Chapter 2
Advanced Control Flow*

This chapter reviews basic control flow and then presents advanced control flow, which is not easily simulated using the basic control structures.

2.1 Basic Control Flow

Basic control-structures allow for virtually any control-flow pattern ***within*** a routine. The basic control-structures are:

- conditional: if-then-else, case
- looping: while, repeat (do-while), for
- routine: call/return (recursion, routine pointers)

which are used to specify most of the control flow in a program. In fact, only the while-loop is necessary to construct any control flow as it can be used to mimic the others. This assertion is demonstrated by showing the transformations from the if-then-else and repeat-loop constructs to the while-loop:

IF	IF using WHILE	REPEAT	REPEAT using WHILE
if (a > b) { b = 2; } **else** { b = 7; }	**flag = true;** **while** (a > b **& flag**) { **flag = false;** b = 2; } **while (flag)** { **flag = false;** b = 7; }	**do** { x += 1; } **while** (x < 10);	**flag = true;** **while (flag** | x < 10) { x += 1; **flag = false;** }

* This chapter is a major revision of "A Case for Teaching Multi-exit Loops to Beginning Programmers" in [6].

P.A. Buhr, *Understanding Control Flow*, DOI 10.1007/978-3-319-25703-7_2

In the if-then-else simulation (left), the flag variable ensures the loop body executes at most once and the second while-loop does not execute if the first one does; in the repeat simulation, the flag variable ensures the loop body executes at least once. The reader can generalize to the case statement, which is a series of disjuncted if-statements, and the for-loop, which is a while-loop with a counter.

Another example of restricted control-flow is any program can be written using a single while-loop, any number of if-statements, and extra variables, where the if-statements re-determine what was happening on the previous loop iteration. However, even though the while-loop can mimic the other control structures or only one while-loop is necessary, it is clear that programming in this manner is awkward. Therefore, there is a significant difference between being able to do a job and doing the job well; unfortunately, it is hard to quantify the latter, and so it is often ignored. A good analogy is that a hammer can be used to insert screws and a screwdriver can pound nails so they are weakly interchangeable, but both tools have their own special tasks at which they excel. This notion of **weak equivalence** is common in computer science, and should be watched for because it does not take into account many other tangible and intangible factors. Weak equivalences appear repeatedly in this book.

The remainder of this chapter discusses several advanced forms of control flow: multi-exit loop and multi-level exit. Each form allows altering control flow in a way not trivially possible with the basic control-structures. Again, there is a weak equivalence between the new forms of control and the while-loop, but each new form provides substantial advantages in readability, expressibility, maintainability and efficiency, which justifies having specific language constructs to support them.

2.2 GOTO Statement

Conspicuous by its absence from the list of basic control-structures is the GOTO statement. The reason is most programming styles do not advocate the use of the goto, and some modern programming-languages no longer provide a GOTO statement (e.g., Java [10]). However, the much maligned goto is an essential part of control flow and exists implicitly in virtually all control structures. For example, the following shows the implicit GOTO statements in both the if-then-else and while-loop.

Implicit Transfer	Explicit Transfer
if (*C*) { *// false => transfer to else* . . . *// then-clause* *// transfer after else* } **else** { . . . *// else-clause* }	**if** (**!** *C*) **goto L1;** . . . *// then-clause* **goto L2;** **L1:** . . . *// else-clause* **L2:**
while (*C*) { *// false => transfer after while* . . . *// loop-body* } *// transfer to start of while*	**L3: if** (**!** *C*) **goto L4;** . . . *// loop-body* **goto L3;** **L4:**

For the if-then-else (top), there are two gotos: the first transfers over the then-clause if the condition is false (notice the reversal of the condition with the **not** operator), and the second transfers over the else-clause after executing the then-clause if the condition is true. For the while-loop (bottom), there are also two gotos: the first transfers over the loop-body if the condition is false (notice the reversal of the condition with the **not** operator), and the second transfers from the bottom of the loop-body to before the loop condition. The key point is that programming cannot exist without the goto (unconditional branch) to transfer control. Therefore, it is necessary to understand the goto even when it is used implicitly within high-level control-structures. By understanding the goto, it is possible to know when it is necessary to use it explicitly, and hence, when it is being used correctly.

Given that the goto is essential, why is it a problem? During the initial development of programming, it was discovered that arbitrary transfer of control results in programs that are difficult to understand and maintain:

> For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce [7, p. 147].

During the decade of the 1970s, much work was done to understand the effect of different restrictions on transfer of control with regard to expressibility versus understandability and maintainability. This research has the generic name **structured programming**. The conclusion was that a small set of control structures used with a particular programming style makes programs easier to write and understand, as well as maintain. In essence, a set of reasonable conventions was adopted by programmers, similar to adopting a set of traffic rules for car drivers. Not all structured programming conventions are identical, just as traffic conventions vary from country to country. Nevertheless, the conventions have a large intersection, and therefore, it is only necessary to learn/adjust for any minor differences.

The basis for structured programming is the work of Böhm and Jacopini [5], which shows any arbitrary control-flow written with gotos can be transformed into an equivalent restricted control-flow uses only IF and WHILE control structures. The transformation requires the following modifications to the original goto program:

- copy code fragments of the original program, and
- introduce **flag variables** used only to affect control flow, and thus, not containing data from the original program.

Interestingly, even when there are no gotos, constructing control flow with just IF and WHILE frequently requires code duplication and the use of flag variables to overcome the limitations of WHILE. For example, when using a WHILE loop, it is often necessary to copy code before the loop and at the end of the loop (see page 15), and to effect early termination of nested WHILE loops, it is often necessary to use flag variables (see page 21). However, copying code has the problem of maintaining consistent copies, i.e., every programmer has experienced the situation of changing the code in one copy but forgetting to make the corresponding change in the other copies. Although this problem is mitigated by putting the duplicated code

in a routine, routines introduce additional declarations, calls and execution cost, especially if many parameters are required. Similarly, introducing flag variables requires extra declarations and executable statements to set, test and reset the flags. ***In fact, a flag variable used strictly for control flow purposes is the variable equivalent of a GOTO statement.*** This equivalence exists because modification of a flag variable can be buried in a loop body or even appear in remote parts of the program, and thus, has the same potential to produce unintuitive control flow. This additional complexity is often a consequence of restricting control flow, e.g., only using IF and WHILE.

Peterson, Kasami and Tokura [23] demonstrate some code copying and all flag variables required by the Böhm and Jacopini transformation can be eliminated by using a new control-structure. This control structure involves an EXIT statement and requires labelling some loops (labelled **break** in Java). The EXIT statement transfers control *after* the appropriately labelled control-structure, exiting any intervening (nested) control-structures, e.g.:

```
A: LOOP                          // outer loop A
    ...
    B: LOOP                      // inner loop B
        ...
        WHEN C2 EXIT B           // exit inner loop B
        ...
        WHEN C1 EXIT A           // exit outer loop A
        ...
    END LOOP
    ...
END LOOP
```

EXIT A transfers to the statement after (the END LOOP of) the outer loop labelled A and EXIT B transfers to the statement after the inner loop labelled B. Fundamentally, EXIT is a GOTO but restricted in the following ways:

- EXIT cannot be used to create a loop, i.e., cause a backward branch in the program, which means only looping constructs can be used to create a loop. This restriction is important so all situations resulting in repeated execution of statements are solely delineated by the looping constructs.
- Since EXIT always transfers out of containing control-structures, it cannot be used to branch into a control structure. This restriction is important for languages allowing declarations within the bodies of control structures. Branching into the middle of a control structure may not create the necessary local variables or initialize them.

Therefore, exits are a valuable programming tool to simplify or remove techniques required when control flow is restricted.

Three different forms of exits are identified:

1. one in which control leaves only the current loop control-structure at possibly several different locations, called a **multi-exit** or **mid-test** loop;
2. one in which control leaves multiple levels but to a statically determinable level, called **static multi-level exit**;

3. one in which control leaves multiple levels but to a level that is determined dynamically, called **dynamic multi-level exit**.

In the case of multi-exit loop, the level of exit is determined implicitly, and hence, it is unnecessary to name program levels nor to use that name to exit the loop. However, terminating multiple control-structures either statically or dynamically requires additional naming. Because the multi-exit loop occurs more frequently than the other kinds of exits, and absence of names reduces program complexity, some languages have a specific construct for direct exit from a loop (e.g., unlabelled **break** in C/C++/Java). However, static multi-level exit subsumes multi-exit loop, so having separate constructs for multi-exit and static multi-level exit is a programmer convenience.

2.3 Multi-Exit (Mid-Test) Loop

The anatomy of a loop construct involves two parts: an infinite loop and a mechanism to stop the loop. This anatomy can be seen by decomposing the two most common looping constructs into these two constituent parts. The while-loop (left) is composed of an infinite loop with one exit located at the top (right, Ada [30]):

```
while ( i < 10 ) {          loop                        -- infinite loop
                              exit when i >= 10;        -- loop exit
    ...                         ...       ↑ reverse condition
}                           end loop
```

The repeat-loop (left) is composed of an infinite loop with one exit located at the bottom (right):

```
do                          loop                        -- infinite loop
    ...                       ...
                              exit when i >= 10;        -- loop exit
while ( i < 10 );           end loop      ↑ reverse condition
```

Both right examples have one important difference, the condition of the **if** statement is reversed relative to the **while** or **do**-**while**. In this case, the condition specifies why the loop exits instead of why the loop should continue execution.

Unlike the fixed **while** and **do**-**while** looping constructs, the **loop** construct allows the exit to be located in positions other than the top or bottom of the infinite loop. A multi-exit (mid-test) loop can have one or more exit locations anywhere in the body of the loop. Therefore, the anatomy of a multi-exit loop is:

```
loop
  exit when C1; -- top
    ...
  exit when C2; -- middle
    ...
  exit when C3; -- bottom
end loop
```

The one advantage of loop exit is the ability to change the kind of loop solely by moving the exit line; i.e., no other text in the program needs to be modified.

Given the basic anatomy of a multi-exit loop, what is the best way to implement it in C/C++? An infinite loop can be created in the following ways:

```
while ( true ) {          for ( ;; ) {          do {
    ...                       ...                 ...
}                         }                     } while ( true );
```

In this book, the **for** version (middle) is used because it is more general; i.e., it is easily modified to have a loop index:

```
for ( int i = 0; i < 10; i += 1 ) { // loop index
```

without having to change existing text in the program. If a **while/do-while** construct is used to generate an infinite loop, it must be changed to a **for** construct if a loop index is introduced.[1] This style is used wherever possible in this book, i.e., programs are written and language constructs are used so new text can be inserted or existing text removed without having to change any other text in the program. This approach ensures any errors introduced by a change are only associated with new code, never existing code. (Occasionally it is necessary to violate this style so examples fit.)

The loop exit is done with two statements: **if** and **break**. The **if** statement checks if the loop should stop, and the **break** statement terminates the loop body and continues execution after the end of the loop body. As above, implementing a **while** or **do-while** loop is trivial by placing the exit at the top or bottom of the loop body:

```
for ( ;; ) {   // WHILE          for ( ;; ) {   // REPEAT
  if ( i >= 10 ) break;              ...
    ...                            if ( i >= 10 ) break;
}                                  }
```

and the kind of loop can be changed by simply moving the exit location in the loop body. In contrast, changing a while loop to a repeat loop requires modifying several lines, which increases the possibility of introducing errors. A disadvantage is confusion using an **if** statement that does not continue execution in the loop body; i.e., the usual flow of control associated with an **if** statement is not followed in this case because of the exit. Having special syntax, like **exit when** *C*, removes this confusion. Having to reverse the exit condition from a while loop can be an advantage: most programmers think, talk and document the situations that cause a loop to stop versus continue looping, which often leads to errors specifying the loop conditional expression. For example, when merging two files using high-values, the loop performing the merge exits when both keys become high-value:

```
loop
  exit when f1.key = high and f2.key = high;
```

Using a **while** or **do-while**, the expression must be inverted. Without an understanding of de Morgan's law, many programmers fail to make this transformation correctly, negating the relational operators but not the logical, producing:

[1] Alternatively, the loop index could be added to the **while** construct by adding multiple statements. However, this defeats the purpose of having a **for** construct and increases the chance of additional errors when adding or removing the loop index because three separate locations must be examined instead of one.

```
while ( f1.key ~= high & f2.key ~= high ) do
                        ↑ should be |
```

Thus, the specification of loop termination in the form of when it should stop rather than when it should continue produces programs that are often easier to write and understand.

The more interesting use of the loop exit is exiting from locations other than the top or bottom of the loop, e.g.:

```
for ( ;; ) {
    ...
  if ( ... ) break;  // middle exit
    ...
}
```

For many programmers, this control flow seems peculiar, but in fact, exits of this form are necessary to eliminate copied code needed with while and repeat loops. The best example of duplicated code is reading until end of file, as in the code fragment:

```
cin >> d; // priming
while ( ! cin.fail() ) {
    ...
    cin >> d;
}
```

```
for ( ;; ) {
    cin >> d;
  if ( cin.fail() ) break;
    ...
}
```

The left example shows traditional **loop priming**, which duplicates the reading of data before and at the end of the loop body. The reason for the initial read is to check for the existence of any input data, which sets the end-of-file indicator. However, all duplicate code is a significant software-maintenance problem. The right example uses loop exit to eliminates the duplicate code, and hence, has removed the software-maintenance problem solely by altering the control flow. A similar C++ idiom for this example is:

```
while ( cin >> d ) {  // cin returns status of read
    ...
}
```

but results in side-effects in the expression (changing variable d) and precludes analysis of the input stream (cin) without code duplication, e.g., print the status of stream cin after every read for debugging purposes, as in:

```
while ( cin >> d ) {
    cout << cin.good() << endl;

    ...
}
cout << cin.good() << endl;
```

```
for ( ;; ) {
    cin >> d;
    cout << cin.good() << endl;
  if ( cin.fail() ) break;
    ...
}
```

In the C++ idiom (left), printing the status of the read, i.e., did it fail or succeed, must be duplicated after the loop to print the status for end-of-file. In the multi-exit case (right), the status is printed simply by inserting one line of code after the read.

