CS 326
Programming Languages, Concepts and Implementation

Instructor: Mircea Nicolescu

Subroutines and Control Abstraction

Language Specification

- General issues in the design and implementation of a language:
  - Syntax and semantics
  - Naming, scopes and bindings
  - Control flow
  - Data types
  - Subroutines

Subroutines and Control Abstraction

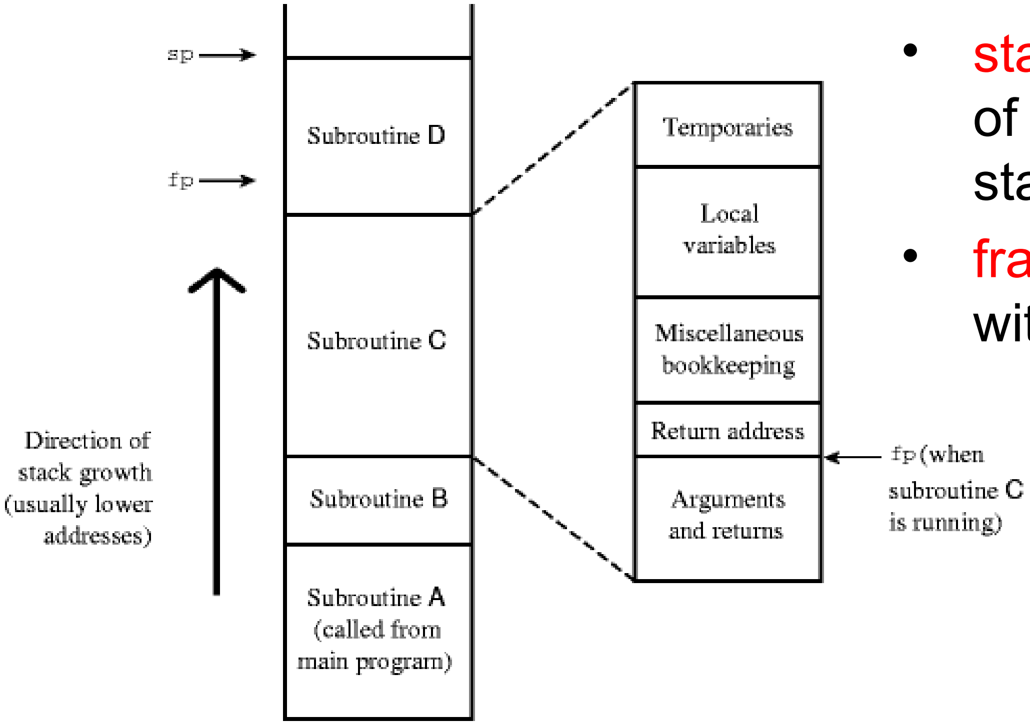
- Abstraction - associate a name with a potentially complex program fragment
  - consider the fragment in terms of its purpose, not implementation
- Control abstraction - purpose corresponds to an operation
  - subroutines (functions, procedures), coroutines, exceptions
- Data abstraction - purpose is to represent information
  - generic data structures, classes (also include control abstraction)
- Subroutine parameters
  - formal parameters - specified in subroutine definition
  - actual parameters - provided when the subroutine is called

Stack Layout

- Recall allocation strategies:
- Static
  - code
  - global variables
  - "own" (static) local variables
  - explicit constants (including strings, sets, other aggregates)
  - small scalars may be stored in the instructions themselves
- Stack
  - parameters
  - local variables
  - temporaries
  - bookkeeping information
- Heap
  - any variables allocated (explicitly or implicitly) on the heap

