Ns-3 Application Note

Why Don't Those #@(%!~& Libraries Work?

Craig Dowell

craigdo@ee.washington.edu

Background

Consider the following program

```
  int
main(int argc, char **argv)
{
  myprintf("Hello World\n");
}
```

When you compile this program, the compiler determines that there is no function called myprintf in the compilation unit (the source file) and marks the function as an *unresolved reference*.

When the executable is linked, the linker looks through its list of libraries looking for an implementation of the function myprintf. The linker handles providing an implementation in one of two ways depending on the kind of file providing the implementation.

- If the implementation is found in a static library (e.g., libcore.a), the linker copies the implementation of myprintf into the executable. The linker must then recursively search for further unresolved references in the myprintf function and resolve them by (possibly) looking in other libraries. This continues until the executable contains all of the code required to execute myprintf (there are no unresolved references).

- If the implementation of myprintf is found in a dynamic library (e.g., libcore.so), the linker adds a note to itself that the unresolved reference can be resolved by loading the dynamic library. The linker recursively searches for further unresolved references and continues until all references are resolved.

What happens when you run the resulting program depends on the kind of library to which you linked.

- If you linked to static libraries, your program is self-contained. It will be loaded into memory and executed, starting at a system-dependent entry point. The code at this entry point will perform system-dependent initialization and then eventually call your program's entry point, i.e., main().

- If you linked to dynamic libraries, the system will load your program, but will also need to load any required shared libraries into memory and resolve all of the references that were left marked as unresolved by the linker. This must be done before your program is executed, i.e., before main() is called.

C++ Adds a New Twist

You may have noticed that the actions taken by the linker and by the system during initialization are *not* in general language-dependent. The C++ language adds a new twist to the initialization game with its use of *global constructors* (also called static constructors). An example of this is shown below.

```
class Object
{
public:
  static const InterfaceId iid;
};

…

const InterfaceId Object::iid = MakeObjectInterfaceId();
```

Here the `InterfaceId` named `Object::iid` is initialized by a function call to `MakeObjectInterfaceId()`.[1] This function call *must* happen before the main program which references the library is executed.[2]

The unfortunate thing here is that the C++ language cannot define how this happens – it's the language, not the linker or the run-time system after all. This creates serious problems in trying to compile C++ programs with static or global constructors using static linkage.

Problems with Static Libraries and Global or Static Constructors

The first issue to consider relates to how static linkers work in general. Often "bugs" relating to static linkage of C++ programs reflect absolutely *correct behavior* of the linker.

Why is My Global or Static Constructor not Called?

Recall that a static linker's purpose is to look through a compilation unit for symbols marked as unresolved and then copy code from a library to define the symbol.

Consider what happens if your program does not directly reference any symbol in a *compilation unit* contained in a given library.[3] The linker never sees an unresolved reference to code in that object file and so it assumes that no code from that particular

---

[1] Note that if you have the color version of this document, your attention is called to the function by the red color. If the figure contains characters you must type, they will be shown in blue.

[2] If you are on a Linux system, your library name will be something like libcore.a or libcore.so; if you are on a Cygwin system, your library name will be something like libcore or core.dll.

[3] A compilation unit is a particular C++ file. For example, src/node/drop-tail-queue.cc is a compilation unit.

compilation unit is used, and therefore will ignore any global or static constructors in that file as well.

In order to ensure that the global or static constructors of a compilation unit are called, you must directly reference some symbol in the compilation unit object file defining the global constructor. Note that this is a reference to an *object file* used to build the library, not a symbol in the library file. This means that you must have knowledge of the files that were used to build the library to make this work. For example, consider a situation where you have a static library called `libnode.a`; and that library is made from a number of object files. Imagine that there is a static constructor defined in a file called `drop-tail-queue.cc`. The code in `drop-tail-queue.cc` may look something like,

```
const ClassId DropTailQueue::cid =
  MakeClassId<DropTailQueue> ("DropTailQueue", Queue::iid);
```

In order to get this constructor to execute under static linkage, you must reference another function in the corresponding compilation unit, i.e., `drop-tail-queue.o`. It will not do to reference a symbol in another execution unit of the library. If you want all of the global or static constructors to be called in the library, you must reference something in each of the *compilation units* that have global constructors defined within. This is not a reasonable thing to do.

