Procedural Abstraction

In the Bowling Alley of Tomorrow, there will even be machines that wear rental shoes and throw the ball for you. Your sole function will be to drink beer. -- Dave Barry

- Subprograms 8.2
  - body
  - head
- Parameter passing 8.5
  - value, result, value-result, reference, name
- Functions 8.10
- Subprograms as parameters 8.6
- Overloading 8.7
- Generics 8.8

Subprograms

A *subprogram* is a named, parameterized sequence of program statements that can be reused. It typically *encapsulates* (packages) an algorithm.

In particular, it:

- is defined by means of lower-level operations
- is *called* (referred to) by name
- accepts *arguments* (“parameters”)
- may deliver results to the *calling program*
- suspends execution of the calling program during execution
- returns control to the calling program after execution

- *is the fundamental procedural abstraction mechanism!*

Bodies, Heads

Subprograms consist of two main parts:

- a body
  - a sequence of statements
- a head
  - name
  - parameters
    - parameter type
    - parameter passing mode
    - parameter name
  - type of return value

A fairly standard format:

- *name(mode$_1$ type$_1$ param$_1$, mode$_2$ type$_2$ param$_2$, ...) return_val_type S;*

220

Parameters

By giving a subprogram a name, we can re-execute its statements by calling its name. This alone would be useful. But subprograms are much more powerful. A subprogram

- performs the same operation over and over

  but
- it can perform the same operation on different operands

The *operands* of a subprogram are called *subprogram parameters*.

221

Parameters: Formal and Actual

- When we *define* a subprogram we are saying:
  - when called, perform this operation on any given thing X
  - X is called a *formal parameter*
    - there is no actual object X, it's just the name we use to refer to the object that the subprogram will operate on when called
- When we *call* a subprogram we are saying:
  - perform the operation on this particular thing Y now
  - Y is called an *actual parameter*
    - Y is the actual object that will be operated on
- When a subprogram is called, the actual parameters are *somehow* substituted in for the formal parameters in the order they appear in the header.

"How are they substituted?" you ask?

Parameter Passing

Parameter passing is the substitution of actual parameters in a subprogram call for the formal parameters in the subprogram definition.

"How much" of the actual parameter gets passed depends on the *parameter passing mode*.

- only the *value* of the actual parameter?
- both the *value* and the *address* of the actual parameter?
- only the *address* of the actual parameter?

What if the actual parameter is a constant?

What if the actual parameter is an expression?

What does it mean if only the value *is passed?*

What does it mean if only the address *is passed?*

Parameter Passing: Pass-by-Value

With the *pass-by-value* parameter passing mode, the actual parameter is evaluated before being passed to the subprogram: only its *value* is accessible by the subprogram.

- typically with *pass-by-value*, the subprogram only gets a *copy* of the value of the actual parameter

- *Ada:* `in` parameters
- *Pascal:* value parameters
- *Algol-68:* all parameters
- *C:* all parameters

Parameter Passing: Pass-by-Value Example

With *pass-by-value* parameter passing, the formal parameter in the subprogram is like a local variable that gets initialized to the value of the actual parameter.

```
procedure nwrite(ch: char; N: byte);
var
   i: byte;
begin
   for i := 1 to N do
      write(ch);
   ch := '.';
   write(ch)
end;
```

```
mc: 'j'

ch: '.'
 N: 5
 i: 6
```

```
p> mc := 'j';
p> nwrite('j',5);
p>
```

```
Borland Pascal  Version 7.0  Copyright (c) 1983,92 Borland International

jjjjj.
```

Parameter Passing: Pass-by-Result

With the *pass-by-result* parameter passing mode, the formal parameter in the subprogram acts as a local variable; at the *end* of processing in the subprogram, the value of the formal parameter is assigned to the actual parameter.

- *pass-by-result* can be thought of as *write-only* parameter passing

- *Ada:* `out` parameters
- *Pascal:* N/A

226

Parameter Passing: Pass-by-Result Example

With *pass-by-result* parameter passing, the formal parameter in the subprogram is like a local variable that is *not* initialized to the value of the actual parameter.

```
procedure rand(out N: integer) is
   seed: integer;
begin
   seed := secs();
   N := randomize(seed);
end rand;
```

```
R:     68
seed:  13542
N:     68
```

```
a> R := 0;
a> rand(R);
a> put(R);
```

```
Borland Ada  Version 13.5  Copyright (c) 1985 Borland International

 68
```

227

Parameter Passing: Pass-by-Value-Result

With the *pass-by-value-result* parameter passing mode, the formal parameter in the subprogram acts as a local variable.

