THIS SECTION NOT FOR PUBLICATION

ISSUE:	0.6
PREVIOUS ISSUE:	0.5
DATE	2001-09-14
AUTHOR:	Pete Goodliffe
CONTACT:	pete.goodliffe@pace.co.uk
MASTER FORMAT:	Word 97
COPYRIGHT:	© Pete Goodliffe 2001 *(retained)*

Techniques for debugging in C++

by Pete Goodliffe <pete.goodliffe@pace.co.uk>

What happens when programs die? Do they go to digital heaven? Are they reincarnated (if they're really bad, perhaps they come back as a daemon)? Or do they leave a legacy for their children – a guide on how to live a better life?

In this article I present a debugging library 'dbg' I have written that has been specifically designed to integrate well with modern C++ programming idioms. I also describe a number of common debugging methods that can employ these utilities.

Before I present the library, we'll cover a little background.

Traditional mechanisms

People have been debugging ever since the first program was written. In fact, it's been going on even before this. So how do we debug in these enlightened times? Not much more methodically than when we did it ten years ago.

Debuggers

When talking about debugging techniques here I'm going to deliberately avoid mention of a debugger. Why? Well, a debugger is not always present on every platform (I've recently been playing with some embedded environments where a debugger was not easily available). Nothing can compare to the power of using a debugger to "get into" program execution, but when a debugger is not available we still need an arsenal of techniques to allow us to debug effectively.

Besides, oranges are not the only fruit – a debugger is not the only tool to aid debugging. It's good for dissecting a program that doesn't work properly, but it's not too handy to preemptively detect code that could cause problems later down the line.

Constraints

In this article I focus more on constraint-based techniques for ensuring our code is bug free. That is, writing some extra code (that can be compiled out at build-time, if necessary) that enforces a set of logical constraints. These constraints define whether or not the program is operating correctly.

If constraints are added in at the smallest granularity in the code-base then the maximum benefit is achieved – most of the code is sanity tested. Of course there is a possible downside to this; the extra code may make your program run slower. Whether this is significant is a moot point, and will depend to a large extent on the platform and target for your code.

Judicious use of these constraints in your code can leave you satisfied in the correct behaviour and can help identify problem situations early, before they escalate into major problems.

The traditional language mechanism for doing this is the standard C library `assert` macro. Any programmer who knows a `for` from a `while` knows that `assert` is a simple macro used to enforce a run time constraint in the manner described above.

Logging

Another mechanism used to aid the debugging process is logging – for example the Win32 `RETAILMSG` system, or the Unix `syslog`. Good logging mechanisms allow us to trace execution, report program state, and are flexible enough to allow us to divert output to different destinations, perhaps also prioritising the significance of certain messages.

Defensive programming

Of course, other good debugging (or bug avoiding) practices are the set of defensive programming techniques. This library and article were inspired by my C Vu Professionalism in Programming column on defensive programming [1]. We won't cover the same ground as in that article.

Moving forward

Constraints are a very valuable method for getting your code to work and keeping it working. However, `assert` is not really satisfactory. This is mainly because of what it is: a C-style macro. Why was it ever allowed into the standard library as it is – in lower case is anyone's guess. It should really have been an upper case macro.

If we can avoid use of macros (and the preprocessor as a whole) we should – it jars with the modern C++ idiom. As Stroustrup says, "The first rule about macros is: Don't use them unless you have to." [2]. So do we have to?

As far as logging is concerned, we need a high-quality portable logging mechanism. Perhaps it can fall back to the existing platform-dependant logging mechanisms at the implementation level. However, you can't write platform independent code without a good logging abstraction.

What we'd like to achieve in a good debugging library

There are a number of requirements for a good "debugging" library that I worked from:

- Constraints must be better than assert in all areas
- Must compile out to nothing in production builds, if required
- Minimal performance overhead whilst running
- Easy to use, non-intrusive code
- Avoids the evil preprocessor
- Must integrate well with the existing standard library and modern C++ idioms
- Obviously, must be completely portable across compilers and OSes

There are a number of secondary feature requirements I was aiming for:

- Ability to differentiate different diagnostic levels – and enable/disable them independently.
- Control at run time whether constraints are fatal (i.e. cause an `abort()`, throw exceptions or are disabled completely).
- Differentiate different logging sources, and can selectively enable/disable output for a single device driver in an entire project, for example.
- Provide execution tracing capabilities.
- Support writing post conditions (this is hard to model in C++)
- Provide an easy mechanism for printing tracing diagnostics out when needed

There are a number of very common constraints traditionally tested for with the `assert` macro. These cases are:

- execution reaches unimplemented functionality
- execution reaches a place where it "shouldn't get"
- a pointer value is invalid i.e. it equals zero
- array access bounds are out of range
- finally, any other general boolean constraint (assertion)

The library should provide different versions of an `assert`-like function that makes the intent of the constraints listed above explicit. When you come across something like "`check_pointer_isnt_null(p->ptrval)`" it's immediately clear what is meant, instead of something like "`assert(0 != p->ptrval)`".