There is a way to get around this problem, but it essentially turns your static library into a dynamic library.

How to Get that Global Constructor Called

The first thing to remember is that global constructors are a C++ language concept. Your Linux or Cygwim linker program `ld` isn't necessarily going to understand that it has got to do something special to get global constructors initialized properly. This can lead to all kinds of subtle errors, including our fundamental problem of not calling global or static constructors at all. A better approach is to make sure that you use the compiler to drive the link process. In this approach the compiler will call the linker and ask it to do (more or less) the right thing.

A good source of information about how the linker tries to deal with C++ can be found at:

http://www.gnu.org/software/binutils/manual/ld-2.9.1/html_mono/ld.html

Here are a few things to note well. First, here is documentation on several important `ld` command line options:

--relocateable

> Generate relocatable output--i.e., generate an output file that can in turn serve as input to ld. This is often called *partial linking*. As a side effect, in environments that support standard Unix magic numbers, this option also sets the output file's magic number to OMAGIC. If this option is not specified, an absolute file is produced. When linking C++ programs, this option *will not* resolve references to constructors; to do that, use `-Ur'. This option does the same thing as `-i'.
>
> **-Ur**
>
> For anything other than C++ programs, this option is equivalent to `-r': it generates relocatable output--i.e., an output file that can in turn serve as input to ld. When linking C++ programs, `-Ur' *does* resolve references to constructors, unlike `-r'. It does not work to use `-Ur' on files that were themselves linked with `-Ur'; once the constructor table has been built, it cannot be added to. Use `-Ur' only for the last partial link, and `-r' for the others.
>
> **`--whole-archive`**
>
> For each archive mentioned on the command line after the `--whole-archive` option, include every object file in the archive in the link, rather than searching the archive for the required object files. This is normally used to turn an archive file into a shared library, forcing every object to be included in the resulting shared library. This option may be used more than once.

And a final note on how your global constructors will actually get called:

> When linking using the `a.out` object file format, the linker uses an unusual set construct to support C++ global constructors and destructors. When linking object file formats which do not support arbitrary sections, such as `ECOFF` and `XCOFF`, the linker will automatically recognize C++ global constructors and destructors by name. For these object file formats, the `CONSTRUCTORS` command tells the linker where this information should be placed. The `CONSTRUCTORS` command is ignored for other object file formats. The symbol `__CTOR_LIST__` marks the start of the global constructors, and the symbol `__DTOR_LIST` marks the end. The first word in the list is the number of entries, followed by the address of each constructor or destructor, followed by a zero word. The compiler must arrange to actually run the code. For these object file formats GNU C++ calls constructors from a subroutine `__main`; a call to `__main` is automatically inserted into the startup code for `main`. GNU C++ runs destructors either by using `atexit`, or directly from the function `exit`. For object file formats such as `COFF` or `ELF` which support multiple sections, GNU C++ will normally arrange to put the addresses of global constructors and destructors into the `.ctors` and `.dtors` sections.

The interesting piece of implicit information here is that the mechanism whereby the global constructors get executed is run-time system dependent. In order to use the –whole-archive mechanism, you'll have to convince the build system to do all of the

linkage correctly as defined by the ld document. In my opinion, it's a bad idea, so I won't encourage it by providing a recipe here.

If you want to trace through initialization using the Cygwin runtime, you will need to set a breakpoint on a function called `__main ()` -- that's two underscores before the word main. If you are on a Linux GNU-based runtime, you will need to set a breakpoint on a function called `_init ()` – that's one underscore before the word init.

What is the Global Constructor Call Order?

