The Lost Art of C Structure Packing

Eric S. Raymond

<esr@thyrsus.com>

Table of Contents

1. Who should read this

This page is about a technique for reducing the memory footprint of C programs - manually repacking C structure declarations for reduced size. To read it, you will require basic knowledge of the C programming language.

You need to know this technique if you intend to write code for memory-constrained embedded systems, or operating-system kernels. It is useful if you are working with application data sets so large that your programs routinely hit memory limits. It is good to know in any application where you really, *really* care about minimizing cache-line misses.

Finally, knowing this technique is a gateway to other esoteric C topics. You are not an advanced C programmer until you have grasped it. You are not a master of C until you could have written this document yourself and can criticize it intelligently.

2. Why I wrote it

This webpage exists because in late 2013 I found myself heavily applying a C optimization technique that I had learned more than two decades previously and not used much since.

I needed to reduce the memory footprint of a program that used thousands - sometimes hundreds of thousands - of C struct instances. The program was cvs-fast-export and the problem was that it was dying with out-of-memory errors on large repositories.

There are ways to reduce memory usage significantly in situations like this, by rearranging the order of structure members in careful ways. This can lead to dramatic gains - in my case I was able to cut the working-set size by around 40%, enabling the program to handle much larger repositories without dying.

But as I worked, and thought about what I was doing, it began to dawn on me that the technique I was using has been more than half forgotten in these latter days. A little web research confirmed that C programmers don't seem to talk about it much any more, at least not where a search engine can see them. A couple of Wikipedia entries touch the topic, but I found nobody who covered it comprehensively.

There are actually reasons for this that aren't stupid. CS courses (rightly) steer people away from

micro-optimization towards finding better algorithms. The plunging price of machine resources has made squeezing memory usage less necessary. And the way hackers used to learn how to do it back in the day was by bumping their noses on strange hardware architectures - a less common experience now.

But the technique still has value in important situations, and will as long as memory is finite. This document is intended to save C programmers from having to rediscover the technique, so they can concentrate effort on more important things.

3. Alignment requirements

The first thing to understand is that, on modern processors, the way your C compiler lays out basic C datatypes in memory is constrained in order to make memory accesses faster.

Storage for the basic C datatypes on an x86 or ARM processor doesn't normally start at arbitrary byte addresses in memory. Rather, each type except char has an *alignment requirement*; chars can start on any byte address, but 2-byte shorts must start on an even address, 4-byte ints or floats must start on an address divisible by 4, and 8-byte longs or doubles must start on an address divisible by 8. Signed or unsigned makes no difference.

The jargon for this is that basic C types on x86 and ARM are *self-aligned*. Pointers, whether 32-bit (4-byte) or 64-bit (8-byte) are self-aligned too.

Self-alignment makes access faster because it facilitates generating single-instruction fetches and puts of the typed data. Without alignment constraints, on the other hand, the code might end up having to do two or more accesses spanning machine-word boundaries. Characters are a special case; they're equally expensive from anywhere they live inside a single machine word. That's why they don't have a preferred alignment.

I said "on modern processors" because on some older ones forcing your C program to violate alignment

rules (say, by casting an odd address into an int pointer and trying to use it) didn't just slow your code down, it caused an illegal instruction fault. This was the behavior, for example, on Sun SPARC chips. In fact, with sufficient determination and the right (e18) hardware flag set on the processor, you can still trigger this on x86.

Also, self-alignment is not the only possible rule. Historically, some processors (especially those lacking barrel shifters) have had more restrictive ones. If you do embedded systems, you might trip over one of these lurking in the underbrush. Be aware this is possible.

4. Padding

Now we'll look at a simple example of variable layout in memory. Consider the following series of variable declarations in the top level of a C module:

```
char *p;
char c;
int x;
```

If you didn't know anything about data alignment, you might assume that these three variables would occupy a continuous span of bytes in memory. That is, on a 32-bit machine 4 bytes of pointer would be immediately followed by 1 byte of char and that immediately followed by 4 bytes of int. And a 64-bit machine would be different only in that the pointer would be 8 bytes.

