Chapter 6. Control Flow

- ***Ordering in program execution*** is fundamental to most models of computing.

- We can organize the language mechanism used to specify ordering into eight principal categories:

  (1) ***Sequencing***: Statements are to be executed in a certain specified order—usually ***the order in which they appear*** in the program text.

  (2) ***Selection***: A ***choice*** is to be made among two or more statements or expressions. (e.g.) ***if*** and ***case (switch)*** statements

  (3) ***Iteration***: A given fragment of code is to be ***executed repeatedly***, either a certain number of times, or until a certain run-time condition is true. (e.g.) ***for/do***, ***while***, and ***repeat*** loops

  (4) ***Procedural abstraction***: A potentially complex collection of control constructs (a subroutine) is encapsulated in a way that allows it to be treated as a single unit.

  (5) ***Recursion***: An expression is ***defined in terms of (simpler version of) itself***, either directly or indirectly.

  (6) ***Concurrency***: Two or more program fragments are to be ***executed/evaluated "at the same time", either in parallel on separate processors, or interleaved on a single processor*** in a way that achieves the same effect.

  (7) ***Exception handling and speculation***: If some ***expected condition turns out to be false, execution branches to*** a handler that executes in place of the remainder of the protected fragment (: exception handling), or in place of the entire protected fragment (: speculation).

  (8) ***Nondeterminacy***: The ordering or choice among statements or expression is ***deliberately left unspecified***, implying that any alternative will lead to correct results.

- These eight principal categories cover all of the control-flow construct and mechanisms found in most programming languages.

6.1 Expression Evaluation

- An expression generally consists of either (1) ***a simple object*** (e.g. a literal constant, or a named variable or constant) or (2) ***an operator or function applied to a collection of operands or arguments***, each of which in turn is an expression.

(Note) It is conventional to use the term ***"operator" for built-in functions that use special, simple syntax***, and to use the term ***"operand" for an argument of an operator***.

(ex) A typical function call: myFunc (A, B, C)

name — a parenthesized, comma-separated list of arguments

(ex) Typical operators take only one or two arguments, and dispensing with the parentheses and commas: a+b, -c

(Note) Some languages define their operators for more normal-looking functions:
(1) In ***Ada***, a+b is short for ***"+" (a, b)***
(2) In ***C++***, a+b is short for ***a.operator+ (b)***

- In general, a language may specify that function calls (operator invocations) employee ***prefix***, ***infix***, or ***postfix*** notation.

(1) prefix: ***op a b*** or ***op(a, b)*** or ***(op a b)***

"Cambridge Polish" places the function name inside the parentheses

(2) infix: ***a op b***

(3) postfix: ***a b op***

- Most imperative languages use ***infix notation for binary operators*** and ***prefix notation for unary operators and (with parentheses around the arguments) other functions***.

- LISP uses Cambridge Polish notation.

(ex) (* (+ 1 3) 2) → (1 + 3) * 2 in infix
1 3 + 2 * in postfix

(* (+ A B) (- C D)) → (A + B) * (C – D) in infix
A B + C D - * in postfix

- Multiword infix notation

(ex) In Algol, a:= if b<>0 then a/b else 0 ;

three-operand infix operator

In C, a = b!=0 ? a/b : 0 ;

- Postfix notation
  (ex) In Pascal, pointer dereferencing operator (^)
  In C and its descendants, post-increment and decrement operators (++ and --)

6.1.1 Precedence and Associativity

- When written in infix notation, without parentheses, these operators lead to ambiguity as to what is an operand of what.

  (ex) Fortran uses ** for exponentiation.
  How should we parse a + b * c ** d ** e / f ? ***Ans. 3***

  (1) ((((a + b) * c) ** d) ** e) / f or
  (2) a + (((b * c) ** d) ** (e / f)) or
  (3) a + ((b * (c ** (d ** e))) / f)

  (Note) ** : ***right to left***
  *, / : ***left to right***

  In Fortran, 2 ** 1 ** 2 = = 2
  In Ada, A ** B ** C is illegal, should be (A ** B) ** C or A ** (B ** C)

- In any given language, the choice among alternative evaluation orders depends on the ***precedence*** and ***associativity*** of operators

  → specify that certain operators, in the absence of parentheses, group "more tightly" than other operators

  → specify whether sequence of operators of equal precedence group to the right or to the left

(Note) ***Issue of precedence and associativity do not arise in prefix or postfix notation.***

- See Figure 6.1 Operator precedence levels in Fortran, Pascal, C, and Ada (p228 of the textbook)

- Because the rules for precedence and associativity vary so much, ***it is wise to make liberal use of parentheses***.

