Control

It would seem, then, that the detail is worked out with more precision if the control is private.
-- Aristotle

- Sequence 7.2
  - statement blocks
- Selection 7.3
  - if-then-else, if-then
  - case, switch, break
- Iteration 7.4
  - conditional loops
  - indexed loops
- Goto! 7.5
- Guards! 7.6

Control Statements

Only the simplest programs would consist of a single statement or of a group of statements all executed in turn. More power can be gained from the dynamic selection of alternate paths through a program.

There are three fundamental ways to arrange statements in a program to control the order of their execution.

- sequence (compound statement)
  - begin-end
- selection (conditional statement)
  - if-then-else
- iteration (loop statement)
  - while-do

The Fundamentals

Given the three fundamental mechanisms of sequence, selection and iteration, all other control structures can be derived.

- `if` *C* `then` *S*
  - `if` *C* `then` *S* `else nop`
- `repeat` *S* `until` *C*
  - *S*
`while` ~*C* `do` *S*
- `for` $i \leftarrow lo$ `to` *hi* `do` *S*
  - $i \leftarrow lo$
`while` $i \le hi$ `do`
    S
    $i \leftarrow succ(i)$
`end`

Sequence

The *sequence mechanism* allows the execution of two or more statements in the order in which they are written.

Duh!

Why is this even worth mentioning?

Because it is an important abstraction to be able to consider a group of statements and a single statement to be basically the same thing.

By defining *sequence* we can define the other control mechanisms without requiring special definitions for single or multiple statements.

Sequence Syntax

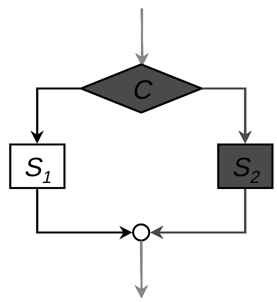
The sequence mechanism is implemented through *blocks* (as already discussed (♬ 105)) — *delimited groups of statements*.

- Algol, Pascal, Ada
  - ```
    begin
       statement1 ... statementN
    end
    ```
- C, C++, Java
  - ```
    {
       statement1 ... statementN
    }
    ```
- Fortran
  -

Selection

The fundamental *selection mechanism* allows for two alternate paths of execution based on the evaluation of a conditional expression. The two paths and the conditional expression form a triple, making selection a *ternary operator*.

- *select(C, S_1, S_2)*
  - evaluate *C*, execute either statement *S1* or statement *S2*



Nested Selection

In general, for the selection statement *select(C, S_1, S_2)*, S_1 and S_2 can be any statement: a single statement, a compound statement, or even another selection statement. S_2 may also be nop (which is implemented as an if-then statement instead of an if-then-else statement in most languages).

- `if` C_1 `then if` C_2 `then` S_1 `else` S_2
  - `if` C_1 `then ( if` C_2 `then` S_1 `else` S_2 `)`
  - `if` C_1 `then ( if` C_2 `then` S_1 `) else` S_2

Resolving the Ambiguity

Different languages resolve selection statement ambiguity for `if` C `then` S_1 `else` S_2 in different ways:

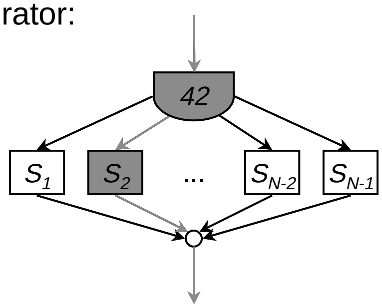
- Algol 60
  - S_1 may be a compound statement but not a selection statement
  - `if` C_1 `then begin if` C_2 `then` S_1 `else` S_2 `end`
  - `if` C_1 `then begin if` C_2 `then` S_1 `end else` S_2
- Pascal, C, C++, Java
  - S_2 is assumed to be nested with the most recent `then`
  - `if` C_1 `then if` C_2 `then` S_1 `else` S_2
- Ada, Modula-2
  - require an “end of if statement” indicator
  - `if` C_1 `then if` C_2 `then` S_1 `else` S_2 `endif endif`
  - `if` C_1 `then if` C_2 `then` S_1 `endif else` S_2 `endif`

Multiple Selection: Case

The *selection statement* allows for two different paths of execution based on the evaluation of a *boolean expression* (one of two possible values).

