2 Exceptions

2.1 Dynamic Multi-Level Exit

- **Modularization**: any contiguous code block can be factored into a (helper) routine and called from anywhere in the program (modulo scoping rules).

- Modularization fails when factoring exits, e.g., multi-level exits:

```
B1: for ( i = 0; i < 10; i += 1 ) {
        . . .
        B2: for ( j = 0; j < 10; j += 1 ) {
                . . .
                if ( . . . ) break B1;
                . . .
        }
        . . .
}
```

```
void rtn( . . . ) {
        B2: for ( j = 0; j < 10; j += 1 ) {
                . . .
                if ( . . . ) break B1;
                . . .
        }
}
B1: for ( i = 0; i < 10; i += 1 ) {
        . . .
        rtn( . . . )
        . . .
}
```

- **Fails to compile because labels only have routine scope.**

- ⇒ ***among*** routines, control flow is controlled by call/return mechanism.

  - given A calls B calls C, it is impossible to transfer directly from C back to A, terminating B in the transfer.

- Fundamentally, a routine can have multiple kinds of return.

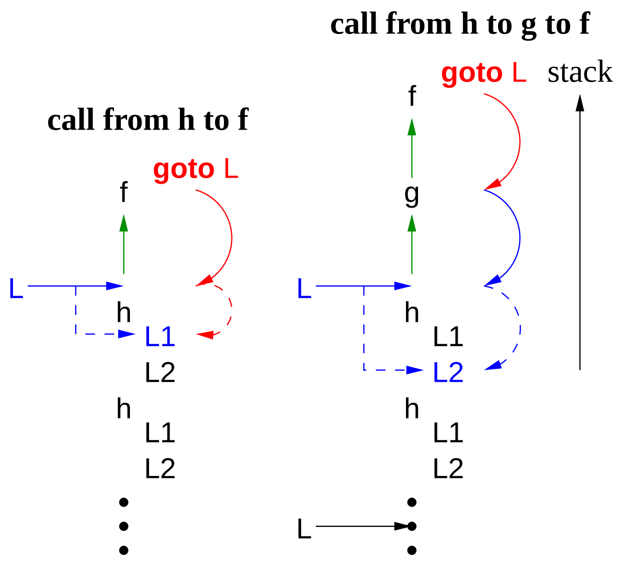
  - routine call returns normally, i.e., statement after the call
  - exceptional returns, i.e., control transfers to statements **not** after the call

```
C    Two alternate return parameters, denoted by * and implicitly named 1 and 2
          subroutine AltRet( c, *, * )
               integer c;
               if ( c == 0 ) return       ! normal return
               if ( c == 1 ) return 1     ! alternate return
               if ( c == 2 ) return 2     ! alternate return
          end
C    Statements labelled 10 and 20 are alternate return points
          call AltRet( 0, *10, *20 )
          print *, "normal return 1"
          call AltRet( 1, *10, *20 )
          print *, "normal return 2"
          return
10        print *, "alternate return 1"
          call AltRet( 2, *10, *20 )
          print *, "normal return 3"
          return
20        print *, "alternate return 2"
          stop
          end
```

- Generalization of multi-exit loop and multi-level exit.
  - control structures end with or without an exceptional transfer.
- Pattern addresses fact that:
  - algorithms can have multiple outcomes
  - separating outcomes makes it easy to read and maintain a program
- Pattern does not handle case of multiple levels of nested modularization.
  - if AltRet is further modularized, new routine has to have an alternate return to AltRet and then another alternate return to its caller.
  - Rather than two step operation, simpler for new modularized routine to bypass intermediate step and transfer directly to caller of AltRet.
- **Dynamic multi-level exit** extend call/return semantics to transfer in the ***reverse*** direction to normal routine calls, called **non-local transfer**.

Many routines have non primary exits (dont go to the next statement after the call). All looping structures are stealthily gotos. Despite this we should only ever manually use gotos to simulare a labled break.

```
label L;
void f( int i ) {
      // non-local return
      if ( i == ... ) goto L;
}
void g( int i ) {
      if ( i > 1 ) { g( i - 1 ); return; }
      f( i );
}
void h( int i ) {
      if ( i > 1 ) { h( i - 1 ); return; }
      L = L1; // set dynamic transfer-point
      f( 1 ); goto S1;
  L1:  // handle L1 non-local return
  S1:  // continue normal execution
      L = L2; // set dynamic transfer-point
      g( 1 ); goto S2;
  L2:  // handle L2 non-local return
  S2:  // continue normal execution
}
```



- Non-local transfer mechanism is a label variable containing the tuple:
  1. pointer to a block activation on the stack
  2. transfer point within the block.
- Non-local transfer, **goto** L, in f is a two-step operation:
  1. direct control flow to the specified activation on the stack;
  2. then go to the transfer point (label) within the routine.
- Therefore, a label value is not statically/lexically determined.
  - recursion in g ⇒ unknown distance between f and h on stack.
  - what if L is set during the recursion of h?
- ***Transfer between goto and label value causes termination of stack block.***
- First non-local transfer from f transfers to the label L1 in h's routine activation, terminating f's activation.
- Second non-local transfer from f transfers to the static label L2 in the stack frame for h, terminating the stack frame for f and g.
- Termination is implicit for direct transferring to h or requires stack unwinding if activations contain objects with destructors or finalizers.
- Non-local transfer is possible in C using:
  - jmp_buf to declare a label variable,

Usually labels are constants, but by saying `label L` we make a variable label.