WG 14 N1298 March 14 2008

Cleanup and Try Finally for C

Nick Stoughton

This paper is a work-in-progress, and is incomplete. It is submitted as a statement of direction and to obtain early feedback.

Introduction

During the 2007 Kona meeting, WG 14 discussed the possibility of adding both the gcc based `cleanup` attribute, and the MSVC based `try {} finally {}` construct. While both these mechanisms provide a method for some simple destructors to be created, and both deal with actions associated with the end of a scope block, they differ somewhat in what can be achieved. This paper describes both, and proposes detailed semantics for including these concepts into the revision. Since the two constructs do offer different possibilities, and both are implemented in popular, shipping products, both are considered necessary for the revised standard. Since no real wording yet exists for attributes in general, this proposal follows the form of N1273.

The `cleanup` Attribute

The cleanup attribute is associated with an object with automatic or allocated storage duration, and contains a pointer to a function that is passed a pointer to the object in question. The function is called when the object goes out of scope.

It can be associated with any block scope object, except static duration objects or parameters.

The cleanup attribute can be thought of as a poor-man's destructor. When the cleanup function exits, its return value is ignored. Hence a cleanup function can be thought of as being of type:

```
void cleanup_func(void *);
```

This construct has existed in gcc for some time, but is not very widely used; principle applications that make use of the feature are gcc itself, glibc, and a small handful of other open source applications. However, this is at least partially because, in standard C, it is possible to write portable code which manually covers every scope exit point and explicitly deals with the cleanup at that point. Many programmers end up missing some scope-exit point and leaking resources as a result. Making the cleanup handler explicitly part of the standard would be a major benefit.

Gcc does not define the behavior when a variable goes out of scope because of a longjmp. Experimental observation suggests that the cleanup handler is not called in this case.

The `try {} finally {}` Construct

Microsoft extended the **try{} catch{}** metaphor from C++ into C. At the same time, they extended it to include a **finally {}** block also. In C++, the **catch{}** block is used to handle exceptions, which starts unwinding the stack until a catch block in a calling function that handles this exception is reached. The **finally{}** block simply extends the metaphor to include code to run whenever and however the try block exits (including in the MSVC case, via the **catch{}** block). This paper does not seek to include the exception handling mechanisms from C++ and MSVC, but concentrates on the **try{} finally{}** part.

MSVC avoids namespace pollution by using the keywords **__try{} __finally{}**.

Consider the following code fragment:

```
FILE *f = NULL;
struct foo *m = NULL;
extern _Bool chkFlg;
__try {
     do {
          if(!chkFlg)
               return;   // will return via the finally block
          if ((f = fopen(filename, "r")) == NULL)
               break;    // will jump to the finally block
          if ((m = calloc(1, sizeof(struct foo))) == NULL)
               break;
          if (fread(m, sizeof(struct foo), 1, f) != 1)
               break;    // will jump to the finally block
          // do something with m
          printf("Read foo structure ok\n");
     } while(0);
     // now fall through to the finally block
}
__finally {
     if (f)
          fclose(f);
     if (m)
          free(m);
}
```

This program will open a file and read, into dynamically allocated memory, a data structure. When finished, it will cleanup after itself, closing the file (if it was opened), and freeing the memory (if it was allocated). The program is considerably simpler than the equivalent written in pure C99.

Microsoft have also added a **__leave** keyword, to jump out of the try block (avoiding the do{}while(0) construct used above). Although a **goto** statement can be used to accomplish the same result, a **goto** statement causes stack unwinding. The **__leave** statement is more efficient because it does not involve stack unwinding.

Exiting a **try-finally** statement using the **longjmp** run-time function is considered abnormal termination. It is illegal to jump into a **__try** statement, but legal to jump out of one. All **__finally** statements that are active between the point of departure (normal termination of the **__try** block) and the destination must be run. This is called a local unwind.

If a **try** block is prematurely terminated for any reason, including a jump out of the block, the system executes the associated **finally** block as a part of the process of unwinding the stack.

The **finally** block is not called if a process is killed in the middle of executing a **try-finally** statement.

The Way Forward

Both mechanisms have much in their favor, particularly with respect to making the programmer's job easier in avoiding common problems associated with resource leakage.

The try-finally approach is substantially more complex, and involves interactions with exceptions (via the raise() function) as well as complexities with respect to gotos.

The cleanup attribute is simpler, but suffers from not covering the non-local goto exit (this might be considered a bug in the implementation, but consider the code that does a non-local goto from a function called from inside a scope with one or more objects with cleanup handlers associated).

The whole issue of stack-unwinding when a low-level function causes a higher level function to need to divert into special handler code is poorly handled by both constructs, at least grammatically.

Detailed Semantics

stdc_cleanup

Syntax: To be defined. `stdc_cleanup(fptr).`

Constraints: `fptr` shall be a pointer to a function returning `void`, that takes a single parameter of type compatible with pointer to the associated object.

Semantics: The function shall be called with a pointer to the object when the object goes out of scope. It is unspecified if the function shall be called if the scope is exited via a nonlocal jump. The attribute can only be applied to auto function scope variables; it may not be applied to parameters or variables with static storage duration. The function must take one parameter, a pointer to a type compatible with the variable. The return value of the function (if any) is ignored.

Notes: The function should take a pointer to a compatible type.