Motivation

So why write a new debugging library?

There are already a number of other debugging libraries available. However, they don't all fulfill the requirements laid out above. Some don't compile out to nothing in a production build. Some make very heavy use of the preprocessor. Others don't fit naturally into modern C++, or the debugging commands are too intrusive, or they don't read easily in the code. I've yet to find the perfect tool for the job.

About the dbg library

Having set the scene in some detail we can now look at the dbg library. To a large extent it meets the ideals set out above. It does so in a manner that fits in well with modern C++ idioms. We'll see the basics of how to use it, and then discuss some of the design decisions involved.

As you've probably guessed, the main facilities provided by the dbg library are:

- Expressive and easy to use constraint utilities
- Flexible integrated logging support based on C++ streams

The dbg library aims to be something that could credibly be bundled with the standard C++ library. That is, it should be of the highest quality possible and follow standard library conventions where possible.

The dbg library is entirely contained in the `dbg` namespace (hence the name). It comes with full online API documentation – see "Availablity" below for details.

Constraints using dbg

Firstly, the dbg library defines a number of debugging levels, used to describe the severity of a constraint. The levels specifically related to constraints are:

- `info` - Not a problem, just a message for interest
- `warning` - Potentially bad things that can occur, but can be recovered from
- `error` - For errors that can't be recovered from
- `fatal` - Errors at this level will cause the dbg library to abort execution

The different types of constraint provided by the dbg library are:

1. check_ptr

One of the most common constraint checks is that pointers are non-zero. I often wonder in C++ how valid a check this is, since new does not return zero any more (unless you ask it to, of course). However, when integrating with legacy code you may get invalid zero-pointers flying around. Checking that a pointer parameter is non-null is still a very handy check[1].

```
void example(int *ptr)
{
    dbg::check_ptr(ptr, DBG_HERE); // [2]
    // do something
}
```

2. check_bounds

This checks your array access indexes to ensure they are in bounds. If the array is in scope, the dbg library can deduce the size of the array so you don't have to supply the bounds manually. This can be very valuable when array sizes change frequently.

```
int array[10]
int index = 10;
dbg::check_bounds(index, array, DBG_HERE);
array[index] = 5;
```

3. sentinel

You should put this directly after a "should never get here" comment, just to ensure that you never get there.

```
switch (variable)
{
    ...
    default:
    {
        // Should never get here
        dbg::sentinel(DBG_HERE);
    }
}
```

4. unimplemented

When you start to develop code, you'll probably stub functionality to be implemented later. Use of this constraint can prove when areas of code that are not finished off get called.

```
switch (variable)
{
    ...
    case SOMETHING:
    {
        dbg::unimplemented(dbg::warning, DBG_HERE);
        break; // just in case constraint behaviour
               // is non-fatal
    }
    ...
}
```

5. assertion

Finally, the general purpose assertion, which allows you to put an arbitrary boolean expression as a test. This is like the C-style `assert`.

```
int i = 5;
dbg::assertion(DBG_ASSERTION(i != 6));
```

Assertion behaviour

The behaviour of the dbg library constraints can be set with `dbg::set_assertion_behaviour` to be one of

- `assertions_abort` - Assertions cause a program `abort`
- `assertions_throw` - Assertions case a `dbg_exception` to be thrown
- `assertions_continue` - Assertions cause the standard diagnostic printout to occur (as it does with the above behaviours) but execution continues regardless

Constraint activity and the logging output can be enabled/disabled at run time, as well as being compiled out in its entirety.

assertion_period

As well as specifying the `assertion_behaviour`, you can set an assertion period. This is helpful for constraints that get triggered very frequently. If you `set_assertion_period` of one second, then no matter how many times a constraint fails, it will only produce output once a second. However, it will produce information on how many times it has really triggered for reference.