Many programmers have a tendency to write an unnecessary **else** with loop exits:

BAD	GOOD
for (;;) { S1 **if** (C1) { **S2** } **else** { **break**; } S3 }	**for** (;;) { S1 **if** (**!** C1) **break**; **S2** S3 }

BAD	GOOD
for (;;) { S1 **if** (C1) { **break**; } **else** { **S2** } S3 }	**for** (;;) { S1 **if** (C1) **break**; **S2** S3 }

In both examples, the code S2 is logically part of the loop body not part of an **if** statement. Making S2 part of the loop exit is misleading and makes it more difficult to locate the exit points. Hence, it is good programming style to not have an **else** clause associated with a loop exit.

It is also possible to have multiple exit-points from the middle of a loop (left):

```
for ( ;; ) {
    S1
 if ( C1 ) break;

    S2
 if ( C2 ) break;

    S3


}
```

```
bool flag1 = false, flag2 = false;
while ( ! flag1 & ! flag2 ) {
    S1
    if ( C1 ) flag1 = true;
    } else {
        S2
        if ( C2 ) flag2 = true;
        } else {
            S3
        }
    }
}
```

This loop has two reasons for terminating, such as no more items to search or found the item during the search. Contrast the difference in complexity between these two code fragments that are functionally identical. The while loop (right) requires two flag variables, an awkward conjunction in the loop condition for termination, and testing and setting the flags in the loop body. The reduction in complexity for the multi-exit loop comes solely by altering the control flow.

Finally, an exit can also include code that is specific to that exit (left):

```
for ( ;; ) {
    S1
 if ( C1 ) { E1; break; }

    S2
 if ( C2 ) { E2; break; }

    S3


}
```

```
bool flag1 = false, flag2 = false;
while ( ! flag1 & ! flag2 ) {
    S1
    if ( C1 ) flag1 = true;
    } else {
        S2
        if ( C2 ) flag2 = true;
        } else {
            S3
        }
    }
}
if ( flag1 ) E1; // which exit ?
else E2;
```

The statements E1 and E2 in the then-clause of the exiting **if** statement are executed *before* exit from the loop (meaning they are in the scope of loop declarations, like a

loop index). For a single exit, any specific exit code can be placed after the loop. For multiple exits with specific exit code for each exit, placing the exit codes after the loop requires additional **if** statements to determine which code should be executed based on the reason for the loop exit (right). In some situations, it is impossible to retest an exit condition after the loop because the exit condition is transitory, e.g., reading a sensor that resets after each read, so flag variables must be used.

2.3.1 Loop-Exit Criticisms

While the multi-exit loop is an extremely powerful control-flow capability, it has its detractors. Critics of the multi-exit loop often claim exits are difficult to find in the loop body. However, this criticism is simply dealt with by outdenting the exit (**eye-candy**), as in all the examples given thus far, which is the same indentation rule used for the **else** of the if-then-else:

```
if ( ... ) {                        if ( ... ) {
    XXX XXX                             XXX XXX
    } else { // not outdented       } else { // outdented, good eye-candy
    XXX XXX                             XXX XXX
}                                   }
```

In the left example, it is difficult to locate the else-clause because the **else** keyword is indented at the same level as the statements in the then/else clauses; in the right example, the else-clause is outdented from these statements to make it trivial to locate. In addition (or possibly as an alternative to outdenting), comments can be used to clearly identify loop exits. As well, folding specific exit-code into a loop body can make it difficult to read when the exit-specific code is large. This criticism is justified, and so programmers must decide when such an approach is best.

Critics of multi-exit loop also claim the control-flow is "unstructured", i.e., it does not follow the doctrine originally presented for structured programming, which only uses **if** and **while**. (See [16] for a comprehensive discussion of both sides of the issue.) However, this is a very restricted and conservative view of structured programming and does not reflect current knowledge and understanding in either programming-language design or software engineering. That is, it makes no sense to summarily reject the multi-exit loop when its complexity is virtually equivalent to a while loop and yet it simplifies important coding situations. The key point is to understand the potential techniques available to solve a problem and select the most appropriate technique for each individual case.

2.3.2 Linear Search Example

A good example that illustrates the complexity of control flow is a linear search of an array to determine if a key is present as an element of the array. In addition, if

the search finds the key, the position in the array where the key is found must be available for subsequent operations. The two common errors in this search are:

- using an invalid subscript into the array. This problem often happens when the array is full and the key is not present; it is easy to subscript one past the end of the array.
- off-by-one error for the index of the array element when there is a match with the search key. This problem often happens if the loop index is advanced too early or too late with respect to stopping the search after finding the key.

Several solutions to this problem are examined.

The first solution uses only **if**-**else**, **while** and basic relational/logical operators:

```
i = -1; found = 0;
while ( i < size - 1 & ! found ) {
    i += 1;
    found = key == list[i];
}
if ( found ) { . . .      // found
} else { . . .            // not found
}
```

Why must the program be written in this way? First, if the condition of the while-loop is written like this:

```
while ( i < size & key != list[i] )
```

a subscript error occurs when i has the value size because the bitwise logical operator & evaluates both of its operands (and C arrays are subscripted from zero). There is no way around this problem using only **if**-**else**, **while** constructs and basic relational/logical operators. Second, variable i must be incremented *before* key is compared in the loop body to ensure it has the correct value should the comparison be true and the loop stop. This results in the unusual initializing of i to -1, and the loop test succeeding in the range -1 to $size-2$. Finally, the variable found is used solely for control flow purposes, and therefore, is a flag variable.

Many C programmers believe there is a simple second solution using only the **if**-**else** and **while** constructs (**for** is used to simplify it even further):

```
for ( i = 0; i < size && key != list[i]; i += 1 ) {}
                          ↑ only executed if i < size
if ( found ) { . . .      // found
} else { . . .            // not found
}
```

However, there are actually three control structures used here: **if**-**else**, **for**, and &&. Unlike the bitwise operator &, the operation && is not a traditional operator because it may not evaluate both of its operands; if the left-hand operand evaluates to false, the right-hand operand is not evaluated. By not evaluating list[i] when i has the value size, the invalid subscript problem is removed. Such partial evaluation of an expression is called **short-circuit** or **lazy** evaluation. A short-circuit evaluation is a control-flow construct and not an operator because it cannot be written in the form of a routine as a routine call eagerly evaluates its arguments before being invoked

in most programming languages.[2] All operators in C++ can be written in the form of a routine. Therefore, the built-in logical constructs && and || are not operators but rather control structures in the middle of an expression.

Finally, to understand both solutions requires a basic understanding of boolean algebra, i.e., the truth tables for boolean operators $\wedge$ (and) and $\vee$ (or), and for &&, it is necessary to understand the relationship $false \wedge ? = false$, and for ||, the relationship $true \vee ? = true$. In some cases, students may not have taken boolean algebra before attempting to write a linear search in a programming course, making the explanation of the linear search doubly difficult.

The third solution uses a multi-exit loop:

```
for ( i = 0; ; i += 1 ) {
  if ( i == size ) break;
  if ( key == list[i] ) break;
}
if ( i == size ) { . . .        // not found
} else { . . .                  // found
}
```

If the first loop-exit occurs, it is obvious there is no subscript error because the loop is terminated when i has the value size. If the second loop-exit occurs, i must reference the position of the element equal to the key. As well, no direct knowledge of boolean algebra is required (even though the logic of the boolean $\wedge$ occurs implicitly); hence, this form of linear search can be learned very early.

Finally, all the linear-search solutions have an additional test after the search to re-determine the result of the search; i.e., when the search ends, it is known if the key is or is not found in the array, but this knowledge is thrown away. Hence, after the search completes, it is necessary to check again what happened in the search. This superfluous check at the end of the search can be eliminated by folding this code into the search using exit code, e.g.:

```
for ( i = 0; ; i += 1 ) {
  if ( i == size ) { . . . /* not found */ break; }
  if ( key == list[i] ) { . . . /* found */  break; }
}
```

Again, a programmer has to judge whether folding exit code into the loop body is appropriate for each specific situation with respect to functionality and readability.

The key observation is that a multi-exit loop is a viable alternative to other control-flow techniques, e.g., **while**, flag variables and short-circuit logical operators, and is often simpler to understand, eliminates duplicated code and is equally efficient with respect to runtime performance. As well, multi-exit allows new code

[2] Some programming languages, such as Algol 60 [4] and Haskell [13], support deferred/lazy argument evaluation. If C++ had this mechanism, the built-in operators && and || could be written as routines. However, deferred/lazy argument evaluation produces a substantially different kind of routine call, and has implementation and efficiency implications. Interestingly, C++ allows the operators && and || to be overloaded with user-defined versions, but these overloaded versions *do not* display the short-circuit property. Instead, they behave like normal routines with eager evaluation of their arguments. This semantic behaviour can easily result in problems if a user is relying on the short-circuit property. For this reason, operators && and || should never be overloaded in C++. Fundamentally, C++ should not allow redefinition of operators && and || because they do not follow the same semantic behaviour as the built-in versions.

to be inserted between the exits and on exit termination. Accomplishing similar additions, when other constructs are used, requires program restructuring, which introduces the potential for errors.

2.4 Static Multi-Level Exit

A static multi-level exit is one in which control leaves multiple levels of *nested* control-structures to an exit point within the same routine; hence, the exit point is known at compile time, i.e., can be determined before starting program execution. It is the fact that the criteria for completing/terminating a computation is embedded within the nested control-structures that introduces the complexity. Programming languages provide different mechanisms for multi-level exit, some similar to that proposed by Peterson et al. (see Sect. 2.2, p. 10). For example, both μC++ and Java provide a labelled **break** and **continue** statement.

```
L1: {                                   // compound, good eye-candy for labels
    ...
    L2: switch ( ... ) {                // switch
      case ...:
        L3: for ( ... ) {               // loop
            ... continue L3; ...        // next iteration
            ... break L3; ...           // exit loop
            ... break L2; ...           // exit switch
            ... break L1; ...           // exit compound statement
        } // for
        break;                          // exit switch
        ... // more case clauses
    } // switch
    ...
} // compound
```

Note, a label has routine scope in C/C++/Java, meaning it must be unique within a routine (member), and it must be attached to a statement. For **break**, the target label may be associated with a **for**, **while**, **do**, **switch** or compound ({}) statement; for **continue**, the target label may be associated with a **for**, **while** or **do** statement. While labelled **break/continue** eliminate the explicit use of the **goto** statement to exit multiple levels of control structure, the underlying mechanism to perform the transfer is still a **goto**.

Notice, the simple case of multi-level exit (exiting 1 level) is just a multi-exit loop and could have been done using a **break** without a label. The advantage of using an unlabelled **break** is not having to generate unique labels in a routine for each control structure with an exit. The disadvantage of using an unlabelled **break** is the potential for errors when nested control-structures are added:

```
for ( ;; ) {
    ...
  if ( C1 ) break;
    ...
}
```

```
for ( ;; ) {
    for ( ;; ) {  // add new loop
        ...
      if ( C1 ) break; // exit wrong level
        ...
    } // incorrect exit point
} // correct exit point
```

```
B: for ( ;; ) {
    for ( ;; ) {
        ...
      if ( C1 ) break B;
        ...
    }
} // correct exit point
```

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

```
bool flag1 = false;
for ( i = 0; i < 10 && ! flag1; i += 1 ) {
    bool flag2 = false;
    for ( j=0; j<10 && ! flag1 && ! flag2; j += 1 ) {
        ...
        if ( ... ) flag2 = true;
        else {
            ... // rest of loop
            if ( ... ) flag1 = true;
            else {
                ... // rest of loop
            } // if
        } // if
    } // for
    if ( ! flag1 ) {
        ... // rest of loop
    } // if
} // for
```

Fig. 2.1 Contrast multi-loop exit with basic control structures

Here, the unlabelled break must be transformed into a labelled break to maintain correct control flow.

As stated previously (page 12), without multi-level exit, flag variables are required to achieve equivalent control flow. Fig. 2.1 shows that if two nested loops must exit to different levels depending on certain conditions, there is a significant difference between the multi-loop exit and using only **if**-**else** and **while**. Each loop requires a flag variable for termination. If an additional nested loop is added between the existing two loops, an additional flag variable must be introduced and set, tested, and reset in the new and containing loops. Similarly, removing a nested loop requires removing all occurrences of its associated flag variable. Hence, changes of this sort are tedious and error-prone. In contrast, the multi-level exit version has no flag variables, which reduces both complexity and execution cost; as well, nested loops can be added or removed without changing existing code. Notice the exaggerated outdenting scheme for termination of the outer loop, which shows exactly the level the exit transfers to. Remember, a program is a utilitarian entity; indentation should reflect this and not be just an esthetic consideration.

