Chapter 9
Functions

It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures.
A. Perlis

Functions

- Fundamental to all programming languages
- Essential for:
  - *Abstraction*
  - *Separation of concerns*
  - *Top-down design*
- Behavior of functions is tightly related to dynamic memory management and the stack

Contents

- 9.1 Basic Terminology
- 9.2 Function Call and Return
- 9.3 Parameters
- 9.4 Parameter Passing Mechanisms
- 9.5 Activation Records
- 9.6 Recursive Functions
- 9.7 Run Time Stack

9.1 Basic Terminology

- Some languages syntactically distinguish functions that return a value from functions that do not
  - *Fortran: Functions and Subroutines*
  - *Ada/Pascal: Functions and Procedures*
- C-Like languages do not make a syntactic distinction
  - *Functions that do not return a value are called void functions*
- At the machine level there is no distinction

Function calls

- Value-returning functions are rvalues and can appear in expressions:

  e.g., x = (b*b - sqrt(4*a*c))/2*a

- Non-value-returning functions are invoked as a separate statement

  e.g., strcpy(s1, s2);

Functions and Side-Effects

- Side effects include performing I/O or modifying non-local variables
- The generally accepted rule is that value returning functions should not have side-effects
- Less generally accepted is the notion that procedures should not have side effects except by modifying one or more arguments
- But most imperative and OO languages have no mechanisms to enforce side-effect rules

9.2 Function Call and Return

Ex C/C++ Program
Fig 9.1

```
int h, i;
void B(int w) {
       int j, k;
       i = 2*w;
       w = w+1;
}
void A(int x, int y) {
       bool i, j;
       B(h);
}
int main() {
       int a, b;
       h = 5; a = 3; b = 2;
       A(a, b);
}
```

Function Call

- For value returning functions the function *usually* appears as an r-value
  - *Some languages allow value returning functions to be called and the result discarded by calling as a void function*
- For void functions:
  - *Most languages allow use of function name as if it were a statement, without an explicit CALL statement*
  - *A few such as Fortran require a CALL statement*
  - *Many other languages also provide an explicit CALL*
  - *BASIC syntax varies according to whether CALL was used or not*

Function Return Values

- Most languages return control automatically from a void function when the end of the function body is reached
- Explicit return statements also provided
- For value returning functions:
  - *Many languages provide a "return" statement with an expression to return a value*
  - *Pascal-like and BASIC: function name is treated as a local variable. Return value by assigning to function name*
  - *Haskell & functional langs: designate result by writing an expression*

9.3 Parameters

- Definitions
  - An *argument* is an expression that appears in a function call. (aka *actual parameter*)
  - A *parameter* is an identifier that appears in a function declaration. (aka *formal parameter*)

- The call A(a, b) has arguments a and b.
- The function declaration A has parameters x and y.

Parameter-Argument Matching

- Most language require that arguments match parameters in number and type
  - *Some languages (e.g, Visual Basic, T-SQL, PHP) allow specification of optional parameters with default values*

```
Public Sub Foo(ByVal A as Integer, _
    Optional ByVal S as String = "")
```

  - *Some languages allow variable length argument lists (e.g, C printf function)*
- Arguments and parameters are usually matched by number and by position.
  - *I.e., any call to A must have two arguments, and they must match the corresponding parameters' types.*

Some Exceptions

- Perl - parameters aren't declared in a function header. Instead, parameters are available in an array @_, and are accessed using a subscript on this array.
- Ada - arguments and parameters can be linked by name. the call A(y=>b, x=>a) is the same as A(a, b)
- T-SQL – similar to Ada

```
exec dbo.update_orders @date=GetDate(), @loc='oro'
```

- Smalltalk: unusual infix notation for method names

```
array at: index + offset put: Bag new
array at: 1 put: self
x < 4 ifTrue: ['Yes'] ifFalse: ['No']
```

9.4 Parameter Passing Mechanisms

- Lifetime and semantics of variable references are determined by parameter passing mechanisms:

1. By value
2. By reference
3. By value-result
4. By result
5. By name

- First two are most common
- Pass by Name was primarily used in Algol 68 and was quickly abandoned