Here's what actually happens (on an x86 or ARM or anything else with self-aligned types). The storage for `p` starts on a self-aligned 4- or 8-byte boundary depending on the machine word size. This is *pointer alignment* - the strictest possible.

The storage for `c` follows immediately. But the 4-byte alignment requirement of `x` forces a gap in the layout; it comes out as though there were a fourth intervening variable, like this:

```
char *p;        /* 4 or 8 bytes */
char c;         /* 1 byte */
char pad[3];    /* 3 bytes */
int x;          /* 4 bytes */
```

The `pad[3]` character array represents the fact that there are three bytes of waste space in the structure. The old-school term for this was "slop". The value of the padding bits is undefined; in particular it is not guaranteed that they will be zeroed.

Compare what happens if `x` is a 2-byte short:

```
char *p;
char c;
short x;
```

In that case, the actual layout will be this:

```
char *p;        /* 4 or 8 bytes */
char c;         /* 1 byte */
char pad[1];    /* 1 byte */
short x;        /* 2 bytes */
```

On the other hand, if `x` is a long on a 64-bit machine

```
char *p;
char c;
long x;
```

we end up with this:

```
char *p;      /* 8 bytes */
char c;       /* 1 byte
char pad[7]; /* 7 bytes */
long x;       /* 8 bytes */
```

If you have been following carefully, you are probably now wondering about the case where the shorter variable declaration comes first:

```
char c;
char *p;
int x;
```

If the actual memory layout were written like this

```
char c;
char pad1[M];
char *p;
char pad2[N];
int x;
```

what can we say about `M` and `N`?

First, in this case `N` will be zero. The address of `x`, coming right after `p`, is guaranteed to be pointer-aligned, which is never less strict than int-aligned.

The value of `M` is less predictable. If the compiler happened to map `c` to the last byte of a machine word, the next byte (the first of `p`) would be the first byte of the next one and properly pointer-aligned. M would be zero.

It is more likely that `c` will be mapped to the *first* byte of a machine word. In that case M will be whatever padding is needed to ensure that `p` has pointer alignment - 3 on a 32-bit machine, 7 on a 64-bit machine.

Intermediate cases are possible. M can be anything from 0 to 7 (0 to 3 on 32-bit) because a char can start on any byte boundary in a machine word.

If you wanted to make those variables take up less space, you could get that effect by swapping `x` with `c` in the original sequence.

```
char *p;      /* 8 bytes */
long x;       /* 8 bytes */
char c;       /* 1 byte
```

Usually, for the small number of scalar variables in your C programs, bumming out the few bytes you can get by changing the order of declaration won't save you enough to be significant. The technique becomes more interesting when applied to nonscalar variables - especially structs.

Before we get to those, let's dispose of arrays of scalars. On a platform with self-aligned types, arrays of char/short/int/long/pointer have no internal padding; each member is automatically self-aligned at the end of the next one.

In the next section we will see that the same is **not** necessarily true of structure arrays.

5. Structure alignment and padding

In general, a struct instance will have the alignment of its widest scalar member. Compilers do this as the easiest way to ensure that all the members are self-aligned for fast access.

Also, in C the address of a struct is the same as the address of its first member - there is no leading

padding. Beware: in C++, classes that look like structs may break this rule! (Whether they do or not depends on how base classes and virtual member functions are implemented, and varies by compiler.)

(When you're in doubt about this sort of thing, ANSI C provides an offsetof() macro which can be used to read out structure member offsets.)

