Your browser does not support inline frames or is currently configured not to display inline frames.

Code Complete: A Practical Handbook of Software Construction. Redmond, Wa.: Microsoft Press, 880 pages, 1993. Retail price: $35. ISBN: 1-55615-484-4.

16.1 Using *goto*s

Computer scientists are zealous in their beliefs, and when the discussion turns to *goto*s, they get out their jousting poles, armor, and maces, mount their horses, and charge through the gates of Camelot to the holy wars.

No one quarrels with using *goto*s to emulate structured constructs in languages that don't support structured control constructs directly. The debate is about languages that support structured constructs, in which *goto*s are theoretically not needed. Here's a summary of the points on each side.

The Argument Against gotos

The general argument against *goto*s is that code without *goto*s is higher-quality code. The famous letter that sparked the original controversy was Edsger Dijkstra's "Go To Statement Considered Harmful" in the March 1968 *Communications of the ACM*. Dijkstra observed that the quality of a programmer's code was inversely proportional to the number of *goto*s the programmer used. In subsequent work, Dijkstra has argued that code without *goto*s can more easily be proven correct.

Code containing *goto*s is hard to format. Indentation should be used to show logical structure and *goto*s have an effect on logical structure. Trying to use indentation to show the logical structure of a *goto*, however, is difficult or impossible.

Use of *goto*s defeats compiler optimizations. Some optimizations depend on a program's flow of control remaining within a few statements. An unconditional *goto* makes the flow harder to analyze and reduces the ability of the compiler to optimize it. Thus, even if introducing a *goto* produces an efficiency at the source-language level, it may well reduce overall efficiency by thwarting compiler optimizations.

Proponents of *goto*s sometimes argue that they make code faster or smaller. But code with *goto*s is rarely the fastest or smallest possible. Donald Knuth's marvelous article, "Structured Programming with go to Statements," gives several examples of cases in which using *goto*s is slower and larger (Knuth 1974).

In practice, use of *goto*s tends to violate structured programming principles. Even if *goto*s aren't confusing when used carefully, once *goto*s are allowed, they spread through the code like termites in a rotting house and aren't used carefully. If any *goto*s are allowed, the bad creep in with the good, so it's better not to allow any of them.

Overall, in the two decades since publication of Dijkstra's original letter, experience has shown the badness of *goto*-laden code. In a survey of the literature, Ben Shneiderman concluded that the evidence supports the badness of the *goto* (Shneiderman 1980).

The Argument for gotos

The argument for the *goto* is characterized by advocating its use in specific circumstances rather than its indiscriminate use. Most arguments against *goto*s are based on indiscriminate use, rather than careful use. The *goto* controversy began when Fortran was the most popular language. Fortran lacked any presentable loop structures, and, in the absence of any good advice on programming structured loops with *goto*s, programmers wrote a lot of spaghetti code. Such code undoubtedly was correlated with low quality programs but has little to do with careful use of a *goto* to make up for a gap in a structured language's capabilities.

A well-placed *goto* can eliminate duplicate code. Duplicate code leads to problems with the two sets of code being modified differently. It increases the size of source and executable files. The bad effect of the *goto* is outweighed in such a case by the worse effect of duplicate code.

The *goto*s is useful in a routine that allocates resources, performs operations on those resources, then deallocates the resources. With *goto*s, you can cleanup in one section of code, and they reduce the danger of forgetting to deallocate the resources in each place you detect an error.

In some cases, *goto*s can result in faster and smaller code. Knuth's marvelous 1974 article cited a few cases in which *goto*s produce a legitimate gain (Knuth 1974).

Good programming doesn't mean eliminating *goto*s. Methodical decomposition, refinement, and selection of control structures automatically leads to *goto*-free programs in most cases. *goto*lessness is not the aim, but the outcome, and putting the focus on no *goto*s isn't helpful.

Two decades worth of research with *goto*s has been inconclusive in demonstrating their badness. In a survey of the literature, B. A. Sheil concluded that unrealistic test conditions, poor data analysis, and inconclusive results failed to support the claim that the number of bugs was proportional to the number of *goto*s (Sheil 1981). That criticism applies to Shneiderman's survey of the literature, cited in the argument against *goto*s, as well as other studies. Sheil did not conclude that *goto*s were good, rather that experimental evidence against them was not conclusive.

