- Java uses it for primitive types

Pass by Reference

- The formal parameters of the function become aliases for the arguments
- Function arguments now represent two-way communication
- Can cause confusion and introduce difficulties for the compiler:

```
int a, b, *p, *q;

a = *p;
*q = 3;
b = *p;
```

Variations

- Pass by Value/Result
  The initial value is passed as a copy, and the final value on return is copied back to the actual parameter. Behaves like pass by reference unless the actual parameter is accessed during the function call.

```
int x = 1;

void f(int & a)
{
  a = 2;
  x = 0;
}

main()
{
  f(x);
  cout << x << endl;
}
```

Variations

- Pass by Sharing
  Actual and formal parameters both reference some shared data. But they are not aliases; functions can change the object that is references by cannot set which object is referenced.

Pass by Sharing

```
class Share {
  static class Small {
    public int x;
    public Small(int thex) { x = thex; }
  }

  public static void test(Small s) {
    s.x = 10;
    s = new Small(20);
  }

  public static void main(String[] args) {
    Small mainsmall = new Small(5);
    test(mainsmall);
    System.out.println(mainsmall.x);
  }
}
```

Argument evaluation

Question: When are function arguments evaluated?

There are three common options:

- Applicative order: Arguments are evaluated *just before the function body is executed.*
  This is what we get in C, C++, Java, and even SPL.
- Call by name: Arguments are evaluated *every time they are used.*
  (If they aren't used, they aren't evaluated!)

Lazy Evaluation

(A.K.A. *normal order evaluation*)

Combines the best of both worlds:

- Arguments are not evaluated *until they are used*.
- Arguments are only evaluated *at most once*.

(Related idea to *memoization*.)

Why not use lazy evaluation everywhere? Why doesn't C++ use it?

What about functional languages?

Method calls in objects

What does a call like *obj.foo(x)* do in an OOP language such as C++ or Java?

foo must be a method defined in the class of *obj*.
The method also has access to what object it was called on (called *this* in C++ and Java).

This is syntactic sugar for having a globally-defined method *foo*, with an extra argument for "*this*".
So we can think of *obj.foo(x)* as *foo(obj,x)*.

Overloading

Function overloading: one name, many functions.
Which function to call is determined by the *types* of the arguments.

```
class A { void print() { cout << "in␣A" << endl; } };
class B { void print() { cout << "in␣B" << endl; } };

void foo(int a) { cout << a << "␣is␣an␣int\n"; }
void foo(double a) { cout << a << "␣is␣a␣double\n"; }

int main() {
  cout << (5 << 3) << endl;
  A x;
  B y;
  x.print();
  y.print();
  foo(5);
  foo(5.0);
}
```

How does overloading occur in this C++ example?

Quirk of C++

```
struct Point {
  int x;
  int y;
};

Point operator+ (Point a, Point b) {
  Point result;
  result.x = a.x + b.x;
  result.y = a.y + b.y;
  return result;
}

int main() {
  Point p1, p2;
  /* ... */
  Point p3 = p1 + p2;
  int x = 1 + 2;
}
```

Polymorphism

Subtype polymorphism is like overloading, but the called function depends on the object's *actual type*, not its declared type!

Each object stores a *virtual methods table* (vtable) containing the address of every virtual function.

This is inspected at run-time when a call is made.

Polymorphism Example

```
class Base { virtual void aha() = 0; };

class A :public Base {
  void aha() { cout << "I'm an A!" << endl; }
};

class B :public Base {
  void aha() { cout << "I'm a B!" << endl; }
}

int main(int argc) {
  Base* x;
  if (argc == 1 )
    x = new A;
  else
    x = new B;
  x.aha(); // Which one will it call?
}
```

Macros

```
int y = 10;

#define X (y + 2)

void foo(int y) {
  cout << X << endl;
}

int main() {
  cout << X << endl;
  y = y * 20;
  cout << X << endl;
  foo(50);
}
```

Constant Macros

```
#define PI 3.14159
#define TWOPI PI + PI

double circum (double radius)
{
  return TWOPI * radius;
}
```

Function Macros

```
#define CIRCUM (radius) 2*3.14159*radius

...
cout << CIRCUM(1.5) + CIRCUM(2.5) << endl;
...
```

Why the extra parentheses here?

```
#define DIVIDES (a, n) (!((n) % (a)))
```

Macros

- The advantage is SPEED - pre-compilation!
- Notice: no types, syntactic checks, etc.
  — *lots of potential for nastiness!*
- The literal text of the arguments is pasted into the function wherever the parameters appear.
  This is called call by name
- The *inline* and *constexpr* keywords in C++ are compiler suggestions that may offer a compromise.
- Scheme has a very sophisticated macro definition mechanism
  — allows one to define "special forms".