Pass by Value

- Compute the *value* of the argument at the time of the call and assign that value to the parameter.
  - *Ex: in the call* A(a, b) *in Fig. 9.1,* a *and* b *are passed by value. So the values of parameters* x *and* y *become 3 and 2, respectively when the call begins.*
- So passing by value doesn't normally allow the called function to modify an argument's value.
- All arguments in C and Java are passed by value.
- But references can be passed to allow argument values to be modified.
- Pass by value is also call *copy-in*
- Pass by value parameters are also call *in* parameters

Fig 9.1 – h is unmodified

```
int h, i;
void B(int w) {
        int j, k;
        i = 2*w;
        w = w+1;
}
void A(int x, int y) {
        bool i, j;
        B(h);
}
int main() {
        int a, b;
        h = 5; a = 3; b = 2;
        A(a, b);
}
```

Swap Function

- This won't work in C

```
void swap (int a, int b) {
  int temp = a;
  a = b;
  b = temp;
}
```

- This will work

```
void swap (int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}
```

- To call

```
swap (&x, &y)
```

Swap in Java

- Same reasoning in Java

```
void swap (Object a, Object b) {
  Object temp = a;
  a = b;
  b = temp;
}
```

- But you can swap array elements

```
void swap (Object [] A, int i, int j) {
  int temp = A[i];
  A[i] = A[j];
  A[j] = temp;
}
```

Java Objects

- An Object in Java is really a reference to a memory block in the heap
- However, Objects are not passed by reference.
- A correct statement would be that Object references are passed by value.

Java Swapping Using a Wrapper Class

```
// MyInteger: similar to Integer, but can change value
class MyInteger {
   private int x;                    // single data member
   public MyInteger(int xIn) { x = xIn; } // constructor
   public int getValue() { return x; }  // retrieve value
   public void insertValue(int xIn) { x = xIn;} // insert
}
public class Swapping {
   // swap: pass references to objects
   static void swap(MyInteger rWrap, MyInteger sWrap) {
      // interchange values inside objects
      int t = rWrap.getValue();
      rWrap.insertValue(sWrap.getValue());
      sWrap.insertValue(t);
   }
   public static void main(String[] args) {
      int a = 23, b = 47;
      System.out.println("Before. a:" + a + ", b: " + b);
      MyInteger aWrap = new MyInteger(a);
      MyInteger bWrap = new MyInteger(b);
      swap(aWrap, bWrap);
      a = aWrap.getValue();
      b = bWrap.getValue();
      System.out.println("After.  a:" + a + ", b: " + b);
   }
}
// from http://www.cs.utsa.edu/~wagner/CS2213/swap/swap.html
```

Swapping with C++ preprocessor

- Also from

http://www.cs.utsa.edu/~wagner/CS2213/swap/swap.html

```
#define swap(type, i, j) {type t = i; i = j; j = t;}

int main() {
    int a = 23, b = 47;
    printf("Before swap. a: %d, b: %d\n", a, b);
    swap(int, a, b)
    printf("After swap.  a: %d, b: %d\n", a, b);
    return 0;
}
```

- Preprocessed output

```
int main() {
    int a = 23, b = 47;
    printf("Before swap. a: %d, b: %d\n", a, b);
     { int t = a ; a = b ; b = t ; }
    printf("After swap.  a: %d, b: %d\n", a, b);
    return 0;
}
```

Pass by Reference

- Pass *by reference* means that the memory address of the argument is copied to the corresponding parameter
- The parameter is therefore a *pointer* or *indirect reference* to the argument
- All assignments to the formal parameter within the lifetime of the function affect the value of the argument
- Pass by reference paramters are also called *in-out* parameters

Pass by Reference

- Ex 9.3
- h is passed by reference, so its value changes during the call to B.

```
int h, i;
void B(int* w) {
        int j, k;
        i = 2*(*w);
        *w = *w+1;
}
void A(int* x, int* y) {
        bool i, j;
        B(&h);
}
int main() {
        int a, b;
        h = 5; a = 3; b = 2;
        A(&a, &b);
}
```

Distinguishing reference and value parameters

- If languages support both pass-by-value and pass-by-reference, the distinction must be made explicit in the in the parameter list
- Ex: Pascal

  by value

  `function f(x, y: integer): integer;` by value

  by reference

  `procedure swap(var x, y: integer);`

C++ Reference Parameters

- Use & to denote a reference parameter

```
void swap(int& x, int& y) {
    int temp = x;
    x = y;
    y = temp;
}
```

