6 Control Structures

Control statements are used to modify the order of execution. There are two classes of well-structured control statements: choice statements (if and case) which select one alternative from two or more possible execution sequences, and loop statements (for and while) which repeatedly execute a series of statements.

6.1 switch-/case-statements

A choice statement is used to select one of several possible paths for the computation to pursue (Figure 6.1).[1] The generalized choice statement is called a switch-statement in C and a case-

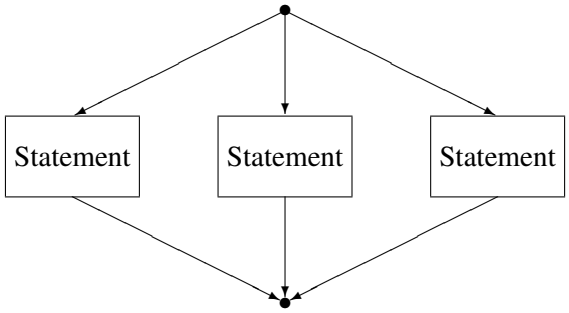

Figure 6.1: Choice statement

statement in other languages. A switch-statement is composed of an expression and a statement for each possible value of the expression:

C

```
switch (expression) {
    case value_1:
        statement_1;
        break;
    case value_2:
        statement_2;
        break;
    ...
```

[1] Throughout this chapter we will use the term "statement" to describe a path of the computation. This should be interpreted to include a *compound statement* which is a sequence of one or more statements.

```
}
```

The expression is evaluated and the result of the expression is used to select a statement to execute; in terms of Figure 6.1, the selected statement represents a path. This requires that there be *exactly* one case alternative for each possible value of the expression. If the expression is of integer type, this is impossible since it is not practical to write a statement for each 32-bit integer value. In Pascal the case-statement is usable only for types which have a small number of values, while C and Ada provide a default alternative, so that the case-statement can be used even for types like Character that have hundreds of values:

C

```
default:
    default_statement;
    break;
```

If the value of the expression is not explicitly listed, the default statement is executed. C actually assumes an empty default statement if the default alternative is missing. This option should not be used because the reader of the program has no way of knowing if an empty default statement is intended, or if the programmer has simply forgotten to provide the necessary statements.

In many cases, the statement for two or more alternatives is identical. C has no direct way of expressing this (see below); in Ada there is an extensive set of syntactical constructions for grouping alternatives:

Ada

```
C: Character;
case C is
    when 'A' .. 'Z'          => statement_1;
    when '0' .. '9'          => statement_2;
    when '+' | '-' | '*' | '/' => statement_3;
    when others              => statement_4;
end case;
```

In Ada the alternatives are introduced by the reserved keyword when, and the default alternative is called others. A case alternative may contain a range of values value_1..value_2, or it may contain a set of values separated by "|".

The break-statement in C

In C you must explicitly terminate each case alternative with the break-statement, otherwise the computation will "fall-through" to the next case alternative. A valid use of fall-through is to mimic the multiple-alternative construct in Ada:

C

```
char c;
switch (c) {
    case 'A':   case 'B':   ...   case 'Z':
```

```
        statement_1;
        break;
    case '0':  ...  case '9':
        statement_2;
        break;
    case '+':  case '-':  case '*':  case '/':
        statement_3;
        break;
    default:
        statement_4;
        break;
}
```

Since each value must be explicitly written, the switch-statement in C is rather less convenient than the case-statement in Ada.

Fall-through should not be used in ordinary programming:

C

```
switch (e) {
    case value_1:
        statement_1;        /* After statement_1, */
    case value_2:
        statement_2;        /*   fall through to statement_2. */
        break;
}
```

Referring to Figure 6.1, the switch-statement is intended to be used to choose one of a set of possible paths. Fall-through is confusing because the end of a path loops back to the beginning of the tree of choices. Furthermore, no semantic importance should be attributed to the sequence in which the choices are written (though the order may be significant in terms of efficiency). Maintainers of the program should be free to rearrange existing choices, or to insert new choices, without fear of introducing a bug. This program is also difficult to test and debug: if a bug is traced to statement_2, it is difficult to know if the statement was reached by direct selection or by fall-through. Rather than using fall-through, common code should be placed in a procedure:

C

```
switch (e) {
    case value_1:
        statement_1;
        common_code( );
        break;
    case value_2:
        common_code( );
        break;
}
```

Implementation

The simplest way of compiling a case statement is as a sequence of tests:

```
        compute   R1,expr          Compute expression
        jump_eq   R1,#value_1,L1
        jump_eq   R1,#value_2,L2
        ...                        Other values
        default_statement          Instructions for default
        jump      End_Case

L1:     statement_1                Instructions for statement_1
        jump      End_Case
L2:     statement_2                Instructions for statement_2
        jump      End_Case
        ...                        Instructions for other statements
End_Case:
```

In terms of efficiency, it is apparent that the closer an alternative is to the top of the statement, the more efficient it is to choose it; you can reorder the alternatives to take advantage of this fact (provided that you don't use fall-through!).

Certain case statements can be optimized to use jump tables. If the set of values of the expression form a short contiguous sequence, then the following code can be used (where we assume that the expression can take values from 0 to 3):

```
        compute   R1,expr
        mult      R1,#len_of_addr   expression*length_of_addr
        add       R1,&table         Start of table
        jump      (R1)              Jump to address in R1

table:                              Jump table
        addr(L1)
        addr(L2)
        addr(L3)
        addr(L4)

L1:     statement_1
        jump      End_Case
L2:     statement_2
        jump      End_Case
L3:     statement_3
        jump      End_Case
L4:     statement_4
End_Case:
```

The value of the expression is used as an index into a table of addresses of statements, and the jump instruction jumps to the address contained in the register. In terms of efficiency, the overhead of a jump-table implementation is fixed and small for *all* choices.

It is imperative that the value of the expression be within the expected range (here 0 to 3), otherwise an invalid address will be computed and the computation will jump to a location that might not even be a valid instruction! In Ada, the expression can often be checked at compilation time:

Ada

```
type Status is (Off, Warm_Up, On, Automatic);
S: Status;
case S is ...                -- There are exactly four values
```

In other cases, a run-time check will be needed to ensure that the value is within range. Jump tables are even compatible with a default alternative, provided that the explicit choices are contiguous. The compiler simply inserts the run-time check before attempting to use the jump table; if the check fails, the computation continues with the default alternative.

The choice of implementation is usually left to the compiler and there is no way of knowing which one is used without examining the machine code. The documentation of an optimizing compiler may inform its users under what circumstances it will compile into a jump table. Even if you take this knowledge into account when programming, your program is still portable, because the case-statement is portable; however, a different compiler may implement it in a different manner, so the improvement in efficiency is not portable.

6.2 if-statements

An if-statement is a special case of a case- or switch-statement where the expression is of Boolean type. Since Boolean types have only two possible values, an if-statement chooses between two possible paths. if-statements are probably the most frequently used control structures; the reason is that relational operators are extensively used and they return values of Boolean type:

C

```
if (x > y)
    statement-1;
else
    statement-2;
```

As we discussed in Section 4.4, C does not have a Boolean type. Instead, integer values are used with the convention that zero is false and non-zero is true.

A common mistake is to use an if-statement to create a Boolean value:

Ada

```
if X > Y then
    Result = True;
else
    Result = False;
end if;
```

instead of using a simple assignment statement:

Ada

```
Result := X > Y;
```

Remember that values and variables of Boolean type are first-class objects: in C they are just integers, and in Ada they are a distinct type but no different from any other enumeration type. The fact that Boolean types have a special status in if-statements does not otherwise restrict them.

Nested if-statements

The alternatives of an if-statement are themselves statements; in particular, they can also be if-statements:

C

```
if (x1 > y1)
    if (x2 > y2)
        statement-1;
    else
        statement-2;
else
    if (x3 > y3)
        statement-3;
    else
        statement-4;
```

It is best not to nest control structures (especially if-statements) too deeply, three or four levels at most. The reason is that it becomes difficult to follow the logic of the various paths. Furthermore, the indentation of the source code is only a guide: if you omit an else, the statement may still be syntactically valid even though the processing is no longer correct.

