Size Does Matter

by John Fehr

Caveat: This technique to reduce code size will work successfully with shared libraries, applications, and static libraries that do not contain many independent functions. (In other words, the application that links to these static libraries would have to directly or indirectly access most of the functions inside the library to get any reduction in final executable size.)

These days, when it's hard to know if your code will wind up on the desktop with oodles of storage space, or in a tiny embedded device with very limited storage space, it's a good idea to use every trick you know to reduce the size of your footprint. The very best solution is to optimize your algorithms so that the resulting implementation is smaller and faster. But assuming you've done that, and used the appropriate compiler optimization flags, what's left?

Like it or not, the code in static (*.a*) libraries are (usually) smaller than shared (*.so*) libraries, because you don't need to specify position-independent code generation for static libraries. This gives you an extra register to play with, which makes all the difference when you only have a handful to start with. Usually though, your static libraries are much bigger than your shared libraries. This is because static libraries are linked with **ar**, which (generally) just concatenates the *.o* files together, while the shared libraries are linked with **ld**, the same linker that links an application's .o and libraries together to create the executable. If we want small static libraries, then our object is to get the *.o* files as small as possible.

Gcc actually has a decent optimizer. One of its problems though (not really its fault) is that it can only optimize what it's given. If you give it lots of little snippets of code, it'll optimize those as best it can. If, however, you give it more code (or event those little snippets all combined into one file), it's able to optimize more code, resulting in smaller code size! (And hopefully faster as well.) So, instead of trying to make our individual .o files smaller, why not try to make fewer .o files, and let the compiler make those as small as possible?

The best way to illustrate this is with an example. We'll use the latest (1.1.3) zlib library. Download the archive, and unarchive it. **Cd** into the *zlib-1.1.3* directory, and type:

```
./configure
```

Then we want to make sure that we're using our version of the static library, since the QNX RTOS comes with its own static and dynamic versions. (This is for minigzip and example.) Change "`CC=gcc`" to "`CC=qcc`" in the Makefile, and the "`LDFLAGS=-L. -lz`" to "`-Bstatic -L. -lz -Bdynamic`." (This just tells the compiler to link the z library statically from the current directory, and the other libraries as dynamic.) Then, type:

```
make
ls -l libz.a example minigzip
```

You'll notice that the *libz.a* file is 64634 bytes, the example executable is 59423 bytes, and the minigzip executable is 55269 bytes. You might also notice that everything was compiled using optimization level 3 (-O3). Unfortunately, the code generated by -O3 is not necessarily smaller than -O2. Since size is our goal for this article, try changing -O3 to -O2 in the Makefile, and typing:

```
make clean
make
ls -l libz.a example minigzip
```

Now our *libz.a* file is 60314 bytes, the example executable is 55050 bytes, and the minigzip executable is 51088 bytes. We've already saved an average of about 7% in size just by changing the optimization level.

Now, let's see if we can get fewer *.o* files. Change the "`OBJS =`" line in the Makefile to read "`OBJS = all_zlib.c`," and create an *all_zlib.c* file that contains:

```
#include "adler32.c"
#include "compress.c"
#include "crc32.c"
#include "deflate.c"
#include "gzio.c"
#include "infblock.c"
#include "infcodes.c"
#include "inffast.c"
#include "inflate.c"
#include "inftrees.c"
#include "infutil.c"
#include "trees.c"
#include "uncompr.c"
#include "zutil.c"
```

(These are all the files whose *.o* versions were previously in the "`OBJS =`" line.)

Now try to **make** it. You'll get lots of warnings as well as quite a few errors. These will usually fall into the following categories:

- "**"(symbol)" redefined**," which means that you had a **#define** in one file that was **#defined** again later as something else...
- "**redefinition of "struct (structure)"**," which means that you used a certain structure name for some data in one source file, and then used the same structure name for some other data in another source file.

Just a little side note here. Whenever possible, make sure that your code does NOT do either of these. Defining a symbol or structure with a certain name differently in different source files makes it very difficult to read and follow.

So, how do we fix these warnings and errors without making major modifications?

- If it's a **define** and both symbols were defined in source files (not header files), simply **#undef** those symbols at the end of your source file.
- If one symbol is defined in a header file, and the other in a source file, change the one in the source file so that it's unique.
- If both symbols are defined in the same header file, on the same line, you've somehow included the same header file twice. Make sure you have appropriate include detection **#ifdefs** in the file. For example, for a file called *cooltools.h*, you might have:

```
#ifndef __COOLTOOLS_H__
#define __COOLTOOLS_H__

... the body of your header file

#endif
```

This way, even if you have multiple `#include cooltools.hs`, it will only be processed once.

