Writing Small And Fast Software

Felix von Leitner
Convergence
`felix@convergence.de`

January 2001

Abstract

Software is becoming larger and slower quicker than RAM prices fall and computer become faster. Good programmers are rare and expensive, so people rather write bad software and buy new hardware than going for quality.

Introduction

Academia says: trade code size for speed (inlining, macros, loop unrolling, immensely complex monster algorithms...)

In practice, more code is almost always bad: it trashes the CPU cache, hogs the memory bus, uses up more CPU cycles, makes a security audit of the code harder, is harder to follow and maintain, and - last but not least - more code needs more documentation.

Executive Summary

Indirection Induces Bloat And Kills Performance!

Remove Unnecessary Indirection!

Know Thy Enemy!

The main reasons for large code are:

1. negligence
2. over-eagerness
3. bad management
4. multiple layers of wrappers

Enemy: Negligence

Tidying up old code is often more work than writing new code. Common problems are:

1. merging code but don't remove overlap
2. leaving debug code in
3. using general instead of specific APIs (e.g. using `fprintf` instead of `write` for a static string)
4. implementing the same stuff all over the place (often found in team environments)

Enemy: Over-Eagerness

Especially young and inexperienced programmers fall for the utopia to create the ultimate, perfect, all-encompassing API.

Object oriented programs tend to define many unneeded methods and routines ”just in case”. Point in case: C++ iostream and libg++.

Fact: Experience shows that your API will not be perfect.

Fact: Using APIs that have been extended is often confusing.

Enemy: Bad Management

When merging code into a common source tree, it is important to spend time looking for code that solves the same problem. The instances of that code then have to be reduced to one (example: AVL trees in the Linux kernel).

When more than one person works on a project, every CVS check-in has to be regarded as a code merge in this sense!

Paying programmers by the line of code does not help, of course.

Enemy: Wrappers Wrapping Wrappers Around Wrappers

This is a particularly common problem in C++ code.

Classic rationale: portability.

Classic example: thread libraries wrapping pthread, which in turn wraps clone().

Even more classic examples: kde, wrapping qt, wrapping xlib. gtk–, wrapping gtk+, wrapping xlib.

Apparently, there even is a Qt emulation wrapper around gtk+!

Enemy: Dynamic Run-Time Resolving

This can be done very efficiently, but more often it is done very poorly.

Examples: signals in Qt, hash tables for symbolic attribute lookup in gtk+, late binding in DYLAN.

What Can I Do?

1. generalize routines only if it’s free
2. write specific interfaces first
3. use specific routines instead of generic
4. be aware of the bloat you generate
5. *your idea here*

Generalize Routines Only If It's Free

An internal function in diet libc:

```
int __ltostr(char *s, int size, unsigned long i,
             int base, char UpCase);
unsigned int fmt_ulong(char *dest,unsigned long src);
```

A good test whether your interface is too generic is: if you leave the names of the arguments away, is it still obvious what each argument does?

```
int __ltostr(char *, int, unsigned long, int, char);
int fmt_ulong(char *,unsigned long);
```

Write Specific Interfaces First

- Programmer tries to write reusable code, defines very broad and generic API
- Initially, the problem is not well known.
- Much time is wasted on implementing unneeded functionality in broad interfaces.
- In the end, removing the bloat would alter the API (which programmers know is **very bad**).
- Also, someone might already rely on the API

When You Need To Extend An API

Do not make an existing function more powerful!

Add a new function instead!

But Doesn’t The Compiler Remove Unneeded Code?

```
$ cat > t.c
int foo(int bar) { return bar+2; }
int bar(int baz) { return baz*23; }
int main() { printf("%d\n",bar(17)); }
$ gcc -c t.c
$ nm t.o
0000000000000014 T bar
0000000000000000 T foo
0000000000000000 t gcc2_compiled.
0000000000000034 T main
                 U printf
$
```

But Doesn't The Linker Remove Unneeded Code?

```
$ cat a.c
char magic[]="bloat";
$ cat b.c
int main() { puts("hello, world"); }
$ gcc -o b a.c b.c
$ grep bloat b
Binary file b matches
$ gcc -o b b.c
$ grep bloat b
$
```

And When I Use Shared Libraries?

Shared libraries are the single technology that has contributed most to bloated software, because **programmers no longer see the actual size of their code.**

All the bloat is still there and wastes disk space.

Modern operating system support demand paging, i.e. only the used pages are actually loaded into memory. This gives bloat offenders a good conscience but in reality it still means that a lot of memory is wasted.

The page granularity is 4k. Smaller quantums of memory can not be not mapped because they are not needed. The data segment is coalesced!

And When I Use Static Libraries?

ld does not include unreferenced object files from static libraries.

That's why the diet libc is initially only a static library.

Prefer Specific Over Generic Routines

This advice is for when you use only a subset of the generic routine!

For temporary storage, use `alloca` instead of `malloc`.

Use strchr instead of strstr.

Use `qsort` and `bsearch` instead of using AVL trees for everything but very dynamic or huge sets.

Optimize For The Common Case, Not The Worst Case

Arrays are great! The code is easy to understand and less error prone than lists or trees.

Arrays can be resized easily (`realloc`), you can traverse an array without trashing the cache, you can free all elements by freeing the array (unless they contain pointers, of course).