Finally, *goto* was included as part of the Ada language, the most carefully engineered programming language in history. Ada was developed long after both sides of the *goto* debate were fully developed, and, after considering all sides of the issue, included the *goto*.

The Phony goto Debate

The primary feature of most *goto* discussions is a shallow approach to the question. People on the "*goto*s are evil" side usually present a trivial code fragment that uses *goto*s, then show how easy it is to rewrite the code fragment without *goto*s. This proves mainly that it's easy to write trivial code without *goto*s. On the other hand, people on the "I can't live without *goto*s" side usually present a case in which eliminating a *goto* results in an extra comparison or two lines of duplicated code. The significance of the gain is questionable, and this proves mainly that there's a case in which using a *goto* results in one less comparison, rarely a significant gain on today's computers.

Most textbooks don't help since they merely provide a trivial example of rewriting code without a *goto* and think they're done. Here's an example of a trivial piece of code from such a textbook:

Pascal Example of Code that's Supposed to be Easy to Rewrite Without gotos

```
repeat
   GetData( InputFile, Data );
   if eof( InputFile ) then
      goto LOOP_EXIT;
   DoSomething( Data );
until ( Data = -1 );
LOOP_EXIT:
```

The book quickly replaces this with *goto*less code:

Pascal Example of Supposedly Equivalent Code, Rewritten Without gotos

```
GetData( InputFile, Data );
while ( not eof( InputFile ) and ( Data <> -1 ) do
   begin
   DoSomething( Data );
   GetData( InputFile, Data )
   end;
```

This "trivial" example is disguised because it contains an error. In the case in which *Data* equals *-1*, the translation

detects the *-1* and exits the loop before executing *DoSomething()*. The original code executes *DoSomething()* before the *-1* is detected. In other words, a programming book trying to show how easy coding is using only structured programming techniques translated its own example incorrectly! The author of that book shouldn't feel too bad, however, because other books make similar mistakes! Even the pros have difficulty achieving *goto*less nirvana.

Here's a faithful translation of the code with no *goto*s:

Pascal Example of Truly Equivalent Code, Rewritten Without gotos

```
repeat
   GetData( InputFile, Data );
   if ( not eof( InputFile )) then
      DoSomething( Data );
until ( Data = -1 or eof( InputFile ) );
```

Even with a correct translation of the code, the debate is still phony because it trivializes the case in which a *goto* is needed. Cases like this are not the ones in which thoughtful programmers choose a *goto* as the preferred form of control.

It would be hard by now to add anything worthwhile to the theoretical debate about *goto*s. One level of discussion isn't usually addressed, however, and that's the situation in which a programmer who is fully aware of *goto*less alternatives chooses to use a *goto* on the basis of its readability and maintainability.

The following sections present cases in which some experienced programmers argue for using *goto*s. The sections provide examples of rewriting the code without *goto*s, and discuss the tradeoffs between the various versions.

Error Processing and gotos

Writing highly interactive code creates additional programming demands. In particular, it demands that you pay a lot of attention to error processing and cleaning up resources when errors occur. Here's an example of code that purges a group of files. It first gets a group of files to purge, then finds each file, opens it, overwrites it, and erases it. It checks for errors at each step:

Pascal Example of Code with gotos that Processes Errors and Cleans up Resources

```
PROCEDURE PurgeFiles( var ErrorState: ERROR_CODE );
{ This routine purges a group of files }
var
   FileIndex:        Integer;
   FileHandle:       FILEHANDLE_T;
   FileList:         FILELIST_T;
   NumFilesToPurge:  Integer;

label
   END_PROC;

begin
   MakePurgeFileList( FileList, NumFilesToPurge );

   ErrorState := Success;
   FileIndex   := 0;
   while ( FileIndex < NumFilesToPurge ) do
      begin
      FileIndex := FileIndex + 1;

      if not FindFile( FileList[ FileIndex ], FileHandle ) then
         begin
         ErrorState := FileFindError;
         goto END_PROC
         end;
```

```
      if not OpenFile( FileHandle ) then
         begin
         ErrorState := FileOpenError;
         goto END_PROC
         end;

      if not OverwriteFile( FileHandle ) then
         begin
         ErrorState := FileOverwriteError;
         goto END_PROC
         end;

      if Erase( FileHandle ) then
         begin
         ErrorState := FileEraseError;
         goto END_PROC
         end

      end; { while }

   END_PROC:

   DeletePurgeFileList( FileList, NumFilesToPurge )

end; { PurgeFiles }
```

