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*.

References provide a way for functions to return multiple things.

They also avoid the (potentially expensive) overhead of copying without having to deal with pointers.
This is especially useful with (large) data structures stored in objects.

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.?

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 is when the same name refers to 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

Sybtype 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?
}
```


Class outcomes

You should know:

- The differences between the four parameter passing modes we have discussed
- How class method calls are typically interpreted by compilers
- How overloading works with operators, classes, and functions
- What subtype polymorphism is and its benefits and drawbacks

You should be able to:

- Follow and create programming examples demonstrating pass by value vs. pass by reference