Be Aware Of The Bloat You Generate

```
$ cat > t.cc
#include <string>
#include <iostream>
main() {
  string s="foo";
  cout << s << endl;
}
$ g++ -static -s -o t t.cc
$ du -H t
387k   t
$
```

Treat malloc As Expensive Operation

Malloc and friends are potentially very expensive.

```
$ cat > mymalloc.c
#include <stdio.h>
static int mallocs=0;
void sayit() { fprintf(stderr,"%d mallocs\n",mallocs); }
int malloc(unsigned long size) {
  if (!mallocs++) atexit(sayit);
  return __libc_malloc(size);
}
$ gcc -shared -o mymalloc.so mymalloc.c
$ LD_PRELOAD=$PWD/mymalloc.so grip
20794 mallocs
```

Bad API vs. Good API

Bad:

```
int fd=open("foo",O_WRONLY|O_CREAT,0600);
```

Good:

```
int fd=open_write("foo");
```

Bad API vs. Good API

Bad:

```
int i;
char buf[100];
sprintf(buf,"the number is %d",i);
```

Good:

```
stralloc s={0};
stralloc_copys(&s,"the number is ");
stralloc_catuint(&s,i);
```

Profile, Don’t Speculate

Trading space for performance is not bad per se!

Use `gprof` to find the code segments that are performance critical. Then only enable tradeoff optimizer options on those code, not the whole project.

Little known `gcc` option: `-Os` is like `-O2` but without trade-offs that waste space.

Other gcc options to save space: `-fomit-frame-pointer -mcpu=i386`

Use `nm` and `objdump` to look at the symbol table.

Inline Functions

```
$ cat > t.c
inline int foo() { return 3; }
int bar() { return foo(); }
$ gcc -O2 -c t.c
$ nm t.o
00000000 T bar
0000000c T foo
00000000 t gcc2_compiled.
$
```

Static Inline Functions

```
$ cat > t.c
static inline int foo() { return 3; }
int bar() { return foo(); }
$ gcc -O2 -c t.c
$ nm t.o
00000000 T bar
00000000 t gcc2_compiled.
$
```

Learn how to use `static` to save space (and keep the name space clean)!

Alloca

Alloca allocates storage **from the stack.** The storage is returned when the program leaves the scope (i.e. no need to free it explicitly).

Malloc needs to keep elaborate data structures and do locking to be thread-safe. Alloca decrements a register.

In C++, malloc() is often hidden by using the `new` operator. C++ is very dangerous because it hides complexity and bloat from the programmer.

Introducing libowfat

Why the name libowfat? Because you link it with "`-lowfat`"!

libowfat is a GPL reimplementation Dan Bernstein's internal helper functions.

Why reimplement them? His APIs are exemplary for good API design.

libowfat: byte.a

Return index, not pointer. Operand being operated on (i.e. destination, main operand) is #1, the rest are parameters.

```
int byte_chr(void*,int,char);
int byte_rchr(void*,int,char);
void byte_copy(void* out, int, void* in);
void byte_copyr(void* out, int, void* in);
int byte_diff(void*, int, void*);
void byte_zero(void*, int);
#define byte_equal(s,t) (!byte_diff((s),(t)))
```

libowfat: str.a

What did the people smoke who made `strcpy` and `strcat` return pointers?!

```
int str_copy(char* out,char* in);
int str_diff(char*,char*);
int str_diffn(char*,char*,int);
int str_len(char*);
int str_chr(char*,char);
int str_rchr(char*,char);
int str_start(char*,char*);
#define str_equal(s,t) (!str_diff((s),(t)))
```

libowfat: fmt.a

```
int fmt_long(char*,signed long);
int fmt_ulong(char*,unsigned long);
int fmt_xlong(char*,unsigned long);
int fmt_8long(char*,unsigned long);
int fmt_ulong0(char*,unsigned long,unsigned int);
int fmt_plusminus(char*,int );
int fmt_minus(char*,int);
int fmt_str(char*,char*);
int fmt_strn(char*,char*,int);
```

libowfat: scan.a

Pass NULL and you get just the length it would have taken.

```
int scan_long(char*,signed long*);
int scan_ulong(char*,unsigned long*);
int scan_xlong(char*,unsigned long*);
int scan_8long(char*,unsigned long*);
int scan_plusminus(char*,int*);
int scan_whitenskip(char*,int);
int scan_nonwhitenskip(char*,int);
int scan_charsetnskip(char*,int);
int scan_noncharsetnskip(char*,int);
```

libowfat: open.a

```
int open_read(const char *filename);
int open_excl(const char *filename);
int open_append(const char *filename);
int open_trunc(const char *filename);
int open_write(const char *filename);
```

libowfat: stralloc.a

```
int stralloc_ready(stralloc*,int);
int stralloc_readyplus(stralloc*,int);
int stralloc_copyb(stralloc*,char*,int);
int stralloc_copys(stralloc*,char*);
int stralloc_copy(stralloc*,stralloc*);
int stralloc_catb(stralloc*,char*,int);
int stralloc_cats(stralloc*,char*);
int stralloc_cat(stralloc*,stralloc*);
int stralloc_append(stralloc*,char*);
int stralloc_starts(stralloc*,char*);
int stralloc_catulong0(stralloc*,unsigned,int);
int stralloc_catlong0(stralloc*,signed,int);
void stralloc_free(stralloc* sa);
```

Duh!

Be aware of what code the compiler generates.

Use gprof and objdump.