Advanced logging with dbg

The dbg logging support is very flexible. You can attach and detach arbitrary ostreams to/from each different diagnostic level. You can attach any number of ostreams to each level. Initially, `std::cerr` is attached to the `dbg::error` and `dbg::fatal` diagnostic levels.

At any point in your code you can log diagnostic information in the usual C++ ostream style. You just write

```
dbg::out(dbg::info) << "Hello this is some information\n";
```

There are a number of other neat logging features. For example, you can enable a number of diagnostic prefixes with `enable_level_prefix` and `enable_time_prefix`. Doing so would produce output like this:

```
*** Tue Jul 31 09:37:08 2001:    info: sentinel reached
```

Separating diagnostic sources

Well structured code-bases split code into distinct functional units, like libraries, layers, or separate device drivers. The use of the dbg library can compliment this. For each constraint, you can describe a diagnostic source (e.g. “foo_lib”) and then at run time enable/disable each source selectively. This allows you to pin-point diagnostics from only particular sections of your code.

For example if you have a number of device drivers, with one called “disk” and one called “keyboard” then in the first you might write a constraint like

```
dbg::check_ptr("disk_drv", disk->address);
```

and in the second

```
dbg::check_ptr("keyboard_drv", keyboard->event_queue);
```

Now if the keyboard driver appears to be doing odd things, you can switch on its diagnostics by putting this in your main code:

```
// Enable the keyboard diagnostics at all levels
dbg::enable(dbg::all, "keyboard_drv", true);
```

The keyboard driver code will now produce diagnostic output, and the constraints will be made active, whilst the disk driver will still have all diagnostics dormant.

Design decisions

That was a pretty quick tutorial on the use of the dbg library. See the online documentation for more information. Now that we know how to use the library, let's see how dbg achieves the goals set out for it.

I present this library partly as a call for peer review. Whilst I have used dbg successfully for some time, if there are comments from the ACCU members that would cause the library to improve I'd be interested in hearing them.

Avoiding macros

This is done by making the constraint utilities real functions, not macros. If you do a build that disables the dbg library, then the functions are replaced by empty inline stubs that optimise away to nothing.

This requires a little template magic to produce a credible ostream substitute class, see the `dbg.h` header file for details.

Do we really need to avoid macros? I believe that in modern C++ code it makes more sense to see a line like:

```
dbg::check_ptr(p);
```

rather than something like:

```
DBG_CHECK_PTR(p);
```

It reads naturally – in the same way that standard algorithms express intent so much more clearly than a for loop.

Why is the general purpose constraint called 'assertion' not 'assert'?

This is one of my annoyances. I can't call the general purpose constraint `assert` since if someone inadvertently mixes `"dbg.h"` with `<assert.h>` nasty things happen. This is a perfectly good reason for not wanting all of our constraints to be macros as well.

Perhaps this is a good argument for a scope-respecting preprocessor? :-)

How to implement differentiated diagnostic sources

This is perhaps the least satisfactory part of the design. Of course, you could remove the facility and remove the problem altogether, but it is genuinely useful.

The reason I'm not happy with the implementation is that it seems to make production builds grow, since the source is identified as a character string; most optimisers don't seem to be able to remove the string if an inline function doesn't use it. This is very annoying.

Other alternatives to using a string to identify the source would be to use integer source codes – but that would require the programmer to have some kind of shared error code header file – a maintenance nightmare. Alternatively, some source definition object could be created and referenced in each dbg call. For diagnostic sources that can span multiple files, this can become clumsy too – you'd have to declare the object in a header file to share it. That's too intrusive.

I'm open to suggestions on this one.

Post conditions

I've done what I can with the post conditions, but it's still not a full and general post condition implementation. The problem here is with the C++ language itself. In order to create a safe post condition you need to use the RAII idiom [3]. When you create a class to run the post condition you can't include arbitrary code. You can't even do this with a disgusting macro construct unfortunately.

Perhaps we should all give up and use Eifel? Either that or be happy writing unsafe code...

Non-debug constraint functions don't have the same signature as the debug versions

This has been mentioned to me as an undesirable feature of the library. I don't personally have a big religious problem with it, and can't see a better way of implementing what this does.

The mechanism used has the overwhelming advantage that it is very simple, and does (on the whole) compile out to nothing.

Tracing needs a named object

You have to write:

```
dbg::trace trace_object(DBG_HERE);
```

and can't miss out the variable name. If you do, the variable is anonymous and gets destroyed at the same place it is created, rather than when the variable goes out of scope. Unfortunately we have no way of enforcing this in the code, which may lead to confusing results when used incorrectly.