If you've gone through the pain and gotten your global and static constructors called, the next issue is that the order in which the global constructors are called is not defined by the C++ standard except *within a compilation unit*.

Unfortunately, this results in what is commonly known as, "The Static Initialization Order Fiasco." The initialization order is defined by the run-time system and not the language. GNU systems will initialize global constructors in link order, so there is a way around the issue, but it is certainly not portable and uses lots of implicit knowledge about the structure of the program.

If you must use global or static constructors, make absolutely sure that they are not interdependent, as the initialization order can be broken by changing the order of libraries in the build system.

Just Say No

Statically linking a program that uses global or static constructors and destructors in C++ can be a confusing and sometimes tricky proposition. The simplest answer is a common recommendation: just don't do it. If you find you must use global constructors, then avoid statically linked programs and dependencies within the constructors. If you absolutely must use global constructors and static linkage, then prepare yourself for a bumpy ride. It is (theoretically) possible to use global constructors in C++ with static linkage, but it is a can of worms we would prefer not to open. A solution that allows using static linkage quickly degenerates into what is essentially a non-portable shared library anyway.

Problems with Shared Libraries and Global Constructors

You can run into problems with are constructors for global or static constructors in shared libraries as well. As mentioned above, global or static constructors must be executed before any regular function in a shared library is entered. Unfortunately, some systems don't do this properly – meaning objects may be left initialized.

In Ns-3, we work with two main systems: GNU on Linux and GNU on Cygwin. The basics for how shared libraries work and how to debug them on these two systems follow,

ELF / GNU / Linux

In the case of ELF-based shared libraries (GNU on Linux is a good example) built using the GNU C++ compiler, there is a good chance that your program will be initialized properly. You only have to deal with "The Static Initialization Order Fiasco" which can be worked around using the "Initialize on First Use" idiom in some cases.

When a dynamic library is constructed using ELF, the GNU compiler provides two auxiliary files crtbegin.o and crtend.o that are used to ensure the global constructors and destructors are called. These small programs use the ELF sections `.ctors` and `.dtors` which contain arrays of constructors and destructors respectively.

The constructors are called via a function named `__do_global_ctors_aux` (two underscores before the function) and the destructors are called via a function named `__do_global_dtors_aux`. If you ever need to debug global construction remember these functions. If you're using gdb you can set a breakpoint at the construction function,

```
(gdb) break __do_global_ctors_aux
Breakpoint 1 at 0x403e84 ...
(gdb) run
```

At that point you'll be able to see the symbols for your dynamic libraries. The interesting function at this point is,

`__static_initialization_and_destruction_0`

If you look for functions of this name, you'll find something like,

```
(gdb) info function __static
All functions matching regular expression "__static":

File examples/simple-p2p.cc:
static void __static_initialization_and_destruction_0(int,
int);

File src/internet-node/internet-node.cc:
static void __static_initialization_and_destruction_0(int,
int);

File src/internet-node/l3-demux.cc:
static void __static_initialization_and_destruction_0(int,
int);

File src/internet-node/l3-protocol.cc:
static void __static_initialization_and_destruction_0(int,
int);
```

```
File src/node/queue.cc:
static void __static_initialization_and_destruction_0(int,
int);

File src/node/drop-tail-queue.cc:
static void __static_initialization_and_destruction_0(int,
int);
```

Notice that these are the functions that need to be called in order to do the per-compilation-unit initialization where the global and static constructors are called. Note that *these are also the functions that were optimized out in the static linkage case*. As an example, let's try and look at the global and static constructor initialization in the drop-tail-queue compilation unit. The source file is found in src/node/drop-tail-queue.cc and is linked into the dynamic library libnode.so which is, in turn, dynamically linked into the executable file simple-p2p which we are actually debugging. If you want to set a breakpoint there, you can qualify the function name with the file name and enter the following into gdb.

```
(gdb) break src/node/drop-tail-
queue.cc:__static_initialization_and_destruction_0
Breakpoint 2 at 0x2aaaab196b2c: file src/node/drop-tail-
queue.cc, line 111.
(gdb)
```

