# Construction and Destruction of Statics in the WATCOM C++ Compiler

**James W. Welch**

**WATCOM International Corp.**

**jww@watcom.on.ca**

**Document No.: X3J16/96-0053 WG21/N0871**

# Overview

Once upon a time, I went to Tokyo where I met with a most˝interesting group of people. While discussing earth-shattering topics, such as the construction and˝destruction of static variables, I put my foot in my mouth and volunteered to write a paper giving an overview of˝how the WATCOM C++ compiler accomplishes this remarkable feat.

This paper accomplishes that commitment. It proposes˝nothing. Its only purpose is exposition. I don't consider anything that we do to be particularly novel. The˝other compilers with which we compete in the marketplace also manage to accomplish the same tasks in a˝similar manner.

# Compiler Platforms

The compiler produces code for INTEL and Digital Alpha˝platforms. The code and associated run-time libraries run on the Microsoft (DOS, Windows, Windows 95,˝Windows NT) , IBM (OS/2) , and Quantum (QNX) operating systems. Some of these platforms involve˝multi-threaded capabilities and dynamically-linked libraries (DLLs).

For discussion purposes, the term *image* will be used to refer to either a program (contains the˝mainline) or a dynamic library (contains routines and data which are˝accessed by other images). The term *global static* means a static variable that is not contained within a˝function.

The main discussion will ignore the complications that arise˝in multi-threaded environments. These will be discussed in a later section.

Global statics are constructed when an image is loaded. For˝a program, this is when the program starts, before control has been passed to the mainline. DLLs can be˝created to have a unique data area for each program that accesses them, or to have one data area that˝remains usable until no more programs access that DLL. Global statics are initialized in DLLs whenever a˝new copy of the data is acquired. Global statics are destructed when an image is unloaded.

# Initialization of Statics

During the compilation process, if the contents of a global˝static is determined to be constant values, the static is initialized at compilation time to have those˝values. Otherwise, code is emitted to perform the initialization. This code is placed within a˝compiler-generated initialization function on a per-module basis. For each global static which requires destruction, an˝entry

```
{ address of variable, address of destructor }
```

is pushed upon a stack (stacks exist on a per-image basis)˝once the variable has been constructed (or at the place such code would exist if compile-time initialization had˝not been used). This stack will be called the *static-destruction stack.*

For each module in which a compiler-generated initialization˝function exists, an entry

```
{ priority, address of initialization routine }
```

is generated into the object file. These entries are˝collected together by the linker and placed in a contiguous data area. The initialization of global statics˝is accomplished by calling all the initialization routines in that area, in increasing order of priority.

The system uses 256 different priority levels with˝conventions for reserved initialization ( 0-31 ), library initialization (32-63), and program initialization˝(64-255). Priorities can be set using a `#pragma` statement, giving users a crude way to control order of˝initialization.

Static variables within functions are initialized the first˝time flow of control passes through the declaration of the data item. For such items, two flags are˝defined. The first indicates if that item has been initialized while the second indicates that the item is˝being initialized. Code is emitted to skip initialization when the first flag is set. Otherwise, the˝second flag is set, the initialization code is executed, a static-destruction entry is pushed on the˝static-destruction stack as with global statics, the first flag is set, and the second flag is reset. The compiler uses˝compile-time initialization if possible and pushes a destruction entry only if required. If no code is˝generated, then the flags are unnecessary and are eliminated.

When the static occurs within an inline function, the data˝item and associated flags are stored using a “common-data” convention so that multiple copies from˝separate compilations will end up as one copy after linking.

# Destruction of Statics

Destruction of statics is performed on a per-image basis. It˝is accomplished by a loop which terminates when the static-destruction stack for the image is empty.˝ The body of the loop pops the top entry and calls the appropriate destructor for the data item referenced in˝the popped entry.

During the destruction process, new items may be pushed on˝the stack when a function containing a destructable static item is called for the first time.

There has been no attempt to prevent or detect the situation˝which arises when a destructed item is again initialized. This is considered to be a “user-beware”˝area. Things seem to work when the initialization does not rely on the memory contents of the item being˝re-initialized. WATCOM provides no guarantees that such behaviour would be supported in the future.

# Multi-thread Considerations

The compiler requires a command-line switch to indicate that code is being compiled for a multi-threaded situation. This is because the run-time data structure must be protected against an interleaved update from more than one thread of execution. Consequently, semaphores must be used to protect the pushing and popping of static-destruction stack for an image. Similarly, the initialization of a static within a function must be protected.

A single thread is used to execute the construction of global statics and the destruction of statics. Since the construction or destruction of items could cause new execution threads to be created, the protection mechanism is not relaxed when these functions are active

The compiler does not provide any automatic protection against interleaved update of user-defined data items. It is the programmer's responsibility to ensure that such protection is in place when required.

# Conclusions

The success enjoyed by the WATCOM C++ compiler in the marketplace is one measure that the engineered schemes outline above are practicable. As a developer, I have received little feedback in regard to this area and so have concluded that either what is done is sufficient or that the development community doesn't care.

The lack of a standard method to specify order of initialization does offend me personally as a computer scientist. However, the lack of user feedback must mean that users are not too concerned about this area. There are work-arounds (using functions) which have been discussed by other committee members.

Statics are destructed in reverse order of construction on a per-image, not a per-program basis. This is the only practicable method known to the developers. DLLs are considered to be extensions to the language. Within an image, statics are destructed in reverse order of the time of their construction, without consideration to individual threads of execution which might have constructed them. Again, multi-threaded capabilities are viewed as extensions.

The `atexit` changes voted into the language in Tokyo have not yet been implemented. It is anticipated that each exit routine will be treated as if it was a destructor. An entry will be pushed on the static-destruction stack each time the function is invoked.