Chapter 8 :: Subroutines and Control Abstraction

1

Programming Language Pragmatics

Michael L. Scott

1

Introduction

2

- abstraction
  - association of a name with a potentially complicated program fragment that can be considered in terms of its purpose or function rather than its implementation
  - most data abstractions include control abstractions

2

Introduction

3

- subroutines
  - principal mechanism for control abstraction
  - performs operations for caller
  - most involve parameters
    - actual parameters (arguments)
    - formal parameters
  - function: returns a value
  - procedure: does not return a value
  - most must be declared

3

Review Of Stack Layout

4

- programs/subroutines use stack space
  - static area
    - code
    - globals
    - explicit constants (including strings, sets, other aggregates)
    - small scalars may be stored in the instructions themselves

4

Review Of Stack Layout

5

- stack frames
  - also called activation records
  - contains
    - arguments/return values
    - local variables
    - temporaries
    - bookkeeping information (return address and saved registers)
  - pushed and popped as subroutines called/return

5

Review Of Stack Layout

6

- stack frames (cont.)
  - sp: stack pointer
    - register containing last used location, or first unused location
  - fp: frame pointer
    - objects in the frame accessed by offset from frame pointer
  - variable sized objects placed at top of frame
    - address and dope vector in fixed-size portion of frame
    - if none, all objects can be offset from sp and no fp is needed

6

Review Of Stack Layout

7

- subroutine nesting
  - in a language with nested subroutines and static scoping
    - Pascal, Ada, list, Scheme
    - static chain used to locate objects
    - static links points to frame of surrounding subroutine
    - guaranteed surrounding subroutine active
    - dynamic link: saved value of fp for return

ELSEVIER

7

Review Of Stack Layout

8

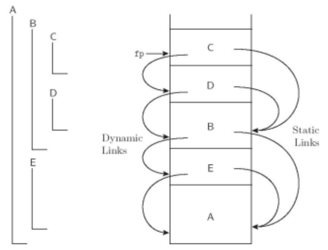


Figure 8.1: **Example of subroutine nesting, taken from Figure 3.5.** Within B, C, and D, all five routines are visible. Within A and E, routines A, B, and E are visible, but C and D are not. Given the calling sequence A, E, B, D, C, in that order, frames will be allocated on the stack as shown at right, with the indicated static and dynamic links.

ELSEVIER

8

Calling Sequences

9

- maintenance of stack is responsibility of *calling sequence*
  - code executed by caller immediately before and after a subroutine call
  - *subroutine prologue* and *epilogue*
    - code performed at beginning/end of subroutine
  - sometimes calling sequence includes all three

ELSEVIER

9

Calling Sequences

10

- tasks executed on the way into a subroutine
  - passing parameters
  - saving return address
  - changing program counter
  - changing stack pointer to allocate space
  - save registers
  - changing frame pointer to point to new frame
  - executing initialization code for any new objects

ELSEVIER

10

Calling Sequences

11

- tasks executed on the way out of a subroutine
  - passing return parameters or function values
  - executing finalization code for any objects
  - deallocating the stack frame
  - restoring saved registers
  - restoring program counter

ELSEVIER

11

Calling Sequences

12

- some tasks must be performed by the caller because they differ from call to call
- other tasks may be performed by the callee
  - space is saved by putting as much in the callee prologue and epilogue as possible
    - appear only once in target program
  - time *may* be saved by assigning tasks to the caller, where more information may be known
    - e.g., there may be fewer registers in use at the point of call than are used somewhere in the callee

ELSEVIER

11

12

Calling Sequences

13

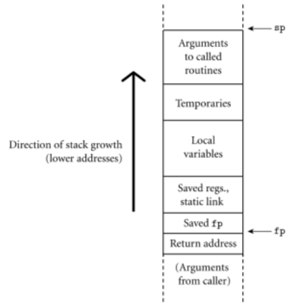
- maintaining the static chain
  - in languages with nested subroutines, caller must perform due to lexical nesting of the caller
- some registers saved by caller and some by callee

13

Calling Sequences

14

- typical calling sequence



14

Calling Sequences

15

- many parts of the calling sequence, prologue, and/or epilogue can be omitted in common cases
  - particularly LEAF routines (those that do not call other routines)
    - leaving things out saves time
    - simple leaf routines do not use the stack – do not even use memory – and are exceptionally fast