You can now go ahead and continue in order to end up at the static constructor initialization code that is associated with the particular file in question. Note that the line number is off the end of the source file since there is no source line associated with this piece of code.

```
(gdb) cont
Continuing.

Breakpoint 2, __static_initialization_and_destruction_0 (
    __initialize_p=10922, __priority=-1424380912)
    at src/node/drop-tail-queue.cc:111
111     }; // namespace ns3
```

At this point you can disassemble and see the actual initialization code if you like. You can pick out the calls to the constructors by looking for the call instructions. For example, you might see,

```
(gdb) disassemble
Dump of assembler code for function
__static_initialization_and_destruction_0:
…
lea    2205373(%rip),%rdi        # 0x2aaaab3b1241 <g_debug>
```

```
callq  0x2aaaab18b560 <_ZN3ns314DebugComponentC1EPKc@plt>
…
(gdb)
```

You can also just step into the initialization function. In the example we're still pursuing, `drop-tail-queue.cc`, there is a call to initialize the IO stream subsystem which you get for free, followed by a definition of a debug component and a call to `MakeClassId()` that initializes `DropTailQueue::cid`. This can be seen by stepping through the initialization.

```
(gdb) next
76        static ios_base::Init __ioinit;
(gdb) next
23      NS_DEBUG_COMPONENT_DEFINE ("DropTailQueue");
(gdb) next
28        MakeClassId<DropTailQueue> ("DropTailQueue",
Queue::iid);
(gdb)
```

If you are a fan of gdb under emacs, you can follow the above recipe and end up looking at the source for drop-tail-queue.h as you might have hoped:

```
#include "ns3/debug.h"
#include "drop-tail-queue.h"

=>_DEBUG_COMPONENT_DEFINE ("DropTailQueue");

namespace ns3 {

const ClassId DropTailQueue::cid =
  MakeClassId<DropTailQueue> ("DropTailQueue", Queue::iid);
```

If you are a fan of graphical debuggers, you can follow the above recipe in the console of the insight debugger and you'll be happy as well.

Finding Shared Libraries (Under Cygwin)

Before getting into some of the deeper issues with DLLs, it is probably worthwhile to make a small detour and discover how to make sure your DLLs are being loaded by the system. This is sometimes not obvious.

First of all, you will most likely find thousands of DLLs on your Windows machine[4] and each of them has been made accessible to Windows in some way. Each of those DLLs was written by someone who wanted to make the DLL name descriptive, with names like

[4] There are over 6,000 DLLs on my XP box at work which doesn't have much in the way of software installed on it.

core.dll and common.dll, for example. There also may be different versions of the DLLs hanging around in unexpected places. Welcome to what Windows programmers lovingly call, "DLL hell."

The first thing you will need to do is to find out where your executable is going to look for your DLLs. In Cygwin, the environment variable PATH is imported from windows and the path names therein are converted to unix-style names. You will need to set an environment variable `LD_LIBRARY_PATH` to point to the directory in which you built your DLLs.