Reference Parameters must be l-values

- Since an address is passed, you can't pass a literal value as a reference parameter

```
swap (a, b)  //OK
swap(a+1, b) // Not OK
swap(x[j],x[j+1]) // OK
```

- In Fortran all parameters are reference parameters
- Some early compilers had an interesting bug

```
Subroutine inc(j)
  j = j + 1
End Subroutine
```

- A call to inc(1) would leave the constant with the value 2 for the remainder of the program

Using r-values as arguments

- Some languages (e.g., Fortran) allow non l-values as arguments for reference parameter
- Solution is to create a temporary variable and pass that address

Pass by Value-Result and Result

- Pass by value at the time of the call and then copy the result back to the argument at the end of the call.
  - *E.g., Ada's* in out *parameter can be implemented as value-result.*
  - *Value-result is often called* copy-in-copy-out.
- Reference and value-result are the same, except when *aliasing* occurs. That is, when:
  - *the same variable is both passed and globally referenced from the called function, or*
  - *the same variable is passed for two different parameters.*
  - *Having 2 or more references (pointers) to the same location*

Example

- C++ and Ada functions

```
void f(int& x, int& y) {
  x = x + 1;
  y = y + 1;
}
procedure f(x,y: in out integer) is
begin
  x := x + 1;
  y := y + 1;
End f;
```

- Consider f(a, b) and f(a, a)
- f(a,a) will leave a incremented by 2 in C++ function
- *If* the Ada compiler implements in-out parameters using pass by value-result then a will be incremented by 1 only

Aliasing problems

- Simple examples can be detected by a compiler
- But in general aliasing problems are intractable
- A fruitful source of bugs

Pass by Name

- Example of *late binding*, since evaluation of the argument is delayed until its occurrence in the function body is actually executed.
- An argument *passed by name* behaves as though it is textually substituted for every occurrence of the corresponding parameter in the function body.
- Originally used in Algol 60/Algol 68
- Dropped by successors: Pascal, Modula, Ada
- Associated with lazy evaluation in functional languages (see, e.g., Haskell discussion in Chapter 14).

Pass by Name

- Implications of pass-by-name:
  1. *The argument expression is re-evaluated each time the formal parameter is accessed.*
  2. *The procedure can change the values of variables used in the argument expression and thus change the expression's value.*

Swap again

- Cannot write a general purpose swap

```
procedure swap(a, b);
  integer a, b;
begin
  integer t;
  t := a; a := b; b := a;
End;
```

- Swap (i , j) works fine but swap (i, a[i]) does not because the second assignment a := b changes the value of i

Another Simple Pass by Name example

```
procedure inc2(i, j);
   integer i, j;
   begin
      i := i+1;              => k <- k+1
      j := j+1               => A[k] <- A[k]+1
   end;
inc2 (k, A[k]);
```

Jensen's Device

```
real procedure SIGMA(x, i, n);
   value n;
   real x; integer i, n;
begin
   real s;
   s := 0;
   for i := 1 step 1 until n do
      s := s + x;
   SIGMA := s;
end
```

- Consider

```
s := SIGMA(a, b, c);       (computes a * c)
s := SIGMA(x[i], i, n);    (computes x1+x2+…+xn )
s := SIGMA(x[i]*y[i],i,n); (x1*y1+ x2*y2+…+ xn*yn)
s := SIGMA(1/i, i, n);     (1/1 + 1/2 + 1/3 … + 1/n)
```

Parameter passing in Ada

- Ada does not specify how parameters are passed, but how they are used (parameter *modes*)
- Modes are in, out, and in out
- An *in* parameter can only be read and cannot be written
- An *out* parameter can only be written and cannot be read (pass by result)
- *In out* parameters can be read and written

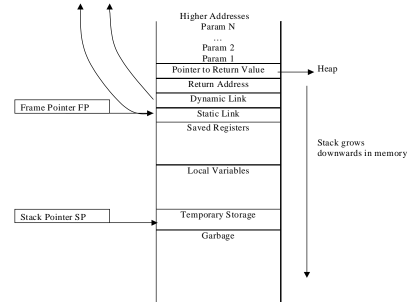
Parameter modes

- Note that text mentions that Ada is unique among languages discussed in the book
- Other languages also use parameter modes
- Most SQL variants use parameter modes for stored procedures

