An Argument for the Use of goto Statements

Carl Youngblood
Embedded Systems Engineer
The Boeing Company

Following E. W. Dijkstra's publication of "GOTO Statement Considered Harmful," there has been an ongoing debate among computer scientists regarding the merits of the goto statement. While many argue that modern programming languages eliminate the need for the goto, Donald Knuth, Steve McConnell, and other experts have demonstrated that there are many valid scenarios in which gotos make code more efficient, cleaner, and more concise. Most notable among these is the use of gotos to handle error processing.

According to Steve McConnell, author of *CODE COMPLETE* , "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." There are basically three approaches to error handling and cleanup. The following fictional code samples illustrate the advantages and disadvantages of each approach.

The first approach uses multiple function return paths to halt execution after an error has been detected and resources have been cleaned up. While this method is fairly straightforward, it has some drawbacks, namely that it requires a lot of code duplication for resource cleanup. Code duplication is one of the cardinal sins of software engineering, because every time a section of code is duplicated it must be managed separately. Changes or bug fixes in one section are often not fixed in duplicate sections, and overall code complexity and bugginess skyrocket. Beyond this drawback, having multiple return paths is in itself considered by many to be undesirable, since it increases code complexity, especially during debugging. One additional minor drawback of this method is that the overall length of code is increased.

```
//////////////////////////////////////////////////////////
//
// METHOD ONE - MULTIPLE RETURN PATHS
//
//    Pros: No goto statements
//    Cons: Code bloat, Increased complexity,
//          code duplication, some argue that multiple
//          return paths alone are undesirable
//
//////////////////////////////////////////////////////////

status smart_token::login(string pin)
{
   status returnval(status::FAILED);
   CK_SESSION_HANDLE session = NULL_PTR;
   CK_KEY_HANDLE key_handle = NULL_PTR;
   CK_RV status;
```

```
   CK_ULONG count;

   // open a session with the smart token
   status = C_OpenSession(m_SlotID, &session);
   if (status != CR_OK)
   {
      returnval.setMessage("Couldn't open a session with the token");
      return returnval;
   }

   // login to the token
   status = C_Login(session, pin.c_str(), pin.length());
   if (status != CR_OK)
   {
      returnval.setMessage("Couldn't log into the token");
      return returnval;
   }

   // initialize key search for all RSA keys on token
   status = C_FindObjectsInit(session,  CKO_RSA_KEY);
   if (status != CR_OK)
   {
      C_Logout(session);
      returnval.setMessage("Couldn't find any RSA keys on the token");
      return returnval;
   }

   // determine how much space is need to retrieve all RSA keys
   status = C_FindObjects(session, NULL_PTR, &count);
   if (status != CKR_OK || count < 1)
   {
      C_Logout(session);
      returnval.setMessage("Couldn't find any RSA keys on the token");
      return returnval;
   }

   // allocate memory to hold all keys
   key_handle = new CK_KEY[count];
   if (!key_handle)
   {
      C_Logout(session);
      returnval.setMessage("A memory allocation error occurred.");
      return returnval;
   }

   // retrieve keys
   status = C_FindObjects(session, key_handle, &count);
   if (status != CKR_OK)
   {
      delete [] key_handle;
      C_Logout(session);
      returnval.setMessage("Couldn't retrieve RSA keys from the token");
      return returnval;
   }
```

```
   // much more code follows, and each command has to duplicate a lot
   // of code to ensure that proper cleanup occurs

   delete [] key_handle;
   C_Logout(session);
   returnval = status::SUCCESS;
   returnval.setMessage("SUCCESS");
   return returnval;
}
```