- the value of the actual parameter is used to intialize the formal parameter
- at the *end* of processing in the subprogram, the value of the formal parameter is assigned to the actual parameter

- *Ada:* `in-out` parameters
- *Pascal:* N/A

Parameter Passing: Pass-by-Value-Result Example

```
procedure rand(in out N: integer) is
  begin
     N := randomize(seed);
  end rand;
```

```
R: 13
N: 13
```

```
a> R := sec();
a> rand(R);
a> put(R);
```

```
Borland Ada  Version 13.5  Copyright (c) 1985 Borland International

 13
```

Parameter Passing: Pass-by-Reference

With the *pass-by-reference* parameter passing mode, the formal parameter is an alias of the actual parameter.

- any modification of the formal parameter is a modification to the actual parameter

- *Ada:* N/A
- *Pascal:* `var` parameters

Parameter Passing: Pass-by-Reference Example

```
procedure addto(var X: integer;
                Y: integer; N: byte);
var
   i: integer;
begin
   for i := 1 to N do
      X := X + Y
end;
```

```
Z: 17
X: 17
Y: 5
N: 3
i: 4
```

```
p> Z := 2;
p> addto(Z, 5, 3);
p>
```

```
Borland Pascal  Version 7.0  Copyright (c) 1983,92 Borland International
```

What if the same variable is passed as two *actual parameters?* *(e.g.* `expon(X, X)`*)*

Comparing the Modes

Have a look at the following little *Pascada* program:

```
program p;
var j: integer;
    A: array [1..10] of integer;
procedure swap(mode X, Y: integer) is
   var tmp: integer;
   begin
      tmp := X;  X := Y;  Y := tmp;
   end swap;

begin
   j := 3;
   A[j] := 6;
   swap(j, A[j]);
end p.
```

What is the result if:

- mode = `in`
  - `j = 3`
  - `A[3] = 6`
- mode = `out`
  - `j =`
  - `A[3] =`

  illegal
- mode = `var`
  - `j = 6`
  - `A[3] = 3`
- mode = `var`
  - `j = 6`
  - `A[3] = 3`

One More: Pass-by-Name

With the *pass-by-name* parameter passing mode, the formal is replaced *textually* with the actual parameter.

- actual parameters are *not* evaluated before being passed to the subprogram
  - they are passed as is (textually)
  - they are evaluated every time they're used inside the subprogram
- for this reason, we say that we have *delayed evaluation* or *late binding* of parameters

The Swap Example: Pass-by-Name

```
procedure swap(name X, Y: integer) is
                    j   A[j]
   var tmp: integer;
   begin
      tmp := Y;  X := Y;  Y := tmp;
             j    j   A[j] A[j]
   end swap;


...
```

```
begin
   j := 3;
   A[j] := 6;
   swap(j, A[j]);
end p.
```

Now what's the result?

- `j = 6`
- `A[6] = 3`

A Closer Look at Delayed Evaluation

Have a look at the following little *PBN* program:

```
program myprog;
   var X, Y: integer;
   procedure p(name X: integer);
      var Y: integer;
   begin
      ...
      write(X);
      ...
   end;
begin
   ...
   p(X+Y);
   ...
end.
```

What gets printed?

Pass-by-name *requires that the entire environment of the caller be passed to the subprogram!*

Jensen's Device

Have a look at the following little procedure in some unknown language:

```
proc sum(name E, I: int;
         in   N: int;
         out  S: int);
begin
   S := 0;
   I := 1;
   while I <= N loop
      S := S + E;
      I := I + 1;
   end loop;

end proc sum;
```

What is the result of:

- `X := sum(2, K, 50, Res);`
  - `2 + 2 + ... + 2 = 100`
- `Y := sum(A[J], J, 4, Res);`
  - `A[1] + A[2] + A[3] + A[4]`
- `Z := sum(2*M-1, M, 5, Res);`
  - `1 + 3 + 5 + 7 + 9 = 25`

Functions

A *function* is a subprogram that returns a value. Because it is a subprogram, we know that it:

- is defined by means of lower-level operations
- is *called* (referred to) by name
- accepts *parameters*
- suspends execution of the calling program during execution
- returns control to the calling program after execution

Because it returns a value, we also know that the calling program can:

- use the function call as an *operator*
- embed the function call inside *expressions*
- use a function call anywhere an *r-value* is expected

Function Call “Issues”

Because a function call returns a value, it can be used inside expressions:

- `i := j * myfunc(k);`

This powerful mechanism allows the programmer to extend the programming language. But there are “issues” to be dealt with:

- what types may be returned by a function?
  - only numeric types?
  - any primitive type?
  - complex types?
- if a function participates in an expression, what about the rules of associativity?
  - `y + sq(y)`
  - `sq(y) + y`

Return Types of Functions

Functions in programming languages are inspired by the mathematical *functions*, which suggests that numeric types as return values would be appropriate. Most languages are not so restrictive as to allow only numeric types as return values. But most languages still restrict the types:

- *Fortran77, Pascal, Modula-2*
  - ✓ primitive types
  - ✗ complex types
- *C*
  - ✗ functions, arrays
  - ✓ all other types (including pointers to functions & pointers to arrays)
- *C++*
  - ✓ all allowable *C* return types
  - ✓ user-defined types (classes)

- *Ada*
  - ✓ anything goes!

Handling Side Effects

Way, way back in our discussion of expressions (♬ 105) we saw this little *Pascal* function:

- ```
  function sq(var x: real): real;
  begin
    x := x * x;
    sq := x
  end;
  ```

This function has a *side-effect* (badthing) because it is allowed to modify its parameter.

- *Ada*
  - functions are allowed `in` mode formal parameters only
- *Everybody else*
  - whatever!

240

Subprograms as Parameters

In our discussion of parameters, we said that much of the power of subprograms is that they are parametric: they can "perform the *same* operation on different *operands*".

Wouldn't it be cool if we could parametrize *operations* too?

```
function integral(function F(Y: real): real;
                  Lo, Hi: real): real;
   const Slices = 1000;
   var   Delta, Sum, X: real;
         I: integer;
begin
   Delta := (Hi - Lo) / Slices;
   X := Lo;
   Sum := F(X);
   for I := 1 to Slices - 1 do begin
      X := X + Delta;
      Sum := Sum + 2.0 * F(X)
   end;
   X := X + Delta;
   Sum := Sum + F(X);
   integral := 0.5 * Delta * Sum
end;
```

241
```

Metafunctions

Our `integral` function will find the definite integral of any *real→real* function. It is a function that operates on functions: a *metafunction*!

```
function sq(X: real): real;
begin   sq := X * X   end;
...
writeln(integral(sq, 0.0, 9.0));
```

- The answer?
  - 243.0 (approximately)
- $\int_0^9 x^2\,dx = x^3/3 \Big|_0^9$

Which of our favourites allow subprograms as parameters?

Pascal	*Modula-2*	*Fortran90*	*Fortran77*	*C/C++*	*Ada*	*Algol-68*
✓*	✓	✓	✓	✗	✗	✓

** But don't try this at home!*

Subprogram Overloading

We talked about overloading operators: the practice of using the same symbol for more than one operation.

In some programming languages (Ada, C++, Java, ...), subprograms can be overloaded.

- Ada

```
function incr(X: integer) return integer;
begin
   return X + 1;
end;
```

```
function incr(D: day) return day;
begin
   if D = day'last then
      return day'first;
   else
      return day'succ(D);
end;
```

```
type day is (mon, tues,
             wed, thurs,
             fri, sat, sun);
```

Generics

Subprogram overloading is cool and all, but why should we have to write two (or three or ten) different `incr` functions?

Why can't we just write one `incr` *function that checks its parameter types and then executes the appropriate code?*

Because parameter type binding *is almost always* static, *that's why!*

But let's suppose for a moment that we have a programming language where parameter type binding could be dynamic (parameters get their types bound at runtime). Now we might be in business.

Parametric Polymorphism

Parametric polymorphism allows a subprogram to be defined without specifying the exact types of its parameters.

- one way to achieve parametric polymorphism is to have parameters whose types are bound at run time
- languages with static binding of parameter types can support a kind of compile time parametric polymorphism
  - define a subprogram leaving parameter types unspecified
  - request one or more *instances* of the subprogram by specifying instances of parameter types
  - the compiler generates code for each of the requested subprogram instances, each with well defined parameter types

Generic Units

- *Ada*
  - ```
    generic
       type INDEX is (<>);
       type ELEM is private;
       type VECTOR is array (INDEX range <>) of ELEM;

       procedure get_max(LIST: in VECTOR; MAX: out ELEM);
       procedure get_max(LIST: in VECTOR; MAX: out ELEM) is
       begin
          MAX := LIST(LIST'FIRST);
          for i in LIST'FIRST..LIST'LAST loop
             if MAX < LIST(i) then
                MAX := LIST(i);
          end loop;
       end get_max;
    ```
  - ```
    procedure int_max is new get_max(INDEX => integer;
                ELEM => integer; VECTOR is array (1..20) of ELEM);
    ```
  - ```
    procedure float_max is new get_max(INDEX => BYTE;
                ELEM => float; VECTOR is array (6..100) of ELEM);
    ```