The mechanism providing this flexibility is the labelled exit, but unlike μC++/Java, C++ does not have a labelled **break** (or **continue**). Hence, the previous multi-level exit example must be written as follows in C++:

```
for ( i = 0; i < 10; i += 1 ) {
    for ( j = 0; j < 10; j += 1 ) {
        ...
        if ( ... ) goto L2;
        ...
    if ( ... ) goto L1;
        ...
    } // for
    L2: ;                   // exit point, bad eye-candy
    ...
} // for
L1: ;                       // exit point
```

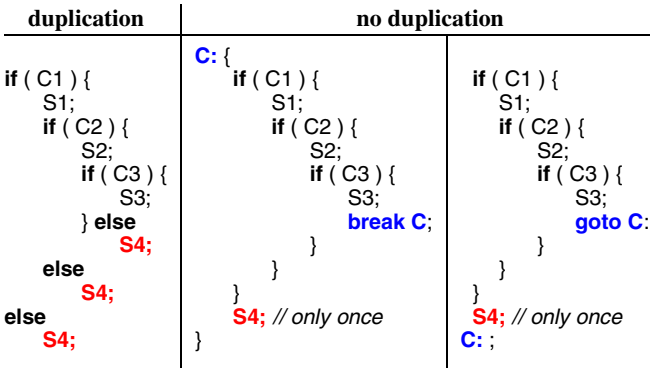


Fig. 2.2 Nested if-statements

There are several points to note. First, because the **break** statement exits only one level, a **goto** statement *must* be used to exit multiple levels. The **goto** transfers control to the corresponding label, terminating any control structures between the **goto** and the label. Second, labels L1 and L2 are necessary to denote the transfer points of the exit but their location is at the *end* of the terminated control-structure rather than the beginning. In the example, a label is associated with a null statement by putting a semicolon after the label, making the label independent of any statement that might be moved or removed.

The advantage of the labelled **break/continue** is allowing static multi-level exits without having to use the **goto** statement and ties control flow to the target control-structure rather than an arbitrary point in a program. Furthermore, the location of the label at the *beginning* of the target control-structure informs the reader (eye candy) of complex control-flow occurring in the body of the control structure. With **goto**, the label at the end of the control structure fails to convey this important clue early enough to the reader. Finally, using an explicit target for the transfer instead of an implicit target allows new nested loop or **switch** constructs to be added or removed without affecting other constructs.

Fig. 2.2 shows this discussion applies equally well to a series of nested if-statement with common **else** clauses, e.g., print an error message if any one of the conditions fails. The goal is to remove the duplicate code S4. Using a routine is one approach, but requires factoring the duplicate code into a routine with necessary parameters. Instead, the problem is solved in situ by changing the control flow. The labelled **break** or **goto** jumps over the common code if control reaches the inner most **if** statement. Otherwise, if one of the **if** statements is false, the common code is executed.

An alternative mechanism for performing some forms of exit is to use **return** statements in a routine body, as in:

```
int rtn( ... ) {
    ... return expr1; ...
    ... return expr2; ...
}
```

Non-nested Returns

```
int rtn( int x, int y ) {
  if ( x <  3 ) return 3;
  if ( x == 3 ) return y;
     x += 1;
     if ( x > y ) {
          x = y + y;
     } else {
          x += 3;
     }
     return x;
}
```

Nested Returns

```
int rtn( int x, int y ) {
     if ( x > 3 ) {
          x += 1;
          if ( x > y ) {
               x = y;
          } else {
               return x + 3;
          }
          return x + y;
     } else if ( x < 3 ) {
          return 3;
     } else {
          return y;
     }
}
```

Fig. 2.3 Nested return-statements

These **return** statements may appear in nested control-structures, and hence, cause termination of both the control structures and the routine. But since **return** terminates a routine, using it for all exit situations in a routine is impossible. As with exits, return points should be highlighted by outdenting or comments to make them easy to locate. Fig. 2.3 shows it is sometimes possible to factor multiple returns towards the beginning of a routine to eliminate nesting. This approach can enhance readability and understandability by reducing the number of return points and nested control-structures with return points.

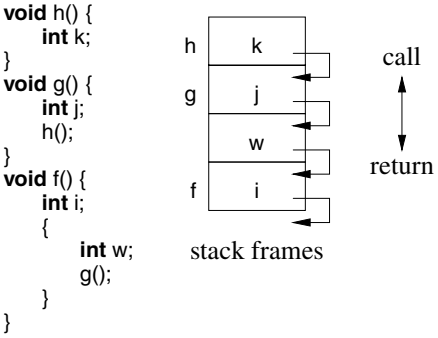
In summary, while multi-exit loops appear often, static multi-level exits appear infrequently. Do not expect to find many static multi-level exits even in large programs. Nevertheless, they are extremely concise and execution-time efficient when needed; use them, but do not abuse them.

2.5 Routine

Routines, subroutines, subprograms, functions or procedures are used to parameterize a section of code so it can be reused without having to copy the code with different variables substituted for the parameters. Routines provide reuse of knowledge and time by allowing other programs to call the routine, dynamically binding actual argument variables to formal parameter variables for the computation of the routine. It is common for one routine to call another, making routines one of the basic building blocks for program construction. Modularization of programs into reusable routines is the foundation of modern software engineering. To support this capability, a programming language allows virtually any contiguous, complete block of code to be factored into a subroutine and called from anywhere in the program; conversely, any routine call is replaceable by its corresponding routine

body with appropriate substitutions of arguments for parameters. The ability to support modularization is an important design factor when developing control structures to facilitate software-engineering techniques.

Routine activation (call/invocation) introduces a complex form of control flow. When control reaches a routine call, the current execution context is temporarily suspended and its local state is saved, and the routine starts execution at its beginning along with a new set of its local variables, called a **routine activation** or **stack frame**.



Note, a language construct defining a declaration scope, i.e., one that can have local declarations, also generates a stack frame that is treated as a call. When a routine returns, it implicitly returns to the point of the routine call, reactivating any saved state. From the control perspective, the programmer does not have to know where a routine is or how to get back to the call; it just happens. In fact, there is no magic, the compiler/linker stores the location of the routine at each call site. When a routine call occurs, this location is used to transfer to the routine and the location of the call is saved so the routine can transfer back. As routines call one another, a chain of suspended routines is formed. A call adds a routine activation, containing a routine's local-state, to one end of the chain; a return removes an activation from the same end of the chain, destroying the routine's local-state. Therefore, the chain behaves like a stack, and it is normally implemented that way. In most programming languages, it is impossible to build this call/return mechanism using the basic control-flow constructs; hence, routine call/return is a fundamental mechanism to affect control flow.

In general, routine-call complexity depends on the references within the routine. If a routine is self-contained, i.e., it only references parameters and local variables:

```
int rtn( int p ) {              // self-contained: local variable references
    int i = 0;
    {   int j = 1;
        {   int k = 2;
            cout << p << i << j << k << endl;
        }
    }
}
```

then the routine does not depend on its lexical context (surrounding code) to execute. In this case, local variable references are statically determined and bound at compile time, even though there may be multiple instances of rtn on the stack (recursion).

C

```
int i = 1;      // external namespace
int rtn() {
    int i;          // hide global i
    {   int j;          // nested blocks
        {   int k;          // nested blocks
            printf( "%d %d %d\n", i, j, k );
        }
    }
}
```

C++

```
int i = 1;                  // external namespace
namespace NS1 {             // in extern namespace
    int i = 2;
    namespace NS2 {         // in namespace NS1
        int i = 3;
        class B {           // in namespace NS2
          protected:
            int i;
          public:
            B() { i = 4; }
        };
        class D : public B { // in class B
            int i;          // via inheritance
          public:
            D() { i = 5; }
            int rtn() {     // member in class D
                int i = 6;
                cout << ::i << NS1::i << NS2::i
                     << B::i << D::i << i << endl;
            }
        };
    }
}
```

Fig. 2.4 Lexical scopes

These fixed references are possible because a routine activation has a pointer to its stack frame (like an object **this** pointer) and all local variables are at fixed offsets from this pointer (base-displacement addressing).

However, the lexical scoping of a programming language allows a routine to reference variables and routines not defined within it (see Fig. 2.4). The lexical scope of a routine in C (left) is only one level, the external namespace, due to the lack of nested routines (see also Fig. 1.1, p. 3). This additional scope level still has fixed references to it because there is only one instance of the external namespace so the compiler can generate a fixed address for any reference; such a pointer is called a **lexical link** or **access link** [3, § 7.3.7]. The lexical scope of a member routine in C++ (right) is significantly more complex: a routine can have inheritance nesting in one or more classes,[3] which can be nested in one or more namespaces. Nevertheless, references within a member routine are still fixed even though there can be multiple routine activations and object instances. Again, each routine activation has a pointer to its stack frame, and a pointer to its containing object instance, i.e. **this**, and there is only one instance of each namespace so the compiler can statically generate references to each variable using offsets within frames/objects or fixed lexical links to namespaces.

This issue is generalized further for languages with more complex lexical scoping, such as nested routines, classes and resumption exceptions (see page 87). Nested routines are examined for both GNU C and C++11 in Fig. 2.5. The nested

[3] While classes can be lexically nested, there is no lexical scoping.

GNU C

```
int f() {
    int i = 0;
    void h(void) { // nest
        void g(void) { // nest
            i += 1;
            if ( i < 3 ) h();
        }
        if ( rand() % 5 == 0 ) g();
        else h();
        i += 1;
    }
    h();
}
int main() {
    f();
}
```

C++11

```
#include <functional>
int f() {
    int i = 0;
    function<void (void)> h = [&](void) {
        function<void (void)> g = [&](void) {
            i += 1;
            if ( i < 3 ) h();
        };
        if ( rand() % 5 == 0 ) g();
        else h();
        i += 1;
    };
    h();
}
int main() {
    f();
}
```

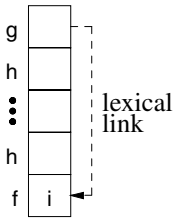


Fig. 2.5 Nested routines

routine g increments the variable f::i, which is accessible in its lexical scope. However, unlike the previous cases, the location of i is not easily found or at a static location because there is an unknown distance on the stack from g to f because the number of recursive calls to h between them is determined dynamically. Hence, when g is called it must be implicitly passed a lexical link, which is a pointer to f's stack frame to access i. Furthermore, unlike the single external area, there may be multiple calls to f on the stack (assume f is recursive), where each one of these calls generates a g with a lexical link to its specific instance of f. Thus, it is impossible to have a static pointer to f, as for the external area, because there can be multiple fs. Lexical links are a standard technique in the implementation of nested routines to give a routine access to variables from the lexical context of its definition [8, Section 9.5].

2.6 Recursion

A **recursive algorithm** is a problem-solving approach that subdivides a problem into two major components:

1. A recursive case, where an instance of the problem is solved by reapplying the algorithm on a refined set of data,
2. and a base case, where an instance of the problem is solved directly.

There is a strong analogy between a recursive algorithm and the mathematical principle of induction. The class of problems suitable for divide-and-conquer solutions lends itself to recursive algorithms because the data is subdivided into smaller and smaller pieces, until there is only one value or a small number of values

that can be manipulated directly. When a recursive algorithm is implemented as a program, a technique called **recursion** is used where a routine calls itself directly or indirectly, forming a call cycle. A direct example of recursion is routine X calling itself any number of times; an indirect example is routine X calling Y, routine Y calling Z, and routine Z calling back to X, and this cycle occurs any number of times. Hence, programs without call cycles are non-recursive and programs with call cycles are recursive.

Many programmers find designing, understanding and coding recursion to be difficult (see also page 4), even though the call mechanism does not differentiate between calls forming and not forming cycles.

> You cannot be *taught* to think recursively, but you can *learn* to think recursively [31, p. 92].

The difficulty occurs because recursive programs form a more complex dynamic program-structure than non-recursive programs. For non-recursive programs, the set of calls form an acyclic graph; for recursive programs, the set of calls form a cyclic graph. Hence, it is the dynamic structure (cycles versus no cycles) that denotes recursion not the call mechanism, which is identical for all calls. And it is the cycles in the call graph that make it more difficult to understand recursive programs, simply because acyclic graphs are simpler than cyclic ones.

The simplest form of recursion is similar to looping:

```
    Looping                 Recursion

int i = 0;                  loop( 0 );              // initialization / start recursion
for ( ;; ) {                void loop( int i ) {
    int j = i + 3;              int j = i + 3;          // local variable
    ...                         ...                     // loop body / repeated statements
  if ( i == N ) break;        if ( i == N ) return    // loop test / base case
    ...                         ...                     // loop body / repeated statements
    i += 1;                     loop( i + 1 );          // increment loop-index / argument
}                           }                       // no operations after return
```

Initializing the loop index corresponds to the initial call to begin the recursion. The test to stop looping corresponds to the base case of the recursion to stop further calls. Incrementing the loop index at the end of the loop corresponds to the recursive call at the end of the routine passing an incremented argument. Of course the loop index and recursive parameter can be an arbitrary entity, such as a pointer traversing a linked list.

This simple form of recursion is called **tail recursion**, where a recursive routine ends solely with a single call to itself, i.e., no other operations can occur after the call except returning a result. What is special about this case? From outside the routine, there is nothing special, i.e., a caller is pending (stopped) waiting for state to be returned so it can process it. However, from inside the routine, the caller sees it call itself at the end. Hence, the next stack frame in the recursion is identical in structure to the current frame, where the arguments are bound to the parameters at each call. As well, no computation after the return means returns can cascade, i.e., the return value is returned, then returned, then returned, etc. without needing to restart callers as there is no future computation, until the initial caller is restarted. Therefore, an optimization of reusing the current stack frame is possible where the argument expression i + 1 is bound to the next i of the call and other local variables

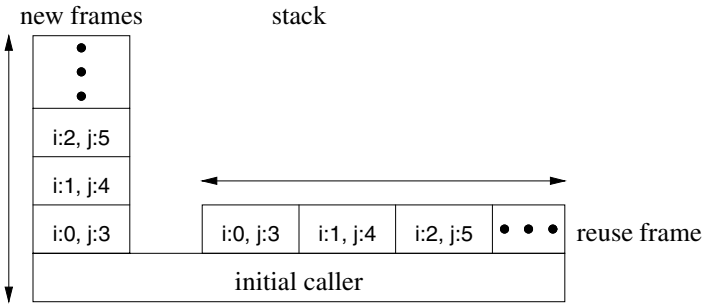


Fig. 2.6 Tail recursion/continuation passing

are reinitialized at the start of the routine, but the location and size of these variables are the same as the previous frame. Hence, the parameters can be mutated for the next call after evaluating the argument expression(s), and any local state is mutated during initialization of the next call.