The tool of choice to look at DLL dependencies is called `cygcheck`. Change into your binary directory and (for the program simple-p2p.exe) type the following:

```
~/repos/ns-3-dev/build-dir/dbg-shared/bin >cygcheck
./simple-p2p.exe
.\simple-p2p.exe
  C:\cygwin\bin\cygwin1.dll
    C:\WINDOWS\system32\ADVAPI32.DLL
      C:\WINDOWS\system32\ntdll.dll
      C:\WINDOWS\system32\KERNEL32.dll
      C:\WINDOWS\system32\RPCRT4.dll
  C:\cygwin\usr\craigdo\repos\ns-3-dev\build-dir\dbg-
shared\lib\core.dll
  C:\cygwin\usr\craigdo\repos\ns-3-dev\build-dir\dbg-
shared\lib\simulator.dll
  C:\cygwin\usr\craigdo\repos\ns-3-dev\build-dir\dbg-
shared\lib\common.dll
  C:\cygwin\usr\craigdo\repos\ns-3-dev\build-dir\dbg-
shared\lib\node.dll
  C:\cygwin\usr\craigdo\repos\ns-3-dev\build-dir\dbg-
shared\lib\applications.dll
  C:\cygwin\usr\craigdo\repos\ns-3-dev\build-dir\dbg-
shared\lib\internet-node.dll
  C:\WINDOWS\system32\p2p.dll
    C:\WINDOWS\system32\msvcrt.dll
    C:\WINDOWS\system32\CRYPT32.dll
      C:\WINDOWS\system32\USER32.dll
        C:\WINDOWS\system32\GDI32.dll
      C:\WINDOWS\system32\MSASN1.dll
    C:\WINDOWS\system32\iphlpapi.dll
      C:\WINDOWS\system32\WS2_32.dll
        C:\WINDOWS\system32\WS2HELP.dll
    C:\WINDOWS\system32\ole32.dll
    C:\WINDOWS\system32\OLEAUT32.dll
~/repos/ns-3-dev/build-dir/dbg-shared/bin >
```

This will look through all of the DLL dependencies and figure out where the DLL will be loaded from. You can see that the cygwin1.dll is going to be found in the proper place, i.e., `C:\cygwin\bin\cygwin1.dll` as expected. You can see that the ns-3 DLLs are going to be picked up from the dbg-shared build directory as expected. You can see that there are windows DLLs getting linked in by Cygwin for free, but everything looks great, right? Wrong. Take a closer look at the DLLs. Notice that we've linked to the following DLL:

```
  C:\WINDOWS\system32\p2p.dll
```

That is the Windows peer-to-peer communications DLL. It is supposed to be the ns-3 point-to-point topology DLL, also called `p2p.dll`. This is a common problem. The Windows DLL search path has trumped your LD_LIBRARY_PATH and sucked in the wrong DLL.

What happens if you go ahead and run simple-p2p.exe with this error present?

```
~/repos/ns-3-dev/build-dir/dbg-shared/bin >./simple-p2p.exe
~/repos/ns-3-dev/build-dir/dbg-shared/bin >
```

It looks like the program ran just fine -- there are no complaints from Cygwin. It turns out that nothing happened. To see this, run the program under the debugger.

```
~/repos/ns-3-dev/build-dir/dbg-shared/bin >gdb ./simple-
p2p.exe
GNU gdb 6.5.50.20060706-cvs (cygwin-special)
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public
License, and you are
welcome to change it and/or distribute copies of it under
certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show
warranty" for details.
This GDB was configured as "i686-pc-cygwin"...
(gdb) run
Starting program: /usr/craigdo/repos/ns-3-dev/build-
dir/dbg-shared/bin/simple-p2
p.exe
gdb: unknown target exception 0xc0000139 at 0x7c964ed1

Program received signal ?, Unknown signal.

Program exited with code 030000000471.
You can't do that without a process to debug.
(gdb)
```

The program died with exception `0xc0000139` and never actually ran. To people familiar with the Microsoft Way ™ you may recognize this exception as an SCODE. It turns out that this is the error

`0xc0000139` `Entry Point Not Found`

This error was caused when the Cygwin runtime tried to load the p2p.dll and begin initializing it. Since the DLL is not a Cygwin DLL, it didn't have the expected entry point and the exception was raised. When you run the program directly, it exits silently.

Another common error is

`0xc0000135` `Failed to Initialize Properly`

This happens when a DLL is not found at all.