Templates

- *C++*
  - ```
    template <class Type>
    Type get_max(Type list[], int length)
    {
       int i;
       Type Max;

       for(Max = list[0], i = 1; i < length; i++)
          if(Max < list[i])
             Max = list[i];

       return Max;
    }
    ```
  - ```
    int imax, intlist[NUMELEMS];
    float fmax, floatlist[NUMELEMS * 10];
    ...
    imax = get_max(intlist, NUMELEMS);
    fmax = get_max(floatlist, NUMELEMS * 10);
    ```
    ```

Implementing Subprograms

We've seen that calling a subprogram requires a fair amount of communication between the *caller* and the *callee*:

- control is transferred from execution of statements in the *caller* to execution of statements in the *callee*
- the value of *value* parameters must be copied from actual parameters to formal parameters
- *reference* parameters must be aliased
- the value of *result* parameters must be copied back from formal parameters to actual parameters
- control is transferred back to the statement in the *caller* immediately following the subprogram call

The Activation Stack

Subprogram calls are strictly nested; the *caller* calls the subprogram, then waits until the *callee* has terminated:

A

```
...
mov  ax, offset X
push ax
call B
add  sp, 2
mov  Y, ax
...
```

B

```
push  bp
mov  bp, sp
mov  bx, [bp+4]
mov  ax, [bx]
...
call C
...
mov  [bx], ax
pop  bp
ret
```

C

```
push bp
mov  bp, sp
...
pop  bp
ret
```

If subprogram calls can be nested arbitrarily deep and need access only to the immediate caller, *what would be a good data structure?*

Activation Records

We'll use a stack to communicate between *callers* and *callees*. Every time a subprogram is called, a new activation record is created and stored on the stack. Each activation record contains info about the *caller* and the *callee*:

- information about the *caller*
  - a pointer to the *caller*'s activation record
  - the return address
  - the address where the return value (if any) should be stored
- information about the *callee*
  - a pointer to the activation record of the enclosing block (for the referencing environment!)
  - local variables and constants
  - formal parameters
  - temporary storage

250

Building Activation Records...

It is common for the run time memory allocated for a program to be divided into three parts:

- *code area*
  - program statements
  - subprogram statements
- *data area*
  - global variables
  - global constants
  - explicit dynamic allocations
- *stack area*
  - local variables
  - subprogram parameters
  - return addresses
  - etc.

activation records!

251

More Building Activation Records

Important control information about the three memory areas is usually kept in *registers*:

- a *register* is a special small memory area right on the CPU
  - usually one word (e.g. 32 bits)
- some relevant registers:
  - SP: address of the top of the runtime stack
  - BP: address of the base of the runtime stack
  - IP: address of the next instruction to be executed
  - AX, BX: general purpose registers (for arithmetic, etc.)
  - CX: general purpose register (for loop indexing, etc.)
  - etc.

Subprogram Code Generation

For every subprogram call, the compiler generates:

- a *prologue*:
  - extra code to get stack space, build the activation record, copy parameter values, initialize local variables, jump to the subprogram code, etc.
- the *subprogram body*:
  - translation of the subprogram statements
- an *epilogue*:
  - extra code to clean up the stack, copy *result* parameters, copy the return value (if any) to the appropriate place, return control to the *caller*, etc.

Subprogram Code Generation Example

Consider this tiny Pascal program:

```
program P;
var X, Y: integer;
function A(arg: integer): integer;
   begin
      A := arg * arg
   end;
begin
   ...
   X := 5;
   Y := A(X);
   ...
end.
```