Consider this struct:

```
struct foo1 {
    char *p;
    char c;
    long x;
};
```

Assuming a 64-bit machine, any instance of `struct foo1` will have 8-byte alignment. The memory layout of one of these looks unsurprising, like this:

```
struct foo1 {
    char *p;      /* 8 bytes */
    char c;       /* 1 byte
    char pad[7]; /* 7 bytes */
    long x;       /* 8 bytes */
};
```

It's laid out exactly as though variables of these types has been separately declared. But if we put `c` first, that's no longer true.

```
struct foo2 {
    char c;       /* 1 byte */
    char pad[7]; /* 7 bytes */
```

```
    char *p;      /* 8 bytes */
    long x;       /* 8 bytes */
};
```

If the members were separate variables, `c` could start at any byte boundary and the size of `pad` might vary. Because `struct foo2` has the pointer alignment of its widest member, that's no longer possible. Now `c` has to be pointer-aligned, and following padding of 7 bytes is locked in.

Now let's talk about trailing padding on structures. To explain this, I need to introduce a basic concept which I'll call the *stride address* of a structure. It is the first address following the structure data that has the **same alignment as the structure**.

The general rule of trailing structure padding is this: the compiler will behave as though the structure has trailing padding out to its stride address. This rule controls what `sizeof()` will return.

Consider this example on a 64-bit x86 or ARM machine:

```
struct foo3 {
    char *p;      /* 8 bytes */
    char c;       /* 1 byte */
};

struct foo3 singleton;
struct foo3 quad[4];
```

You might think that `sizeof(struct foo3)` should be 9, but it's actually 16. The stride address is that of `(&p)[2]`. Thus, in the `quad` array, each member has 7 bytes of trailing padding, because the first member of each following struct wants to be self-aligned on an 8-byte boundary. The memory layout is as though the structure had been declared like this:

```
struct foo3 {
    char *p;      /* 8 bytes */
    char c;       /* 1 byte */
    char pad[7];
};
```

For contrast, consider the following example:

```
struct foo4 {
    short s;      /* 2 bytes */
    char c;       /* 1 byte */
};
```

Because `s` only needs to be 2-byte aligned, the stride address is just one byte after `c`, and `struct foo4` as a whole only needs one byte of trailing padding. It will be laid out like this:

```
struct foo4 {
    short s;      /* 2 bytes */
    char c;       /* 1 byte */
    char pad[1];
};
```

and `sizeof(struct foo4)` will return 4.

Here's a last important detail: If your structure has structure members, the inner structs want to have the alignment of longest scalar too. Suppose you write this:

```
struct foo5 {
    char c;
    struct foo5 inner {
```

```
        char *p;
        short x;
    } inner;
};
```

The `char *p` member in the inner struct forces the outer struct to be pointer-aligned as well as the inner. Actual layout will be like this on a 64-bit machine:

```
struct foo5 {
    char c;             /* 1 byte*/
    char pad1[7];       /* 7 bytes */
    struct foo5_inner {
        char *p;        /* 8 bytes */
        short x;        /* 2 bytes */
        char pad2[6]; /* 6 bytes */
    } inner;
};
```

This structure gives us a hint of the savings that might be possible from repacking structures. Of 24 bytes, 13 of them are padding. That's more than 50% waste space!

6. Bitfields

Now let's consider bitfields. What they give you the ability to do is declare structure fields of smaller than character width, down to 1 bit, like this:

```
struct foo6 {
    short s;
    char c;
    int flip:1;
```

```
    int nybble:4;
    int septet:7;
};
```

The thing to know about bitfields is that they are implemented with word- and byte-level mask and rotate instructions operating on machine words, and cannot cross word boundaries. C99 guarentees that bit-fields will be packed as tightly as possible, provided they don't cross storage unit boundaries (6.7.2.1 #10).