This routine is typical of circumstances in which experienced programmers select a *goto*. Other, similar cases occur when a routine needs to allocate and clean up resources such as memory or handles to fonts, windows, brushes, and printers. The alternative to *goto*s in those cases is usually duplicating code to clean up resources. In such cases, a programmer might balance the evil of the *goto* against the maintenance headache of duplicate code and decide that the *goto* is the lesser evil.

You can rewrite the above routine in a couple of ways that avoid *goto*s, and you make tradeoffs in both cases. Here are the possible rewrite strategies:

Rewrite with Nested *if* Statements. To rewrite with nested *if* statements, nest the *if* statements so that each is executed only if the previous test succeeds. This is the standard, textbook, structured-programming approach to eliminating *goto*s. Here's a rewrite of the routine using the standard approach:

Pascal Example of Code that Avoids gotos by using Nested *if*s

```
PROCEDURE PurgeFiles( var ErrorState: ERROR_CODE );

{ This routine purges a group of files }

var
   FileIndex:        Integer;
   FileHandle:       FILEHANDLE_T;
   FileList:         FILELIST_T;
   NumFilesToPurge:  Integer;

begin
   MakePurgeFileList( FileList, NumFilesToPurge );

   ErrorState  := Success;
   FileIndex    := 0;

   while ( FileIndex < NumFilesToPurge and ErrorState = Success ) do
      begin
      FileIndex := FileIndex + 1;

      if FindFile( FileList[ FileIndex ], FileHandle ) then
         begin
```

```
            if OpenFile( FileHandle ) then
               begin
               if OverwriteFile( FileHandle ) then
                  begin
                  if not Erase( FileHandle ) then
                     begin
                     ErrorState := FileEraseError
                     end
                  end
               else { couldn't overwrite file }
                  begin
                  ErrorState := FileOverwriteError
                  end
               end
            else { couldn't open file }
               begin
               ErrorState := FileOpenError
               end
            end
         else { couldn't find file }
            begin
            ErrorState := FileFindError
            end

         end; { while }

      DeletePurgeFileList( FileList, NumFilesToPurge )

   end; { PurgeFiles }
```

For people used to programming without *goto*s, this code might be easier to read than the *goto* version, and if you use it, you won't have to face an inquisition from the *goto* goon squad.

The main disadvantage of this approach is that the nesting level is deep. Very deep. Deeeeeeeep. With nesting like this, to understand the code, you have to keep the whole set of nested *if*s in your mind at once. Moreover, the distance between error-processing code and code that invokes it is too far: the code that sets *ErrorState* to *FileFindError*, for example, is 23 lines from the *if* statement that invokes it.

With the *goto* version, no statement is more than four lines from the condition that invokes it. Moreover, it doesn't require that you keep the whole structure in your mind at once. You can essentially ignore any preceding conditions that were successful and focus on the next operation. For these reasons, in this case, the *goto* version is more readable and more maintainable than the nested-*if* version.

Rewrite with a Status Variable. To rewrite with a status variable (also called a state variable), create a variable that indicates whether the routine is in an error state or not. In this case, the routine already uses the *ErrorState* status variable, so you can use that:

Pascal Example of Code that Avoids gotos by Using a Status Variable

```
PROCEDURE PurgeFiles( var ErrorState: ERROR_CODE );

{ This routine purges a group of files }

var
   FileIndex:        Integer;
   FileHandle:       FILEHANDLE_T;
   FileList:         FILELIST_T;
   NumFilesToPurge:  Integer;

begin
   MakePurgeFileList( FileList, NumFilesToPurge );

   ErrorState := Success;
```

```
      FileIndex  := 0;
      while ( FileIndex < NumFilesToPurge ) and ( ErrorState = Success ) do
         begin
         FileIndex := FileIndex + 1;

         if not FindFile( FileList[ FileIndex ], FileHandle ) then
            begin
            ErrorState := FileFindError
            end;

         if ( ErrorState = Success ) then
            begin
            if not OpenFile( FileHandle ) then
               begin
               ErrorState := FileOpenError
               end
            end;

         if ( ErrorState = Success ) then
            begin
            if not OverwriteFile( FileHandle ) then
               begin
               ErrorState := FileOverwriteError
               end
            end;

         if ( ErrorState = Success ) then
            begin
            if not Erase( FileHandle ) then
               begin
               ErrorState := FileEraseError
               end
            end

         end; { while }

      DeletePurgeFileList( FileList, NumFilesToPurge )

   end; { PurgeFiles }
```