6.1.2 Assignments

- Each ***assignment takes a pair of arguments***: a value and a reference to a variable into which the value should be placed.

- In general, a programming language construct is said to have a "***side effect***" if it influences subsequent computation (and ultimately program output) in any way other than by returning a value for use in the surrounding context.

- Assignment is perhaps the most fundamental side effect.

- References and Values

(ex) d = a ;
→ refers to the value of a : ***r-value***

a = b + c ;
→ refers to the location of a : ***l-value***

Both interpretations are possible because a variable in C (and in Pascal, Ada, and many other languages) is a named container for a value.

(ex) Not all expressions can be l-values:
2+3 = a , a = 2+3
if constant

In C, (f(a) + 3) → b [c] = 2 ;
c-th element of field b of the structure
pointed at by the third array element after the one to which
f's return value points

In C++, g(a).b[c] = 2 ;
returns a reference to a structure

(ex) Languages like Algol68, Clu, Lisp/Scheme, ML, Haskell, and Smalltalk employ a "***reference model***" of variables

```
b:=2;
c:=b;
a:= b+c ;
```

➔ **Value Model** **Reference Model**

(Note) In a language that uses the reference model, every variable is an l-value.
Use dereferencing for an r-value.

Pascal: value model
Java: value model for built-in types and reference model for user-defined types (classes)
C# and Eiffel: programmer choose a model for each individual user-defined type
C#: a class is a reference and a struct is a value type

- Combination Assignment Operators

(ex) a = a+1 ;
    b.c[3].d = b.c[3].d * e ; redundant address calculation

To eliminate the clutter and compile- or run-time cost of redundant address calculations, and to avoid the issue of repeated side effects, many languages, beginning with Algol68, and including C and its descendants, provide so-called combination "assignment operators" to update a variable.

```
(ex) a += 1;
     b.c[3].d *= e;
     A[ index_fn(i) ] += 1;
```

```
(ex) prefix and postfix inc/dec
     A[ index_fn(i) ]++; or ++A[ index_fn(i) ];
     A[-- i] = b; //if i=3, b is assigned to A[2]
     *p++ = *q++ //if p and q point to the initial elements of a pair of arrays, copy the initial
                 //elements of the arrays (i.e.) *p = *q;
                 //                                p = p + 1;
                 //                                q = q + 1;
```

- Multiway Assignment
  : In several languages, including Clu, ML, Perl, Python, and Ruby, it is also possible to write
  a, b = c, d ;

```
(ex) Advantages
     a, b = b, a; (* swap  a and b *)
     a, b, c = foo (d, e, f ) (* allows functions to return tuples, as well as single values *)
```

6.1.3 Initialization

- Because they already provide a construct (the assignment statement) to set the value of a variable, imperative languages ***do not always provide*** a means of specifying an initial value for a variable ***in its declaration***.

- There are reasons why such initial values may be useful: (see p238)

- Most languages allow variables of built-in types to be initialized in their declarations.

  (ex) int a = 5;

- Initialization saves time only for variables that are statically allocated. Variables allocated in the stack or heap at run time must be initialized at run time.

- If a variable is not given an initial value explicitly in its declaration, the language ***may specify*** a default value.

(ex) In **C**, ***statically allocated variables*** for which the programmer does not provide an initial value are guaranteed to be initialized to ***zero***.

(ex) Java and C# provide a similar guarantee for the fields of all class-typed objects, not just those that are statically allocated.

(ex) Most scripting languages provide a default initial value for all variables, of all types, regardless of scope or lifetime.

- Dynamic Checks
: Instead of giving every uninitialized variable a default value, a language or implementation ***can choose to define the use of an uninitialized variable as a dynamic semantic error***, and can catch these errors at run time.

- Definite Assignment
: For local variables of methods, Java and C# define a notion of ***"definite assignment" that precludes the use of uninitialized variables***. This notion is based on the control flow of the program, and ***can be statically checked by the compiler***. Roughly speaking, every possible control path to an expression must assign a value to every variable in that expression.