Assuming we're on a 32-bit machine, that implies that the layout may look like this:

```
struct foo6 {
    short s;        /* 2 bytes */
    char c;         /* 1 byte */
    int flip:1;     /* total 1 bit */
    int nybble:4;   /* total 5 bits */
    int pad1:3;     /* pad to an 8-bit boundary */
    int septet:7;   /* 7 bits */
    int pad2:25;    /* pad to 32 bits */
};
```

But this isn't the only possibility, because the C standard does not specify that bits are allocated low-to-high. So the layout could look like this:

```
struct foo6 {
    short s;        /* 2 bytes */
    char c;         /* 1 byte */
    int pad1:3;     /* pad to an 8-bit boundary */
    int flip:1;     /* total 1 bit */
    int nybble:4;   /* total 5 bits */
    int pad2:25;    /* pad to 32 bits */
```

```
    int septet:7;  /* 7 bits */
};
```

That is, the padding could precede rather than following the payload bits.

Note also that, as with normal structure padding, the padding bits are not guaranteed to be zero; C99 mentions this.

Note that the base type of a bit field is interpreted for signedness but not necessarily for size. It is up to implementors whether "short flip:1" or "long flip:1" are supported, and whether those base types change the size of the storage unit the field is packed into.

Proceed with caution and check with -Wpadded if you have it available (e.g. under clang). Compilers on exotic hardware, might interpret the C99 rules in surprising ways; older compilers might not quite follow them.

The restriction that bitfields cannot cross machine word boundaries means that, while the first two of the following structures pack into one and two 32-bit words as you'd expect, the third (`struct foo9`) takes up **three** 32-bit words, in the last of which only one bit is used.

```
struct foo7 {
    int bigfield:31;       /* 32-bit word 1 begins */
    int littlefield:1;
};

struct foo8 {
    int bigfield1:31;      /* 32-bit word 1 begins /*
    int littlefield1:1;
    int bigfield2:31;      /* 32-bit word 2 begins */
    int littlefield2:1;
};
```

```
struct foo9 {
    int bigfield1:31;     /* 32-bit word 1 begins */
    int bigfield2:31;     /* 32-bit word 2 begins */
    int littlefield1:1;
    int littlefield2:1;   /* 32-bit word 3 begins */
};
```

On the other hand, `struct foo8` would fit into a single 64-bit word if the machine has those.

7. Structure reordering

Now that you know how and why compilers insert padding in and after your structures we'll examine what you can do to squeeze out the slop. This is the art of structure packing.

The first thing to notice is that slop only happens in two places. One is where storage bound to a larger data type (with stricter alignment requirements) follows storage bound to a smaller one. The other is where a struct naturally ends before its stride address, requiring padding so the next one will be properly aligned.

The simplest way to eliminate slop is to reorder the structure members by decreasing alignment. That is: make all the pointer-aligned subfields come first, because on a 64-bit machine they will be 8 bytes. Then the 4-byte ints; then the 2-byte shorts; then the character fields.

So, for example, consider this simple linked-list structure:

```
struct foo10 {
    char c;
    struct foo10 *p;
    short x;
};
```

With the implied slop made explicit, here it is:

```
struct foo10 {
    char c;           /* 1 byte */
    char pad1[7];     /* 7 bytes */
    struct foo10 *p; /* 8 bytes */
    short x;          /* 2 bytes */
    char pad2[6];     /* 6 bytes */
};
```

That's 24 bytes. If we reorder by size, we get this:

```
struct foo11 {
    struct foo11 *p;
    short x;
    char c;
};
```

Considering self-alignment, we see that none of the data fields need padding. This is because the stride address for a (longer) field with stricter alignment is always a validly-aligned start address for a (shorter) field with less strict requirements. All the repacked struct actually requires is trailing padding:

```
struct foo11 {
    struct foo11 *p; /* 8 bytes */
    short x;         /* 2 bytes */
    char c;          /* 1 byte */
    char pad[5];     /* 5 bytes */
};
```

Our repack transformation drops the size from 24 to 16 bytes. This might not seem like a lot, but

suppose you have a linked list of 200K of these? The savings add up fast - especially on memory-constrained embedded systems or in the core part of an OS kernel that has to stay resident.