The program cygcheck will report this more clearly than a hidden DLL. For example,

```
>cygcheck ./simple-p2p.exe
  C:\cygwin\usr\craigdo\repos\ns-3-dev\build-dir\dbg-
shared\lib\core.dll
  C:\cygwin\usr\craigdo\repos\ns-3-dev\build-dir\dbg-
shared\lib\simulator.dll
  C:\cygwin\usr\craigdo\repos\ns-3-dev\build-dir\dbg-
shared\lib\common.dll
  C:\cygwin\usr\craigdo\repos\ns-3-dev\build-dir\dbg-
shared\lib\node.dll
Error: could not find applications.dll
…
>
```

I had deleted the applications.dll file from the dbg-shared/lib directory. You may find other DLLs hidden depending on your configuration. For example, my machine at home has a common.dll from the Microsoft Outlook Business Contacts Manager that originally hid my ns-3 common.dll.

It is a good idea to create your own "namespace" for DLL file names by adding some characters to more uniquely identify your DLLS. The people at Cygwin prepend the characters "cyg" to their DLLS. I have recommended that we prepend the characters "ns3-" to our DLLS.

Now, we return to global and static constructor initialization.

PE / COFF / GNU / Cygwin

If you're on Cygwin, things aren't as straightforward as they were on the ELF-based systems . The Cygwin environment bridges the gap between Linux and Windows and so

you have a situation where part of your environment is Linux-like and part is Windows-like. You're using the GNU toolchain to generate Microsoft DLLs in Portable Executable (PE) format. PE descends from an older Common Object File Format (COFF) developed for VAX / VMS systems. You are going to have to navigate through the cygwin1.dll code before you get to the global and static constructors of interest.

We will perform the same examination as we did on the ELF-based system, running the simple-p2p example program, but this time it is simple-p2p.exe since we're actually running on Windows.

The first thing to do after firing up the debugger is to force a load for the cygwin1 DLL and set a breakpoint at the function that is responsible for eventually calling the static constructors. The odd syntax (*&'') is required to force gdb to correctly interpret the double colons.

```
> gdb ./simple-p2p.exe
GNU gdb 6.5.50.20060706-cvs (cygwin-special)
...
This GDB was configured as "i686-pc-cygwin"...
(gdb) dll cygwin1.dll
(gdb) break *&'dll::init'
Breakpoint 1 at 0x61010270
(gdb) run
Loaded symbols for ... node.dll

Breakpoint 1, 0x61010270 in dll::init () from
/usr/bin/cygwin1.dll
(gdb)
```

At this point, you have the *symbols* for all of your DLLs loaded, but you need to actually load the DLL you're interested in before setting a breakpoint. From the example above, we were interested in the static constructors for the file `drop-tail-queue.cc` which is part of the node.dll Windows DLL.

```
(gdb) dll node.dll
(gdb)
```

We are still interested in the function

`__static_initialization_and_destruction_0`

since Cygwin uses the GNU toolchain. I don't like typing much, so I usually do an `info functions` with only a few letters of the desired function in order to get gdb to fill out everything I need. This is the per-compilation-unit initialization, so there will be one of these functions for every file in every DLL that needs its static constructors initialized.

You have to make sure you're getting the right one. In this case, it's the one for the file `src/node/drop-tail-queue.cc`

```
(gdb) info functions __static

...

File src/node/llc-snap-header.cc:
static void __static_initialization_and_destruction_0(int,
int);

File src/node/drop-tail-queue.cc:
static void __static_initialization_and_destruction_0(int,
int);

File src/node/queue.cc:
static void __static_initialization_and_destruction_0(int,
int);

...
(gdb)
```

Now you can set a breakpoint on the function, qualified by the file name just like you did for the ELF case as follows.

```
(gdb) break src/node/drop-tail-
queue.cc:__static_initialization_and_destructio
n_0
Breakpoint 2 at 0x828538: file /bin/../lib/gcc/i686-pc-
cygwin/3.4.4/include/c++/
iostream, line 77.
(gdb)
```