(ex)
```
int i;
int j=3;
...
if ( j > o ) { i = 2; }
...                                        // no assignments to j in here
If ( j > o ) { system.out.println (i); }  // error: i was not initialized
```

- Constructors:

(1) Many object-oriented languages (Java and C# among them) allow the programmer to define types for which initialization of dynamically allocated variables occurs automatically, even when no initial value is specified in the declaration.

(2) **C++** distinguishes carefully between initialization and assignment. ***Initialization is interpreted as a call to a constructor function*** and ***assignment is interpreted as a call to the type's assignment operator***. Initialization with a nontrivial value is generally cheaper than default initialization followed by assignment, because it avoids deallocation of the space allocated for the default value.

(3) ***Neither Java nor C# distinguishes between initialization and assignment***. Java uses a reference model for all variables of user-defined object types, and provides for automatic storage reclamation, so assignment never copies values. C# allows the programmer to specify a value model when desired, but otherwise mirrors Java.

6.1.4 Ordering within Expressions

- Consider a - f(b) - c*d
  which one will be evaluated first?

  Similarly, f(a, g(b), h(c)) which argument will be evaluated first?

- The main reasons why the order can be important: ***side effects*** and ***code improvement***

- Because of the importance of code improvement, most language manuals say that the order of evaluation of operands and arguments is undefined. (***Java*** and ***C#*** are unusual in this regard: ***require left-to right evaluation***.) ***In the absence of an enforced order, the compiler can choose whatever order results in faster code***.

6.1.5 Short-Circuit Evaluation

- Boolean expressions provide a special and important opportunity for code improvement and increased readability.

- A compiler that performs "***short-circuit evaluation***" of Boolean expression will generate code that ***skips the second half*** of both of these computations when the overall value can be determined from the first half.

  (ex) if (very_unlikely_condition && very_expensive_function( )) ...
  ➔ can save significant amounts of time in certain situations

  (ex) short-circuiting changes the semantics of Boolean expressions

  In C, which does short-circuit the following code works:

```
p = my_list;
while ( p && p->key ! = val )
p = p->next;
```

  In Pascal, which doesn't short-circuit the following code doesn't work:

```
p := my_list;
while ( p <> nil) and (p^.key <> val) do
p := p^.next;                    run-time error when p is nil
```

- Short circulating is not necessarily as attractive for situations in which a Boolean subexpression can cause a side effect.

  (ex) if expression E1 and E2 both has side effects

  (ex) ...

```
    while not eof (doc_file) do begin
          w := get_word (doc_file);
          if ( tally(w) = 10 ) and misspelled(w) then
              writeln (w)                 supposed to increment global count of misspellings
    end;
    writeln (total_misspellings);
```

- To accommodate such situations while still allowing short-circuit evaluation, a few languages include both regular and short-circuit Boolean operators.

  (ex) In Ada, regular—***and*** and ***or***; the short-circuit—***and then*** and ***or else***
  In Visual Basic, regular—***AND*** and ***OR***; the short-circuit—***AndAlso*** then and ***OrElse***

6.2 Structured and Unstructured Flow

- Language designers debated the merits and evils of gotos.
  won

- Ada and C# allow gotos only in limited contexts. Modular (1,2, and 3), Clu, Eiffel, Java, and most of the scripting languages do not allow them at all. Fortran 90 and C++ allow them primarily for compatibility with their predecessor languages.

- The abandonment of goto was part of a larger "revolution" in software engineering known as "***structured programming***", which emphasizes top-down design, modularization of code, and so on.

6.3 Sequencing

- Like assignment, sequencing is central to imperative programming. It is the ***principal means of controlling the order***.

- In most imperative languages, list of statements can be enclosed with begin ... end or {...} delimiters and then used in any context in which a single statement is expected.
  ←usually called "compound statement".

- Even in imperative languages, there is debate as to the value of certain kinds of side effects. In Euclid and Turing, for example, functions are not permitted to have side effects.

- Unfortunately, there are ***some situations in which side effects in functions are highly desirable***.