Fig. 2.6 shows how this optimization can continue independent of recursion: if control does not need to return to a routine for further computation, the next call can continue on the top of the stack. This style of programming is called **continuation passing**. Continuation passing can be explicit by always passing any state previously needed to the next call, making prior stack frames unnecessary. Taken ad nauseam, a program can execute with a single stack frame, at the high cost of pushing sufficient state forward on each call to perform the next portion of the computation. In general, programmers balance the mix of state saving and state forwarding to efficiently perform computations.

However, recursion is more powerful than looping because of its ability to remember both data and execution state. In the previous looping example, there is a single loop-index variable i and all loop transfers are to static locations (start or end of loop). In the previous recursion example, each routine call creates a new instance of the routine's local variables, which includes parameters, and stores the return location for each call even though the return point is the same for all calls (except the first call); hence, there are $N+1$ instances of parameter variable i, local variable j and a corresponding number of return locations. (Why $N+1$ rather than N?) Hence, the left example uses $O(1)$ storage while the right example uses $O(N)$ storage, and there is an $O(N)$ cost for creating and removing these variables. Therefore, using tail recursion to implement a simple loop may not produce the same efficiency. However, tail recursion is easily transformed into a loop, removing any recursion overhead. Most compilers implicitly convert a tail-recursive routine into its looping counterpart; some languages guarantee this conversion so programmers know tail recursive solutions are efficient, e.g., Scheme [1, § 3.5].

In detail, recursion exploits the ability to remember both data and execution state (return points) created via routine calls; both capabilities are accomplished implicitly through the routine-call stack. Data is explicitly remembered via parameters and local variables created on each call to a routine. Execution state is

implicitly remembered via the call/return mechanism, which saves the caller's location on the stack so a called routine knows where to return. Visualizing the implicit state information pushed and popped on the call-stack during recursion may help understand recursion is just normal call/return with cycles in the call graph manipulating an implicit stack data-structure.

Non-tail recursive situations can take advantage of the additional power of recursion. A simple example is reading an arbitrary number of values and printing them in reverse order:

Looping/Array

```
void printRev() {
    int n, v[100]; // dimension
    for ( n=0; n<100; n+=1 ) {
        cin >> v[n];
      if ( cin.fail() ) break;
    }
    for ( n-=1; n>=0; n-=1 ) {
        cout << v[n] << endl;
    }
}
```

Looping/Stack

```
void printRev() {
    stack<int> s; int v;
    for ( ;; ) {
        cin >> v;
      if ( cin.fail() ) break;
        s.push( v );
    }
    for ( ; ! s.empty(); s.pop() ) {
        cout << s.top() << endl;
    }
}
```

Recursion

```
void printRev() {
    int v;
    // front side
    cin >> v;
  if ( cin.fail() ) return;
    printRev();

    // back side
    cout << v << endl;
}
```

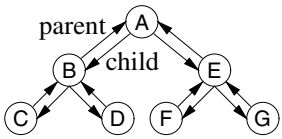
What is interesting about this problem is that *all* values must be read and stored before any printing can occur because the last value appears first.

The looping solution using an array (left) creates a fixed-size array to store the values during reading. Normally, this array is over-dimensioned to handle the worst case number of input values (wasting storage), and the program fails if the worst case is exceeded. As well, the loop index must be managed carefully as it is easy to generate an off-by-one error with the zero-origin arrays in C++.

Alternatively, a variable-sized data-structure can be used to store the values to eliminate the fixed-sized array problems. The looping solution using a stack (centre) creates the simplest data-structure needed to solve this problem, allowing the values to be accessed in Last-In First-Out (LIFO) order.[4] As well, this solution has no explicit loop index, so off-by-one errors are eliminated.

The recursive solution (right) uses the implicit routine-call stack to replace the explicit stack for storing the input values, and replaces the two **for** loops (input/output) by an input and output phase retaining execution-state. In general, recursive control-flow is divided into a front side (before) and a back side (after) for each recursive call. The front side is like a loop going in the forward direction over parameter data, and the back side is like a loop going in the reverse direction over the parameter data with possibly a return value. In this program, the action on the front side of the recursion is to allocate a new local variable v on the stack to remember data and to read a value into this variable. The action on the back side of the recursion is to print the value read and deallocate the local variable, which occurs

[4] While a variable-sized array, e.g., C++ Standard Library vector, is also possible, it is more complex and expensive than is needed for this problem.




```
void printTree( Tree *tree ) {
    Tree *prev = NULL;
    while ( tree != NULL ) {
        if ( prev == tree->parent ) {       // from above ?
            cout << tree->val << endl;   // prefix printing
            prev = tree;
            if ( tree->left != NULL ) tree = tree->left;
            else if ( tree->right != NULL ) tree = tree->right;
            else tree = tree->parent;
        } else if ( prev == tree->left ) { // from left ?
            prev = tree;
            if ( tree->right != NULL ) tree = tree->right;
            else tree = tree->parent;
        } else {                            // from right
            prev = tree;
            tree = tree->parent;
        } // if
    } // while
}
```

Fig. 2.7 Bidirectional tree traversal

in reverse order to the front-side action. The print on the back side of the recursion makes this program non-tail recursive. Note, an arbitrary amount of work can occur between the front and back side of a particular recursion step during intervening recursive (and non-recursive) calls.

If the problem is changed to require more general access to the data, such as printing the values in the forward order for an even number of values and reverse order for an odd number of values, there is no direct recursive solution. The reason is that for an even number of values, the values must be printed on the front side of the recursion, but all the values must be read *before* it is known if there are an even number of them, and by then, all the front-side actions have occurred. Hence, an explicit data-structure is required, like a deque, which allows efficient forward and backward access to the values. If a problem requires even more general access to the data, e.g., efficient random access, an array data-structure is needed.

The next example shows a more complex use of the ability of recursion to implicitly remember data and execution location. In theory, tree traversal, i.e., visiting all nodes of a tree data structure, requires both parent and child links, where a parent link moves up the branch of a tree and a child link moves down the branch of a tree. Fig. 2.7 shows a tree traversal for a tree with bidirectional links. The traversal keeps track of the previous node visited and uses that information along with the current and parent nodes to determine whether to traverse up or down the tree. The amount of storage needed for the traversal is the tree parameter and the prev pointer, as each tree node already contains the parent/child pointers.

If a tree node has no parent pointer, it is impossible to walk up the tree, and hence, to traverse the tree. Traversing a unidirectional tree (child only links) requires a stack to temporarily hold a pointer to the node above the current node (parent link), and this stack has a maximum depth of the tree height. This temporary stack

Looping/Stack

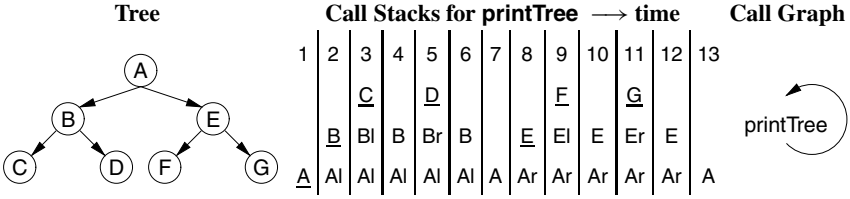
```
struct State {
    Tree *t;                        // parameter
    int es;                         // execution state
    State( Tree *t, int es ) : t( t ), es( es ) {}
};
void printTree( Tree *tree ) {
    stack<State> cs;
    int es;

    for ( ;; ) {
        if ( tree == NULL ) {       // general case ?
      if ( cs.empty() ) break;      // final return ?
            tree = cs.top().t;      // parent
            es = cs.top().es;       // prior exec state
            cs.pop();               // pop stack frame
        } else {                    // normal return
            es = 0;                 // call
        } // if
        switch( es ) {              // execution state ?
          case 0:                   // call
            cout << tree->val << endl; // prefix
            cs.push( State( tree, 1 ) );
            tree = tree->left;
            break;
          case 1:                   // return from left
            cs.push( State( tree, 2 ) );
            tree = tree->right;
            break;
          case 2:                   // return from right
            tree = NULL;            // cause return
        } // switch
    } // for
}
```

Recursion

```
// prefix traversal
void printTree( Tree *tree ) {
    if ( tree != NULL ) {
        cout << tree->val << endl;
        printTree( tree->left );
        printTree( tree->right );
    } // if
}
```

Fig. 2.8 Unidirectional tree traversal

can be an explicit data-structure or implicit using the call-stack. Fig. 2.8 shows a tree traversal for a tree with unidirectional links. The looping solution (left) uses an explicit stack to store the parent node and three possible execution states. (Note, the structure of the looping solution accurately mimics the flow of the control in the recursive solution.) The recursive solution (right) uses the routine-call stack to implicitly store the parameter variable tree (data state) and whether the left or right branch was last visited (execution state). In detail, the recursive solution generates the following information on the call stack during a tree traversal (l ⇒ go left, r ⇒ go right, _ ⇒ print):



Hence, when control is at level N of the tree, level $N-1$ of the call stack has the equivalent of a back (parent) pointer in parameter variable tree to allow walking up

```
int partition( int array[ ], int left, int right, int pivotIndex ) {
    int pivotValue = array[pivotIndex];
    swap( array[pivotIndex], array[right] );          // move pivot to end
    int storeIndex = left;
    for ( int i = left; i < right; i += 1 ) {
        if ( array[i] < pivotValue ) {
            swap( array[storeIndex], array[i] );
            storeIndex += 1;
        }
    }
    swap( array[right], array[storeIndex] );          // move pivot back
    return storeIndex;
}
void quicksort( int array[ ], int left, int right ) {
  if ( right <= left ) return;                        // base case, one value
    int pivotIndex = partition( array, left, right, ( right + left ) / 2 );
    quicksort( array, left, pivotIndex - 1 );         // sort left half
    quicksort( array, pivotIndex + 1, right );        // sort right half
}
```

Fig. 2.9 Recursive quick sort

a tree (when a call returns) without explicit back-pointers in the tree. As well, each call stores its return location on the call stack (l for the call with the left subtree and r for the call with the right subtree). Hence, when control returns from level N of the tree, the return location at $N-1$ of the call stack implicitly indicates which branch of the tree has been traversed. Most importantly, the explicit stack and complex control-flow in the looping solution becomes implicit in the recursive solution using the call stack and front/back side of the recursion. Hiding these details in the recursive solution clearly illustrates the power of the routine-call to retain both data and execution state. This complexity is further hidden in the simple call graph for the recursive tree-traversal, which is only a single cycle, making the recursive solution seem magical.

Another non-tail-recursive example is quick sort [12] (see Fig. 2.9), which employs double recursion like a tree traversal. The recursive algorithm chooses a pivot point in an array of unsorted values. Then all values less than the pivot are partitioned to the left of it and all greater values to the right of it, which means the pivot is now in its sorted location. After partitioning, quick sort is called recursively to sort the set of values on both sides of the pivot. The base case for the recursion is when there is only a single value to be sorted, which is a sorted set.

Note, in both tree traversal and quick sort, the structure of the data (tree, array) remains fixed throughout the recursion; only data within the data structure and auxiliary variables associated with the recursion change. However, recursive solutions to some problems may even restructure the data, such as balancing a binary tree, which reshapes the tree. Changing the data structure during traversal can increase the complexity of the recursive algorithm because elements may move and reappear at new locations. Correspondingly, the program recursion may need to increase in complexity to deal with these cases.

Table 2.1 Péter/Robinson values, $\infty \approx$ extremely large value

m \ n	0	1	2	3	4	$\ldots n$
0	1	2	3	4	5	$n+1$
1	2	3	4	5	6	$n+2$
2	3	5	7	9	11	$2n+3$
3	5	13	29	61	125	$2^{n+3}-3$
4	13	65533	$2^{65536}-3$	$2^{2^{65536}}-3$	∞	$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{n+3 \text{ twos}}-3$
5	65533	$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{65536 \text{ twos}}-3$	∞	∞	∞	∞
$\vdots$ m	∞	∞	∞	∞	∞	∞

Interestingly, it is possible for even simple recursion to cause extreme blowup (see also page 5). For example, Ackermann's function [2]:

$$A(a,b,n) = \begin{cases} a+b & \text{if } n = 0 \\ n-1 & \text{if } b = 0 \text{ and } n = 1,2 \\ a & \text{if } b = 0 \text{ and } n > 2 \\ A(a,A(a,b-1,n),n-1) & \text{if } n > 0 \text{ and } b > 0 \end{cases}$$

and its simplified two-parameter variant created by Rózsa Péter [22] and Raphael M. Robinson [25] (which is often incorrectly called Ackermann's function):

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } n = 0 \\ A(m-1,A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

only use addition and subtraction, but very quickly generate extremely deep recursion and correspondingly very large values due to an exponential effect (see Table 2.1).

This example illustrates that recursion can very succinctly generate extremely complex forms of control flow; therefore, recursion needs to be used carefully, with a full understanding of its storage and execution implications.

2.7 Functional Programming

Functional programming is a style of programming that avoids keeping explicit state by returning values from routines and using immutable variables, i.e., **const** variables that are write-once read-only. One consequence of this programming style is the need to use recursion to perform repeated operations versus looping, which has a mutable loop counter. Specialized programming languages exist that

Mutable Summation	Immutable Summation	
C++ & Looping	C++ & Recursion	Haskell & Recursion
`template<typename T>` `T sum( list<T> *lst ) {` `  T acc = 0;` `  for ( list<T> *p = lst;` `  p != 0; p = p->next )` `  acc += p->val;` `  return acc;` `}`	`template<typename T>` `T sum( list<T> *lst ) {` `  if ( lst == 0 ) return 0;` `  else return lst->val +` `  sum( lst->next );` `}`	`sum lst =` `  if null lst then 0` `  else (head lst) +` `  sum (tail lst)`

Fig. 2.10 Sum list elements

facilitate the functional-programming style, e.g., Lisp [28], Scheme [24], ML [20], Haskell, by supporting a programming style that precludes mutable variables. The main advantage of functional programming is a simplified yet powerful model of computation, which allows easy verification, optimization and parallelization of programs. This discussion focuses solely on the use of recursion to replace mutable variables.