Note that reordering is not guaranteed to produce savings. Applying this technique to an earlier example, `struct foo9`, we get this:

```
struct foo12 {
    struct foo12_inner {
        char *p;       /* 8 bytes */
        int x;         /* 4 bytes */
    } inner;
    char c;            /* 1 byte*/
};
```

With padding written out, this is

```
struct foo12 {
    struct foo12_inner {
        char *p;       /* 8 bytes */
        int x;         /* 4 bytes */
        char pad[4];   /* 4 bytes */
    } inner;
    char c;            /* 1 byte*/
    char pad[7];       /* 7 bytes */
};
```

It's still 24 bytes because `c` cannot back into the inner struct's trailing padding. To collect that gain you would need to redesign your data structures.

Since shipping the first version of this guide I have been asked why, if reordering for minimal slop is so simple, C compilers don't do it automatically. The answer: C is a language originally designed for writing

operating systems and other code close to the hardware. Automatic reordering would interfere with a systems programmer's ability to lay out structures that exactly match the byte and bit-level layout of memory-mapped device control blocks.

8. Awkward scalar cases

Using enumerated types instead of #defines is a good idea, if only because symbolic debuggers have those symbols available and can show them rather than raw integers. But, while enums are guaranteed to be compatible with an integral type, the C standard does not specify *which* underlying integral type is to be used for them.

Be aware when repacking your structs that while enumerated-type variables are usually ints, this is compiler-dependent; they could be shorts, longs, or even chars by default. Your compiler may have a pragma or command-line option to force the size.

The `long double` type is a similar trouble spot. Some C platforms implement this in 80 bits, some in 128, and some of the 80-bit platforms pad it to 96 or 128 bits.

In both cases it's best to use `sizeof()` to check the storage size.

Finally, under x86 Linux doubles are sometimes an exception to the self-alignment rule; an 8-byte double may require only 4-byte alignment within a struct even though standalone doubles variables have 8-byte self-alignment. This depends on compiler and options.

9. Readability and cache locality

While reordering by size is the simplest way to eliminate slop, it's not necessarily the right thing. There are two more issues: readability and cache locality.

Programs are not just communications to a computer, they are communications to other human beings. Code readability is important even (or especially!) when the audience of the communication is only your future self.

A clumsy, mechanical reordering of your structure can harm readability. When possible, it is better to reorder fields so they remain in coherent groups with semantically related pieces of data kept close together. Ideally, the design of your structure should communicate the design of your program.

When your program frequently accesses a structure, or parts of a structure, it is helpful for performance if the accesses tend to fit within a *cache line* - the memory block fetched by your processor when it is told to get any single address within the block. On 64-bit x86 a cache line is 64 bytes beginning on a self-aligned address; on other platforms it is often 32 bytes.

The things you should do to preserve readability - grouping related and co-accessed data in adjacent fields - also improve cache-line locality. These are both reasons to reorder intelligently, with awareness of your code's data-access patterns.

If your code does concurrent access to a structure from multiple threads, there's a third issue: cache line bouncing. To minimize expensive bus traffic, you should arrange your data so that reads come from one cache line and writes go to another in your tighter loops.

And yes, this sometimes contradicts the previous guidance about grouping related data in the same cache-line-sized block. Multithreading is hard. Cache-line bouncing and other multithread optimization issues are very advanced topics which deserve an entire tutorial of their own. The best I can do here is make you aware that these issues exist.

10. Other packing techniques

Reordering works best when combined with other techniques for slimming your structures. If you have several boolean flags in a struct, for example, consider reducing them to 1-bit bitfields and packing them

into a place in the structure that would otherwise be slop.

You'll take a small access-time penalty for this - but if it squeezes the working set enough smaller, that penalty will be swamped by your gains from avoided cache misses.

More generally, look for ways to shorten data field sizes. In cvs-fast-export, for example, one squeeze I applied was to use the knowledge that RCS and CVS repositories didn't exist before 1982. I dropped a 64-bit Unix `time_t` (zero date at the beginning of 1970) for a 32-bit time offset from 1982-01-01T00:00:00; this will cover dates to 2118. (Note: if you pull a trick like this, *do a bounds check whenever you set the field* to prevent nasty bugs!)