Stack Layout

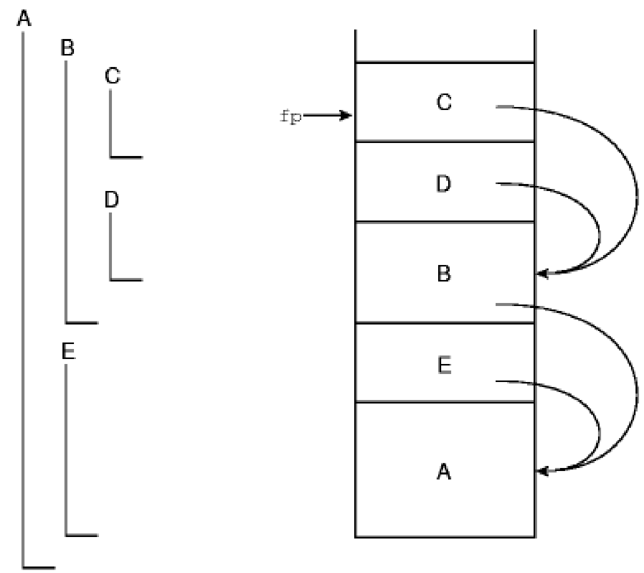
- When calling a subroutine, push a new entry on the stack - stack frame (activation record)
- When retuning from a subroutine, pop its frame from the stack



- stack pointer register (sp) - address of the first unused location at top of stack
- frame pointer register (fp) - address within current stack frame

Stack Layout

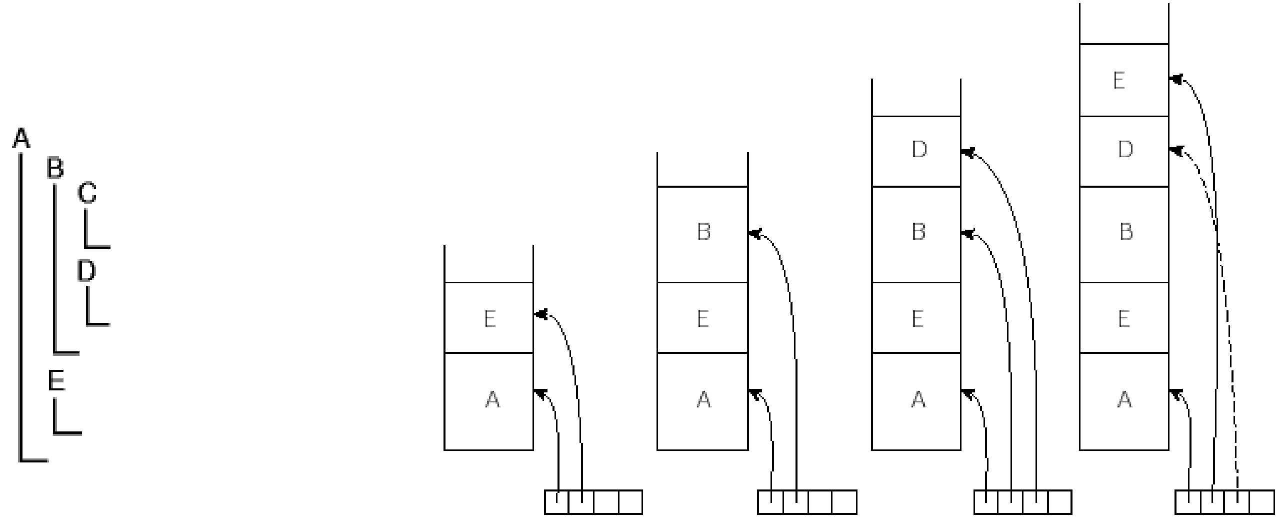
- In a language with nested subroutines - how can we access non-local objects?
  - static chain
  - display



- Static chain - composed of static links:
- Static link - from a subroutine to the lexically surrounding subroutine
- Disadvantage - to access an object k levels deeper → need to dereference k pointers

Stack Layout

- Display:



- Element j in display - reference to most recently called subroutine at lexical nesting level j
- From a subroutine at lexical level i, to access an object k levels outwards:
  - follow only one pointer, stored in element i-k in display
  - constant access time

Calling Sequences

- Maintenance of the stack - responsibility of the calling sequence:
  - code executed by caller immediately before and after subroutine call
  - code executed by subroutine at the beginning (prologue)
  - code executed by subroutine at the end (epilogue)

- Tasks to do around the call:
  - passing parameters
  - saving return address
  - changing program counter
  - allocate a new frame, change stack pointer
  - save registers (including frame pointer)
  - change frame pointer
  - initialization code for local objects

Calling Sequences

- Tasks to do around the return:
  - pass return values
  - finalization code for local objects
  - deallocate frame, restore stack pointer
  - restore saved registers (including frame pointer)
  - restore program counter

- Caller and callee - what does each do?
  - Some tasks must be performed by caller (passing actual parameters), most others can be performed by either caller or callee