Another potential problem is that of the dangling else:

C

```
if (x1 > y1)
    if (x2 > y2)
        statement-1;
    else
        statement-2;
```

As implied by the indentation, the language definition associates the else with the inner-most if-statement. If you want to associate it with the outer if-statement, brackets must be used:

C

```
if (x1 > y1) {
    if (x2 > y2)
        statement-1;
}
```

```
else
    statement-2;
```

Nested if-statements can define a full binary tree of choices (Figure 6.2(a)), or any arbitrary subtree. In many cases, however, what is needed is to choose one of a sequence of outcomes (Fig-

(a) Fully nested (b) Sequential

Figure 6.2: if-statements

ure 6.2(b)). If the selection is done on the basis of an expression, a switch-statement can be used. Otherwise, if the selection is done on the basis of a sequence of relational expressions, a sequence of nested if-statements is needed. In this case, it is conventional not to indent the statements:

C

```
if (x > y) {
    ...
} else if (x > z) {
    ...
} else if (y ¡ z) {
    ...
} else {
    ...
}
```

Explicit end if

The syntax of the if-statement in C (and Pascal) requires that each alternative be a single statement. If the alternative consists of a sequence of statements, they must be grouped into a single *compound* statement using braces ({, }) in C and (begin, end) in Pascal. The problem with this syntax is that if the terminating bracket is omitted, the compilation will continue without noticing the mistake. At best, the missing bracket will be noticed at the end of the compilation; at worst, a missing *opening* bracket will balance out the bracket count and an obscure run-time bug will be introduced.

This problem can be alleviated by explicitly terminating the if-statement. If the terminating bracket is omitted, it will be flagged as soon as another construct (a loop or procedure) is terminated with a different bracket. The if-statement syntax in Ada is:

Ada

```
if expression then
    statement-list-1;
else
    statement-list-2;
end if;
```

The problem with this construct is that in the case of a sequence of conditions (Figure 6.2(b)), there will be a confusing sequence of end if's at the end. To avoid this, a special construct is used; elsif introduces another condition and statement, but not another if-statement so no addition termination is required:

Ada

```
if x > y then
    ...
elsif x > z then
    ...
elsif y > z then
    ...
else
    ...
end if;
```

Implementation

The implementation of an if-statement is straightforward:

```
        compute   R1,expression
        jump_eq   R1,L1          False is represented as 0
        statement-1
        jump      L2
L1:     statement-2
L2:
```

Note that the False alternative is slightly more efficient than the True alternative since the latter executes an extra jump instruction.

On the surface it would seem that a condition of the form:

C

```
if (!expression) ...
```

would require an extra instruction to negate the value. However, compilers are smart enough just to change the initial jump_false instruction to jump_true.

Short circuit and full evaluation

Suppose that the expression in an if-statement is not just a simple relational expression, but a compound expression:

```
if (x > y) and (y > z) and (z ¡ 57) then...
```
Ada

There are two possible implementations of this statement. The first, called *full evaluation*, evaluates each of the components, then takes the Boolean and of the components and jumps according to the result. The second implementation, called *short-circuit evaluation*, evaluates the components one-by-one: if any component evaluates to False, a jump is executed to the False alternative since the entire expression must obviously be False. A similar situation occurs if the expression is a compound or-expression: if any component is True, the entire expression is obviously True.

The choice between the two implementations can usually be left to the compiler. Short-circuit evaluation will in general require fewer instructions to be executed. However, these instructions include many jumps and on a computer with a large instruction cache (see Section 1.7), it may be more efficient to compute all the components and only jump after the full evaluation.

Pascal specifies full evaluation because it was originally developed for a computer with a large cache. Other languages have two sets of operators: one for (full) evaluation of Boolean values and the other for short-circuit evaluation. For example, in Ada, and is used for fully evaluated Boolean operations on Boolean and modular types, while and then specifies short-circuit evaluation:

```
if (x > y) and then (y > z) and then (z < 57) then...
```
Ada

Similarly, or else is the short-circuit equivalent of or.

C contains three logical operators: "!" (not), "&&" (and), and "||" (or). Since C lacks a true Boolean type, these operators take integer operands and are defined in accordance with the interpretation described in Section 4.4. For example, a && b is equal to one if both operands are non-zero. Both "&&" and "||" use short-circuit evaluation. Also, be sure not to confuse these operators with the bitwise operators (Section 5.9).

In terms of programming style, Ada programmers should choose one style (either full evaluation or short circuit) for an entire program, using the other style only if necessary; C programmers always use the short-circuit operators.

