Chapter 8 :: Subroutines and Control Abstraction

Introduction

• 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

Review Of Stack Layout

• 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

Review Of Stack Layout (cont.)

• stack frames
  – 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
Review Of Stack Layout

- 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

Calling Sequences

- 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

Calling Sequences

- 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

Calling Sequences

- 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

Calling Sequences

- 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
Calling Sequences

• 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

Calling Sequences

• typical calling sequence

   Direction of stack growth
   (lower addresses)

   Arguments to called
   routines

   Temporaries

   Local variables

   Saved regs.,
   call/return

   Saved flag

   Retain address
   (Arguments
   from caller)

Calling Sequences

• 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

Calling Sequences

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

Calling Sequences

• 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);
      ```
Calling Sequences

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

```c
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

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

```c
infix & tothe; (* exponentiation *)
fun x tothe n = 1.0
    | x tothe n = x * (x tothe (n-1));  (* assume n >= 0 *)

-- right-associative, binary, at precedence level 8
```

Parameter Passing

• parameter passing modes
  – ex.: \texttt{p (x)}
  – call-by-value: \texttt{p} gets a copy of \( x \)’s value
  – call-by-reference: \texttt{p} gets a copy of \( x \)’s address
    • introduces aliases in subroutines, which may be tricky

```c
x : integer
procedure foopy (integer)
    y := 3
    print x
    ...
    x := 2
    foopy()
    print x
```

Parameter Passing

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

• some languages (e.g., ML) allow infix notation (cont.)
  – Fortran: \texttt{A .cross. B}
• some languages use same syntax for control expressions

```c
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

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

```c
x : integer
procedure foopy (integer)
    y := 3
    print x
    ...
    x := 2
    foopy()
    print x
```
Parameter Passing

- 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
procedure foo(x: integer;
    y: integer := 3;
    
    x := 2;
    foo(x);
    print x
    
    -- global
    
Enter function parameters 'x' and 'y'
    
    Boundaries 'x' to 15
    ```

- purpose of call-by-reference
  - to change the value of an actual parameter
  - to avoid time-consuming value copies
    - for some parameters, copying 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

- 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

- C/C++: functions
  - parameters passed by value (C)
  - parameters passed by reference can be simulated with pointers (C)
    ```
    void proc(int *x, int y)
    
    proc(&a, b);
    ```
  - programmers did not like the extra syntax required
    - references introduced in C++

- 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

- 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

- **C/C++**: functions
  - references introduced in C++
    
    ```c
    void proc(int x, int y)
    { x = x + y }
    proc(a, b);
    ```
  - another example
    
    ```c
    void swap(int a, int b) { int t = a; a = b; b = t; }
    ```
- references can be used in other ways as well
  - can also be used as return values
    
    ```c
    int i, j = 1;
    ... j = 2;
    cout << i; // prints 3
    cout << a << b << c;
    ((cout.operator<<(a)).operator<<(b)).operator<<(c);
    ```

- 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

- named parameters
  - examples
    
    ```c
    put(item => 37, base => 8);
    put(base => 8, item => 37);
    put(37, base => 8);
    ```
  - good for complex interfaces
    
    ```c
    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 => 12,
                title_point_size => 13,
                justification => true, hyphenation => false,
                page_num => 3, paragraph_indent => 10,
                background_color => white);
    ```

- default parameters
  - need not be provided by caller
    
    ```c
    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);
    ```

- variable number of arguments
  - examples
    ```c
    #include <cstdio.h>  /* macros and type definitions */
    int printf(char *format, ...)
    {
      va_list argp;
      va_start(argp, format);
      ... char cp = va_arg(argp, char);
      ... double dp = va_arg(argp, double);
      ... va_end(argp);
    }
    ```
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(i : int_array) return integer is
    rt: integer;
    begin
      for i in A'first .. A'last loop
        if i > rt then rt := A(i); end if;
      end loop;
      return rt;
      end A_max;
```

Parameter Passing (cont.)

- SR:
  
```
  procedure A_max (ref A[1..n]: int) returns rt : int
  var
    i, a1, a2 : int;
  begin
    if A[i] > rt then rt := A[i]; end if;
  end A_max;
```

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)

<table>
<thead>
<tr>
<th>implementation mechanism</th>
<th>permissible operations</th>
<th>change to actual?</th>
<th>alias?</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>value</td>
<td>read, write</td>
<td>no</td>
</tr>
<tr>
<td>in, out (Ada)</td>
<td>value or reference</td>
<td>read only</td>
<td>no</td>
</tr>
<tr>
<td>value/result</td>
<td>value or reference</td>
<td>write only</td>
<td>yes</td>
</tr>
<tr>
<td>var, ref</td>
<td>value or reference</td>
<td>read, write</td>
<td>yes</td>
</tr>
<tr>
<td>in, out (Ada)</td>
<td>value or reference</td>
<td>read, write</td>
<td>yes</td>
</tr>
<tr>
<td>name (Algol 68)</td>
<td>closure</td>
<td>read, write</td>
<td>yes</td>
</tr>
</tbody>
</table>

Generic Subroutines and Modules (cont.)

- Ada: generics
- C++: templates

```cpp
  template <typename T, int n_elem>
  class my_queue {
  private:
    int size, size_in_use;
    T *data;
  public:
    my_queue () : size(0), size_in_use(0), data(new T[n_elem]) {} // initialization
    void push(T d) { size_in_use++);
    T pop () { return data[size_in_use - 1];
    size_in_use--;
    return data[size_in_use];
  }
  
  // array-based queue template in C++
```
Generic Subroutines and Modules

• 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

• 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

• C++ example

```cpp
try {
  // protected kind of code

  catchErrorCXX5 (
    // exception error of:
    // doesn't matter if error other than end_of_file
    const char* error)
  { ... };
}

// 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

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

Exception Handling

• 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-raise the exception to propagate upward
  – if recovery not possible, an error message can be
    printed
Coroutines

- 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

Examples:
- screen saver program to prevent liquid crystal burn-in
- file system checks for corrupted files (sanity check)

Example (cont.):
- could be written with coroutines

```
coroutine update_screen
  -- initialize
d To
loop
  ... transfer(c)
... transfer(u)
begin
  us = new update_screen
coroutine check_file_system
  -- initialize
d To
loop
  ... transfer(u)
... transfer(c)
```

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

Events

- 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

- 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