At first glance, it looks like our breakpoint is set somewhere else in the middle of nowhere. This turns out not to be the case. Go ahead and continue until you hit the breakpoint (number two in this example). If you leave breakpoint number one active, you'll end up breaking at the initialization of all of the windows DLLs as well. Just continue through them.

```
(gdb) cont
Continuing.

Breakpoint 1, 0x61010270 in dll::init () from
/usr/bin/cygwin1.dll

...
```

```
(gdb) cont
Continuing.

Breakpoint 2, __static_initialization_and_destruction_0
(__initialize_p=1,
    __priority=65535)
    at /bin/../lib/gcc/i686-pc-
cygwin/3.4.4/include/c++/iostream:77
77        static ios_base::Init __ioinit;
(gdb)
```

Now look at the stack trace.

```
(gdb) bt
#0  __static_initialization_and_destruction_0
(__initialize_p=1,
    __priority=65535)
    at /bin/../lib/gcc/i686-pc-
cygwin/3.4.4/include/c++/iostream:77
#1  0x00828760 in global constructors keyed to
_ZN3ns313DropTailQueue3cidE ()
    at src/node/drop-tail-queue.cc:112
#2  0x61010263 in per_module::run_ctors () from
/usr/bin/cygwin1.dll
#3  0x00000001 in ?? ()
#4  0x610102b0 in dll::init () from /usr/bin/cygwin1.dll
#5  0x01140008 in ?? ()
#6  0x00000000 in ?? ()
(gdb)
```

Note that you are in the global constructors keyed to DropTailQueue! The first initialization is for the iostream which was used in drop-tail-queue.cc and that's what you saw when you entered the breakpoint– it turns out you were going to the right place after all. Now disassemble and look for a call to a familiar-looking function:

```
(gdb) disassemble
...
movl   $0xffffffff,0xffffff88(%ebp)
call   0x83c430 <_ZN3ns314DebugComponentC1EPKc>
cmpl   $0xffff,0xc(%ebp)
...
---Type <return> to continue, or q <return> to quit---q
Quit
(gdb)
```

We have found the first static constructor in the drop-tail-queue.cc file. Set a breakpoint, continue and step to get there.

```
(gdb) b *0x83c430
Breakpoint 3 at 0x83c430: file /bin/../lib/gcc/i686-pc-
cygwin/3.4.4/include/c++/
iostream, line 77.
(gdb) cont
Continuing.

Breakpoint 3, 0x0083c430 in
ns3::DebugComponent::DebugComponent ()
    at /bin/../lib/gcc/i686-pc-
cygwin/3.4.4/include/c++/iostream:77
77        static ios_base::Init __ioinit;
(gdb) step
ns3::DebugComponent::DebugComponent (this=0x884198,
    name=0x87f992 "DropTailQueue") at src/core/debug.cc:108
108       : m_isEnabled (false)
(gdb)
```

I feel joy! Now we can get on to the business at hand which is actually finding the problem.

If you are a fan of gdb under emacs, you can follow the above recipe and end up looking at the source for drop-tail-queue.h as you might have hoped:

```
#include "ns3/debug.h"
#include "drop-tail-queue.h"

=>_DEBUG_COMPONENT_DEFINE ("DropTailQueue");

namespace ns3 {

const ClassId DropTailQueue::cid =
  MakeClassId<DropTailQueue> ("DropTailQueue", Queue::iid);
```

If you are a fan of graphical debuggers, you can follow the above recipe in the console of the insight debugger and you'll be happy as well.

Summary

Dealing with libraries and C++ can be a difficult proposition if you have a system that uses global and static constructors. It can be done, but with variable amounts of pain depending on your approach.

We recommend against attempting to combine systems using global and static constructors with static linkage. You may succeed, but will probably end up with a degenerate form of shared libraries in the end.

You will have more success using shared libraries, however debugging problems with global and static constructors can be difficult since they are executed before your main program starts and are therefore much more difficult to debug than ":normal" code.

Cygwin-based systems add a level of complexity with the cygwin1.dll and share an environment with many thousands of other DLLs. There are quirks in the system which may make it appear that your program executes when it is in fact doing nothing, and error messages can sometimes be confusing.

Nonetheless, you can get global and static constructors to work if you have the patience, and it does provide an elegant mechanism for system initialization.