## Parameter Passing Modes

Our programs are littered with *function calls* like $f(x, 5)$.

This is a way of *passing information* from the *call site* (where the code `f(x,5)` appears) to the function itself.

The *parameter passing mode* tells us *how* the information about the arguments (e.g. $x$ and 5) is communicated from the call site to the function.



## Pass by Value

C/C++ use pass by value by default.
Java uses it for *primitive types* (`int`, `boolean`, etc.).

```
void f(int a) {
  a = 2*a;
  print(a);
}

int main() {
  int x = 5;
  f(x);
  print(x);
  return 0;
}
```

What does this C++ program print?



## Pass by Value

In this scheme, the function recieves *copies* of the actual parameters.

The function cannot modify the originals, only its copies, which are destroyed when the function returns.

Function arguments represent *one-way communication* from call site to function.
(How can the function communicate back?)

## Pass by Reference

C++ supports *reference parameters*.
Perl and VB use this mode by default.

```
sub foo {
  $_[0] = "haha changed by foo";
}

my $y = "this is mine!";
foo($y);
print $y, "\n";
```

You can guess what this Perl program prints...

Similar behavior happens if we wrote `void f(int &a) {...}` in the previous C++ example.



## Pass by Reference

The *formal parameters* of the function become *aliases* for the actual parameters!
Normally the actual parameters must be variable names (or more generally, *l-values*...).

Function arguments now represent *two-way communication*.

Most common reasons to use reference parameters:



## Variations

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

- **Pass by Sharing**
  This is what happens with objects in Java.
  Actual and formal parameters both reference some *shared* data (i.e., the object itself).
  But they are not aliases; functions can change the object that is referenced, but cannot set *which* object is referenced.

## Pass by Value/Result

This is the default in Fortran, and for "in out" parameters in Ada:

```
--in file f.adb
procedure f(x : in out Integer) is
begin
  x := 10;
end f;

--in file test.adb
procedure test is
  y : Integer := 5;
begin
  f(y);
  Ada.Integer_Text_IO.Put(y);
end test;
```

Calling test prints 10, not 5!



## Pass by Sharing

This is the default in languages like Java, for non-primitive types:

```
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);
  }
}
```

Does this program print 5, 10, or 20?



## Parameter Passing in Functional Languages

Why haven't we talked about parameter passing in Haskell, Scheme, etc.?

## 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!)
- **Normal order**: next slide...



## 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*.)



## Lazy Examples

Note: lazy evaluation is great for functional languages (why?).

- Haskell uses lazy evaluation for *everything*, by default. Allows wonderful things like infinite arrays!
- Scheme lets us do it manually with *delayed evaluation*, using she *built-in special forms* `delay` and `force`.

# 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?



# 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.

See section 9.4 in your book if you want the details.

## Polymorphism example in C++

```
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 ) // can't know this at compile-time!
    x = new A;
  else
    x = new B;
  x.aha(); // Which one will it call?
}
```



## Different kinds of functions

The code `f(5)` here is definitely a function call:

```
int f(int x) { return x + 6; }

int main() {
  cout << f(5) << endl;
  return 0;
}
```

- *What else is a function call?*



## Operators

Say we have the following C++ code:

```
int mod (int a, int b) {
  return a - (a/b)*b;
}
```

What is the difference between
`23 % 5`
and
`mod(23, 5)`

## Are Operators Functions?

It's language dependent!

- **Scheme**: Every operator is clearly just like any other function.
  Yes, they can be re-defined at will.
- **C/C++**: Operators are functions, but they have a *special syntax*.
  The call `x + y` is *syntactic sugar* for either `operator+(x, y)` or `x.operator+(y)`.
- **Java**: Can't redefine operators; they only exist for some built-in types.
  So are they still function calls?



## Built-ins

A *built-in function* looks like a normal function call, but instead makes something special happen in the compiler/interpreter.

- Usually system calls are this way.
  C/C++ are an important exception!
- What is the difference between a built-in and a library function?



## Macros

Recall that C/C++ has a *preprocessor* stage that occurs before compilation.
These are the commands like `#include`, `#ifndef`, etc.

`#define` defines a *macro*. It corresponds to textual substitution *before* compilation.

## Constant Macros

Here's an example of a basic macro that you might see somewhere:

The program

```
#define PI 3.14159

double circum (double radius)
{ return 2*PI*radius; }
```

gets directly translated by the preprocessor to

```
double circum (double radius)
{ return 2*3.14159*radius; }
```

*before compilation*!

Unit 7
SI 413
Parameter Passing Modes
Parameter Evaluation
Overloading and Polymorphism
Operators, Built-ins
Macros

## Macro Issues #1

What if we wrote the last example differently:

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

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

Unit 7
SI 413
Parameter Passing Modes
Parameter Evaluation
Overloading and Polymorphism
Operators, Built-ins
Macros

## Function-like Macros

We can also do things like this in C++:

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

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

gets translated to

```
...
cout << 2*3.14159*1.5 + 2*3.14159*2.5 << endl;
...
```

(still *prior to compilation*)

## Macro Issues #2

What if we made the following function to print out the larger number:

```
#define PRINTMAX (a,b) \
  if (a >= b) {cout << a << endl;} \
  else {cout << b << endl;}
```

This will work fine for `PRINTMAX(5,10)`,
but what happens with the following:

```
int x = 5;
PRINTMAX(++x, 2)
```



## Thoughts on 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 . . .
- The `inline` keyword in C++ is a compiler suggestion that may offer a compromise.
- Scheme has a very sophisticated macro definition mechanism — allows one to define "special forms".



## Class Outcomes

You should know:

- The way parameter passing works in pass by *value*, by *reference*, by *value/result*, and by *sharing*
- Relative advantages of those parameter passing modes
- The way arguments are evaluated under *applicative order*, *normal order*, and *call by name*
- Why lazy evaluation (normal order) can be terrific
- What *function overloading* is and where it gets used
- What *subtype polymorphism* is and how *virtual tables* are used to implement it
- Differences between *operators*, *built-ins*, *library routines*, and *user-defined functions*
- What *constant macros* and *function-like macros* are, and what are their advantages and drawbacks.