- If the symbols are defined in different header files, you'll have to modify one so that it's unique, and figure out which source files are using which define. (Look at which header file is being included for each source file that contains that symbol, and modify each instance of that symbol in the source file(s) accordingly.) This works for both type a) and type b) problems.

Let's apply these to our library. First, apply 1) to DO1, DO2, DO4, and DO8 in *crc32.c* and *adler32.c*. Next, apply 2) to NEEDBYTE, NEXTBYTE, DONE, and BAD in *inflate.c*, since they are already defined in *infutil.h*. (Change all instances of **NEEDBYTE** to **INFLATE_NEEDBYTE** and **NEXTBYTE** to **INFLATE_NEXTBYTE**, etc. in *inflate.c*.)

Now you'll get a bunch of redefinitions of a **struct**. If you look at them, you'll notice most of them are dummy **structs**, put in for buggy compilers. Just replace anything that looks like:

```
struct (structure name) {int dummy;};
with
struct (structure name);
```

Go ahead and change all of those definitions. You can just type:

```
grep dummy *.c *.h | grep struct
```

to find all of the occurrences.

We do, however, have one redefinition of a **struct** that's a bit troublesome. **Zlib** does something that's a little weird. In *zlib.h* the **z_stream_s struct** has a pointer to an **internal_state** structure, but it doesn't define what's in the **internal_state** structure. The **internal_state** structure is defined in both *deflate.h* and *inflate.c*, but completely differently! We can't simply rename one of the **internal_state** structures, because then it wouldn't be contained in the **z_stream_s** structure! So what do we do?

It's quite simple. Just define an **inflate_internal_state** structure in *zlib.h* the same way that **internal_state** is defined there, and change the state variable to a union instead of just a pointer. However, since gcc doesn't allow unions without a name, we'll name the union and put in defines so that the correct union member is used without changing our source code. So, we change:

```
struct internal_state FAR *state;
```

to

```
union {
   struct internal_state FAR *state;
   struct inflate_internal_state FAR *inflate_state;
} u;
#define state u.state
#define inflate_state u.inflate_state
```

in *zlib.h*.

You'll also have to change the name in the definition of the **internal_state** structure in *inflate.c* to *inflate_internal_state*. Then, change any occurrences of "**`->state`**" to "**`->inflate_state`**" in *inflate.c*

as well. Lastly, again in *inflate.c*, change any occurrences of "**struct internal_state**" to "**struct inflate_internal_state**."

Now, when we **make**, we get a bunch of redefined warnings, but they warn about a structure being redefined in the same file. Welcome to 4). Put the **`#ifndef`** suggested above to *infblock.h*, *inftrees.h*, and *infcodes.h*.

Now, attempt to compile and we get a rather confusing error:

```
trees.c:503: syntax error before ".".
```

If we look at that line, it just reads "`int bits;`." There doesn't seem to be anything wrong with that! If you encounter something like this, (and it happens more often then you'd think), odds are that the name of the variable in question was **#defined** in another source file. Type:

```
grep bits *.c *.h | grep define
```

Wow! There's a lot of them. Go into *infblock.c*, *infcodes.c* *inffast.c*, *inftrees.c*, and *maketree.c*, and put an "**`#undef bits`**" at the end of each file. (Another option would be to **`#undef`** bits just before line 503 in *trees.c*, but the first option is usually the safest bet.)

Zlib.a should now build without a problem. Take a look at the sizes. *Libz.a* is only 46934 bytes now, example is only 53664 bytes, and minigzip is only 50333 bytes! Our *libz.a* is now 22% smaller then our previous version, and 27% smaller than our original! Example and minigzip also shrunk, although not as dramatically.

This trick should work to varying degrees with any project, be it a library or an executable. In one of my projects, the executable's size went from 13.7M to 2.8M using this method!

Another positive side effect of including many source files in a single source file is that your compilation will be much faster. Our example above is quite simple. On my machine, a "normal" build takes about seven seconds. Our single-file build takes five seconds. You should notice much bigger differences in your code depending on how many source files you have. The example I gave above (13.7M to 2.8M) took 16:24 to build 289 source files, but only 2:14 when those files were included into eight "big" source files.

I hope you'll be able to shrink your libraries and executables as much as I have!