The advantage of the status-variable approach is that it avoids the deeply nested *if-then-else* structures of the first rewrite, so it's easier to understand. It also places the action following the *if-then-else* test closer to the test than the first rewrite and completely avoids *else* clauses.

Understanding the nested-*if* version requires substantial mental gymnastics, but this version is easier to understand because it closely models the way people think about the problem. You find the file. If everything is OK, you open the file. If everything is still OK, you overwrite the file. If everything is still OK, ...

The disadvantage of this approach is that the using status variables isn't as common a practice as it should be. Document it carefully, or some programmers might not understand the general approach. In this example, the use of well-named enumerated types helps significantly.

Comparison of Approaches

Each of the three methods has something to be said for it. The first avoids unnecessary tests and deep nesting but has *goto*s. The second avoids *goto*s but is deeply nested and gives an exaggerated picture of the logical complexity of the routine. The third avoids *goto*s and deep nesting but introduces extra tests.

The last approach is slighty preferable to the first two because it's more readable and models the problem better, but that doesn't make it the best approach in all circumstances. Any of these techniques works well when applied consistently to all the code in a project. Consider all the factors that have been presented, then make a project-wide decision about which method to favor in your programs.

gotos and Sharing Code in an else Clause

One challenging case in which some programmers would use a *goto* is the case in which you have two conditional tests and an *else* clause, and want to execute code in one of the conditions and the other *else* clause. Here's an example of a case that could drive someone to *goto*:

C Example of Sharing Code in an else Clause with a goto

```
if ( StatusOK )
   {
   if ( DataAvail )
      {
      ImportantVar = x;
      goto MID_LOOP;
      }
   }
else
   {
   ImportantVar = GetVal();

   MID_LOOP:

   /* lots of code */
   ...
   }
```

This is a good example because it's logically tortuous-it's nearly impossible to read as it stands and one of the hardest cases to rewrite correctly without a *goto*. If you think you can easily rewrite it without *goto*s, ask someone to review your code! Several expert programmers have rewritten it erroneously.

You can rewrite it in several ways. You can duplicate code, put the common code in a routine and call it from two places, or retest the conditions. The rewrite won't be as fast as the original in most languages, but it will be almost as fast. Unless the code is in a really hot loop, rewrite it without thinking about efficiency.

The best rewrite is to put the */* lots of code *\/* part in its own routine. You can then call the routine in the places you would otherwise have used as the origin or destination of a *goto* and preserve the original structure of the conditional. Here's how it looks:

C Example of Sharing Code in an else Clause by Putting Common Code in a Routine

```
if ( StatusOK )
   {
   if ( DataAvail )
      {
      ImportantVar = x;
      DoLotsOfCode( ImportantVar );
      }
   }
else
   {
   ImportantVar = GetVal();
   DoLotsOfCode( ImportantVar );
   }
```

Normally, writing a new routine (or a macro in C) is the best approach. Sometimes, however, it's not practical to put duplicated code in its own routine. In this case you can work around it by restructuring the conditional so that you keep the code in it rather than in a new routine. Here's how it looks:

C Example of Sharing Code in an else Clause Without a goto

```
if ( (StatusOK && DataAvail) || ! StatusOK )
   {
   if ( StatusOK && DataAvail )
      ImportantVar = x;
   else
      ImportantVar = GetVal();

   /* lots of code */
   ...
   }
```

This is a faithful and mechanical translation of the logic in the *goto* version. It tests *StatusOK* two extra times and *DataAvail* one, but the code is equivalent. If retesting the conditionals bothers you, notice that the value of *StatusOK* doesn't need to be tested twice in the first *if* test. You can also drop the test for *DataAvail* in the second *if* test. Try rewriting it yourself if you want the practice.