15

Calling Sequences

16

- in-line expansion
  - certain subroutines can be extended in-line at the point of call
  - a copy of the subroutine is placed in the caller
  - avoids overhead
    - space allocation
    - branch delays from the call and return
    - maintaining static chain
    - saving/restoring registers

16

Calling Sequences

17

- in-line expansion (cont.)
  - allows compiler to perform code improvements
    - global register allocation
    - instruction scheduling
    - common subexpression elimination

17

Calling Sequences

18

- in-line expansion (cont.)
  - compiler chooses which subroutine to expand
    - some languages allow the programmer to suggest that particular routines be in-lined (may be ignored)
    - C/C++

```
inline int max(int a, int b) {return a > b ? a : b;}
```

    - Ada

```
function max(a, b : integer) return integer is
begin
    if a > b then return a; else return b; end if;
end max;
pragma inline(max);
```


18

Calling Sequences

19

- in-line expansion (cont.)
  - preferable to macros
  - disadvantages
    - increases code size
    - cannot be used for recursive subroutines
      - one level can be expanded in-line

```
string fringe (bin_tree *t) {
        // assume both children are nil or neither is
   if (t->left == 0) return t->val;
        return fringe(t->left) + fringe(t->right);
      }
```


Parameter Passing

20

- formal parameters vs. actual parameters
- parameter passing modes
  - value
  - value/result (copying)
  - reference (aliasing)
  - closure/name

Parameter Passing

21

- most languages use prefix notation for calls
  - subroutine name followed by parenthesized arguments
  - List places the function name inside the parentheses: **(max a b)**
- some languages (e.g., ML) allow infix notation

```
infixr 8 tothe;      (* exponentiation *)
fun x tothe 0 = 1.0
  | x tothe n = x * (x tothe(n-1));      (* assume n >= 0 *)
```

  - right-associative, binary, at precedence level 8

Parameter Passing

22

- some languages (e.g., ML) allow infix notation (cont.)
  - Fortran: **A .cross. B**
- some languages use same syntax for control expressions

```
if a > b then max := a else max := b;              (* Pascal *)

(if (> a b) (setf max a) (setf max b))             ; Lisp

(a > b) ifTrue: [max <- a] ifFalse: [max <- b].    "Smalltalk"
```


Parameter Passing

23

- parameter passing modes
  - ex.: **p(x)**
  - call-by-value: p gets a copy of x's value
  - call-by-reference: p gets a copy of x's address
    - introduces aliases in subroutines, which may be tricky

```
x : integer                     -- global
procedure foo(y : integer)
    y := 3
    print x
...
x := 2
foo(x)
print x
```


Parameter Passing

24

- parameter passing modes (cont.)
  - call-by-value: prints 2 twice
  - call-by-reference: prints 3 twice

```
x : integer                     -- global
procedure foo(y : integer)
    y := 3
    print x
...
x := 2
foo(x)
print x
```

Parameter Passing
25

- parameter passing modes (cont.)
  - call-by-value/result: copies the value into the formal parameter at beginning and copies the formal parameter back into the actual parameter upon return

```
x : integer                    -- global
procedure foo(y : integer)
    y := 3
    print x
. . .
x := 2
foo(x)
print x
```


Parameter Passing
26

- parameter passing modes (cont.)
  - call by reference in C: typically, explicit, but implicit with arrays

```
void swap(int *a, int *b) { int t = *a; *a = *b; *b = t; }
...
swap(&v1, &v2);
```

  - Fortran: all parameters passed by value
  - call-by-sharing: similar to call by reference, though while values can change, the identity of the object pointed to cannot

Parameter Passing
27

- parameter passing modes (cont.)
  - purpose of call-by-reference
    - to change the value of an actual parameter
    - to avoid time-consuming value copies
      - for some parameters, copy may be preferable since after a certain number of indirections, cost may be less
      - may not be desirable if it leads to unanticipated modification of actual parameters

Parameter Passing
28

- parameter passing modes (cont.)
  - read-only parameters
    - Modula-3: **READONLY** parameters could not be modified
    - C: implemented with **const**

```
void append_to_log(const huge_record *r) { ...
...
append_to_log(&my_record);
```

      - points to record whose value is constant
      - **huge_record* const r;** for constant pointer

Parameter Passing
29

- Ada provides three parameter passing modes
  - *in*: callee reads only
  - *out*: callee writes and can then read; actual modified
  - *in out*: callee reads and writes; actual modified
- Ada in/out is always implemented as
  - value/result for scalars, and either
  - value/result or reference for structured objects

