Doc No: J16/99-0039, WG21/N1215
Date: October 22, 1999
Reply to: Stephen D. Clamage,
stephen.clamage@sun.com

Exit Processing Order

This paper addresses Library Working Group issue 3: `atexit` registration during `atexit()` call is not described.

Example 1: (C and C++)

```
#include <stdlib.h>
void f1() { }
void f2() { atexit(f1); }

int main()
{
     atexit(f2); // the only use of f2
     return 0;
}
```

At program exit, `f2` gets called due to its registration in `main`. Running `f2` causes `f1` to be newly registered during the exit processing. Is this a valid program? If so, what are its semantics?

Neither the C89 standard nor the C++ standard says directly whether you can register a function with `atexit` during exit processing.

Both standards say that functions are run in reverse order of their registration. Since `f1` is registered last, it ought to be run first, but by the time it is registered, it is too late to be first. If the program is valid, the standards are self-contradictory about its semantics.

Example 2: (C++ only)

```
void F() { static T t; ... } // type T has a destructor

int main()
{
    atexit(F); // the only use of F
}
```

Function `F` registered with `atexit` has a local static variable `t`, and `F` is called for the first time during exit processing. A local static object is initialized the first time control flow passes through its definition, and all static objects are destroyed during exit processing. Is the code valid? If so, what are its semantics?

Section 18.3 "Start and termination" says that if a function `F` is registered with `atexit` before a static object `t` is initialized, `F` will not be called until after `t`'s destructor completes.

In example 2, function `F` is registered with `atexit` before its local static object `t` could possibly be initialized. On that basis, `F` must not be called by exit processing until after `t`'s destructor completes. But the destructor cannot be run until after `F` is called, since otherwise the object could not be constructed in the first place. If the program is valid, the standard is self-contradictory about its semantics.

The new C9X standard requires stack-like behavior in its section 7.20.4.2:

> First, all functions registered by the `atexit` function are called, in the reverse order of their registration, except that a function is called after any previously registered functions that had already been called at the time it was registered.

The corresponding sentence in C++ 18.3/8 should be modified to say the same thing.

Example 1 is thus fully defined: Since `f2` had already been called at the time `f1` is registered, `f1` is run after `f2`, but before any previously registered functions are run.

Example 2 is more complicated. Suppose `F` is first called during exit processing, and is perhaps called more than once. In particular, `F` might be called from the destructor of some static object.

Possible semantic choices:

1. Retain the "destruction in the reverse order of construction" property for non-local static objects. A local static object is destroyed at the same time it would be if a function calling its destructor were registered with `atexit` at the completion of its constructor.
2. If a function called during exit processing uses a local static object, the results are undefined.
3. Remove the requirement to interleave functions registered with `atexit` and static destructors. After all registered functions have been run, all static objects are destroyed. If during destruction of a static object, one or more functions are registered with `atexit`, they are run in reverse registration order after completion of the destructor.

Variations on these semantic choices:

A. Relax the requirements on local static objects to apply only to local static objects having non-trivial destructors (12.4).
B. If the flow of control passes through the definition of a local static object after the object has been destroyed as a result of exit processing, the object is reinitialized.

Consequences of the choices:

1. Consistent handling of local static objects, easy to explain, and with no extra implementation difficulties. It is possible to get undefined behavior if a function is called again after its local static object is destroyed.
2. This is the status quo, since standard is self-contradictory about local static objects. It would be nice to provide some guarantees, however, so programmers have some control over object creation and destruction.
3. If we had recognized this problem before releasing the standard, #3 might be a good choice. But it changes the semantics of existing programs, and some C++ vendors have already implemented the current rule in their products.

A. Superficially attractive, it makes special cases, and makes different guarantees about lifetime of storage depending of the definition of destructors.

B. Might prevent some program crashes, but static objects typically record program state changes. If the object is destroyed and re-created, the program could run but produce nonsensical results.

Recommendation:

Option 1 above, without variations. Change section 18.3/8
from:

> First, objects with static storage duration are destroyed and functions registered by calling `atexit` are called. Objects with static storage duration are destroyed in the reverse order of the completion of their constructor. (Automatic objects are not destroyed as a result of calling `exit()`.) Functions registered with `atexit` are called in the reverse order of their registration. A function registered with `atexit` before an object `obj1` of static storage duration is initialized will not be called until `obj1`'s destruction has completed. A function registered with `atexit` after an object `obj2` of static storage duration is initialized will be called before `obj2`'s destruction starts.

to:

> First, objects with static storage duration are destroyed and functions registered by calling `atexit` are called. Non-local objects with static storage duration are destroyed in the reverse order of the completion of their constructor. (Automatic objects are not destroyed as a result of calling `exit()`.) Functions registered with `atexit` are called in the reverse order of their registration, except that a function is called after any previously registered functions that had already been called at the time it was registered. A function registered with `atexit` before a non-local object `obj1` of static storage duration is initialized will not be called until `obj1`'s destruction has completed. A function registered with `atexit` after a non-local object `obj2` of static storage duration is initialized will be called before `obj2`'s destruction starts. A local static object `obj3` is destroyed at the same time it would be if a function calling the `obj3` destructor were registered with `atexit` at the completion of the `obj3` constructor.