Summary of Guidelines for Using gotos

Use of *goto*s is a matter of religion. My dogma is that in modern languages, you can easily replace nine out of ten *goto*s with equivalent structured constructs. In these simple cases, you should replace *goto*s out of habit. In the hard cases, you can still exorcise the *goto* in nine out of ten cases. In these cases, you can break the code into smaller routines; use nested *if*s; test and retest a status variable; or restructure a conditional. Eliminating the *goto* is harder in these cases, but it's good mental exercise, and the techniques discussed in this section give you the tools to do it.

In the remaining one case out of 100 in which a *goto* is a legitimate solution to the problem, document it clearly and use it. If you have your rain boots on, it's not worth walking around the block to avoid a mud puddle. But keep your mind open to *goto*less approaches suggested by other programmers. They might see something that you don't.

Here's a summary of guidelines for using *goto*s:

- Use *goto*s to emulate structured control constructs in languages that don't support them directly. When you emulate structured constructs, emulate them exactly. Don't abuse the extra flexibility the *goto* gives you.
- Don't use *goto*s when an equivalent structured construct is available.
- Measure the performance of any *goto* used to improve efficiency. In most cases that use *goto*s, you can recode them without *goto*s with improved readability and no loss in efficiency. If your case is the exception, document the efficiency improvement so that *goto*less evangelists won't remove it when they see it.
- Limit yourself to one *goto* label per routine, unless you're emulating structured constructs.
- Limit yourself to *goto*s that go forward, not backward, unless you're emulating structured constructs.
- Make sure all *goto* labels are used. They might be a symptom of missing code, namely the code that goes to them. If they're not used, delete them.
- Make sure a *goto* doesn't create unreachable code.
- If you're a manager, adopt the perspective that a battle over a single *goto* isn't worth the loss of the war. If the programmer is aware of the alternatives and is willing to argue, the *goto* is probably OK.

Further Reading

These articles contain the whole *goto* debate. It erupts from time to time in most workplaces, textbooks, and magazines, but you won't hear anything that wasn't fully explored 20 years ago.

Dijkstra, E. "GOTO Statement Considered Harmful," *Communications of the ACM,* v. 11, no. 3, March 1968, pp. 147-8. This is the classic paper in which Dijkstra put the match to the paper and ignited a controversy that shows no signs of abating.

Wulf, W. A. "A Case Against the GOTO," *Proceedings of the 25th National ACM Conference*, August 1972, pp. 791-97. This paper was another argument against the indiscriminate use of *goto*s. Wulf argued that if programming languages provided adequate control structures, *goto*s would become largely unnecessary. Since the paper was written in 1972, languages such as Pascal, C, and Ada have proven him correct.

Knuth, Donald. "Structured Programming with go to Statements," 1974, in (Yourdon 1979), pp. 259-321. This long paper isn't entirely about *goto*s, but it includes a horde of examples of code that's made more efficient by eliminating *goto*s and another horde of examples of code that's made more efficient by adding them.

Rubin, Frank. "'GOTO Considered Harmful' Considered Harmful" (letter to the editor), *Communications of the ACM*, vol. 30, no. 3 (March 1987), pp. 195-6. In this rather hot-headed letter to the editor, Rubin asserts that *goto*less programming has cost businesses "hundreds of millions of dollars." He then offers a code fragment that uses a *goto* and argues that it's superior to *goto*less alternatives.

The response that the letter generated was more interesting than the letter itself. For five months, the *CACM* published letters that offered different versions of Rubin's original 7-line program. The letters were evenly divided between those defending *goto*s and those castigating them. Readers suggested roughly 17 different rewrites, and the rewritten code fully covered the spectrum of approaches to avoiding *goto*s. The editor of *Communications of the ACM* noted that the letter had generated more response by far than any other issue ever considered in pages of the *CACM*.

For the follow-up letters, see:

Communications of the ACM, vol. 30, no. 5 (May 1987), pp. 351-355.
Communications of the ACM, vol. 30, no. 6 (June 1987), pp. 475-478.
Communications of the ACM, vol. 30, no. 7 (July 1987), pp. 632-4.
Communications of the ACM, vol. 30, no. 8 (August 1987), pp. 659-62.
Communications of the ACM, vol. 30, no. 12 (December 1987), pp. 997, 1085.

Email me at stevemcc@construx.com.