Short-circuit evaluation is essential when the ability to evaluate one relation in a compound expression depends on a previous relation:

```
if (a /= 0) and then (b/a > 25) then ...
```
Ada

Such situations are also common when pointers (Chapter 8) are used:

```
if (ptr /= null) and then (ptr.value = key) then ...
```
Ada

6.3 Loop statements

Loop statements are the most difficult statements to program: they are prone to bugs especially at the boundaries of the loop, that is, the first and last executions of the loop body. Furthermore, an inefficient program is almost certainly spending most of its time in loops, so it is critical that their implementation be fully understood.

The structure of a loop statement is shown in Figure 6.3. A loop statement has an *entry* point,[2]

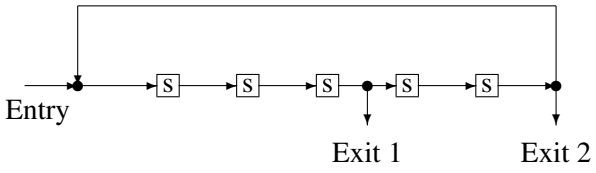

Figure 6.3: Loop statements

a sequence of statements that compose the loop, and one or more *exit* points. Since we (usually) want our loop statements to terminate, an exit point will have a *condition* associated with it, which decides if the exit should be taken, or if the execution of the loop should be continued. Loop statements are distinguished by the number, type and placement of the exit conditions. We will begin by discussing loops with arbitrary exit conditions, called while-statements, and in the next section discuss a specialization called for-statements.

The most common type of loop has its single exit point at the beginning of the loop, that is at its entry point. This is called a while-statement:

C

```
while (s[i].data != key)
    i++;
```

The while-statement is straightforward and reliable. Because the condition is checked at the beginning of the loop, we know that the loop body will be executed in its entirety a number of times that depends on the condition. If the exit condition is initially False, the loop body will not be executed and this simplifies the programming of boundary conditions:

C

```
while (count > 0)
    process(s[count].data);
```

If there is no data in the array, the loop will exit immediately.

In many cases, however, the exit from a loop is naturally written at the end of the loop. This is common when a variable must be initialized before any processing is done. Pascal has the repeat-statement:

[2]We will not even consider the possibility of jumping into a loop!

Pascal

```
repeat
    read(v);
    put_in_table(v)
until v = end_value;
```

The Pascal repeat terminates when the exit condition is True. Do not confuse it with the C do-statement, which terminates when the exit condition is False:

C

```
do {
    v = get( );
    put_in_table(v);
} while (v != end_value);
```

A pure approach to structured programming requires that all loops exit only at the beginning or the end of the loop. This makes a program much easier to analyze and verify, but in practice, exits from within a loop are commonly needed, especially when an error is detected:

Pascal

```
while not found do
    begin
    (* Long computation *)
    (* Error detected, terminate *)
    (* Long computation *)
    end
```

Pascal, which has no way of exiting from within a loop, uses the following unsatisfactory solution: set the exit condition and use an if-statement to skip the remainder of the loop body:

Pascal

```
while not found do
    begin
    (* Long computation *)
    if error_detected then found := True
    else
        begin
        (* Long computation *)
        end
    end
```

In C, the break-statement can be used:

C

```
while (!found) {
    /* Long computation */
    if (error_detected( )) break;
    /* Long computation */
}
```

Ada has the usual while-statement, as well as an exit-statement which is used to exit from an arbitrary location within the loop; the common occurrence of an if-statement together with an exit-statement can be conveniently expressed using a when-clause:

Ada

```
while not Found loop
   -- Long computation
   exit when error_detected;
   -- Long computation
end loop;
```

An operating system or real-time system is not intended to terminate, so there must be a way of indicating non-terminating loops. In Ada, this is directly expressed by a loop-statement with no exit condition:

Ada

```
loop
   ...
end loop;
```

Other languages require you to write an ordinary loop with an artificial exit condition that ensures that the loop does not terminate:

C

```
while (1==1) {
   ...
}
```

Implementation

The implementation of a while-loop:

C

```
while (expression)
   statement;
```

is as follows:

```
L1:  compute    R1,expr
     jump_zero  R1,L2        Skip statement if false
     statement               Loop body
     jump       L1           Jump to check for termination
L2:
```

Note that there are *two* jump instructions in the implementation of a while-statement! Surprisingly, if the loop exit is at the end of the loop, only one jump instruction is needed:

C

```
do {
   statement;
} while (expression);
```

compiles to:

```
L1:  statement
     compute   expr
     jump_nz   L1          Not zero is true
```

Even though the while-loop is a very clear construct, the efficiency of the loop can be improved by changing it to a do-loop. Exiting a loop in the middle requires two jumps just like a while-statement.