The second approach uses deep nesting to ensure that a given section of code is only reached upon successful completion of all preceeding sections, and that resource cleanup code is not duplicated. While this method eliminates code duplication, the deep nesting quickly becomes difficult to follow and overall code complexity increases. One advantage of this approach is that there is only one function return path.

```
////////////////////////////////////////////////////////////
//
// METHOD TWO - DEEP NESTING
//
//    Pros: Less code duplication, no goto statements
//    Cons: Still some code bloat, Greatly increased complexity
//
////////////////////////////////////////////////////////////

status smart_token::login(string pin)
{
   status returnval(status::FAILED);
   CK_SESSION_HANDLE session = NULL_PTR;
   CK_KEY_HANDLE key_handle = NULL_PTR;
   CK_RV status;
   CK_ULONG count;

   // open a session with the smart token
   returnval.setMessage("Couldn't open a session with the token");
   status = C_OpenSession(m_SlotID, &session);
   if (status == CR_OK)
   {
      // login to the token
      returnval.setMessage("Couldn't log into the token");
      status = C_Login(session, pin.c_str(), pin.length());
      if (status == CR_OK)
      {
         // initialize key search for all RSA keys on token
         returnval.setMessage("Couldn't find any RSA keys on the token");
         status = C_FindObjectsInit(session,    CKO_RSA_KEY);
         if (status == CR_OK)
         {
            // determine how much space is need to retrieve all RSA keys
            returnval.setMessage("Couldn't find any RSA keys on the token");
```

```
            status = C_FindObjects(session, NULL_PTR, &count);
            if (status == CKR_OK && count >= 1)
            {
               // allocate memory to hold all keys
               returnval.setMessage("A memory allocation error occurred.");
               key_handle = new CK_KEY[count];
               if (key_handle)
               {
                  // retrieve keys
                  status = C_FindObjects(session, key_handle, &count);
                  if (status == CKR_OK)
                  {
                     // much more code follows,
                     // and each command requires deeper nesting

                     returnval = status::SUCCESS;
                     returnval.setMessage("SUCCESS");
                  }
                  delete [] key_handle;
               }
            }
         }
         C_Logout(session);
      }
   }

   return returnval;
}
```

The third approach is one of the few instances in which the use of goto statements actually increases code clarity and brevity while reducing complexity. Rather than handling each error as a special case, a failed function call causes the execution to jump immediately to a single cleanup section that handles all necessary resource cleanup and returns from the function. This method keeps code clean and compact while reducing complexity. There is no code duplication for resource cleanup and there is only one function return path.

```
////////////////////////////////////////////////////////
//
// METHOD THREE - SINGLE CLEANUP SECTION AT FUNCTION'S END
//
//    Pros: No code duplication, single return path,
//          greatly simplified code
//    Cons: Requires goto or similar construct
//
////////////////////////////////////////////////////////

status smart_token::login(string pin)
{
   status returnval(status::FAILED);
   CK_SESSION_HANDLE session = NULL_PTR;
   CK_KEY_HANDLE key_handle = NULL_PTR;
```

```
   CK_RV status;
   CK_ULONG count;
   bool logged_in = false;

   // open a session with the smart token
   returnval.setMessage("Couldn't open a session with the token");
   status = C_OpenSession(m_SlotID, &session);
   if (status != CR_OK) goto cleanup;

   // login to the token
   returnval.setMessage("Couldn't log into the token");
   status = C_Login(session, pin.c_str(), pin.length());
   if (status != CR_OK) goto cleanup;
   logged_in = true;

   // initialize key search for all RSA keys on token
   returnval.setMessage("Couldn't find any RSA keys on the token");
   status = C_FindObjectsInit(session,  CKO_RSA_KEY);
   if (status != CKR_OK) goto cleanup;

   // determine how much space is need to retrieve all RSA keys
   returnval.setMessage("Couldn't find any RSA keys on the token");
   status = C_FindObjects(session, NULL_PTR, &count);
   if (status != CKR_OK || count < 1) goto cleanup;

   // allocate memory to hold all keys
   returnval.setMessage("A memory allocation error occurred.");
   key_handle = new CK_KEY[count];
   if (!key_handle) goto cleanup;

   // retrieve keys
   returnval.setMessage("Couldn't retrieve RSA keys from the token");
   status = C_FindObjects(session, key_handle, &count);
   if (status != CKR_OK) goto cleanup;

   // Much more code follows, each command having a single
   // short-circuit point that jumps to a single cleanup
   // section.  There is only one function return path.

   returnval = status::SUCCESS;
   returnval.setMessage("SUCCESS");

cleanup:
   if (key_handle) delete [] key_handle;
   if (logged_in) C_Logout(session);
   return returnval;
}
```

Some may argue that the likelihood of goto statements being used improperly far outweighs the slight benefit that they offer in the few instances where their use is preferred. This may have been true for languages in which modern structured programming constructs were not available, but in C, C++ and other high level languages, there is always a better alternative

that is already favored by programmers as the standard approach. In these languages, therefore, the use of gotos is by convention already confined strictly to those few instances where they are beneficial.