(ex) A pseudorandom number generator

```
procedure srand (seed: integer )
  --Initialize internal tables.
  --The pseudorandom generator will return a different
  --sequence of values for each different value of seed.

function rand ( ) : integer
  --No arguments; returns a new "random" number.
```

➔ Obviously rand needs to have ***a side effect, so that it will return a different value each time it is called***. One could always recast it as ***a procedure with a reference parameter: procedure rand (var n: integer), but less appealing***

6.4 Selection

- ***Selection statements*** in most imperative languages employ ***some variant of the if…then…else*** notation introduced in Algol 60:

> **If** *condition* **then** *statement*
> **else if** *condition* **then** *statement*
> **else if** condition **then** *statement*
> …
> **else** *statement*

- Languages differ in the details of the syntax.

6.4.1 Short-Circuited Conditions

- While the condition in an if…then…else statement is a Boolean expression, there is usually ***no need for*** evaluation of that expression to result in ***a Boolean value in a register***.

(why?) The purpose of the Boolean expression in a selection statement is not to compute a value to be stored, but ***to cause control to branch to various locations***.
➔ Allow us to generate particularly efficient code called "***jump code***"

- ***Jump code is applicable*** not only to ***selection statements*** such as if… then…else, but to ***logically controlled loops*** as well.

(ex) Suppose that we are generating code for the following source:

```
If ((A > B) and (C > D)) or (E ≠ F) then
    then-clause
else
    else-clause
```

⇩

In Pascal, which does not use short-circuit evaluation

```
        r1 := A
        r2 := B
        r1 := r1 > r2
        r2 := C
        r3 := D
        r2 : r2 > r3
        r1 := r1 & r2
        r2 := E
        r3 := F
        r2 := r2 ≠ r3
        r1 := r1 | r2
        if r1 = 0 goto L2
    L1 : then-clause
        goto L3
    L2 : else-clause
    L3 :
```

In jump code,

```
        r1 := A
        r2 := B
        if r1 <= r2 goto L4
        r1 := C
        r2 := D
        if r1 > r2 goto L1
    L4 : r1 := E
        r2 := F
        if r1 = r2 goto L2
    L1 : then-clause
        goto L3
    L2 : else-clause
    L3 :
```

6.4.2 Case/Switch Statements

- The case statements of Algol W and its descendants provide alternative syntax for a special case of nested if...then...else.

(ex) In Ada,

```
i := ...
if i = 1 then
  clause_A
elsif i =2 or i = 7 then
  clause_B
elsif i in 3..5 then
  clause_C
elsif i=10 then
  clasue_D
else
  clause_E
end if;
```

can be written as ⟹

```
case ...
is
   when 1        => clause_A
   when 2 | 7    => clause_B
   when 3..5     => clause_C
   when 10       => clause_D
   when others => clause_E
end case;
```

- The ***constants in the label lists*** must be ***disjoint***, and ***must be a type compatible*** with the tested expression.

- Most languages allow this type to be ***anything whose values are discrete***: integers, characters, enumerations, and subranges of the same. C# allows strings as well.

- The principal motivation of the ***case statement*** is ***to facilitate the generation of efficient target code***. Rather than test its controlling expression sequentially against a series of possible values, the ***case statement is meant to compute an address to which it jumps in a single instruction***.

- As with if...then...else statements, the syntactic details of case statements vary from language to language, for example, different punctuation to delimit labels and arms.

- More significantly, languages differ in
  (1) whether they permit label ranges,
  (2) whether they permit (or require) a default (else) clause,
  (3) how they handle a value that fails to match any label at run time

(ex) C's syntax for case (switch) statements (retained by C++ and Java) is unusual in several respects:

```
switch ( ... /* tested expression */ ) {
        case 1: clause_A
                break;
        case 2:
        case 7: clause_B
                break;
        case 3:
        case 4:
        case 5: clause_C
                break;
        case 10: clause_D
                 break;
        default: clause_E
                 break;    /*could be omitted */
}
```

- Most of the time, the need to ***insert a break at the end of each arm*** and the compiler's willingness to accept arms without breaks could produce ***unexpected and difficult-to-diagnosis bugs***.

- C# retains the familiar C syntax, including multiple consecutive labels, but requires every nonempty arm to end with a break, goto, continue, or return.

6.5 Iteration

- ***Iteration*** and ***recursion*** are two mechanism that ***allow a computer to perform similar operations repeatedly***.

- In a very real sense, it is ***iteration and recursion*** that ***make computers useful***.

- In most languages, ***iteration takes the form of "loops"***. The iterations of a loop are generally executed for their side effects: their ***modifications of variables***.