Parameter Passing
30

- C/C++: functions
  - parameters passed by value (C)
  - parameters passed by reference can be simulated with pointers (C)

```
void proc(int* x,int y){*x = *x+y }
 …
proc(&a,b);
```

  - programmers did not like the extra syntax required
    - references introduced in C++

Parameter Passing

31

- C/C++: functions
  - references introduced in C++

```
void proc(int& x, int y)
{x = x + y }

proc(a,b);
```

  - another example

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


31

Parameter Passing

32

- C/C++: functions
  - references can be used in other ways as well

```
int i;
int &j = i;
...
i = 2;
j = 3;
cout << i;          // prints 3
```

  - can also be used as return values

```
cout << a << b << c;

((cout.operator<<(a)).operator<<(b)).operator<<(c);
```


32

Parameter Passing

33

- call-by-name
  - Algol 60
  - call by textual substitution (procedure with all name parameters works like macro)
- conformant arrays
  - arrays as parameters with some unspecified bounds

33

Parameter Passing

34

- default parameters
  - need not be provided by caller

```
type field is integer range 0..integer'last;
type number_base is integer range 2..16;
default_width : field       := integer'width;
default_base  : number_base := 10;
procedure put(item  : in integer;
              width : in field       := default_width;
              base  : in number_base := default_base);
```


34

Parameter Passing

35

- named parameters
  - examples

```
put(item => 37, base => 8);
put(base => 8, item => 37);
put(37, base => 8);
```

  - good for complex interfaces

```
format_page(columns => 2,
    window_height => 400, window_width => 200,
    header_font => Helvetica, body_font => Times,
    title_font => Times_Bold, header_point_size => 10,
    body_point_size => 11, title_point_size => 13,
    justification => true, hyphenation => false,
    page_num => 3, paragraph_indent => 18,
    background_color => white);
```


35

Parameter Passing

36

- variable number of arguments

```
#include <stdarg.h>     /* macros and type definitions */
int printf(char *format, ...)
{
    va_list args;
    va_start(args, format);
    ...
        char cp = va_arg(args, char);
        ...
        double dp = va_arg(args, double);
    ...
    va_end(args);
}
```


36

Parameter Passing

- function returns
  - sometimes returned through function name
  - return can use local variable
  - Ada:

```
type int_array is array (integer range <>) of integer;
    -- array of integers with unspecified integer bounds
function A_max(A : int_array) return integer is
rtn : integer;
begin
    rtn := integer'first;
    for i in A'first .. A'last loop
        if A(i) > rtn then rtn := A(i); end if;
    end loop;
    return rtn;
end A_max;
```


Parameter Passing

- function returns (cont.)
  - SR:

```
procedure A_max(ref A[1:*]: int) returns rtn : int
    rtn := low(int)
    fa i := 1 to ub(A) ->
        if A[i] > rtn -> rtn := A[i] fi
    af
end
```


Parameter Passing

	implementation mechanism	permissible operations	change to actual?	alias?
value	value	read, write	no	no
in, const	value or reference	read only	no	maybe
out (Ada)	value or reference	write only	yes	maybe
value/result	value	read, write	yes	no
var, ref	reference	read, write	yes	yes
sharing	value or reference	read, write	yes	yes
in out (Ada)	value or reference	read, write	yes	maybe
name (Algol 60)	closure (thunk)	read, write	yes	yes

Generic Subroutines and Modules

- generic modules or classes
  - allow a single copy of source code to handle a variety of types
  - parameter types incompletely specified
  - type checking delayed until run time
- Ada: generics
- C++: templates

Generic Subroutines and Modules

- generic modules or classes are particularly valuable for creating *containers*
  - data abstractions that hold a collection of objects
  - operations oblivious to type
  - e.g., stack, queue, heap, set, dictionary
- generic subroutines (methods) are needed in generic modules (classes), and may also be useful in their own right (e.g., max)

Generic Subroutines and Modules

- array-based queue template in C++

```
template<class item, int max_items = 100>
class queue {
    item items[max_items];
    int next_free;
    int next_full;
public:
    queue() {
        next_free = next_full = 0;      // initialization
    }
    void enqueue(item it) {
        items[next_free] = it;
        next_free = (next_free + 1) % max_items;
    }
    item dequeue() {
        item rtn = items[next_full];
        next_full = (next_full + 1) % max_items;
        return rtn;
    }
};
...
queue<process> ready_list;
queue<int, 50> int_queue;
```