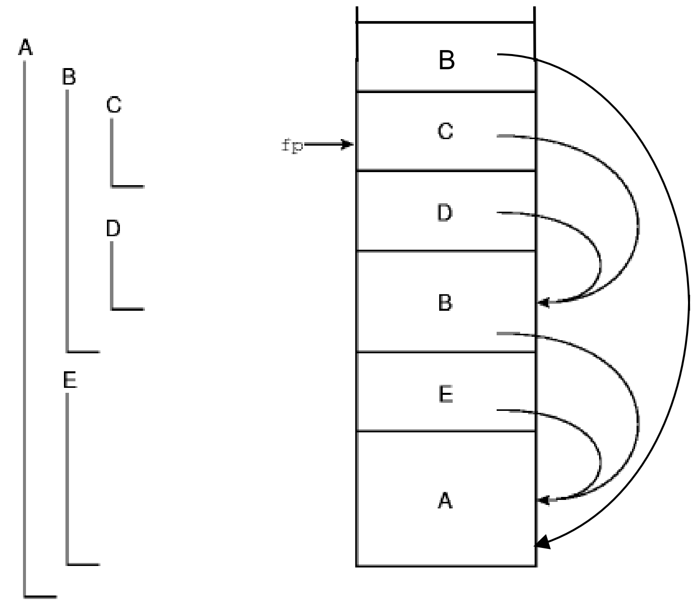
- For space efficiency - put as much as possible in the callee
  - tasks in callee appear once in the target program
  - tasks in caller appear at every point of call

Calling Sequences

- Maintain the static chain (in a language with nested subroutines):
  - done by caller

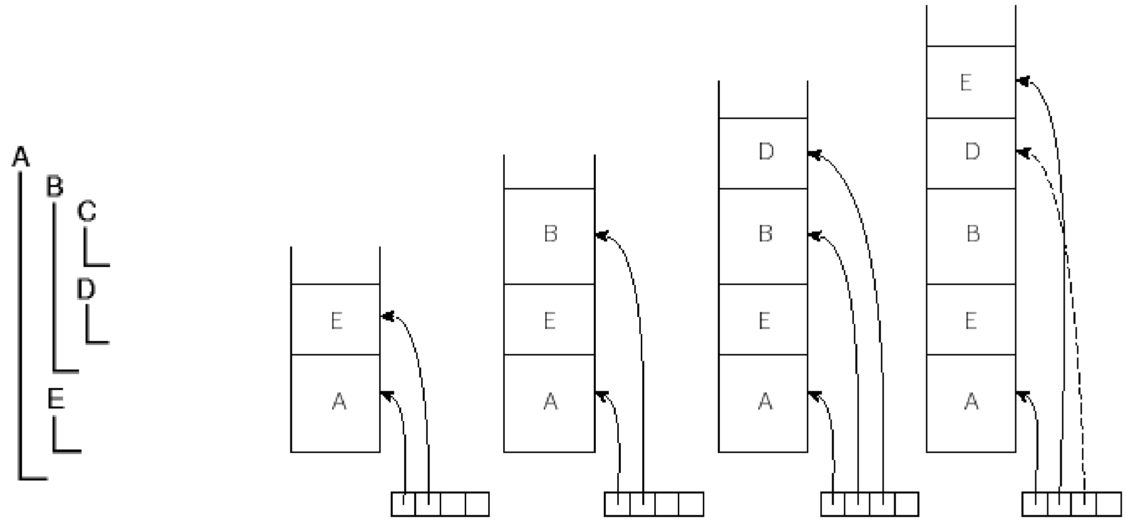
- If callee is nested (directly) inside caller (A calls E)
  - caller passes its own frame pointer as the callee's static link

- If callee is k levels outward (C calls B, k = 1)
  - caller dereferences its own static link k times, and passes the result as the callee's static link



Calling Sequences

- Maintain the display (in a language with nested subroutines):
  - done by callee



- When a subroutine at lexical level j is called (E calls B, j = 1), the callee (B):
  - saves the current j^{th} display element into its stack frame
  - replaces it with a copy of its own frame pointer
  - restores old value upon return