Multiple selection is possible by choosing execution paths based on the evaluation of an expression having one of many possible values (such as an integer-valued expression). The *case* statement is an n-ary operator:

- *case(E,* S_1*,* S_2*, ...,* S_{N-1}*)*



Case Statement Syntax

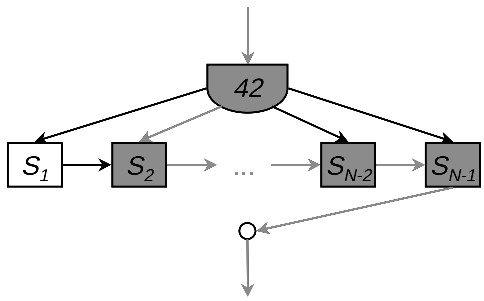
- Pascal
  - `case` *E* `of`
    constlist$_1$: S_1;
    constlist$_2$: S_2;
    ...
    constlist$_{N-2}$: S_{N-2};
    `else` S_{N-1}
`end`
- Ada
  - `case` *E* `is`
    `when` *constlist*$_1$ `=>` S_1;
    `when` *constlist*$_2$ `=>` S_2;
    ...
    `when` *constlist*$_{N-2}$ `=>` S_{N-2};
    `when others` S_{N-1};
`end case`

In either language *constlist* can be:

- a single constant
  - `13`, `'a'`, `tuesday`
- multiple comma-separated constants
  - `1, 3, 5, 7`
- a range constant
  - `mon..fri`, `13..42`

Multiple Selection: Switch

C has a variation of the *case* statement called the *switch* statement, which is more of an elaborate *goto* structure than it is a case statement.



Switch Statement Syntax

- switch(*E*)
  {
  case $const_1$: S_1;
  case $const_2$: S_2;
  ...
  case $const_{N-2}$: S_{N-2};
  default: S_{N-1};
  }

- *E* must evaluate to an integer value (including `char`)
  - `13, 'a'`
- $const_i$ must be a single integer constant
  - `13, 'a'`
- as soon as a constant $const_i$ matches the value of *E*, its statements and all subsequent statements are executed in sequence

Break

C has a special statement invoked by the reserved word `break`. The break statement is basically a *goto* that transfers control unconditionally to the first statement outside the current block.

```
{
   ...
   break;
   ...
}
```

S_o

In the example, when control reaches the `break`, execution continues immediately with statement S_o.

Simulating the Case Statement in C

There are two things about C's *switch* statement that distinguish it from the Pascal/Ada-style *case* statement:

- all statements after the first matching $const_i$ are executed
- each $const_i$ must be a single constant only

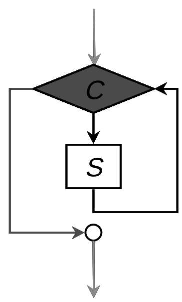
But we can get around these differences using `break`:

```
switch(E) {
  case const1:
  case const2:
  case const3: S1; break;
  case const4: S2; break;
  ...
  case constM: SN-2; break;
  default: SN-1;
}
```

Iteration

The fundamental *iteration mechanism* allows a statement to be executed repeatedly based on the evaluation of a conditional expression. The statement and the conditional expression form a *binary operator*:

- *iterate(C, S)*
  - evaluate *C*, either (execute statement *S* and repeat) or continue



Pretest vs. Posttest

A design decision for iteration is when C is evaluated

- *pretest*
  - evaluate *C*
    - *C* is true
      - execute *S*
      - repeat
    - *C* is false
      - continute executing after *S*
- *posttest*
  - execute *S*
  - evaluate *C*
    - *C* is true
      - repeat
    - *C* is false
      - continute executing after *S*

Iteration Syntax

- Pascal
  - `while` C_1 `do` S
  - `repeat` S `until` C_2
- C
  - `while(`C`)` S
  - `do` S `while(`C`);`
- Ada
  - `while` C `loop` S `end loop;`
  -

Are pretest *and* posttest *the only possibilities?*

Generic Loops

Ada has a very general loop construct:

- `loop` S `end loop;`

The standard *pretest* (while-do) and *posttest* (do-while) can be fashioned from the general loop:

- ```
  loop
      exit when not C;
      S
  end loop;
  ```
- ```
  loop
      S
      exit when not C;
  end loop;
  ```

Intest

Ada's `exit` statement can appear anywhere in a `loop`, allowing very powerful iterative constructs:

```
sum := 0;

loop
   get(x);
   exit when x = 0;
   sum := sum + x;
end loop;

put(sum);
```

```
sum := 0;
get(x);

while x /= 0 loop
   sum := sum + x;
   get(x);
end loop;

put(sum);
```


Indexed Loops

The general iterative construct can have any boolean or relational expression for C, allowing any loop exit condition. Often we need to repeat a loop a known (or predictable) number of times.

