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

Introduction

- 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

Review Of Stack Layout

- 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

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

- 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
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

• 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

• 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

• 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

• typical calling sequence

• 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

• 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

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

• 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++
      \[
      \text{inline int max(int } a, \text{ int } b) \{ \text{return } a > b \ ? a : b; \}\]
    • Ada
      \[
      \text{function max(a, b : integer) return integer is}
      \begin{align*}
      \text{  \quad begin} \\
      \text{    if } a > b \text{ then return } a; \text{ else return } b; \text{ end if;} \\
      \text{  \quad end max;} \\
      \text{  \quad pragma inline(max);} \\
      \end{align*}
      \]

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

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

Parameter Passing

- 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
  ```
  infix 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

- 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].  "Selftalk"
  ```

Parameter Passing

- 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
      procedure footy : integer
      y := 3
      print x
      ...
      x := 2
      footy()
      print x
      ```
  - parameter passing modes (cont.)
    - call-by-value: prints 2 twice
    - call-by-reference: prints 3 twice
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

```plaintext
x : integer
procedure foo :: integer
    y := 2
    print x
    ...
    x := 2
    foo(x)
    print x
```

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

```plaintext
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

• 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

• 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)

```plaintext
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

- 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; }
    ```

- C/C++: functions
  - references can be used in other ways as well
    ```c
    int i;
    int &j = i;
    ...
    j = 2;
    cout << i;  // prints 3
    ```
  - can also be used as return values
    ```c
    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

- 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_base : field = integer’width;
default_base : number_base := 10;
procedure put(item : in integer;
    width : in field := default_width;
    base : in number_base := default_base);
    ```

- 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 => 11, title_point_size => 13,
        justification => true, hyphenation => false,
        page_num => 3, paragraph_indent => 18,
        background_color => white);
    ```

- variable number of arguments
  ```c
  #include <stdio.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);
  }
  ```
Parameter Passing

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

```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
  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 (cont.)

- SR:

```ada
procedure k_max(ref A[:1]: int) returns rtn : int
  begin
    for i := 1 to ub(A) ->
      if A(i) > rtn -> rtn := A(i) fi
eof
  end
```

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

- array-based queue template in C++

```c++
template <class t, int max_items = 100>
class generic {
  int start_idx, end_idx;
public:
  generic() { start_idx = end_idx = 0; // initialization
  void enqueue( t item ) { when( start_idx == end_idx )
    start_idx = (start_idx + 1) % max_items;
    start_idx = (start_idx + 1) % max_items;
  } // enqueue
  void dequeue() { start_idx = (start_idx + 1) % max_items;
    end_idx = (end_idx + 1) % max_items;
  } // dequeue
};
generic<int, 10> gen1;
generic<int, 20> gen2;
```
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
  - 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

- 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

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

Exception Handling

- C++ example
  ```cpp
  try { ...
  // protected block of code
  } catch (std::exception &e) {
  // handle for std::exception not previously caught
  // if this case, the original exception is a valid C++ exception...
  // if not, it's unknown what to do
  }
  // 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
  ```
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

Coroutines

- 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

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

Coroutines

- 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

- 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
• 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