Calling Sequences

- Static chains and displays - tradeoffs:
  - Display element must be saved in procedure prologue and restored in procedure epilogue
    - 2 loads and 2 stores
  - Static link must be passed by caller and saved by callee
    - >=1 load and 1 store
  - Display computes frame pointer for arbitrary non-local variable with 1 load (of display element)
  - Static chain computes frame pointer for non-local variable k levels out with k loads

In-Line Expansion

- In-line expansion - subroutine expanded in-line at the point of call
- Implemented as:
  - macros

    ```
    #define MAX(a,b) ((a) > (b) ? (a) : (b))
    ```

  - in-line functions

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

- Advantages over "regular" subroutines:
  - avoid overhead of stack maintenance
  - allow compiler to perform code improvement across boundaries between subroutines
- Disadvantage
  - increase size of target code (body of subroutine appears at every point of call)

In-Line Expansion

- Comparison - macros vs. in-line functions
  - macros always expand at the point of call
  - declaring a function in-line - suggestion for compiler to expand at the point of call
  - macros use normal-order evaluation
    - problems with side-effects: MAX (x++, y++)
  - in-line functions use applicative-order evaluation
  - macros expand by simple textual replacement (pre-processing time)
  - in-line functions have the same semantics as "regular" functions (compile time)
    - scope rules
    - type checking

In-Line Expansion

- Dealing with recursion
  - cannot expand every call
  - generate a "regular" subroutine, but expand inline just the first instance
    - useful when the recursive call usually does not occur
    - Example – hash table lookup (element usually found in first try):

```
range_t bucket_contents (bucket *b, domain_t x)
{
    if (b->key == x)
        return b->val;
    else if (b->next == 0)
        return ERROR;
    else
        return bucket_contents (b->next, x);
}
```

Parameter Passing

- Issues:
  - implementation mechanism (what is it passed?)
    - value
    - reference
    - name
    - closure
  - legal operations (inside subroutine)
    - read
    - write
  - change performed on actual parameter?
    - yes
    - no
  - change visible immediately?
    - yes
    - no

Parameter Modes

- Main parameter-passing modes:
  - call by value
    - the value of actual parameter is copied into formal parameter
    - the two are independent
  - call by reference
    - the address of actual parameter is passed
    - the formal parameter is an alias for the actual parameter
- Speed – better to pass a large object by reference
- Safety – if a subroutine is allowed to change the actual parameter - pass it by reference
- Semantic issue:
  - argument passed by reference - is it because it's large, or because changes should be allowed?
  - what if we want to pass a large argument, but not to allow changes?

Parameter Modes

- C
  - everything is passed by value
  - arrays are pointers - what is passed by value is a pointer
  - to allow "call by reference" - must pass the address explicitly as a pointer:

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

Parameter Modes

- C
  - Permissible operations – read and write
  - Change on actual parameter – no
  - Alias – no
  - Speed - better to pass the address of a large object
  - How do we prohibit changes to the object?

    void f (const huge_record * r){ ... }

    r <=> pointer to constant huge_record

  - How do we define a constant pointer to huge_record?

    huge_record * const r

Parameter Modes

- Pascal
  - programmer's choice - call by value or call by reference

    procedure (var a : integer, b : integer);   (* a passed by reference *)
    ... (* b passed by value *)

  - if an array is passed without var - it will be passed by value!!
  - var should be used with:
    - arguments that need to be changed
    - large arguments
  - no mechanism to prohibit changes to an argument passed by reference

Parameter Modes

- Languages with reference model (Smalltalk, Lisp, Clu)
  - everything is a reference anyway
  - “call by sharing”
- Ada
  - three parameter modes:
    - in - read only
    - out - write only
    - in out - read and write
  - for scalar types - always pass values
  - call by value/result
    - if it's an out or in out parameter - copy formal into actual parameter upon return
    - change to actual parameter becomes visible only at return

Parameter Modes

- Ada
  - for composite types – pass either a value or a reference
  - program is "erroneous" if the results are different:

```
type t is record
    a, b : integer;
end record;
r : t;

procedure foo (s : in out t) is
begin
    r.a := r.a + 1;
    s.a := s.a + 1;
end foo;

...
r.a := 3;
foo (r);
put (r.a);       -- does this print 4 or 5?
```

  - Passing values – print 4
  - Passing addresses – print 5

Parameter Modes

- C++
  - same modes as in C, plus references:

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

  - safety - use const to prohibit changes to actual parameter
  - references - can be used not only for parameters:

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

  - implementation: j is a pointer to integer
  - semantic: j is treated as an integer (used wherever an integer is expected)

Parameter Modes

- Call by name (Algol 60, Simula)
  - parameters are re-evaluated in the caller's referencing environment every time they are used
  - similar to a macro (textual expansion)
  - pass a hidden routine (thunk) - re-evaluates the parameter
  - Example (Jensen's device):

```
real procedure sum (expr, i, low, high);
    value low, high;
        comment low and high are passed by value;
        comment expr and i are passed by name;
    real expr;
    integer i, low, high;
begin
    real rtn;
    rtn := 0;
    for i := low step 1 until high do
        rtn := rtn + expr;
        comment the value of expr depends on the value of i;
    sum := rtn
end sum
```

To evaluate the sum:

$$y = \sum_{1 \le x \le 10} 3x^2 - 5x + 2$$

Call:

y := sum (3*x*x - 5*x + 2, x, 1, 10);

Special-Purpose Parameters

- Conformant (open) arrays
  - formal parameter array whose shape is determined at run-time
  - Pascal – shape of all arrays must be known at compile time, except for formal parameters (conformant arrays):

```
procedure apply_to_A (function f (n : integer) : integer;
                                var A : array [low..high : integer] of
integer);
var i : integer;
begin
            for i := low to high do
                  A[i] := f (A[i]);
end;
```

Special-Purpose Parameters

- Default (optional) parameters
  - Specify default values for parameters in subroutine declaration
  - If parameters are missing from the call, the default values are used
  - Example in Ada (writes a number on a specific width in characters, in a specific base):

```
procedure put (item : in integer;
               width : in integer := 11
               base : in integer := 10);
...
put (37);          -- equivalent to put (37, 11, 10)
```

  - What if we want to specify the base, but not the width?

Special-Purpose Parameters

- Named (keyword) parameters
  - Syntax allows to specify what parameters are passed (instead of positional notation)
  - Same example in Ada:

```
procedure put (item : in integer;
                width : in integer := 11
                base : in integer := 10);
...
put (item => 37, base => 8);    -- equivalent to put (37, 11, 8)
put (base => 8, item => 37);    -- equivalent to put (37, 11, 8)
put (37, base => 8); -- equivalent to put (37, 11, 8)
```

Special-Purpose Parameters

- Variable numbers of arguments (in C, C++, Common Lisp)
  - Example in C (compute the average of any number of integers):

```
x = average( 2, 3, 4, -1 );   /* use -1 as a terminator */
x = average( 2, 3, 4, 5, -1 );              /* use -1 as a terminator */

int average( int first, ... )     / * must always specify the first parameter */
{                        / * … is part of the syntax */
   int count = 0, sum = 0, i = first;
   va_list marker;

   va_start( marker, first );                 /* Initialize variable arguments */
   while( i != -1 )
   {
      sum += i;
      count++;
      i = va_arg( marker, int);                        /* Returns next argument (must know the type) */
   }
   va_end( marker );                          /* Reset variable arguments */
   return( sum ? (sum / count) : 0 );
}
```

Special-Purpose Parameters

- Variable numbers of arguments
  - Example in C (the main function):

```
void main ( int argc,          /* Number of arguments, including the program name */
            char * argv [] )   /* Array of strings, one for each command-line argument */
{
   int count;

   printf ( "\nCommand-line arguments:\n" );
   for ( count = 0; count < argc; count++ )
      printf ( "  argv[%d]   %s\n", count, argv[count] );
}
```

  - If the program is executed with the command:

```
my_prog.exe 35 foo
```

  - The output is:

```
Command-line arguments:
  argv[0]   C:\MSC\MY_PROG.EXE
  argv[1]   35
  argv[2]   foo
```

Announcements

- Readings
  - Chapter 9
- Homework
  - HW 6 out – due on April 23
  - Submission
    - at the beginning of class
    - with a title page: Name, Class, Assignment #, Date
    - preferably typed

Generic Subroutines and Modules

- Subroutines
  - allow operations on different *values*

- Generic subroutines and modules
  - allow operations on different *types*
  - Generic modules – useful to create containers (lists, stacks, queues, trees) for any elements
  - Generic subroutines – used in generic modules, also by themselves