6.4 for-statements

Very often, we know the number of iterations that a loop requires: either it is a constant known when the program is written, or it is computed before the beginning of the loop. Counting loops can be programmed as follows:

C

```
int i;                  /* Loop index */
int low, high;          /* Loop limits */
i = low;                /* Initialize index*/
while (i ¡= high) {     /* Evaluate exit expression */
   statement;
   i++;                 /* Increment index */
};
```

Since this paradigm is so common, all (imperative) programming languages supply a for-statement to simplify programming. In C the syntax is:

C

```
int i;                  /* Loop index */
int low, high;          /* Loop limits */

for (i = low; i ¡= high; i++) {
   statement;
}
```

The syntax in Ada is similar, except that the declaration and incrementation of the loop variable is implicit:

Ada

```
Low, High: Integer;

for I in Low .. High loop
   statement;
end loop;
```

Later in this section we will discuss the reasons for these differences.

for-statements are notorious for the bugs that they can introduce at the boundary values. The loop is executed for each of the values from low to high; thus the total number of iterations is $high - low + 1$. However, if the value of low is strictly greater than the value of high, the loop is executed zero times. If you wish to execute a loop exactly N times, the for-statement will be:

Ada

```
for I in 1..N loop ...
```

and the number of iterations is $N - 1 + 1 = N$. In C, because arrays are required to start from index zero, ordinary counting loops are usually written:

C

```
for (i = 0; i ¡ n; i++) ...
```

Since $i < n$ is the same as $i <= (n-1)$, the loop is executed $(n-1) - 0 + 1 = n$ times as required.

Generalized for-statements

Even though all imperative languages contain a for-statement, they differ greatly in the additional features that are provided. Two extremes are Ada and C.

Ada takes the point of view that a for-loop should be used *only* for loops with a fixed number of iterations, and that this number can be computed before starting the loop. The rationale for this point of view is: (1) most loops are in fact this simple, (2) other constructions can easily be explicitly programmed, and (3) for-loops are difficult enough to test and verify as it is. Ada even lacks the classic generalization: incrementing the loop variable by values other than 1 (or –1). In Algol, iteration over a sequence of odd numbers can be written:

Algol

```
for I := 1 to N step 2 do ...
```

while in Ada we have to explicitly program:

Ada

```
for I in 1 .. (N+1)/2 loop
   I1 = 2*I-1;
   ...
end loop;
```

In C, all three elements of the for-statement can be arbitrary expressions:

```
for ( i = j*k;   (i ¡ n) && (j+k > m);   i += 2*j ) ...
```
C

The definition of C specifies that:

```
for (expression_1; expression_2; expression_3) statement;
```
C

is equivalent to:

```
expression_1;
while (expression_2) {
    statement;
    expression_3;
}
```
C

In Ada, expressions are also permitted for the loop limits, but these are evaluated only once at the loop entry. That is:

```
for I in expression_1 .. expression_2 loop
    statement;
end loop
```
Ada

is equivalent to:

```
I = expression_1;
Temp = expression_2;
while (I ¡ Temp) loop
    statement;
    I := I + 1;
end loop;
```
Ada

If the body of the loop modifies the value of a variable used in computing expression_2, the upper limit of the Ada loop will not be modified. Compare this with the definition of the C for-loop above which re-evaluates the value of expression_2 on each iteration.

The generalizations in C are not just syntactic sugaring because of the possibility of side-effects, that is, statements within the loop that modify expression_2 and expression_3. Side-effects should be avoided for the following reasons:

- Side-effects make the loop difficult to fully verify and test.
- Side-effects adversely affect the readability and maintainability of the program.
- Side-effects make the loop much less efficient because expression_2 and expression_3 must be re-evaluated on each iteration. If side-effects are not used, an optimizing compiler will be able to move these evaluations outside the loop.