Not every platform provides a __FUNCTION__ like definition

It's a shame, and it should be in the standard IMHO. On platforms without a `__FUNCTION__` the diagnostic output is not so helpful.

How useful are assertions throwing?

It's neat to be able to select a "throwing" style assertion, but what happens in code not expecting to have to handle exceptions? It may cause more trouble than its worth. It's certainly debatable whether dbg exceptions should be caught, since they can be compiled out.

Performance

In order to have some confidence that the dbg library really does compile away to nothing we need concrete proof. I have used the library on a number of platforms, and the results are shown below. Note that here we are not comparing image sizes across compilers – I may not have configured each compiler to generate the smallest code possible (certainly the gcc 3.00 code is very large, and this *can* be reduced greatly). The first two code size columns are for a test program built with debugging support enabled and disabled (i.e. conditionally compiled out) respectively. When disabled, the dbg library is not linked to the executable so we can expect a dramatic size reduction[3]. The final column shows the size of this test program when no dbg APIs are called at all. In an ideal world this size will be identical to the *release build* column.

compiler/version/ platform/options	compiled image sizes		
	-DDBG	release build	removed completely
gcc 2.96 Linux (`-O0`)	69381	21590	15978
gcc 2.96 Linux (`-O1`)	66411	15883	15725
gcc 2.96 Linux (`-O2`)	66635	15851	15661
gcc 2.96 Linux (`-O3`)	66635	15851	15693
gcc 2.96 Linux (`-O4`)	66635	15851	15693
gcc 3.00 Linux (`-O3`)	588147	129680	129584
bcc32 WinNT 5.5.1	201728	138752	138240
MSVC 6.0 WinNT (release, 0=max speed)	54593	11662	11496

Table of compiled image sizes

We can see that in the general case the library performs just as we'd hope – there is no significant overhead associated with its use for non-debug builds. However, when we employ the string literal diagnostic source definitions there is a slight overhead (since the optimiser does not remove the string literal when it removes the empty inline function).

I have not performed any investigations into the overhead of the dbg library in terms of runtime speed. Empirically, it has always run far better than satisfactorily, so it has not been necessary. Perhaps an Overload reader would care to generate this data?

Extensions

There are still some areas the library could be expanded in. Top of my list are:

1. Assertions breaking out into the debugger

Another `assertion_behaviour` could be added to break out to the debugger when a constraint is broken.

It's a nice idea, but unfortunately far too platform specific – it would require the library to be ported to each new platform which is undesirable.

Another approach I investigated was to add an "`on_assert`" callback facility akin to `atexit`. In there you can register your own debugger trap. However, I didn't like this. When the debugger opens, you have to step back through your function, then some internal dbg library logic, before reaching the point of constraint failure – not too neat.

There is an internal dbg library function `do_assertion_behaviour`. You can set a breakpoint here to get the same effect anyway.

2. Only works with ostreams

The dbg library logging only currently works for ostreams. It would be nice to extend it to all basic_ostreams.

Conclusion

The dbg library is a powerful implementation of the constraint defensive programming technique, with a number of other useful capabilities including a flexible and integrated logging mechanism.

To be effective you really need to employ the technique from the start of a project – it's very hard to retrofit constraints on at a later date. You never have the time or impetus to be rigorous enough to make it fully valuable.

Availability

The dbg library is available along with online documentation from `http://cthree.org/dbg/`. Future developments will be posted there.

The API documentation is available from `http://cthree.org/dbg/kdoc/dbg.html`.

References

[1] Pete Goodliffe. Professionalism in Programming #8: Defensive Programming. In: C Vu 13.X. ISSN: 1354-3164

[2] Bjarne Stroustrup. The C++ Programming Language. 3rd Edition. ISBN: 0-201-88954-4. Section 7.8: Macros.

[3] Bjarne Stroustrup. The C++ Programming Language. 3rd Edition. ISBN: 0-201-88954-4. Section 14.4.1: Using Constructors and Destructors.

Footnotes

1. Zero is a special case of a set of invalid pointer values. At least in the C/C++ standard it is defined to be a pointer to nothing. However, on most platforms 1 and 3 are also invalid - does testing for zero give us a false sense of security?
2. DBG_HERE is a helpful macro that describes a source file position – OK we were trying to avoid the preprocessor, but this is unavoidable.
3. At least for a small test program. For larger programs this reduction in size will quickly diminish in significance.