Generic Subroutines and Modules

- Example (generic queue module 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;
```

- How are generics implemented?

Generic Subroutines and Modules

- Static mechanism
  - create code for each specific module/subroutine at compile time
  - separate copy for each instance
- Similar to macros/in-line subroutines
  - "context-sensitive macro facility" (Ada)
  - type checking
  - applicative order evaluation
  - scoping rules

Exception Handling

- Exception
  - unusual condition detected at run time
  - may require to "back-out" from several levels of subroutine calls
- Examples:
  - arithmetic overflow
  - end-of-file on input
  - wrong type for input data
  - user-defined conditions (not necessarily errors), raised explicitly
- "Traditional" ways to handle such situations:
  - return some default value when cannot produce an acceptable one
  - return (or have an extra parameter) an explicit "status" value, to be inspected after each call
  - pass a closure for error handling, to be called in case of trouble

Exception Handling

- C++, Ada, Java, ML – structured approach:
  - handlers (catch in C++) are lexically bound to blocks of protected code (the code inside a try block in C++)

- Exception propagation:
  - if an exception is raised (throw in C++):
    - if the exception is not handled in the current subroutine, return abruptly from subroutine
    - return abruptly from each subroutine in the dynamic chain of calls, until a handler is found
    - if found, execute the handler, then continue with code after handler
    - if no handler is found until outermost level (main program), terminate program

Exception Handling

- Example (C++):

```
void f ()
{
    ...
    try
    {
        g();
    }
    catch (exc)
    {
        // handle exception of type exc
    }
    ...
}
```

```
void g ()
{
    ...
    h();
    ...
}

void h()
{
    ...
    if (...)
        throw exc();
    ...
}
```

Exception Handling

- Usage for exception handling mechanisms:
  - perform operations to recover, and then continue execution
    - allocate more memory
    - recover from errors in a compiler
  - cannot recover locally, but:
    - may want a local handler just to clean up some local resources
    - then re-raise the exception to be handled by a "higher authority"
  - terminate, but first print a helpful error message
- Advantages:
  - uniform manner of handling errors
  - handle errors exactly where we want, without checking for them explicitly everywhere they might occur
  - subroutine documentation – specify what exceptions might be raised by a subroutine → subroutine user may want to catch them

Definition of Exceptions

- Parameterized exceptions – the code which raises the exception can pass additional information with it
  - C++:

```
class duplicate_in_set
{
        item dup;        // element that was inserted twice
        …
}
...
throw duplicate_in_set (d);
```

  - Ada, Common Lisp:
    - exceptions are just tags – no other information than the exception name

Exception Propagation

- C++ (predefined exceptions):

```
try {
...
// protected block of code
...
}
catch (end_of_file) {      // derived from io_error
...
}
catch (io_error e) {       // any io_error other than end_of_file
...
}
catch (...) {              // all other exceptions; ... is part of syntax
...
}
```

  - handler matches exception if it names a class from which exception is derived
  - can declare an exception object (e) – access additional information passed with the exception

Implementation of Exceptions

- How can we keep track of handlers?
  - maintain a separate stack of handlers
  - when entering a subroutine (in the prologue)
    - push all its exception handlers on the handler stack
  - when returning from a subroutine (in the epilogue)
    - pop all its exception handlers from the handler stack
  - if an exception is raised
    - check if any current handler (top of stack) matches the exception
    - if yes, execute it
    - if no, perform the subroutine epilogue and return, then check again in the caller
- Problem
  - run-time overhead (maintaining the handler stack) even in the common case, when no exceptions occur
  - can we do better?

Implementation of Exceptions

- Alternative solution:
  - at compile time, build a table of handlers
  - each entry - two fields:
    - the address of the protected code
    - the address of the corresponding handler
  - if an exception is raised
    - binary search for the address of the current block in the table
    - if found and the handler matches the exception, execute the handler
    - otherwise, return and repeat for the caller's code
- Properties
  - if an exception occurs
    - higher run-time cost than in previous solution - search for addresses in the table
  - in the common case (no exceptions raised)
    - zero run-time cost - the table is built at compile time

Iterators

- (Discussed in Chapter 6 of the textbook)
- Traverse an array
  - compute maximum element, average of all elements, display elements, etc.
  - write an enumeration-controlled (for) loop every time
    - easy
- Traverse a tree
  - compute maximum node, average of all nodes, count number of nodes, etc.
  - write some (recursive) code to do it
    - more complex, and not convenient to do it every time
  - would be nice to just use something similar to a for loop, that hides the details
- Iterator - control abstraction that allows enumerating the items of an abstract data type

Iterators

- Clu – a simple enumeration-controlled loop is implemented as an (built-in) iterator:

```
for i in from_to_by (first, last, step) do
    ...
end
```

```
from_to_by = iter (from, to, by : int) yields (int)
    i : int := from
    if by > 0 then
        while i <= to do
            yield i
            i +:= by
        end
    else
        while i >= to do
            yield i
            i +:= by
        end
    end
end from_to_by
```

  - yield – returns control with current value of i
  - Next iteration – continues from where it has left

Iterators

- Icon – iterators are called generators
  - The enumeration-controlled loop in Icon:

```
every i := first to last by step do
{
   ...
}
```

  - ...to...by... is a built-in infix generator

- More Icon generators – usually operating on strings:
  - find (substr, str) – generates the positions in str where substr appears
  - upto (chars, str) – generates the positions in str where any character in chars appears

Iterators

- Icon – print all positions in string s that follow a blank:

```
every i := 1 + upto (' ', s) do
{
    write (i)
}
```

  - can be also written as:

```
every write (1 + upto (' ', s))
```

- Any user-defined subroutine in Icon can be a generator:
  - needs to use suspend expr instead of return expr
  - suspend is the equivalent of yield in Clu
  - suspend returns control from the iterator (with the generated value), but saves its state for the next iteration

Iterators

- Euclid – iterators are emulated
  - syntax of for loops allows using a generator module
  - the module must export:
    - variables named value and stop
    - a procedure named next

- Example (traverse a tree):

```
for n in TreeIterModule loop
    ...
end loop
```

- which is equivalent to:

```
begin
    var ti : TreeIterModule
    loop
                                exit when
    ti.stop
                                n := ti.value
                                ...
                                ti.next
    end loop
end
```

Coroutines

- Coroutines – execution contexts that exist concurrently, but that execute one at a time
  - transfer control to each other explicitly, by name
- Implementation
  - closure – a code address and a referencing environment
  - transfer – jump through a non-local goto, after saving current state
- Coroutines provided in Simula, Modula-2
- Useful for implementing:
  - iterators
  - threads
  - servers
  - discrete event simulation

Coroutines

- Example – a "screen-saver" program:
  - paints a picture
  - in the background, checks the disk for corrupted files
- Without coroutines:

```
loop
   -- update a portion of the picture on screen
   -- perform next "step" of file system checking
```

- Problem:
  - not every task can be easily broken into "steps"
  - with regular subroutines – upon return, all information in the activation frame is lost
  - the code may have a complex structure (many nested loops) - hard to save/restore the state

Coroutines

- Example – a "screen-saver" program, with coroutines:

  ("loose Simula syntax")

```
us, cfs : coroutine

coroutine update_screen
    -- initialize
    detach
    loop
        ...
        transfer (cfs)
        ...

coroutine check_file_system
    -- initialize
    detach
    for all files
        ...
        transfer (us)
        ...
            transfer (us)
        ...
        transfer (us)
        ...

begin            -- main
    us := new update_screen
    cfs := new check_file_system
    resume (us)
```

Iterators as Coroutines

- Easy to implement iterators with coroutines:

```
for i in from_to_by (first, last, step) do
   ...
end
```

- Compiler translates this to:

```
it := new from_to_by (first, last, step, i, done, current_coroutine)
while not done do
   ...
   transfer (it)
destroy (it)
```

Iterators as Coroutines

- The coroutine that implements the iterator from_to_by is:

```
coroutine from_to_by (from, to, by : int;
        ref i : int; ref done : bool; caller : coroutine)
    i := from
    if by > 0 then
        done := from <= to
        detach
        loop
            i +:= by
            done := i <= to
            transfer (caller)     -- yield i
    else
        done := from >= to
        detach
        loop
            i +:= by
            done := i >= to
            transfer (caller)     -- yield i
```