Implementation

for-statements are common sources of inefficiencies in programs because slight differences in language definition, or small changes in the use of the statement, can have significant consequences. In many cases, the optimizer can solve the problems, but it is better to be aware of and avoid problems, rather than to trust the optimizer. In this section, we will describe the implementation in greater detail at the register level.

In Ada, the loop:

Ada

```
for I in expression_1 .. expression_2 loop
   statement;
end loop;
```

compiles to:

```
      compute   R1,expr_1
      store     R1,I          Lower bound to index
      compute   R2,expr_2
      store     R2,High       Upper bound to index
L1:   load      R1,I          Load index
      load      R2,High       Load upper bound
      jump_gt   R1,R2,L2      Terminate loop if greater
      statement               Loop body
      load      R1,I          Increment index
      incr      R1
      store     R1,I
      jump      L1
L2:
```

An obvious optimization is to dedicate a register to the index variable I and, if possible, another register to High:

```
      compute   R1,expr_1     Lower bound to register
      compute   R2,expr_2     Upper bound to register
L1:   jump_gt   R1,R2,L2      Terminate loop if greater
      statement
      incr      R1            Increment index register
      jump      L1
L2:
```

Now consider a simple loop in C:

C

```
for (i = expression_1; expression_2; i++)
   statement;
```

This compiles to:

```
        compute   R1,expr_1
        store     R1,i          Lower bound to index
L1:     compute   R2,expr_2     Upper bound within loop !
        jump_gt   R1,R2,L2      Terminate loop if greater
        statement               Loop body
        load      R1,i          Increment index
        incr      R1
        store     R1,i
        jump      L1
L2:
```

Note that expression_2, which may be very complicated, is now computed inside the loop. Also, expression_2 necessarily uses the value of the index variable i which is changed each iteration. Thus the optimizer must be able to identify the non-changing part of the evaluation of expression_2 in order to extract it from the loop.

Can the index variable be stored in a register for greater efficiency? The answer is "maybe" and depends on two properties of the loop. In Ada, the index variable is considered to be constant and cannot be modified by the programmer. In C, the index variable is a normal variable; it can be kept in a register only if there is no possibility that its current value will be needed except within the loop. Never use a global variable as an index variable because another procedure may read or modify its value:[3]

C

```
int i;

void p2(void) {
   i = i + 5;
}

void p1(void) {
   for (i=0; i¡100; i++)          /* Global index variable */
      p2( );                      /* Side effect on index */
}
```

The second property that affects the ability to optimize the loop is the potential use of the index variable outside the loop. In Ada, the index variable is implicitly declared by the for-statement and is *not* accessible outside the loop. Thus no matter how the loop is exited, we do not have to save the value in the register. Consider the following loop which searches for a key value in an array a:

C

```
int a[100];
int i, key;
```

[3] Also, in a multi-processing environment, another process may access the value.

```
key = get_key( );
for (i = 0; i ¡ 100; i++)
    if (a[i] == key) break;
process(i);
```

The variable i must contain the correct value regardless of which exit is taken. This can cause difficulty when trying to optimize the code. Note that in Ada, explicit coding is required to achieve the same effect because the index variable does not exist outside the scope of the loop:

Ada

```
Found: Integer := False;

for I in 1..100 loop
    if A(I) = Key then
        Found = I;
        exit;
    end if;
end loop;
```

The definition of the scope of loop indices in C++ has changed over the years, but the final definition is the same as in Ada: the index does not exist outside the scope of the loop:

C++

```
for (int i=0; i¡100; i++) {
        // Index variable is local to loop
}
```

In fact, any statement controlled by a condition (including if- and switch-statements) can have several declarations appear in the condition; their scope is limited to the controlled statements. This feature can contribute to the readability and reliability of a program by preventing unintended use of a temporary name.

6.5 Sentinels

The following section is not about programming languages as such; rather it is intended to show that a program can be improved by using better algorithms and programming techniques instead of fiddling with language details. The section is included because the topic of loop exit in a linear search is the subject of intense debate, and yet there exists a different algorithm that is simultaneously clear, reliable and efficient.