Generic Subroutines and Modules

43

- generic implementation options
  - Ada and C++: purely static
    - compiler takes care of all instances
  - C++: separate code for each instance of the template
  - Java: all instances of generic share code
- similarities to macros, but
  - generics integrated into language and understood by the compiler
  - generic parameters are type-checked
  - names inside generics obey scoping rules

Exception Handling

44

- exception
  - a hardware-detected run-time error or unusual condition detected by software
- examples
  - arithmetic overflow
  - end-of-file on input
  - wrong type for input data
  - user-defined conditions, not necessarily errors

Exception Handling

45

- exception handler
  - code executed when exception occurs
  - may need a different handler for each type of exception
- advantages
  - allow user to explicitly handle errors in a uniform manner
  - allow user to handle errors without having to check these conditions explicitly in the program everywhere they might occur

Exception Handling

46

- exception handlers found in many languages
  - Clu, Ada, Modula-3, Python, PHP, Ruby, C++, Java, C#, and ML

Exception Handling

47

- C++ example

```
try {
    ...
    // protected block of code
    ...
} catch(end_of_file) {
    ...
} catch(io_error e) {
    // handler for any io_error other than end_of_file
    ...
} catch(...) {
    // handler for any exception not previously named
    // (in this case, the triple-dot ellipsis is a valid C++ token;
    // it does not indicate missing code)
}
```

  - handlers examined in order
  - first match is used by name or by parent class
  - all other errors caught by …
  - if no …, exception propagated up the dynamic chain
    - if outermost level reached, predefined handler terminates

Exception Handling

48

- three important operations performed by exception handlers
  - compensate for exception to allow program to continue
    - "out of memory" may request more memory
  - if cannot be handled locally, handler may do local clean-up
    - e.g., call destructors
    - re-raises the exception to propagate upward
  - if recovery not possible, an error message can be printed

Coroutines

49

- coroutines
  - execute one at a time and transfer control back and forth explicitly by name
- coroutines can be used to implement
  - iterators
  - threads
  - because they are concurrent (i.e., simultaneously started but not completed), coroutines cannot share a single stack

Coroutines

50

- example
  - screen saver program to prevent liquid crystal burn-in
  - file system checks for corrupted files (sanity check)
  - could be written as

```
loop
    -- update picture on screen
    -- perform next sanity check
```

    - successive sanity checks may depend on each other

Coroutines

51

- example (cont.)
  - could be written with coroutines

```
coroutine update_screen            coroutine check_file_system
    -- initialize                      -- initialize
    detach                             detach
    loop                               for all files
        ...                                ...
        transfer(cfs)                      transfer(us)
        ...                                ...
                                               transfer(us)
                                           ...
begin         -- main                      transfer(us)
    us := new update_screen                ...
    cfs := new check_file_system
    transfer(us)
```

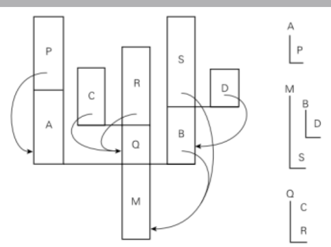

Coroutines

52

- coroutines
  - allow explicit transfer between concurrently running subroutines
  - maintains small context block instead of activation record
  - could be implemented with threads
- cannot share the same stack
  - non-LIFO
  - disjoint, but share same static space
  - use cactus stack instead

Coroutines

53



- each branch to the side is coroutine (A,B,C,D)
- static nesting on right
- static links: arrows; dynamic links: vertical arrangement

Events

54

- event
  - something to which a program needs to respond
  - occurs outside of program at unpredictable time
    - GUI events: keystrokes, mouse motions, button clicks
    - network operations: message arrival
- typically I/O performed synchronously with blocking
- for events, usually want a handler
  - event handler or callback function

Events

55

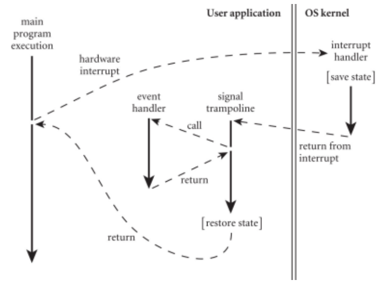
- traditionally, events were handled by interrupts
  - an asynchronous event would trigger an interrupt
  - registers saved
  - jump to predefined address in OS kernel
- in modern systems, most events handled by threads
  - lightweight process
  - threads can be synchronous

55

Events

56

- interrupt handler and signal trampoline



56