- An ***"enumeration-controlled" loop*** is executed once for every value in a given finite set. A ***"logically controlled" loop*** is executed until some Boolean condition changes value.

6.5.1 Enumeration-Controlled Loops

- ***Originated with the do loop for Fortran I***. Similar mechanisms have been ***adopted in some form by almost every subsequent languages***, but syntax and semantics ***vary widely***.

(ex) Fortran 90,

```
do i = 1, 10, 2
  ...
enddo
```

(where, index=i, initial value=1, bound=10, step size=2)

Modula-2,

```
For i := first To last BY step DO
      ...
END
```

- Following the lead of Clu, many modern languages allow ***enumeration-controlled loops to iterate over much more general finite sets—the nodes of a tree, for example, or the elements of collection***.

- If the loop goes "the other direction" (i.e. first > last and step < 0) then we will use the ***inverse test to end the loop***. Many languages restrict the generality of their arithmetic sequences. Commonly, ***step is required to be a compile-time constant***.

(ex) (1) Ada limits the choices +1/-1.

(2) Several languages, including both Ada and Pascal, require special syntax for loops that iterate "backward".

(Ada: for i in reverse 10..1
Pascal: for i := 10 down to 1)

(3) Fortran compilers, is to ***precompute the number of iterations, place this "iteration count" in a register, decrement the register at the end of each of iteration, and branch back to the top of the loop if the count is not zero yet***. The use of the iteration count avoids the need to test the sign of step within the loop.

- ***Semantic Complications***

The choice between requiring and (merely) enabling enumeration manifests itself in several specific questions:

(1) Can control enter or leave the loop in any way other than through the enumeration mechanism?
(2) What happens if the loop body modifies variables that were used to compute the end-of-loop bound?
(3) What happen if the loop body modifies the index variable itself?
(4) Can the program read the index variable after the loop has completed, and if so, what will its value be?

(1) and (2) are easy to resolve: (1) Most languages allow a ***break/exit statement to leave a for loop early*** (2)Most languages (but not C) specify that the ***bound is computed only once***, before the first iteration, and ***kept in temporary location***. ***Subsequent changes to variables used to compute the bound have no effect on how many times the loop iterates***.

Question (3) and (4) are more difficult:

- ***Many languages prohibit change to the loop index within the body of the loop***. ***Fortran*** makes the prohibition a matter of programmer discipline (i.e. ***not required to catch***). ***Pascal*** allow the ***compiler to catch*** all possible updates.

- If control escapes the loop ***with a break/exit***, the natural value for the index would seem to be the ***one that was current at the time of the escape***. For ***"normal" termination***, the natural value would seem to be the first one that ***exceeds the loop bound.***

may be semantically invalid

⇨ Attractive solution pioneered by Algol W and Algol 68, and subsequently adopted by Ada, Modula 3, and many other languages
: The ***header of the loop*** is considered to ***contain a "declaration" of the index***. Its ***type is inferred from the bounds of the loop***, and its ***scope is the loop's body***. Because the index is not visible outside the loop, its value is not an issue.

6.5.5 Logically Controlled Loops

- In comparison to enumeration-controlled loops, logically controlled loops have many fewer semantic subtleties.

- Where within the body of the loop the termination condition is tested?

(1) ***Pre-tested loops***

- By far the most common approach is to test the condition before each iteration.
  (ex) Algol-W: **while** condition **do** statement

- To allow the body of the loop to be ***a statement list***, most modern languages use an explicit concluding keyword (e.g. end), or bracket the body with delimiters (e.g. { ... })

(2) ***Post-tested loops***

- Pascal introduced special syntax for this case, which was retained in Modula but dropped in Ada.

(ex)

repeat
    readln (line)
until line[1]='$';
	instead of	readln (line);
		while line[1] < > '$' do
		    readln(line);

- Note that ***the body of a post-test loop is always executed at least once***.

- C provides a post-test loop whose condition works ***"the other direction"***

(ex)
```
do {
        line = read_line (stdin) ;
    } while ( line [0] != '$' ) ;
```

(3) Midtest loops

- In many languages "midtest" can be accomplished with a special statement nested inside a conditional: ***exit*** in Ada, ***break*** in C, ***last*** n Perl.

(ex) assumed to always be true

```
for ( ; ; ) {
        line = read_line (stdin);
        if ( all_blanks (line) ) break;
        consume_line (line) ;
}
```