In the last example of the previous section (searching an array), there are three jump instructions in every execution of the loop: the conditional jump of the for-statement, the conditional jump of the if-statement and the jump from the end of the loop back to the beginning. The problem with this search is that we are checking two conditions at once: have we found the key, and have we reached the end of the array? By using a *sentinel*, we can reduce the two conditions to one. The

idea is to extend the array by one extra place at the beginning of the array, and to store the key we are searching for in that place (Figure 6.4). Since we will necessarily find the key, either as

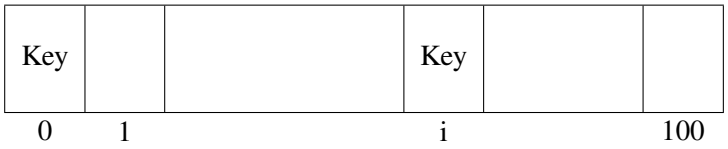

Figure 6.4: Sentinels

an occurrence within the array or as the artificial occurrence, only one condition need be checked within the loop:

Ada

```
type A_Type is array(0..100) of Integer;
          -- Extra place at zero for sentinel

function Find_Key(A: A_Type; Key: Integer)
        return Integer is
    I: Integer := 100;          -- Search from end
begin
    A(0) := Key;                -- Store sentinel
    while A(I) /= Key loop
        I := I - 1;
    end loop;
    return I;
end Find_Key;
```

Upon return from the function, if I is zero then the Key does not exist in the array; otherwise, I contains the index of the occurrence. Not only is this code more efficient, but the loop is extremely simple and can be easily verified.

6.6 * Invariants

The formal definition of the semantics of loop statements is based on the concept of an *invariant*: a formula which remains true after every execution of the loop body. Consider a simplistic program for computing integer division of a by b to obtain the result y:

C

```
y = 0;
x = a;
while (x >= b) {            /* As long as b "goes into" x, */
    x -= b;                 /*   subtracting b means that */
    y++;                    /*   result must be incremented */
}
```

and consider the formula:

$$a = yb + x$$

where an italic letter denotes the value of the corresponding program variable. After the initialization statements, this is certainly true since $y = 0$ and $x = a$. Furthermore, at the end of the program the formula *defines* what it means for y to be the result of integer division a/b, provided that the remainder x is less than the divisor b.

What is not so obvious is that the formula remains true after every execution of the loop body. In this trivial program, that fact is easy to see by simple arithmetic, given the changes to the values of x and y in the loop body:

$$(y+1)b + (x-b) = yb + b + x - b = yb + x = a$$

Thus the body of a loop statement transforms the state of a program from a state that satisfies the invariant to a different state that still satisfies the invariant.

Now note that for the loop to terminate, the Boolean condition in the while-statement must be False, that is the computation must be in a state such that $\neg(x \geq b)$ which is equivalent to $x < b$. Combining this formula with the invariant, we have shown that the program actually computes integer division.

More precisely, we have shown that *if* the program terminates, *then* the result is correct. This is called *partial correctness*. To prove *total correctness*, we must also show that the loop terminates.

This is done as follows. Since b is constant (and assumed positive!) during the execution of the program, what we have to show is that repeatedly decrementing x by b must eventually lead to a state in which $0 \leq x < b$. But (1) since x is repeatedly decremented, its value cannot stay indefinitely above that of b; (2) by the condition for terminating the loop and the computation in the loop body, x is never negative. These two facts imply that the loop must terminate.

Loop invariants in Eiffel

Eiffel supports the specification of assertions in general (see Section 11.5) and loop invariants in particular within the language:

Eiffel

```
from
    y = 0; x = a;
invariant
    a = yb + x
variant
    x
until
    x ¡ b
loop
    x := x − b;
    y := y + 1;
end
```

The from-clause establishes the initial condition, the until-clause gives the condition to terminate the loop, and the statements between loop and end form the loop body. The invariant-clause states the loop invariant and the variant-clause states the expression that will decrease (but stay non-negative) with each iteration of the loop. The correctness of the invariant is checked after each execution of the loop body.

6.7 goto-statements

In the original definition of Fortran there was only one structured control statement: the do-statement which is similar to the for-statement. All additional control used conditional or unconditional jumps to labels, called goto-statements:

Fortran

```
      if (a .eq. b) goto 12
      ...
      goto 5
4     ...
      ...
12    ...
      ...
5     ...
   if (x .gt. y) goto 4
```

In 1968, E. W. Dijkstra wrote a famous letter entitled "goto Considered Harmful" which launched a debate on structured programming. The main thrust of the "anti-goto" argument is that arbitrary jumps are not structured and create *spaghetti code*, that is, code whose possible threads of execution are so intertwined that it becomes impossible to understand or test the code. The "pro-goto" argument is that real programs often require control structures that are more general than those offered by structured statements, and that forcing programmers to use them results in artificial and complex code.