Fig. 2.10 shows a looping and a recursive solution for summing the integer elements of a list in C++ and a recursive solution in Haskell[5]. The C++ solution uses the following list data-structure, similar to that used in most functional systems:

```
template<typename T>
  struct list {
    T val;        // data
    list *next;   // next list element
    list( T val, list *next ) : val( val ), next( next ) {}
};
```

For simplicity, the fields in the structure are accessed directly rather than through accessor routines. Both summing examples are generic and work with any list containing a type with a zero (0) value and plus (+) operator. The recursion eliminates the two mutable variables, acc and p, in the looping solution. Mutable variable acc is replaced by $N+1$ return values and p is replaced by $N+1$ parameter values resulting from the $N+1$ recursive calls to process a list of elements. Because the recursion is not tail recursive, there is no trivial optimization of the recursive programs into the more efficient looping version with mutable state. The program can be converted into tail recursion by changing the interface and restructuring the code:

```
template<typename T>
  T sum( list<T> *lst, T val ) {                    // two parameters versus one
    if ( lst == 0 ) return val;
    else return sum( lst->next, val + lst->val ); // compute sum on front side
}
```

The tail-recursive solution computes the sum on the front side of the recursion, while the non-tail-recursive solution computes on the back side. While the original

[5]Haskell commonly uses pattern-matching versus **if** statements, but there is no semantic difference between these mechanisms.

non-tail-recursive solution has an intuitive structure, the tail-recursive form requires a complex solution for a simple problem. It is possible to regain the original interface by using a helper routine to perform the sum, but that further increases the complexity of the solution. In fact, many compilers can implicitly optimize the non-tail-recursive solution into its more complex form, which retains the simple solution while achieving looping efficiency.

Fig. 2.11 shows two mutable-list and two immutable-list versions for reversing a list in C++. The first mutable-list version uses looping and assumes the list is a deque (doubly linked). The loop traverses the list bidirectionally to its midpoint using two indices, swapping the values from the front and back halves of the list nodes during the traversal. The only difficult part is determining the list midpoint for a list with an even number of values. The second mutable-list solution uses recursion to walk through the list and assumes the list is only singly linked with a list header containing a pointer to the start and end of the list. During the front side of the recursion, the head of the list is removed and remembered. During the back side of the recursion, the node removed on the front side is added to the end of the list. In both cases, the original list is modified to generate the reversed list; in the first approach, the node values are changed, while in the second case, the link fields are changed.

The immutable-list solutions both duplicate the list in reversed order, which is the functional approach. The first immutable-list version uses looping (non-functional) and assumes the list is only singly linked with a list header only containing a pointer to the start of the list. As each node is traversed in the original list, a copy of the node is made and added to the head of the reversed list. The second immutable-list version uses recursion to replace the loop (functional). However, this solution requires access to both the original and the reverse list so it requires two parameters, which changes the routine's signature. The solution hides the second parameter by using a helper routine. During the front side of the recursion, the head of the list is copied and added to the head of the reversed list. During the back side of the recursion, the reversed list is returned. In both cases, the original list is unmodified and new reverse list is generated.

Quicksort is another example where it is possible to have a mutable version that sorts the array of values in place by interchanging values in the array (see Fig. 2.9, p. 32), or immutable version that copies the value into a sorted array, which is subsequently returned. For example, in Haskell, an immutable quicksort can be written as:

```
quicksort [ ] = [ ]
quicksort( p:xs ) = quicksort [x|x <- xs,x < p] ++ [p] ++ quicksort [x|x <- xs,x >= p]
```

Here two patterns are used to deal with the case where the list is empty [ ], which returns an empty set [ ], or non-empty, which returns a new sorted list. In the second case, the argument is divided into its head, p, and the rest of the list, xs, using s:xs, and p becomes the pivot. The new sorted list is constructed by concatenating three lists: the quicksort of a new list with values less than the pivot, [x|x <- xs,x < s], a new list containing the pivot [s], and the quicksort of a new list with values greater than or

Iteration

Mutable List

```
template<typename T>
 deque<T> &reverse( deque<T> &l ) {
    T *fwd, *bwd, *prev;
    for ( fwd = l.head(), bwd = l.tail();
          fwd != bwd && prev != fwd;
          fwd = l.succ( fwd ),
          bwd = l.pred( bwd ) ) {
       swap( fwd->value, bwd->value );
       prev = bwd; // used for even test
    }
    return l;
}
```

Immutable List

```
template<typename T>
 queue<T> *reverse( const queue<T> &l ) {
    queue<T> *r = new queue<T>;
    for ( T *fwd = l.head();
          fwd != NULL;
          fwd = l.succ( fwd ) ) {
       r->addHead( new T( fwd->value ) );
    }
    return r;
}
```

Recursion

Mutable List

```
template<typename T>
 queue<T> &reverse( queue<T> &l ) {
    if ( l.empty() ) return l;
    else {
       T *front = l.dropHead();
       queue<T> &t = reverse( l );
       t.addTail( front );
       return t;
    }
}
```

Immutable List

```
template<typename T>
 queue<T> *helper( queue<T> *r, T *node ) {
    if ( node == NULL ) return r;
    else {
       r->addHead( new T( node->value ) );
       return helper( r, node->next );
    }
}
template<typename T>
 queue<T> *reverse( const queue<T> &l ) {
    queue<T> *r = new queue<T>;
    return helper( r, l.head() );
}
```

Fig. 2.11 Reverse list elements

equal to the pivot, [x|x <- xs,x >= s]. Note, at each level of the recursion, new lists are created that are subsequently sorted rather than modifying the original list of values. The key observation is that functional programming relies on immutable values to achieve its goals of verification, optimization and parallelizing of programs. Recursion is used instead of mutable loop-indices for iteration and creating/copying new data structures is used instead of reusing existing ones.

Finally, as mentioned in Sect. 2.6, p. 26, many programmers find recursion difficult, and hence, struggle to understand and create recursive algorithms and programs. For these programmers, a functional programming-language presents a daunting environment because of the heavy use of recursion. Nevertheless, functional programming techniques are ideal for solving certain kinds of problems. When these techniques are used properly and in the right circumstances, it is possible to create succinct and elegant solutions to complex problems.

2.8 Routine Pointers

The flexibility and expressiveness of a routine comes from abstraction through the argument/parameter mechanism, which generalizes a routine across any argument variables of matching type. However, the code within the routine is the same for all data in these variables. To generalize a routine further, it is necessary to pass code as an argument, which is executed within the routine body. Most programming languages allow a routine to be passed as a parameter to another routine for further generalization and reuse. Java only has routines contained in class definitions so routine pointers must be accomplished indirectly via classes.

C/C++	Java
typedef int (*RP)(int); **int** f(**int** v, **RP p**) { **return** p(v*2) + 2; } **int** g(**int** i) { **return** i-1; } **int** h(**int** i) { **return** i / 2; } cout << f(4, **g**) << endl; cout << f(4, **h**) << endl;	interface RP { **int p(int i);** } **class** F { **static int** f(**int** v, **RP p**) { **return** p.p(v * 2) + 2; } } **class** G implements RP { **public int** p(**int** i) { **return** i-1; } } **class** H implements RP { **public int** p(**int** i) { **return** i / 2; } } System.out.println(F.f(4, **new G()**)); System.out.println(F.f(4, **new H()**));

As for data parameters, routine parameters are specified with a type (return type, and number and types of parameters), and any routine matching that type can be passed as an argument. Routine f is generalized to accept any routine argument of the form: returns an **int** and takes an **int** parameter. Within the body of f, the parameter p is called with an appropriate **int** argument, and the result of calling p is further modified before it is returned. The types of both routines g and h match the second parameter type of f, and hence, can be passed as arguments to f, resulting in f performing different computations rather than a fixed computation.

Fig. 2.12 shows a routine that plots arbitrary functions rather than having a specific function embedded within the routine. Specifically, the plot routine takes start and end points along the X-axis, minimum and maximum points along the Y-axis for scaling, and a function to plot. The X range is divided into 50 intervals, with a star plotted at a scaled distance above/below the X-axis for the Y value returned from the function. Note, the plot has the X-axis rotated along the normal Y-axis so the plot can be arbitrarily long; therefore, the Y-axis is rotated along the normal X-axis, and hence is restricted by the width of the screen or paper.

A routine parameter is passed as a constant reference; in general, it makes no sense to change or copy routine code, like copying a data value. (There are esoteric situations where code is changed or copied during execution, called self-modifying code, but it is rare.) C/C++ requires the programmer to explicitly specify the reference via a pointer, while other languages implicitly create a reference.

A common source of confusion and errors in C/C++ is specifying the type of a routine. A routine type has the routine name and its parameters embedded within the return type, mimicking the way the return value is used at the routine's call site. For example, a routine that takes an integer parameter and returns a routine that takes an integer parameter and returns an integer is declared/used as follows:

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

void plot( double start, double end, double min, double max, double(*f)( double ) ) {
    double points = 50;                         // number of points along X-axis
    double inc = (end - start) / points, range = 1.0 / (max - min),
           offset = -min, height = points;
    cout << fixed << "(" << setw(5) << "X" << ", " << setw(5) << "Y" << ")" << endl;
    for ( double x = start; x <= end; x += inc ) {
        double y = f( x );                      // Y range
        int graph = height * (offset + y) * range;  // Y range scaled into integer domain
        cout << "(" << setw(5) << setprecision(2) << x
             << ", " << setw(5) << setprecision(1) << y << ") "
             << setw(graph) << "*" << endl;
    }
    cout << endl << endl;                       // space between graphs
}
double poly( double x ) { return 7 * pow( x, 2 ) + 2 * x + 1; }

int main() {
    plot( 0, 4 * M_PI, -1, 1, sin );            // plot library and user functions
    plot( 0, 4 * M_PI, -1, 1, cos );
    plot( -1.4, 1.4, -6, 6, tan );
    plot( -1.5, 1.25, 0, 10, poly );
}
```

Fig. 2.12 Graph functions

```
int ( *f( int p ) )( int ) { ... return g; } // return routine
int i = f( 3 )( 4 );  // call returned routine
```

Essentially, the return type is wrapped around the routine name in successive layers (like an onion). While attempting to make the two contexts consistent was a laudable goal, it did work out in practice. The problem is further exacerbated because routine pointers are defined as a pointer but treated as a reference, and hence, automatically dereferenced, allowing the direct call f(3) versus the indirect call (*f)(3).

Two important uses of routine parameters are fixup (see also Sect. 3.1, p. 62) and call-back routines. A **fixup routine** is passed to another routine and is called if an unusual situation is encountered during a computation. For example, when inverting a matrix, the matrix may not be invertible if its determinant is 0, i.e., the matrix is singular. Instead of halting the program if a matrix is found to be singular, the invert routine calls a user supplied fixup routine to see if it is possible to recover and continue the computation with some form of correction (e.g., modify the matrix):

```
int singularDefault( int matrix[][10], int rows, int cols ) { abort(); }
int invert( int matrix[][10], int rows, int cols,
            int (*singular)( int matrix[][10], int rows, int cols ) = singularDefault ) {
    ... if ( determinant( matrix, rows, cols ) == 0 ) {
        correction = singular( matrix, rows, cols ); // possible correction
    } ...
}
int fixup( int matrix[][10], int rows, int cols ) { return 0; }
invert( matrix, 10, 10, fixup ); // fixup rather than abort
```

The fixup routine generalizes the invert routine because the corrective action is specified for each call, and that action can be tailored to a particular usage. By giving the fixup parameter a default value, most calls to invert need not provide a fixup argument.

A **call-back routine** is used in event programming. When an event occurs, one or more call-back routines are called (triggered) and each one performs an action specific for that event. For example, a graphical user interface has an assortment of interactive "widgets", such as buttons, sliders and scrollbars. When a user manipulates the widget, events are generated representing the new state of the widget, e.g., a button changes from up to down, and the widget remembers the new state. A program registers interest in state transitions for different widgets by supplying a call-back routine, and each widget calls its registered call-back routine(s) when the widget changes state. Normally, a widget passes the new state of the widget to each call-back routine so it can perform an appropriate action, e.g.:

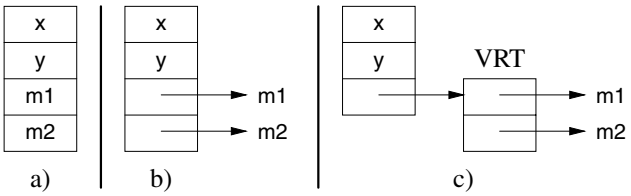
```
int callback( /* information about event */ ) {
    // examine event information and perform appropriate action
    // return status of callback action
}
...
registerCB( closeButton, callback );
```

Event programming with call-backs is straightforward until the call-back routine has multiple states that change depending on the number of times it is called or previous argument values. In this case, it is necessary to retain data and execution state information between invocations of the call-back routine, which can result in an awkward coding style (see also functor page 40 and coroutine Sect. 4, p. 125).

Another area where routine pointers are used implicitly is implementing virtual-routine members. Consider an object containing data and routine members, e.g.:

```
class Base {
    int x, y;                   // data members
    virtual void m1(...);       // routine members
    virtual void m2(...);
};
```

The following are three implementation approaches for objects of type Base.



Implementation a) (left) has each object containing data fields x and y, and copies of routines m1 and m2. However, for N objects of type Base, there are N copies of routines m1 and m2, which are constant, i.e., the code for m1 and m2 does not change during execution and each routine can be tens, hundreds or thousands of bytes long. To remove the routine duplication, implementation b) (middle) factors out the constant values from each object and replaces them with a routine pointer to a constant routine value. While this approach results in significant savings in storage,

each object still duplicates M routine pointers, where M is the number of routine members in the object (two for Base). Implementation c) (right) removes this final duplication by factoring out the common routine pointers into a virtual-routine table (VRT), and each object of type Base points to this common table. Therefore, routine pointers are used implicitly in the implementation of objects containing virtual routine members. Complex variants of the VRT are used to handle multiple and virtual inheritance [29, § 3.5.1].

As mentioned, it is possible to return a routine pointer as well as pass it as an argument, e.g.:

```
double g( double i ) { return i - 1.0; }
double h( double i ) { return i / 2.0; }
double ( *f( int i ) )( double ) {      // returns a pointer to a routine taking
    return i == 0 ? g : h;              //   a double and returning a double
}
```

However, returning a nested routine is complex because of lexical references:

GNU C

```
int ( *f( int i ) )( int ) {
    int w = 2;
    int g( int j ) {
        return i + w + j;
    }
    return g;
}
```

C++11

```
function<int ( int )> f( int i ) {
    int w = 2;
    function<int ( int )> g = [&](int j) {
        return i + w + j;
    };
    return g;
}
```

Nested routine g accesses the parameter i and local variable w. However, when g is returned, routine f terminates and i and w are removed from the stack. As a result, when g is finally called, it references invalid storage.

To make this capability work, in general, requires a mechanism called a **closure**, which does not exist in C and only recently added to C++11, but does exist in other programming languages (first-class routines). The closure is an instance of the environment needed by g to make it work after it is returned from f. In essence, the closure retains information about a particular instance of f's environment existing when g is returned. In this case, the closure is an instance of parameter i and local variable w, and g would need a pointer to this closure as well as any lexical links that existed when the closure was created.

Fig. 2.13 shows a closure can be simulated with a class and made to look like a routine with a functor. A **functor** is a class redefining the function-call operator, which allows its object instances to be called like a routine. In the example, the closure object preserves the environment within f, so after f has terminated, g can be called any number of times in the outer environment. The same effect can be achieved in C++11 by explicitly asking the compiler to make a copy closure for the nested routine g:

```
function<int ( int )> g = [ = ](int j) {      // capture non-local symbols by value
    return i + w + j;
};
cout << f( 3 )( 4 ) << endl;                  // temporary Closure
function<int ( int )> c = f( 3 );             // retain Closure
cout << c( 4 ) + c( 7 ) << endl;              // call g twice
```

```
class Closure {
    int i, w;
  public:
    Closure( int i, int w ) : i( i ), w( w ) {}
    int operator()( int j ) { return i + w + j; } // functor
};
Closure f( int i ) {
    int w = 2;
    return Closure( i, w );                 // capture
}
cout << f( 3 )( 4 ) << endl;                // temporary Closure
Closure c = f( 3 );                         // retain Closure
cout << c( 4 ) + c( 7 ) << endl;            // call g twice
```

Fig. 2.13 Functor

However, a functor is not interchangeable with a routine pointer even if both have the same signature. Functors and routine pointers are only interchangeable through template parameters, where different code is generated for each.

2.9 Iterator

Iterative control flow repeatedly performs an action over a set of data. In some situations, the notion of iteration is abstracted, i.e., it is performed indirectly rather than directly using looping or recursion. One important example is when the elements of a data structure are not directly accessible. For example, a data structure may be implemented such that its elements are structured via a linked-list or a tree. Abstraction requires the data structure interface to be independent of the implementation. Hence, a user cannot construct a loop that accesses any of the link fields in the data structure because the implementation is allowed to change, which precludes any form of direct traversal. Therefore, an indirect abstract mechanism is needed to traverse the structure without requiring direct access to the internal representation. Such indirect traversal is provided by an **iterator** (or generator or cursor) object. A key requirement of indirect traversal is the ability to retain both data and execution state information between calls, e.g., to remember the current location in the traversal between one call to advance the traversal to the next. In some languages, special constructs exist that building special generator objects for iterating [11, 17, 21, 26, 27]. In C++, iterators are created through companion types of a basic data-structure. For example, the C++ Standard Library (stdlib) provides iterator types for many of its data structures. The kinds of traversal provided by an iterator are often those most efficiently possible for the associated data structure. For example, the iterator for a singly linked list may only provide unidirectional traversal (i.e., from head to tail), whereas the iterator for doubly linked list may provide bidirectional traversal (i.e., from head to tail or tail to head). While it is possible to provide reverse traversal of a singly linked list or random access to a

```
int main() {
    list<int> il;                                   // doubly-linked list
    for ( int i = 0; i < 10; i += 1 ) { il.push_back( i ); } // create list elements
    list<int>::iterator fr;                         // forward iterator
    list<int>::reverse_iterator rv;                 // reverse iterator
    for ( fr = il.begin(), rv = il.rbegin(); fr != il.end(); ++fr, ++rv ) { // bidirectionally print
        cout << *fr << " " << *rv << endl;
    }
    for ( fr = il.begin(); fr != il.end(); ++fr ) { il.erase( fr ); } // remove list elements
}
```

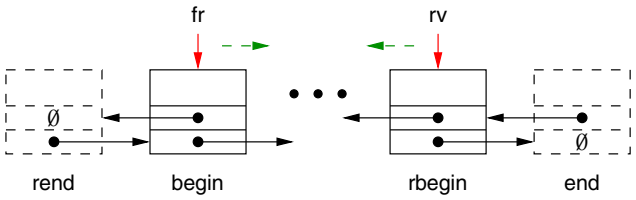


Fig. 2.14 List/iterator

doubly linked list, the cost for these operations can be very expensive; therefore, it is better to change to a more efficient data structure to obtain the necessary iterator access.

Fig. 2.14 demonstrates creating and using a C++ standard-library list and its iterator. This list is doubly-linked, i.e., each node has forward and backward pointers, which allow it to be traversed bidirectionally (forward or backwards). Two kinds of bidirectional iterator, iterator and reverse_iterator, are used to simultaneously traverse the list. An iterator is assigned a starting point, often the beginning/end of the list: begin/rbegin. The prefix operators ++/-- provide the bidirectional movement of an iterator's internal cursor to the next/previous node or to a fictitious node past the end/beginning of the list: end/rend. The forward iterator, iterator, advances from the starting point towards the end of the list, while the reverse iterator, reverse_iterator, advances from the starting point towards the beginning of the list. Because the iterator is an object, there can be multiple iterators instantiated, where each iterator is iterating independently over the same or a completely different list.

To accomplish iteration, the iterator objects fr and rv are closures retaining state information about the traversal in member variables between calls to the traversal operators, ++/--, such as the list being traversed and the current location of the traversal in the list. In effect, the iterator manages a cursor in the closure that moves in a particular direction during the list traversal. Because an iterator can retain arbitrary information between calls, it is possible for some iterators to continue traversing even as elements are being removed (see the last loop in Fig. 2.14). However, the control flow in operators ++/-- cannot be a loop traversing the data structure because it must return after each iteration terminating the loop; hence, the code must be structured sequentially and is controlled by internal execution-state information retain between calls to the iterator movement-operations. This control flow is necessary because a routine member cannot retain state between calls (see also Sect. 4.5, p. 136).

There are two basic kinds of iterators: one that brings the data to the action is called an **external iterator**, and one that brings the action to the data is called an **internal iterator** [9, p. 260]. The previous iterators, iterator/reverse_iterator, are external iterators, i.e., the data is extracted (directly or indirectly) from the data structure and then some action is performed using the data or changing the data. An internal iterator is often called a **mapping iterator** or **applicative iterator** because it maps or applies an action onto each node of the data structure during traversal. For example, the specific recursive printTree routine in Fig. 2.8, p. 31 can be generalized into an internal iterator for performing any action across a complete traversal of the tree data-structure by using a routine pointer, e.g.:

```
template<typename T, typename Action> void map( Tree<T> *tree, Action act ) {
  if ( tree == NULL ) return        // base case ?
    act( tree->data );              // prefix, perform action on data in node
    map( tree->left, act );
    map( tree->right, act );
}
void print( Data &d ) { cout << d << endl; } // action
Tree tree;
map( tree, print );                 // bring action to data: print tree
```

By passing a different routine/functor to the iterator, it is possible to perform a user specified action at each node during the traversal.

Given the two forms of iterator, what are their advantages and disadvantages? In general, an external iterator is simpler than an internal iterator with respect to the following situations:

- not traversing the entire data structure;
- accessing state at the point of traversal;
- traversing multiple data structures and performing some action on the values from each, e.g., comparing two data structures for equality.

For example, Fig. 2.15a shows an external iterator where the loop traverses at most the first 10 nodes of the list, and performs different actions depending on state information local to the traversal context, i.e., variables m and i. Fig. 2.15b shows it is possible to generalize an internal iterator to handle this scenario using a closure to access the state local at the traversal context and to manage its loop counter.

Therefore, there is only a weak equivalence between the two kinds of iterators. External iterators are more general than internal iterators, but simple internal iterators hide more of the traversal details, and hence, are often easier to use.

2.10 Dynamic Multi-Level Exit

Basic and advanced control-structures allow for virtually any control-flow patterns *within* a routine, called a **local transfer**. Modularization states any contiguous, complete block of code can be factored into a routine and called from anywhere in the program (modulo lexical scoping rules). However, modularization fails when factoring exits, in particular, multi-level exits.

```
int i, m = 5;
list<int> l;
list<int>::iterator fr;  // forward iterator
for ( i = 0, fr = l.begin(); i < 10 && fr != l.end();
      i += 1, ++fr ) {
    if ( *fr != m )      // non-matching elements
    else *fr += i;       // matching elements
}



if ( i < 10 )        // all front elements processed
else                 // only front 10 list elements
```

(a) External

```
class Closure {
    int i, &m;
  public:
    Closure( int &m ) : m( m ), i( 0 ) {}
    bool operator()( int &fr ) {
      if ( i == 10 ) return false;
        if ( fr != m ) cout << fr << endl;
        else fr += i;
        i += 1;
        return true;
    }
} c( m );
if ( map( l, c ) ) . . .
else . . .
```

(b) Internal

Fig. 2.15 Kinds of iterator

```
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( . . . )
    . . .
}
```

The inner loop is factored into routine rtn, but fails to compile because the label B1 is in another routine, and labels only have routine scope. (There is a good reason for this restriction.) Hence, control flow *among* routines is rigidly controlled by the call/return mechanism, e.g., given A calls B calls C, it is impossible to transfer directly from C back to A, terminating B in the transfer. Dynamic multi-level exit extends call/return semantics to transfer in the *reverse* direction to normal routine calls, called **non-local transfer**, and allow modularization of static multi-level exit (see Sect. 3.7.2, p. 77).

The control-flow pattern being introduced by non-local transfer is calling a routine and having multiple forms of return. That is, when a routine call returns normally, i.e., control transfers to the statement after the call, it indicates normal completion of the routine's algorithm; whereas, one or more exceptional returns, i.e., control transfers to statements not after the call, indicate some form of ancillary completion (but not necessarily an error). For example, Fig. 2.16 shows a Fortran program with alternate returns from a subroutine. Subroutine AltRet has a normal return and two alternate returns, named 1 and 2. While normal return always transfers to the statement after the call, the caller can specify where the alternative returns transfer on each call, in this case lines labelled 10 and 20. Hence, there are three forms of return from AltRet, normal completion and two exceptional

completions, and the actions for each form are controlled at the call. This pattern is

```
C   Two alternate return parameters, denoted by * and 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
```

Fig. 2.16 Fortran alternate return from subroutine

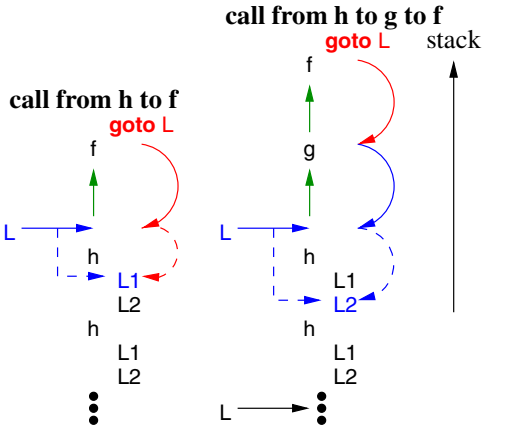
just a generalization of what occurs with multi-exit loop and multi-level exit, where control structures may end with or without an exceptional transfer of control from within them. The pattern also addresses the fact that algorithms can have multiple outcomes, and separating the outcomes from one another makes it easy to read and maintain a program. However, this pattern does not handle the case of multiple levels of nested modularization, where a new modularized routine wants to have an alternate return not to its direct caller but rather an indirect caller several stack frame below it. For example, if AltRet is further modularized, the new routine has to have an alternate return to AltRet and then another alternate return to its caller. Rather than this two-step operation, it is simpler for the new modularized routine to bypass the intermediate step and transfer directly to the caller of AltRet. To accomplish a multiple-step return requires a more complex non-local transfer, which transfers in the same reverse direction as **return** but can return multiple times to an alternate return-point.

The underlying mechanism for non-local transfer is presented in C pseudo-code in Fig. 2.17 using a label variable L, which contains both a pointer to a routine activation on the stack and a transfer point within the routine. In the example, the first nonlocal transfer from f transfers to the static label L1 in the activation record for h, terminating f's activation. The second nonlocal transfer from f transfers to the static label L2 in the activation record for h, terminating the activation records for f and g. Note, the value of the label variable is not statically/lexically determined like a normal label within a routine. Hence, nonlocal transfer, **goto** L in f, involves a two-step operation: direct control flow to the specified routine activation on the stack; and then go to the transfer point (label) within the routine. A consequence of the transfer is that blocks activated between the **goto** and the label value are terminated because of stack unwinding. PL/I [14] is one of a small number of languages (Beta [19],

C pseudo-code

