CS322 Languages and Compiler Design II

Spring 2012

Lecture 9

Procedure Parameter Passing

```
func f(x:integer,y:integer) { ... };
func g(z:@integer,q:@integer) { ... };
var a:@integer := ...;
var w:integer := ...;
.
.
.
f(3,w); ... g(a,a); ... f(17+5,a[3]);
.
.
.
```

- Do we pass addresses (**l-values**) or contents (**r-values**) of variables?
- How do we pass actual values that aren't variables?
- What does it mean to pass an aggregate value like an array?

Call-by-Value (i.e., r-value)

- Each actual argument is **evaluated** to a **value** before call.
- On entry, value is **bound** to formal parameter just like a local variable.
- Updating formal parameter doesn't affect actuals in **calling** procedure.

Example (C) :

```
double hyp(double a,double b) {
  a = a * a;
  b = b * b;
  return sqrt(a+b);
}
r = hyp(a,b); /* values of a,b don't change in caller */
```

- Simple; easy to understand!
- Implement by copying.

Problems with Call-by-Value

- Can be inefficient if value is large.

Example (C): Calls to `dotp` copy 20 doubles:

```
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;
}
vector v1,v2;
double d = dotp(v1,v2);
```

- Cannot affect calling environment directly. (Often a **good** thing, but not always!)

Example: calls to `swap` have no effect:

```
void swap(int i,int j) {
  int t = i ; i = j; j = t;
}
swap(a[p],a[q]); /* contents of array a are unchanged */
```

- Can at best **return** only one result, though this might be a record.

Call-by-Reference (i.e., l-value)

• Pass the **address** (l-value) of each actual parameter.

• On entry, the formal is bound to the address, which must be dereferenced to get value, but can also be **updated**.

• If actual argument doesn't have an l-value (e.g., "`2 + 3`"), either:

- Evaluate it into a temporary location and pass address of temporary, or

- Treat as an error.

• Now `swap`, etc., work fine!

• Accesses are slower: parameters must remain in memory.

• Lots of opportunity for **aliasing** problems, e.g.,

```
func matmult(a,b,c: @@real)
... [* sets c := a * b *]

matmult(a,b,a) [* oops! *]
```

• **Call-by-value-result** (a.k.a. **copy-restore**) addresses this problem, but has other drawbacks.

Hybrid Methods

How might we combine the simplicity of call-by-value with the efficiency of call-by-reference, especially for large values like records and arrays?

Answer depends on what a record or array **r-value** is in a particular language. (This is also important for the semantics of assignment.)

• In Pascal, Ada, and similar languages, r-values of both arrays and records are the actual contents. So passing a record or array by value means copying the contents, whereas passing by reference doesn't. These languages let the programmer specify (in the procedure header) the method to use on each parameter.

• C always uses call-by-value, but programmers can take the l-value of a variable explicitly, and pass that to obtain cbr-like behavior:

```
swap(int *a, int *b) { int t = *a; *a = *b; *b = t; }
swap(&a[p],&a[q]);
```

Of course, it is the programmer's responsibility to make sure that the l-value remains valid (especially when it is **returned** from a function).

C/C++ Records

In ANSI C/C++, record (`struct` or `class`) r-values are the actual contents.

To avoid copying C structures, must use pointers:

```
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;
}
vector v1,v2;
double d1 = dotp(&v1,&v2);
```

C/C++ Arrays

On the other hand, C/C++ array r-values are **pointers** to the contents. In this example, no doubles are copied on call:

```
typedef double vector[10];
double dotp(vector v, vector w) {
   double d = 0.0; int i;
   for (i = 0; i < 10; i++)
     d += v[i] * w[i];
   return d;
}
vector v1,v2;
double d1 = dotp(v1,v2);
```

Complex and Simple Solutions

• C++ supports both call-by-reference parameters and explicit pointers:

```
swap(int &a, int *b) {
  int = a; a = *b; *b = t;
}
...
swap(a[p],&a[q]);
```

Mixing explicit and implicit pointers can be **very** confusing!

• In Java and fab, r-values of both records (objects) and arrays are **pointers** to the actual contents, which are held in the heap. These languages have only call-by-value, but this doesn't actually cause copying, even for record or array values.

• Approach is made more feasible because programmer doesn't have to worry about lifetime of heap data, due to automatic garbage collection.

• Clever compilers can decide whether smallish objects should be heap-allocated or manipulated directly.

Substitution

• Can often get the effect we want using **substitution**, i.e., **macro-expansion**, e.g (in C):

```
#define swap(x,y) {int t; t = x; x = y; y = t;}
...
swap(a[p],a[q]);
```

• **BUT** blind substitution is dangerous because of possible "**variable capture**," e.g.,

```
swap(a[t],a[q])
```

expands to

```
{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

• Really want "substitution with renaming where necessary" = Algol-60's **call-by-Name** facility.

• Flexible, but potentially very confusing if language supports mutation of variables.

• But if language has no mutable variables (as in "pure" functional languages such as Haskell), substitution gives a beautifully simple semantics for procedure calls.

• Inherently less efficient than call-by-value or call-by-reference, because much more than a simple value or address needs to be transmitted at calls.