Each such field shortening not only decreases the explicit size of your structure, it may remove slop and/or create additional opportunities for gains from field reordering. Virtuous cascades of such effects are not very hard to trigger.

The riskiest form of packing is to use unions. If you know that certain fields in your structure are never used in combination with certain other fields, consider using a union to make them share storage. But be extra careful and verify your work with regression testing, because if your lifetime analysis is even slightly wrong you will get bugs ranging from crashes to (much worse) subtle data corruption.

11. Overriding alignment rules

Sometimes you can coerce your compiler into not using the processor's normal alignment rules by using a pragma, usually `#pragma pack`. GCC and clang have an *attributepacked* you can attach to individual structure declarations; GCC has an -fpack-struct option for entire compilations.

Do not do this casually, as it forces the generation of more expensive and slower code. Usually you can save as much memory, or almost as much, with the techniques I describe here.

The only good reason for `#pragma pack` is if you have to exactly match your C data layout to some

kind of bit-level hardware or protocol requirement, like a memory-mapped hardware port, and violating normal alignment is required for that to work. If you're in that situation, and you don't already know everything else I'm writing about here, you're in deep trouble and I wish you luck.

12. Tools

The clang compiler has a -Wpadded option that causes it to generate messages about alignment holes and padding. Some versions also have an undocumented -fdump-record-layouts option that yields more information.

I have not used it myself, but several respondents speak well of a program called `pahole`. This tool cooperates with a compiler to produce reports on your structures that describe padding, alignment, and cache line boundaries.

I've received a report that a proprietary code auding tool called "PVS Studio" can detect structure-packing opportunities.

13. Proof and exceptional cases

You can download sourcecode for a little program that demonstrates the assertions about scalar and structure sizes made above. It is packtest.c.

If you look through enough strange combinations of compilers, options, and unusual hardware, you will find exceptions to some of the rules I have described. They get more common as you go back in time to older processor designs.

The next level beyond knowing these rules is knowing how and when to expect that they will be broken. In the years when I learned them (the early 1980s) we spoke of people who didn't get this as victims of "all-the-world's-a-VAX syndrome". Remember that not all the world is a PC.

14. Related Reading

This section exists to collect pointers to essays on other advanced C topics which I judge to be good companions to this one.

A Guide to Undefined Behavior in C and C++

Time, Clock, and Calendar Programming In C

15. Version history

1.14 @ 2015-12-19
: Typo correction: -Wpadding → -Wpadded.

1.13 @ 2015-11-23
: Be explicit about padding bits being undefined. More about bitfields.

1.12 @ 2015-11-11
: Major revision of section on bitfields reflecting C99 rules.

1.11 @ 2015-07-23
: Mention the clang -fdump-record-layouts option.

1.10 @ 2015-02-20
: Mention *attributepacked*, -fpack-struct, and PVS Studio.

1.9 @ 2014-10-01
: Added link to "Time, Clock, and Calendar Programming In C".

1.8 @ 2014-05-20
: Improved explanation for the bitfield examples,

1.7 @ 2014-05-17

Correct a minor error in the description of the layout of `struct foo8`.

1.6 @ 2014-05-14

Emphasize that bitfields cannot cross word boundaries. Idea from Dale Gulledge.

1.5 @ 2014-01-13

Explain why structure member reordering is not done automatically.

1.4 @ 2014-01-04

A note about double under x86 Linux.

1.3 @ 2014-01-03

New sections on awkward scalar cases, readability and cache locality, and tools.

1.2 @ 2014-01-02

Correct an erroneous address calculation.

1.1 @ 2014-01-01

Explain why aligned accesses are faster. Mention offsetof. Various minor fixes, including the packtest.c download link.

1.0 @ 2014-01-01

Initial release.