In retrospect, this debate was far too emotional and drawn-out because the basic principles are quite simple and are no longer seriously disputed. Furthermore, modern dialects of Fortran include more advanced control statements so that the goto-statement is no longer dominant.

It can be mathematically proven that if- and while-statements are sufficient to express any needed control structure. These statements are also easy to understand and use. Various syntactic extensions such as for-statements are well understood and if used properly pose no difficulty in understanding or maintaining a program. So why do programming languages (including Ada which was designed with reliability as the highest priority) retain the goto-statement?

The reason is that there are several well-defined situations where use of a goto-statement may be preferable. Firstly, many loops do not naturally terminate at their entry point as required by the while-statement. Attempting to force all loops into while-statements can lead to obscure code. With modern languages, the flexibility of exit- and break-statements means that goto-statements are usually unnecessary for this purpose. Nevertheless, the goto-statement still exists and can

occasionally be useful. Note that both C and Ada limit the goto-statement by requiring that the label be in the same procedure.

A second situation that can be easily programmed using a goto-statement is an escape from a deeply nested computation. Suppose that deep within a series of procedure calls an error is detected that invalidates the entire computation. The natural way to program this requirement is to display an error message and terminate or reset the entire computation. However, this requires returning from many procedures, all of which have to know that an error has occurred. It is easier and more understandable just to goto a statement in the main program.

The C language has no means of dealing with this situation (not even with goto-statements which are limited to a single procedure), so facilities of the operating system must be used to handle serious errors. Ada, C++ and Eiffel have a language construct called *exceptions* (see Chapter 11) which directly solves this problem. Thus most of the uses of goto-statements have been superseded by improved language design.

Assigned goto-statements

Fortran includes a construct called an *assigned* goto-statement. A label variable can be defined and a label value assigned to the variable. When a jump is made to the label variable, the actual target of the jump is the current value of the label variable:

Fortran

```
      assign 5 to Label
      ...
      if (x .gt. y) assign 6 to Label
5     ...
6     ...
      goto Label
```

The problem, of course, is that the assignment of the label value could have been made millions of instructions before the goto is executed and it is practically impossible to verify or debug such code.

While assigned goto-statements do not exist in other languages, it is quite easy to simulate such a construct by defining many small subprograms and passing around pointers to the subprograms. You will find it difficult to relate a particular call with the pointer assignment that connected it to a specific subprogram. Thus pointers to subprograms should only be used in highly structured situations such as tables used by interpreters or callback mechanisms.

6.8 Exercises

1. Does your compiler implement all case-/switch-statements the same way, or does it try to choose an optimal implementation for each statement?

2. Simulate a Pascal repeat-statement in Ada and C.

3. The original definition of Fortran specified that a loop is executed at least one time even if the value of low is greater than the value of high! What could motivate this design?

4. The sequential search in C:

```
while (s[i].data != key)
    i++;
```
C

might be written as follows:

```
while (s[i++].data != key)
    ;                          /* Null statement */
```
C

What is the difference between the two computations?

5. Suppose that Ada did allow an index variable to exist after the scope of the loop. Show how optimization of a loop would be affected.

6. Compare the code generated for a search implemented using a break- or exit-statement with the code generated for a sentinel search.

7. Write a sentinel search using do-while rather than while. Is it more efficient?

8. Why did we put the sentinel at the beginning of the array rather than at the end?

9. (Scholten) The game of Go is played with stones of two colors, black and white. Suppose that you have a can with an unknown mixture of stones and that you execute the following algorithm:

```
while Stones_Left_in_Can loop
    Remove_Two_Stones(S1, S2);
    if Color(S1) = Color(S2) then
        Add_Black_Stone;
    else
        Add_White_Stone;
    end if;
end loop;
```
Ada

Show that the loop terminates by identifying a value which is always decreasing but always non-negative. Can you say anything about the color of the last stone to be removed? (Hint: write a loop invariant on the number of white stones.)