Procedures and Functions

- Subroutines: avoid duplicating frequently used code
- Procedural abstraction: divide programs into named components with hidden internals
- Provide framework for managing local data, especially in recursive programs

Invoking a procedure generates a **procedure activation**, which has associated data such as:

- the **return address** of the caller
- the **actual** values corresponding to the **formal** parameters of the procedure
- space for the values of **local variables** associated with the procedure.

The data associated with each activation is independent from that for all other activations, so many activations can exist at once, as in a recursive routine.

In most languages, activation data can be stored on a **stack**, and we speak of pushing and popping activation **frames** from the stack, which is a very efficient way of managing local data.

---

Procedure Parameter Passing

```
TYPE IARRAY IS ARRAY OF INTEGER;
PROCEDURE f(x:INTEGER,y:INTEGER) IS ...
PROCEDURE g(z:IARRAY,q:IARRAY) IS ...
VAR a: IARRAY, 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.

```
double hyp(double a,double b) {
    a = a * a;
    b = b * b;
    return sqrt(a+b);
}
```

- 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

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

- Cannot affect calling environment directly. (Of course, perhaps this is a good thing!)

Example: calls to `swap` have no effect:

```c
void swap(int i,int j) {
    int t;
    t = i; i = j; j = t;
} ...
swap(a[p],a[q]);
```

- Can at best return only one result (as a value), though this might be a record.

Hybrid Methods; Records and Arrays

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, of course.)

- 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 permit the programmer to specify (in the procedure header) which method to use on each parameter.

- In ANSI C/C++, record (struct or class) r-values are the actual contents, although array r-values are pointers to the contents. 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:

```c
void swap(int *a, int *b) {
    int t;
    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).

Complex and Simple Solutions

- C++ supports both cbr parameters and explicit pointers:

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

Mixing explicit and implicit pointers can be very confusing!

- In Java and ML, 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.

- Really want “substitution with renaming where necessary” = call-by-name (first proposed in Algol60).

- Flexible, but potentially very confusing, and inefficient to implement.

- If language has no updatable variables (as in “pure” functional languages), substitution gives a beautifully simple semantics for procedure calls.

**Exceptions**

Programs often need to handle exceptional conditions, i.e., deviations from “normal” control flow.

Exceptions may arise from

- failure of built-in or library operations (e.g., division by zero, end of file)
- user-defined events (e.g., key not found in dictionary)

Awkward or impossible to deal with these conditions explicitly without distorting normal code.

Most recent languages (Ada, C++, Java, etc.) provide a means to define, raise, and handle exceptions.

Java example:

```java
class Help extends Exception {
    int ugh;
    Help(int u) {ugh = u;}
}

foo () {
    int icky;
    try {
        icky = foo ();
    } catch (Blah e) {
        icky = e.ugh++; // e.ugh=yucc
    }
}
```

- A few languages support resumption of the program at the point where the exception was raised.

**What to do in an exception?**

- In most languages, uncaught exceptions propagate to next dynamically enclosing handler. E.g. caller can handle uncaught exceptions raised in callee.

```
class Blah extends Exception {
    int ugh;
    Blah(int u) {ugh = u;}
}

foo () {
    ... throw new Blah(yucc); ...
}
```

```
bar () {
    int icky;
    try {
        icky = foo ();
    } catch (Blah e) {
        icky = e.ugh++; // e.ugh=yucc
    }
}
```

**Exceptions vs. Error Values**

An alternative to user-raised exceptions is to return status values, which must be checked on return:

```java
fun find (k0:string) (env: (string * int) list) : int option =
    case env of
    nil => NONE
    | (k,v)::t =>
        if k = k0 then
            SOME v
        else find k0 t
    ...
```

```
... case find "abc" e0 of
    SOME v => ... v ...
    | NONE => ...error code...
```
Exception vs. Error Values (2)

With exceptions, we can defer checking for (rare) error conditions to a more convenient point.

```haskell
exception NotFound
fun find (k0:string) (env: (string * int) list) : int =
  case env of
    nil => raise NotFound
  | (k,v)::t =>
    if k = k0 then v
    else find k0 t

... let val v = find "abc" e0
in ... v ...
end
handle NotFound => ...error code...
```

Exceptions and the Stack

- In the usual exception model, raising an exception provides a way to return quickly from a deep recursion, with no need to pop stack frames one at a time.

Example:

```haskell
fun product l =
  let exception Zero
  fun prod l =
    case l of
      nil => 1
    | (h::t) =>
      if h = 0 then raise Zero
      else h * (product t)
  in (prod l) handle Zero => 0
end
```

- To implement exceptions, runtime system can maintain a **handler stack** with an entry for each handler context currently alive. When an exception occurs, control passes to the top handler on the stack. If this handler itself raises an exception, control passes to the next handler on the stack, etc.

- Alternatively, some runtime systems use a table from code addresses to handler addresses. If an exception occurs (assumed to be rare), the table is inspected to find the appropriate handler.