Languages and Compiler Design II
Parameter Passing

Material provided by Prof. Jingke Li
Stolen with pride and modified by Herb Mayer
PSU Spring 2010
rev.: 4/19/2010
Agenda

• Parameter Classification
• Call by Value
• Call by Reference
• Call by Copy-Restore
• Call by Name
Parameter Classification

**Basic question concerning parameter passing:**

- For each parameter, what exactly is passed from the caller to callée — the parameter’s address? its content? or something else?

**Various parameter passing methods:**

- call by value: passing r-values
- call by reference: passing l-values
- call by copy-restore: hybrid between by-value and by-reference; AKA Call by Value-Result
- call by name: passing via name substitution; is rarely used; first designed for Algol-60
Call by Value

• Each actual argument is evaluated before call. On entry, the resulting value is copied and bound to the formal parameter; which behaves just like a local variable

• Advantages:
  – Simple; easy to understand!

• Formal parameters can be used as local variables, Updating them doesn’t affect actuals in calling procedure:

```c
double hyp( double a, double b ) {
    a = a * a;
    b = b * b;
    return sqrt( a + b ); // use built-in sqrt()
} //end hyp
```
Problem with Call by Value

Can be inefficient if value is large:
```c
typedef struct { double a1, a2, ..., a10; } vector;
double dotp( vector v, vector w ) {
    return v.a1 * w.a1 + v.a2 * w.a2 + ... + v.a10 * w.a10;
} //end dotp
vector v1, v2;
double d = dotp( v1, v2 ); // copy 20 doubles
```

Cannot affect calling environment directly:
```c
void swap( int i, int j ) {
    int temp;
    temp = i; i = j; j = temp;
} //end swap
swap( a[p], a[q] ); // has no effect!
```

Can return result only through (the single) return value
Call by Reference

*Implemented by passing address of actual parameter:*

- On entry, the formal is bound to the address, providing a reference to actual parameter from within the subroutine.
- If actual argument doesn’t have an l-value (e.g., “2 + 3”), then either:
  - Forbid in language, i.e. treat as an error; compiler catches this.
  - Evaluate it into a temporary location and pass its address; but what will be the modified value of actual after the call?

- Advantages:
  - No more large copying.
  - Actual parameter can be updated — Now swap, etc., work fine!
Problems with Call by Reference

• Accesses are slower: the formal parameter is an address that must be dereferenced to get at the value

• Opportunity for aliasing problems, e.g.,

```
procedure MatrixMult( a, b, c: matrix )
... sets c := a * b;
MatrixMult( a, b, a ) // oops!
```

• Question: Can we combine the simplicity of call by value with the efficiency for large values of call by reference?
C and C++ Solution

- C uses call by value, except for arrays. Structs (records in other languages) pass actual contents, while arrays are passed by address (pointer). Arrays do not get copied when passed as parameters.

- Separately, programmers can take the address of a variable explicitly, and pass that to obtain call by ref-like behavior.

```c
swap1( int *a, int *b ) { // pointer value params, NOT ref
  int t; t = *a; *a = *b; *b = t;
} //end swap1
swap1( &a[p], &a[q] );
```

- C++ inherits the above properties of C. Furthermore, it supports call by reference parameters through the use of reference variables:

```c
swap2( int &a, int &b ) { // genuine reference parameter
  int t; t = a; a = b; b = t;
} //end swap2
swap2( a[p], a[q] );
Swap2( i, j );
```
Pascal and Ada Solution

• In Pascal, Ada, and similar languages, structures and arrays can be passed by value (copying the whole contents) or by reference (passing only the address):
  – **in** parameter: is for passing value of the actual into the subroutine
  – **out** parameter: is for passing value of the formal parameter back to the actual
  – **in out** parameter: is for passing value into as well as out from the subroutine

• The idea is to let the programmer decide, which parameter passing method to use for each parameter

• Later versions of Ada, fix the implementation choice:
  – Scalar types are passed by value, if an **in** parameter, or copy-restore, if an **out** or **in-out** parameter; AKA value-result parameter
  – Composite data types are passed by reference
Java Solution

• Java has only call by value mode. However, in Java, a variable representing an object is a reference to the object, instead of its content. Note that a variable always points to actual heap space.

• When object is passed as a parameter, only its reference is copied.

• This reference model of variables also provides some call by reference-like benefits — when an object is passed as an actual parameter, its content can be updated in the subroutine:

```java
class A { int i = 0; ... }
class B {
    void sub( A a ) { a.i = 5; }
    void test() {
        A x = new A();
        x.i = 4;
        sub( x );
        System.out.println( x.i ); // prints 4? Or 5?
    }
}
```
Difference to Call by Reference?

Consider:

```java
class A { int i = 0; ... }
class B {
    void sub( A a ) {
        a = new A(); // parameter a gets modified here
        a.i = 5;
    } //end sub
    void test() { // <- start here <-
        A x = new A();
        x.i = 4;
        sub( x );
        System.out.println( x.i ); // prints 4? Or 5? Or??
    } //end test
} //end B
```

Updating formal parameter affects only the local copy
Call by Copy-Restore

- Each actual argument is evaluated to a value before call
- On entry, value is bound to formal parameter just like a local
- Updating formal parameters doesn’t affect actuals in calling procedure during execution
- Upon exit, the final contents of formals are copied into the actuals
- Thus, behaves like call by reference in most “normal” situations, but may give different results when concurrency or aliasing are involved:

```plaintext
type t is record a, b: integer; end record;
r : t;

procedure foo( s : in out t ) is
  begin r.a := 2; s.a := s.a + 3;
end foo;
r.a := 1;
foo( r );
print( r.a ); -- what’s the value of r.a?
```
Test: Reference or Copy-Restore?

```pascal
var a : integer := 12;
...
procedure trick( var x : integer )
begin -- trick
  x := 13;
  if x = 13 and a = 13 then
    writeln( "pass by reference." );
  elsif x = 13 and a = 12 then
    writeln( "pass by copy-restore." );
  else
    writeln( "mysterious new method." );
  end if;
end trick;
...
trick( a );
```
Call by Name

• First introduced in Algol-60, first implemented via “thunk” by Ingerman

• Idea derived from substitution or macro-expansion. In C and some other languages, macros are used to define simple functions. The user interface of a macro resembles that of a function.

```c
#define max( a, b ) ((a)>(b) ? (a) : (b))

v = max( x, y );
```

• When a macro is invoked, each formal parameter is literally replaced by corresponding actual parameter; string substitution:

```c
max( x+1, y-2 ) => ( (x+1) > (y-2) ? (x+1) : (y-2))
```

• However, macros and functions are fundamentally different:
  – Macros are invoked at compile-time (or pre-compile-time), functions are called at runtime
  – Macros can not contain recursions
Call by Name, Cont’d

• But blind substitution is dangerous because of possible “variable capture”
  ```c
  #define swap(x, y) { int t; t = x; x = y; y = t; }
  ...
  swap(a[t], a[q]);
  ```

• the call expands to
  ```c
  { int t; t = a[t]; a[t] = a[q]; a[q] = t; }
  ```

• Here “t is captured” by the declaration in the macro, and is undefined at its first use

• Call by name is actually “substitution with renaming where necessary”
  – Flexible, but potentially very confusing, and inefficient to implement; requires small “.dll” or piece of code to evaluate actual parameter
  – If language has no updatable variables (as in “pure” functional languages), call by name renders beautifully simple semantics for function calls