# What are some best practices for reducing memory usage in C?

Asked 15 years, 11 months ago  Modified 3 years, 8 months ago

Viewed 32k times

43

What are some best practice for "Memory Efficient C programming". Mostly for embedded/mobile device what should be the guidelines for having low memory consumptions ?

I guess there should be separate guideline for a) code memory b) data memory

c  optimization  memory-management

Share

Improve this question

Follow

edited Apr 8, 2014 at 11:26

Sebastian Mach

39.1k ● 8 ● 96 ● 130

asked Jan 1, 2009 at 6:05



user27424

1,381 ● 2 ● 12 ● 12

Nice to see such a variety of quite different good answers rather than repitions. – SmacL Jan 1, 2009 at 11:17

1 Let me refine your title. "memory efficient" could also mean read and/or write speed. – Sebastian Mach Apr 8, 2014 at

11:25

## 16 Answers

Sorted by: Highest score (default)

29

In C, at a much simpler level, consider the following;

- Use #pragma pack(1) to byte align your structures
- Use unions where a structure can contain different types of data
- Use bit fields rather than ints to store flags and small integers
- Avoid using fixed length character arrays to store strings, implement a string pool and use pointers.
- Where storing references to an enumerated string list, e.g. font name, store an index into the list rather than the string
- When using dynamic memory allocation, compute the number of elements required in advance to avoid reallocs.

Share Improve this answer

Follow

answered Jan 1, 2009 at 9:56

SmacL

22.9k ● 15 ● 99 ● 151

Ah, completely forgot about the first three points myself. I thought it was too obvious and overlooked it. xD – strager Jan 1, 2009 at 19:04

Yep. The above answer was added after having read yours,nimrodms & monjardins, and was there to add all the

more obvious ommissions. – SmacL Jan 2, 2009 at 9:01

1. That may come with a speed penalty. Better: structure your structs intelligently: struct s { int i; char c; } instead of struct s {char c; int i;} 2. Unions are very good. 3. bitfields can be painfully slow. Beware – Mikeage Feb 23, 2009 at 4:51

1 @Mikeage - Yup, I find that typically you are either optimizing for either memory or speed, where the two are in opposition. – SmacL Feb 23, 2009 at 7:35

most embedded systems have instructions for bit operations, so bitfield is a good choice – phuclv Apr 16, 2014 at 13:49

22

A few suggestions that I've found useful in working with embedded systems:

- Ensure any lookup tables or other constant data are actually declared using `const`. If `const` is used then the data can be stored in read-only (e.g, flash or EEPROM) memory, otherwise the data has to be copied to RAM at start-up, and this takes up both flash and RAM space. Set the linker options so that it generates a map file, and study this file to see exactly where your data is being allocated in the memory map.
- Make sure you're using all the memory regions available to you. For example, microcontrollers often have onboard memory that you can make use of (which may also be faster to access than external RAM). You should be able to control the memory

regions to which code and data are allocated using compiler and linker option settings.

- To reduce code size, check your compiler's optimisation settings. Most compilers have switches to optimise for speed or code size. It can be worth experimenting with these options to see whether the size of the compiled code can be reduced. And obviously, eliminate duplicate code wherever possible.
- Check how much stack memory your system needs and adjust the linker memory allocation accordingly (see the answers to this question). To reduce stack usage, avoid placing large data structures on the stack (for whatever value of "large" is relevant to you).

Share Improve this answer

Follow

edited May 23, 2017 at 12:09

Community Bot

1 ● 1

answered Jan 1, 2009 at 14:40

ChrisN

16.9k ● 10 ● 60 ● 82

16

Make sure you're using fixed point / integer math wherever possible. Lots of developers use floating-point math (along with the sluggish performance and large libraries & memory usage) when simple scaled integer math will suffice.

Share Improve this answer

Follow

answered Jan 1, 2009 at 19:05

Dan

10.4k ● 5 ● 38 ● 53

This assumes the system lacks an FPU, and floating-point operations need to be emulated. Still, floats and doubles take more memory than fixed-point integers in most cases, so it does help to use the latter. – strager Jan 1, 2009 at 19:07

4 Most systems lack FPU. Integers rule! – jakobengblom2 Jan 9, 2009 at 8:14

9

All good recommendations. Here are some design approaches that I've found useful.