9.5 Activation Records

- A block of information associated with each function call, which includes:
  - *The called function's parameters and local variables*
  - *Return address*
  - *Saved registers*
  - *Temporary variables*
  - *Return value*
  - *Static link - to the function's static parent*
  - *Dynamic link - to the activation record of the caller*
- An activation record is also called a Stack Frame

Stack Frame



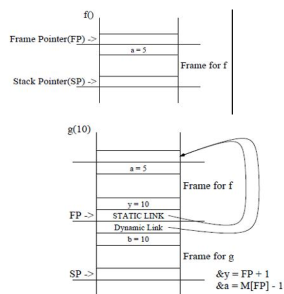
Dynamic and Static Links

- Dynamic link points to the stack frame or activation record of the caller
  - *Text: "the dynamic link is used to facilitate information flow between the called function and the caller"*
  - *Aside from dynamic scoping, the primary function of the dynamic is to restore the frame pointer of the caller on function return*
- The static link points to the static (lexical) parent of the caller.
- Only used for languages with lexically nested scopes

Nested Function Example

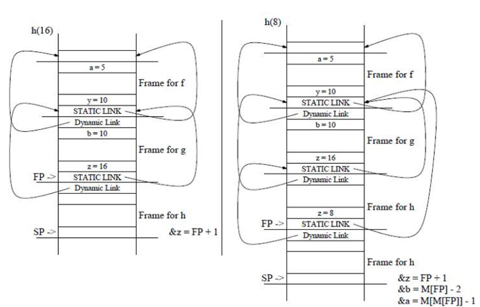
```
function f(): integer;
  var a: integer;
  function g(y: integer): integer;
     var b: integer;
     function h(z: integer): integer;
       if z > 10 then
         h := z / 2;
       else
         h := z + b * a;
       endif;
     end function h;
     b := 10;
     g := y + a + h(16);
  end function g;
  a := 5;
  f := g(10);
end function f;
```

Stack frames



Source: http://eliza.newhaven.edu/lang/attach/Oct_17.pdf

Stack frames



Memory Allocation for Activation Records

- Usually on the stack – which is why they're called stack frames
- Languages such as Fortran that do not allow recursion can allocate activation records statically in global memory

9.6 Recursive Functions

- A function that can call itself, either directly or indirectly, is a recursive function.
- Only a few languages such as Fortran do NOT support recursive function calls
- Primary control structure for functional languages: replaces iteration

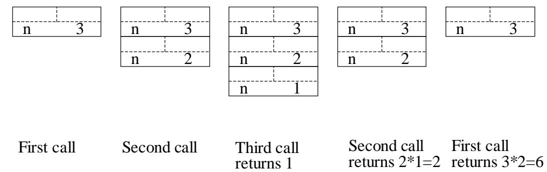
```
int factorial (int n) {
  if (n < 2)
    return 1;
  else
    return n*factorial(n-1)
}
```

9.7 Run Time Stack

- A stack of activation records.
  - Each new call pushes an activation record, and each completing call pops the topmost one.
  - So, the topmost record is the most recent call, and the stack has all active calls at any run-time moment.
- For example, consider the call factorial(3). This places one activation record onto the stack and generates a second call factorial(2). This call generates the call factorial(1), so that the stack gains three activation records.

Stack Activity for the Call factorial(3)

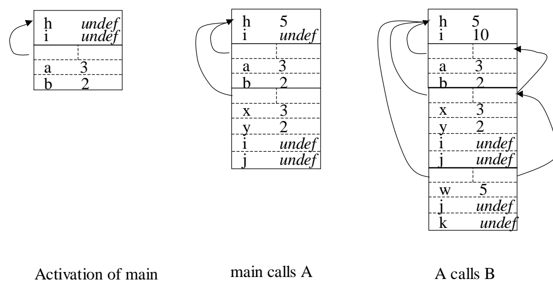
Fig. 9.7



First call | Second call | Third call returns 1 | Second call returns 2*1=2 | First call returns 3*2=6

Stack Activity for Program in Fig. 9.1

Fig. 9.8



Activation of main | main calls A | A calls B