```
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; // dynamic transfer-point
    f( 1 ); goto S1;
 L1: // handle L1 non-local return
 S1: // continue normal execution
    L = L2; // dynamic transfer-point
    g( 1 ); goto S2;
 L2: // handle L2 non-local return
 S2: // continue normal execution
}
```

Control Flow



C setjmp/longjmp

```
jmp_buf L;
void f( int i ) {
    ... longjmp( L, 1 ); ...
}
...
void h( int i ) {
    ...
    if ( setjmp( L ) == 0 ) {
        f( 1 );
        // normal return
    } else {
        // non-local return
    }
    if ( setjmp( L ) == 0 ) {
        g( 1 );
        // normal return
    } else {
        // non-local return
    }
}
```

PL/I

```
TEST: PROCEDURE OPTIONS(MAIN);
    DCL L LABEL;
    F: PROCEDURE( I ); DECLARE I FIXED BIN(31);
        ... GOTO L; ....
    END;
    ...
    H: PROCEDURE;
        ...
        L = L1;
        CALL F( 1 ); GOTO S1;
     L1: /* NON-LOCAL RETURN */

     S1: /* NORMAL RETURN */

        L = L2;
        CALL G( 1 ); GOTO S2;
     L2: /* NON-LOCAL RETURN */

     S2: /* NORMAL RETURN */
    END;
END;
```

Fig. 2.17 Dynamically-scoped transfer-points (nonlocal transfer)

C [15]) supporting nonlocal transfer among dynamic blocks through the use of label variables. PL/I and Beta provide nonlocal transfer via a language control-flow mechanism, while C uses a library approach.

For example, in Fig. 2.17, the C program (left bottom) uses the library routine setjmp to store the current execution context in variable L, which is within the scope of the call to setjmp, setjmp returns a zero value, and a call is made to routine f. Routine f may execute a transfer using the library routine longjmp to the execution-context variable, transferring out of f, through any number of additional scope levels, back within the saved scope of setjmp, which now returns a nonzero value to

indicate alternate return. However, having setjmp return once for the explicit setup call and implicitly again after the longjmp makes this library approach for nonlocal transfer obscure and confusing. The code associated with alternate execution resets execution-context variable L and calls g, which may return normally or to the new alternate code. The PL/I program (right bottom) uses a label variable and works identically to the C pseudo-code (left top). The key is that the transfer point for the longjmp or GOTO is unknown statically; it is determined by the dynamic value of the execution context or label variable.

Unfortunately, nonlocal transfer is too general, allowing branching to almost anywhere, resulting in the structured programming problem. This lack of discipline can make programs less maintainable and error-prone [18, p. 102]. Therefore, a restricted form of nonlocal transfer (**goto**) is necessary, similar to the restrictive form of the local **goto** using labelled break (see Sect. 2.4, p. 20). The next chapter presents different mechanisms to control nonlocal transfer to make it fit into the structured-programming methodology.

2.11 Summary

There is a weak equivalence between the basic and advanced control-structures because simulating advanced control-structures with basic ones is both awkward and inefficient in most cases. Multi-exit loop eliminates duplicated code, which is a maintenance problem. Static multi-level exit allows the elimination of flag variables, which are the variable equivalent of the **goto** statement. As well, static multi-level exit can simulate multi-exit, i.e., multi-exit is a subset (one level exit) of static multi-level exit (multiple levels); nevertheless, some languages provide separate constructs for multi-exit and static multi-level exit. The GOTO statement can be used to simulate both multi-exit and static multi-level exit if these specific control structures do not exist in a programming language. While the GOTO statement can be easily misused, resulting in control flow that is extremely difficult to follow, it can be used for legitimate purposes. Any legitimate purpose *must* satisfy the two restrictions of not creating a loop and not branching into a control structure (see page 11). Therefore, the GOTO should not be maligned or removed from programming languages unless other strongly equivalent constructs are provided. Even more advanced forms of exit are possible when the transfer point is determined dynamically due to routine call. There is a weak equivalence between control structures and routines because control structures can only simulate routine calls via copying code and textually substituting arguments for parameters. Therefore, routines are a fundamental component in control flow. Recursion is a simple extension of basic routine call, i.e., a routine can call itself directly or indirectly, and it is possible because of the basic stack implementation of routine activations. Functional programming uses recursion as the looping mechanism to eliminate mutable data. Routine pointers generalize a routine beyond data parameters allowing the code executed by a routine call to change dynamically. Routine pointers are

the basis for many other programming mechanisms, like virtual routines in object-oriented programming. Returning routine pointers presents problems for nested routines that reference local state, requiring a closure mechanism to capture this local state. Iterators combine a number of control-flow techniques (most importantly a closure) to traverse an abstract and encapsulated data-structure. Iterators may be external or internal, defining who controls the iteration: the client using the iterator or the iterator itself, leading to a weak equivalence between them. Selecting the best control flow for implementing an algorithm is crucial for readability, efficiency and maintainability; but the selection process can only be done if a programmer is aware of the possibilities and understands how each works.

2.12 Questions

1. State the three basic forms of control structures.
2. The following C/C++ code fragment is valid:

```
// assume i, a, b, c, and d are of type int
switch ( i ) {          // unusual switch statement
  case 0:
    if ( a > b ) {
      case 1:
        for ( c = 0; c < 10; c += 1 ) {
          case 2:
            d += 1;
        } // for
    } // if
} // switch
```

   a. Outline very briefly the control flow in this code fragment when i takes on values 0, 1, 2, 4. (Do not explain what the code fragment is trying to accomplish, only what the control flow is doing.)
   b. Explain what control-flow problem this form of control introduces, and why this problem results in both comprehension and technical difficulties.
   c. Find the popular name of a similar *unusually* usage of the **switch** statement.

3. Any program using basic control-flow constructs can be transformed into an equivalent program using only the **while** construct. Rewrite each of the following C++ control structures using **ONLY** expressions (including operators & and |) and **while** statements so that it preserves the exact same execution sequence. The statements **if**/**else**, **switch**, **for**, **do**, **break**, **continue**, **goto**, **throw** or **return**, and the operators &&, || or ? are not allowed. New variables may be created to accomplish the transformation.

a. **if/else** conditional

```
if ( C ) {
    S1
} else {
    S2
}
```

b. **switch** conditional

```
switch ( i ) {
 case 1:
    S1
 case 2:
    S2
    break;
 default:
    S3
}
```

c. **do/while** loop

```
do {
    S1
} while ( C );
```

d. **for** loop

```
for ( int i = 0; i < 10; i += 1 ) {
    S1
}
```

4. Any program using basic control-flow constructs can be transformed into an equivalent program using only the **while** construct. Rewrite the C++ program in Fig. 2.18 using **ONLY** expressions (including operators & and |) and **while** statements so that it preserves the exact same execution sequence. The statements **if/else**, **switch**, **for**, **do**, **break**, **continue**, **goto**, **throw** or **return**, and the operators &&, || or ? are not allowed. New variables may be created to accomplish the transformation. Output from the transformed program must be identical to the original program.
5. Rewrite the following C++ program in Fig. 2.19, p. 51 using **ONLY** expressions (including operators & and |), **one while**, and any number of **if/else** statements so that it preserves the exact same execution sequence. The statements **switch**, **for**, **do**, **break**, **continue**, **goto**, **throw** or **return**, and the operators &&, || or ? are not allowed. New variables may be created to accomplish the transformation. Output from the transformed program must be identical to the original program.
6. Explain the notion of *weak equivalence*.
7. Rewrite an **if/else** and **while** loop using **goto** statements.
8. Convert the C++ program in Fig. 2.20, p. 51 from using **goto** to using only basic control-structures:
9. Define the term *structured programming*.
10. Böhm and Jacopini demonstrated any arbitrary control-flow written with gotos can be transformed into an equivalent restricted control-flow uses only **if** and **while** control structures. What two program modifications are required for the transformation?
11. What is a *flag variable* and why are flag variables a problem?
12. The following code fragment contains duplicate code; show how it can be eliminated.

```
int x
cin >> x;                    // duplicate
while ( ! cin.fail() ) {
    S1;
    cin >> x;                // duplicate
}
```

13. Why is a flag variable the variable equivalent of a **goto** statement?

```
#include <iostream>
#include <cstdlib>                  // atoi
using namespace std;

int main( int argc, char *argv[ ] ) {
    int v1, v2 = 2;
    switch ( argc ) {
      case 3:
        v2 = atoi( argv[2] );
        if ( v2 < 2 || 100 < v2 ) goto usage;
      case 2:
        v1 = atoi( argv[1] );
        if ( v1 < 1 || 100 < v1 ) goto usage;
        break;
      usage:
      default:
        cerr << "Usage: " << argv[0] << " v1 (1-100) [ v2 (2-100) ]" << endl;
        exit( EXIT_FAILURE );
    }
    int i = v1, j = v2, k = 7;
    cout << boolalpha;
    if ( i > 27 ) {
        j = 10;
        cout << j << " " << k << endl;
    } else {
        k = 27;
        cout << j << " " << k << endl;
    }
    for ( ;; ) {
        cin >> i;
        cout << cin.good() << endl;
      if ( cin.fail() ) break;
        cout << i << endl;
    }
}
```

Fig. 2.18 Only while

14. Give two restrictions on static multi-level exit that makes it an acceptable programming language construct, i.e., it cannot be misused like a **goto**.
15. What control-flow pattern is necessary to eliminate all flag variables, and what safe construct provides this pattern?
16. A loop exit in C++ is created with the **if** and **break** statements; however, there are *good* and *bad* ways to code the loop exit in a loop. Rewite the loop bodies in each of the following code fragments to use the *good* pattern for loop exit and explain why the change is better:

a.
```
for ( int i = 0;; i += 1 ) {
    if ( key == list[i] ) {
        break;
    } else {
        cout << list[i] << endl;
    }
}
```

b.
```
for ( int i = 0;; i += 1 ) {
    if ( key != list[i] ) {
        cout << list[i] << endl;
    } else {
        break;
    }
}
```

```
#include <iostream>
using namespace std;

int main() {
    char ch;
    int g, b;
    cin >> noskipws;                          // turn off white space skipping
    for ( ;; ) {                              // loop until eof
        for ( g = 0; g < 5; g += 1 ) {        // groups of 5 blocks
            for ( b = 0; b < 4; b += 1 ) {    // blocks of 4 characters
                for ( ;; ) {
                    cin >> ch;                // read one character
  if ( cin.fail() ) goto fini;                // eof ?
                    if ( ch != '\n' ) break;  // ignore newline
                } // for
                if ( ch == '\t' ) ch = ' ';   // convert tab to blank
                cout << ch;                   // print character
            } // for
            cout << "  ";                     // block separator
        } // for
        cout << endl;                         // group separator
    } // for
 fini: ;
    if ( g != 0 || b != 0 ) cout << endl;
} // main
```

Fig. 2.19 One while and if/else

```
int c[2] = { 1, 1 }, turn = 1;

void dekker( int me, int other ) {
    int i = 0;
 A1:
    c[me] = 0;
 L1:
    if ( c[other] == 0 ) {
        if ( turn == me ) goto L1;
        c[me] = 1;
  B1:
        if ( turn == other ) goto B1;
        goto A1;
    }
    CS();
    turn = other;
    c[me] = 1;
    i += 1;
    if ( i <= 1000000 ) goto A1;
}
```

Fig. 2.20 Goto program

17. Rewrite the following code using only **if/else** and **while** to eliminate the exits from the middle of the loop.

```
for ( ;; ) {
    S1
  if ( i >= 10 ) { E1; break; }
    S2
  if ( j >= 10 ) { E2; break; }
    S3
}
```

18. Describe an indentation technique (eye-candy) so exits in a loop body are easy to read.
19. Write a linear search that looks up a key in an array of elements, and if the key does not appear in the array, it is added to the end of the array; otherwise, if the key does exist the number of times it has been looked up is recorded by incrementing a counter associated with the element in the array. Use the following data structures:

```
struct elem {
    int data;     // data value examined in the search
    int occ;      // number of times data is looked up
};
```

Use a multi-exit loop with exit code in the solution and terminate the program if adding an element results in exceeding the array size.

20. a. Explain how the short-circuit *AND* (&&) and *OR* (||) work.
    b. Use de Morgan's law to convert this **while** loop into an equivalent **for** (;;) loop with an exit using a **break**.