P

```
...
mov  ax, 5
mov  X, ax
push ax
call A
add  sp, 2
mov  Y, ax
...
```

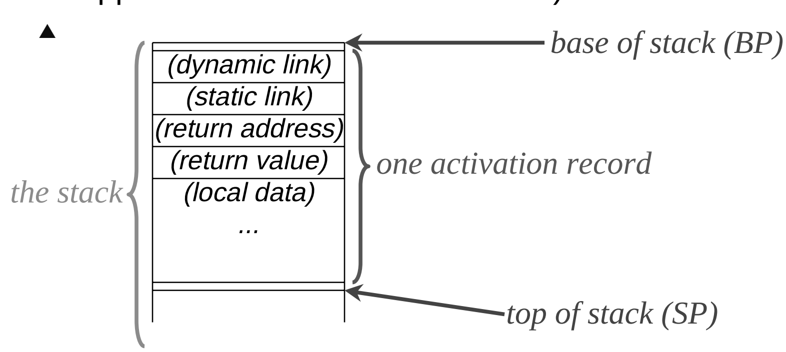
A

```
push bp
mov  bp, sp
mov  bx, [bp+4]
mov  ax, bx
imul ax, bx
pop  bp
ret
```


A Closer Look at Activation Records

Different languages have slightly different requirements for the content of activation records. We'll stick to Pascal.

- Here is the general format of the activation stack (we'll grow the stack from the top of the slide towards the bottom, the opposite of what's in the textbook)



Activation Records: An Extended Example...

- We're going to walk through the life of the activation stack during the execution of this little Pascal

```
program Main;
  var A, B: integer;
  procedure P;
  begin
    A := A + 1;   B := B + 1
  end;
  procedure Q;
    var B: integer
    procedure R;
      var A: integer;
    begin
      A := 16;   P;   write(A, B)
    end;
  begin
    B := 11;   R;   P;   write(A, B)
  end;
begin
  A := 1;   B := 6;
  P;   write(A, B);
  Q;   write(A, B)
end.
```

256

An Extended Example Extended...

Instead of translating this program into Assembly language like we did last time, let's assume the *code area* contains Pascal instructions:

Main

```
L1m  A := 1
L2m  B := 6
L3m  P
L4m  write(A, B)
L5m  Q
L6m  write(A, B)
L7m  ...
```

P

```
L1p  A := A + 1
L2p  B := B + 1
L3p
```

Q

```
L1q  B := 11
L2q  R
L3q  P
L4q  write(A, B)
L5q
```

R

```
L1r  A := 16
L2r  P
L3r  write(A, B)
L4r
```

257

Growing the Activation Stack...

IP=L3m

s01	(dynlink)	*Main*
s02	(statlink)	
s03	(retaddr)	
s04	A 1	
s05	B 6	
s06		
s07		
s08		
s09		

IP=L1q

s01	(dynlink)	*Main*
s02	(statlink)	
s03	(retaddr)	
s04	A 2	
s05	B 7	
s06	s01	*Q*
s07	s01	
s08	L6m	
s09	B	
s10		
s11		
s12		
s13		
s14		

IP=L1r

s01	(dynlink)	*Main*
s02	(statlink)	
s03	(retaddr)	
s04	A 2	
s05	B 7	
s06	s01	*Q*
s07	s01	
s08	L6m	
s09	B 11	
s10	s06	*R*
s11	s06	
s12	L3q	
s13	A	
s14		
s15		
s16		
s17		

258

Growing the Activation Stack Some More

IP=L1p

s01	(dynlink)	*Main*
s02	(statlink)	
s03	(retaddr)	
s04	A 2	
s05	B 7	
s06	s01	*Q*
s07	s01	
s08	L6m	
s09	B 11	
s10	s06	*R*
s11	s06	
s12	L3q	
s13	A 16	
s14	s10	*P*
s15	s01	
s16	L3r	
s17		

IP=L1p

s01	(dynlink)	*Main*
s02	(statlink)	
s03	(retaddr)	
s04	A 3	
s05	B 8	
s06	s01	*Q*
s07	s01	
s08	L6m	
s09	B 11	
s10	s06	*P*
s11	s01	
s12	L4q	
s13		
s14		

IP=L4q

s01	(dynlink)	*Main*
s02	(statlink)	
s03	(retaddr)	
s04	A 4	
s05	B 9	
s06	s01	*Q*
s07	s01	
s08	L6m	
s09	B 11	
s10		
s11		

259

Smaller Progam, Bigger Example

```
L01   program main;
L02   var A: integer;

L03     function F(N: integer): integer;
L04     begin
L05       if N <= 1 then
L06         F := 1
L07       else
L08         F := N * F(N-1)
L09     end;

L10   begin
L11     A := F(3);
L12     writeln(A)
L13   end.
```

IP	L02
AX	

S01	(dynamic link)	(main)
S02	(static link)	
S03	(return addr)	
S04	(return val)	
S05		
S06		
S07		
S08		
S09		
S10		
S11		
S12		
S13		
S14		
S15		
S16		
S17		
S18		
S19		
S20		
S21		
S22		