An *indexed loop* (or *counter-controlled loop* or *for-loop*) combines a counter variable, an initial counter value, a counter adjustment and a final counter value, giving a 5-ary operator:

- *count(Ctr, V_1, V_2, succ, S)*
  - set *Ctr* to V_1
  - compare *Ctr* to V_2
    - *Ctr* > V_2?
  - execute S
  - set *Ctr* to *succ(Ctr)*

For-loop Syntax

- Pascal
  - `for` *Ctr* `:=` V_1 `to` V_2 `do` *S*
  - `for` *Ctr* `:=` V_1 `downto` V_2 `do` *S*
  - *Ctr*, V_1 and V_2 must be ordinals and compatible
- Ada
  - `for` *Ctr* `in` V_1`..`V_2 `loop` *S* `end loop;`
  - `for` *Ctr* `in reverse` V_1`..`V_2 `loop` *S* `end loop;`
  - the `for` keyword declares *Ctr* as local to the loop
  - *Ctr*'s value cannot be changed explicitly within the loop
  - `end loop` marks the end of the lifetime of *Ctr*

For-loop Weirdos

- Algol 60
  - `for` *Ctr* `:=` *exprlist* `do` *S*
  - `for` *Ctr* `:=` *V1* `step` *Incr* `until` *V2* `do` *S*
  - `for` *Ctr* `:=` *expr* `while` *C* `do` *S*
  - combinations are legal
    - `for i := 1, 4, 9, i+1 while i <= 50, 2, 3 step 2 until 21 do S`
- C
  - `for(`*expr1list*`;` *expr2*`;` *expr3list*`)` *S*
  - execute *expr1list*
  - evaluate *expr2*
    - nonzero?
      - execute *S*
      - execute *expr3list*
    - zero?

Goto!

Most imperative languages allow *unconditional branching* (goto!). It was decided (pretty much before most of us were born) that unconditional branching is a *bad thing*.

The unconditional branching mechanism transfers control to a specified location in a program, usually indicated by a *label* on a statement (making *goto* a unary operator):

- *goto(Label)*
  - evaluate *Label* (if necessary/allowed)
  - go there

Goto Syntax

Pascal

- ```
  label 1, 5;
  begin
    ...
    goto 1;
    ...
  1: ...
  end;
  ```
- labels must be declared
- labels must be numbers
- all gotos local

Ada

- ```
  ...
    goto MYLABEL;
    ...
  <<MYLABEL>>
    ...
  ```
- cannot jump into an `if` or `loop`
- cannot jump between `then` and `else`
- cannot jump into or out of subprograms
- can jump out of `loop`

C

- ```
  ...
    goto mylabel;
    ...
  mylabel: ...
  ```
- labels have “function scope”
- cannot jump into a block that declares a dynamically-typed variable
  ```

Goto Varieties

- Restricted goto:
  - some languages place restrictions on the legal targets of goto
    - cannot jump into S in an iterative construct
    - cannot jump into S in a selection construct
    - cannot jump into or out of subprograms
- Super turbo goto 2000 SE:
  - some languages allow Label to be a variable
    - “landing site” not known until runtime
    - impossible to verify “legality” of target
    - label value can be passed as a parameter to a subprogram

Guards!

Dijkstra proposes very general select and iterative constructs:

- if-fi
  - if

    guard$_1$ → statements$_1$ []
    guard$_2$ → statements$_2$ []
    ... []
    guard$_m$ → statements$_m$
    fi
- do-od
  - do

    guard$_1$ → statements$_1$ []
    guard$_2$ → statements$_2$ []
    ... []
    guard$_m$ → statements$_m$
    od

- a *guard* is a boolean expression
- [] is a separator
- all guards are evaluated in parallel
- choose *any* true guard and execute its statements
- choice (when more than one true guard) is non-deterministic