```
while ( Values[i] != HighValue && Values[i] <= Max )  { . . . }
```

    c. What is the inconsistency between the built-in operators && and ||, and user defined versions of these operators?
21. What is *static* about **break** and labelled **break**?
22. Rewrite the following program removing the labelled breaks.

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

23. a. Transform routine do_work in Fig. 2.21 so it preserves the same control flow but removes the **for** and **goto** statements, and replaces them with **ONLY** expressions (including & and |), **if/else** and **while** statements. The statements **switch**, **for**, **do**, **break**, **continue**, **goto**, **throw** or **return**, and the operators &&, || or ? are not allowed. In addition, setting a loop index to its maximum value, to force the loop to stop is not allowed. Finally, copying significant amounts of code or creating subroutines is not allowed, i.e., no transformation where the code grows exponentially with the number of nested loops. New

```
#include <cstdlib>                      // atoi
#include <iostream>
using namespace std;

// volatile prevents dead-code removal
void do_work( int C1, int C2, int C3, int L1, int L2, volatile int L3 ) {
    for ( int i = 0; i < L1; i += 1 ) {
        cout << "S1 i:" << i << endl;
        for ( int j = 0; j < L2; j += 1 ) {
            cout << "S2 i:" << i << " j:" << j << endl;
            for ( int k = 0; k < L3; k += 1 ) {
                cout << "S3 i:" << i << " j:" << j << " k:" << k << " : ";
      if ( C1 ) goto EXIT1;
                cout << "S4 i:" << i << " j:" << j << " k:" << k << " : ";
       if ( C2 ) goto EXIT2;
                cout << "S5 i:" << i << " j:" << j << " k:" << k << " : ";
        if ( C3 ) goto EXIT3;
                cout << "S6 i:" << i << " j:" << j << " k:" << k << " : ";
            } // for
         EXIT3:;
            cout << "S7 i:" << i << " j:" << j << endl;
        } // for
      EXIT2:;
        cout << "S8 i:" << i << endl;
    } // for
  EXIT1:;
} // do_work

int main( int argc, char *argv[] ) {
    int times = 1, L1 = 10, L2 = 10, L3 = 10;
    switch ( argc ) {
      case 5:
        L3 = atoi( argv[4] );
        L2 = atoi( argv[3] );
        L1 = atoi( argv[2] );
        times = atoi( argv[1] );
        break;
      default:
        cerr << "Usage: " << argv[0] << " times L1 L2 L3" << endl;
        exit( EXIT_FAILURE );
    } // switch

    for ( int i = 0; i < times; i += 1 ) {
        for ( int C1 = 0; C1 < 2; C1 += 1 ) {
            for ( int C2 = 0; C2 < 2; C2 += 1 ) {
                for ( int C3 = 0; C3 < 2; C3 += 1 ) {
                    do_work( C1, C2, C3, L1, L2, L3 );
                    cout << endl;
                } // for
            } // for
        } // for
    } // for
} // main
```

Fig. 2.21 Static multi-level exit

variables may be created to accomplish the transformation. Output from the transformed program must be identical to the original program.

b. i. Compare the original and transformed program with respect to performance by doing the following:
   - Remove (comment out) *all* the print (cout) statements in the original and transformed version.
   - Time the execution using the time command:

     ```
     % time ./a.out
     3.21u 0.02s 0:03.32 100.0%
     ```

     (Output from time differs depending on the shell, but all provide user, system and real time.) Compare the *user* time (3.21u) only, which is the CPU time consumed solely by the execution of user code (versus system and real time).
   - Use the program command-line argument (if necessary) to adjust the number of times the experiment is performed to get execution times approximately in the range 0.1 to 100 s. (Timing results below 0.1 s are inaccurate.) Use the same command-line value for all experiments.
   - Run both the experiments again after recompiling the programs with compiler optimization turned on (i.e., compiler flag -O2). Include all 4 timing results to validate the experiments.

   ii. State the observed performance difference between the original and transformed program, without and with optimization.

   iii. Speculate as to the reason for the performance difference between the coding styles.

   iv. Does compiler optimization affect either coding style? (Yes/No answer).

24. Why is it good practice to label all exits?

25. The following form of control flow appears occasionally:

```
if C1 then
    S1
    if C2 then
        S2
        if C3 then
            S3
        else
            S4
        endif
    else
        S4
    endif
else
    S4
endif
```

Notice, if any of the conditions are false, the same code is executed (often printing an error message or back-tracking), resulting in code duplication. One way to deal with the code duplication is to put code S4 into a routine and call it. Alternatively, imagine fixing this duplication with a labelled **if** for use in the

else clause, where the **else** terminates all **if** statements up to the corresponding labelled one, as in:

```
L1: if C1 then
        S1
        if C2 then
            S2
            if C3 then
                S3
    else L1 // terminates 3 if statements
        S4
    endif
```

In this example, all 3 **if** statements transfer to the same **else** clause if the conditional is false. Unfortunately, the syntactic form of the **if** statement in C/C++ makes it impossible to implement this extension. Explain the problem.

26. Explain the control flow of *call/return*.
27. What is *modularization*, and why is it the basic building block of software engineering.
28. Why may a block require a stack frame?
29. Why do stack frames form a stack?
30. Explain the term *lexical scope*, and give two explanations of different lexical scoping.
31. Why do nested routines need a *lexical link*?
32. Explain the control flow that occurs with *recursion*
33. Convert the following looping form of factorial into a recursive form.

```
unsigned long long int factorial( int n ) {
    for ( unsigned long long int fact = 1; n > 1; n -= 1 ) {
        fact *= n;
    }
    return fact;
}
```

34. What is *tail recursion*?
35. What is meant by the *front* and *back* side in recursion?
36. What class of problems are handled best by recursive solutions?
37. Convert the following recursive form of the Euclidean algorithm computing greatest common-divisor into a looping form.

```
int gcd( int x, int y ) {
    if ( y == 0 ) {
        return x;
    } else {
        return gcd( y, x % y );
    }
}
```

38. What two properties define *functional programming*
39. Convert the following generic, mutable summation routine into one that is an immutable summation routine.

```
template<typename T>
 T sum( list<T> *lst ) {
    T acc = 0;
    for ( list<T> *p = lst;
         p != 0; p = p->next )
         acc += p->val;
    return acc;
```

40. Routine pointers introduce what new form of generalization in a routine.
41. What is the problem with routine pointer syntax in C/C++?
42. Explain the term *fixup* and *call back* routine.
43. What is the relationship between routine pointers and virtual routines?
44. How are virtual routines implemented in object-oriented languages to conserve space?
45. Why is returned a nested routine complex?
46. What is a *functor* and explain its purpose?
47. What is an iterator and why must it save state between invocations?
48. Explain *mapping* and *application* iterator.
49. What is a *non-local transfer*?
50. a. Transform the program in Fig. 2.22 replacing **throw/catch** with longjmp/setjmp. Except for a jmp_buf variable to replace the exception variable created by the **throw**, no new variables may be created to accomplish the transformation. Output from the transformed program must be identical to the original program, **except for one aspect, which you will discover in the transformed program**.

    b. i. Compare the original and transformed program with respect to performance by doing the following:
    - Compile the original **throw/catch** and setjmp/longjmp programs without print statements.
    - Time each execution using the time command:

      ```
      % time ./a.out
      3.21u 0.02s 0:03.32 100.0%
      ```

      (Output from time differs depending on the shell, but all provide user, system and real time.) Compare the *user* time (3.21u) only, which is the CPU time consumed solely by the execution of user code (versus system and real time).
    - Use the program command-line arguments to adjust the amount of program execution to get execution times in the range 10 to 100 s. (Timing results below 1 s are inaccurate.) Use the same command-line values for all experiments, if possible; otherwise, increase/decrease the arguments as necessary and scale the difference in the answer.
    - Run both the experiments again after recompiling the programs with compiler optimization turned on (i.e., compiler flag -O2). Include all 4 timing results to validate the experiments.

```
#include <iostream>
using namespace std;
#include <cstdlib>                          // exit, atoi

struct T {
    ~T() { cout << "~T" << endl; }
};
struct E {};
int hc, gc, fc, kc;

void f( volatile int i ) {                  // volatile, prevent dead-code optimizations
    T t;
    cout << "f enter" << endl;
 if ( i == 3 ) throw E();
    if ( i != 0 ) f( i - 1 );
    cout << "f exit" << endl;
    kc += 1;                                // prevent tail recursion optimization
}
void g( volatile int i ) {
    cout << "g enter" << endl;
    if ( i % 2 == 0 ) f( fc );
    if ( i != 0 ) g( i - 1 );
    cout << "g exit" << endl;
    kc += 1;
}
void h( volatile int i ) {
    cout << "h enter" << endl;
    if ( i % 3 == 0 ) {
        try {
            f( fc );
        } catch( E ) {
            cout << "handler 1" << endl;
            try {
                g( gc );
            } catch( E ) {
                cout << "handler 2" << endl;
            }
        }
    }
    if ( i != 0 ) h( i - 1 );
    cout << "h exit" << endl;
    kc += 1;
}
int main( int argc, char *argv[] ) {
    switch ( argc ) {
      case 4: fc = atoi( argv[3] );         // f recursion depth
      case 3: gc = atoi( argv[2] );         // g recursion depth
      case 2: hc = atoi( argv[1] ); break;  // h recursion depth
      default: cerr << "Usage: " << argv[0] << " hc gc fc" << endl;
               exit( EXIT_FAILURE );
    }
    if ( hc < 0 || gc < 0 || fc < 0 ) {
        cerr << "Input less than 0" << endl;
        exit( EXIT_FAILURE );
    }
    h( hc );
}
```

Fig. 2.22 Throw/catch

ii. State the observed performance difference between the original and transformed program, without and with optimization.
iii. Speculate as to the reason for the performance difference.

51. Why does modularization (refactoring) cause problems with multi-level exit?
52. Explain why a label variable for non-local transfer must be a tuple of two values.
53. Why does longjmp not work properly in C++?

References

1. Abelson, H., Adams IV, N.I., Bartley, D.H., Brooks, G., Dybvig, R.K., Friedman, D.P., Halstead, R., Hanson, C., Haynes, C.T., Kohlbecker, E., Oxley, D., Pitman, K.M., Rozas, G.J., Jr., G.L.S., Sussman, G.J., Wand, M., *Ed. by* Richard Kelsey, Clinger, W., Rees, J.: Revised[5] report on the algorithmic language Scheme. SIGPLAN Not. **33**(9), 26–76 (1998)
2. Ackermann, W.: Zum hilbertschen aufbau der reellen zahlen. Math. Ann. **99**(1), 118–133 (1928)
3. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools, 2nd edn. Addison-Wesley Longman Publishing, Boston, MA, USA (2006)
4. Backus, J.W., Bauer, F.L., Green, J., Katz, C., McCarthy, J., Naur, P., Perlis, A.J., Rutishauser, H., Samuelson, K., Vauquois, B., Wegstein, J., van Wijngaarden, A., Woodger, M.: Revised report on the algorithmic language algol 60. Commun. ACM **6**(1), 1–17 (1963)
5. Böhm, C., Jacopini, G.: Flow diagrams, turing machines and languages with only two formation rules. Commun. ACM **9**(5), 366–371 (1966)
6. Buhr, P.A.: A case for teaching multi-exit loops to beginning programmers. SIGPLAN Not. **20**(11), 14–22 (1985)
7. Dijkstra, E.W.: Go to statement considered harmful. Commun. ACM **11**(3), 147–148 (1968). Reprinted in [32] pp. 29–36.
8. Fischer, C.N., LeBlanc, Jr., R.J.: Crafting a Compiler. Benjamin Cummings (1991)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Professional Computing Series. Addison-Wesley, Boston (1995)
10. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, 2nd edn. Addison-Wesley, Boston (2000)
11. Griswold, R.E., Griswold, M.T.: The Icon Programming Language. Prentice-Hall, Englewood Cliffs (1983)
12. Hoare, C.A.R.: Algorithms 63/64: Partition/quicksort. Commun. ACM **4**(7), 321 (1961)
13. Hudak, P., Fasel, J.H.: A gentle introduction to haskell. SIGPLAN Not. **27**(5), T1–53 (1992)
14. International Business Machines: OS and DOS PL/I Reference Manual, 1st edn. (1981). Manual GC26-3977-0
15. Kernighan, B.W., Ritchie, D.M.: The C Programming Language, 2nd edn. Prentice Hall Software Series. Prentice-Hall, Englewood Cliffs (1988)
16. Knuth, D.E.: Structured programming with go to statements. ACM Comput. Surv. **6**(4), 261–301 (1974). DOI http://doi.acm.org/10.1145/356635.356640
17. Liskov, B., Atkinson, R., Bloom, T., Moss, E., Schaffert, J.C., Scheifler, R., Snyder, A.: CLU Reference Manual. Lecture Notes in Computer Science, vol. 114. Springer, New York (1981)
18. MacLaren, M.D.: Exception handling in PL/I. SIGPLAN Not. **12**(3), 101–104 (1977). Proceedings of an ACM Conference on Language Design for Reliable Software, March 28–30, 1977, Raleigh, North Carolina, U.S.A.
19. Madsen, O.L., Møller-Pedersen, B., Nygaard, K.: Object-oriented Programming in the BETA Programming Language. Addison-Wesley, Boston (1993)
20. Milner, R.: A theory of type polymorphism in programming. J. Comput. Syst. Sci. **17**, 348–375 (1978)

21. Murer, S., Omohundro, S., Stoutamire, D., Szyperski, C.: Iteration abstraction in sather. ACM Trans. Progr. Lang. Syst. **18**(1), 1–15 (1996)
22. Péter, R.: Konstruktion nichtrekursiver funktionen. Math. Ann. **1**(111), 42–60 (1935)
23. Peterson, W.W., Kasami, T., Tokura, N.: On the capabilities of while, repeat, and exit statements. Commun. ACM **16**(8), 503–512 (1973)
24. Rees, J., Clinger, W.: Revised[3] report on the algorithmic language Scheme. SIGPLAN Not. **21**(12), 37–79 (1986)
25. Robinson, R.M.: Recursion and double recursion. Bull. Am. Math. Soc. **54**, 987–993 (1948)
26. van Rossum, G.: Python Reference Manual, Release 2.5. Python Software Foundation (2006). Fred L. Drake, Jr., editor
27. Schemenauer, N., Peters, T., Hetland, M.L.: Simple generators. Tech. rep. (2001). http://www.python.org/peps/pep-0255.html
28. Steele, G.: COMMON LISP: The Language. Digital Press, New York (1984)
29. Stroustrup, B.: The Design and Evolution of C++. Addison-Wesley, Boston (1994)
30. United States Department of Defense: The Programming Language Ada: Reference Manual, ANSI/MIL-STD-1815A-1983 edn. (1983). Springer, New York
31. Weissman, C.: Lisp 1.5 Primer. Dickenson Publishing, Belmont (1967)
32. Yourdon, E.N. (ed.): Classics in Software Engineering. Yourdon Press, New York (1979)

http://www.springer.com/978-3-319-25701-3

Understanding Control Flow
Concurrent Programming Using μC++
Buhr, P.A.
2016, XXI, 741 p. 100 illus. in color., Hardcover
ISBN: 978-3-319-25701-3