- Byte Coding

Write an interpreter for a special-purpose byte-code instruction set, and write as much of the program as possible in that instruction set. If certain operations require high performance, make them native-code and call them from the interpreter.

- Code Generation

If part of the input data changes very infrequently, you could have an external code generator that creates an ad-hoc program. That will be smaller than a more general program, as well as running faster and not having to allocate storage for the seldom-changing input.

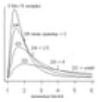
- Be a data-hater

Be willing to waste lots of cycles if it will let you store absolutely minimum data structure. Usually you will find that performance suffers very little.

Share Improve this answer Follow

edited Jan 27, 2009 at 22:41

answered Jan 27, 2009 at 22:36



Mike Dunlavey

**40.6k** ● 15 ● 94 ● 138

What do you mean by "cycles" in this case? – Someone Jan 20, 2021 at 13:24

@Someone: Machine cycles, instruction cycles, the little steps that machines do that take time. If you've got a 1-gigahertz machine, each instruction takes a number of cycles, like maybe 10 on average. So a program that takes 100 million instructions will take 1 billion cycles, which is one second. That sounds like a big number, but in typical modern software it's not big at all. – Mike Dunlavey Jan 20, 2021 at 20:50

8

Most likely you will need to chose your algorithms carefully. Aim for algorithms which have O(1) or O(log n) memory usage (i.e. low). For example, continuous resizable arrays (e.g. `std::vector`) in most cases require less memory than linked lists.

Sometimes, using look-up tables may be more benefitial to both code size *and* speed. If you only need 64 entries

in a LUT, that's 16*4 bytes for sin/cos/tan (use symmetry!) compared to a large sin/cos/tan function.

Compression sometimes helps. Simple algorithms like RLE are easy to compress/decompress when read sequentially.

If you're dealing with graphics or audio, consider different formats. Paletted or bitpacked* graphics may be a good tradeoff for quality, and palettes can be shared across many images, further reducing data size. Audio may be reduced from 16-bit to 8-bit or even 4-bit, and stereo can be converted to mono. Sampling rates can be reduced from 44.1KHz to 22kHz or 11kHz. These audio transformations greatly reduce their data size (and, sadly, quality) and are trivial (except resampling, but that's what audio software is for =]).

* I guess you could put this under compression. Bitpacking for graphics usually refers to reducing the number of bits per channel so each pixel can fit in two bytes (RGB565 or ARGB155 for example) or one (ARGB232 or RGB332) from the original three or four (RGB888 or ARGB8888, respectively).



edited Jan 1, 2009 at 6:50

answered Jan 1, 2009 at 6:41

strager
89.9k • 27 • 138 • 179

8

Avoid memory fragmentation by using your own memory allocator (or carefully using the system's allocator).

One method is to use a 'slab allocator' (see this article for example) and multiple pools of memory for objects of different sizes.

Share Improve this answer

Follow

answered Jan 1, 2009 at 7:52

nimrodm

**23.7k** ● 7 ● 61 ● 62

1 +1. This was a winning factor when we needed to get a full HTML 4 browser working in 8mb of RAM. – JBRWilkinson Sep 23, 2012 at 23:38

wow… how!? 8mb is TINY by modern standards! – Someone Jan 20, 2021 at 13:27

7

Pre-allocating all memory upfront (i.e. no malloc calls except for startup initialization) is definitely helpful for deterministic memory usage. Otherwise, different architectures provide techniques to help out. For example, certain ARM processors provide an alternative instruction set (Thumb) that almost halves code size by using 16-bit instructions instead of the normal 32 bits. Of course, speed is sacrificed in doing so...

Share Improve this answer

Follow

answered Jan 1, 2009 at 6:10

Judge Maygarden

27.5k • 9 • 83 • 100

For complicated algoriths, ARM mode may reduce code size. For simple operations, THUMB mode may be better. It all depends on the context, really. But for general code, I agree that THUMB is smaller than ARM. – strager Jan 1, 2009 at 6:37

Pre-allocating works well for speed and size of certain applications. For others, it can worsen size requirements, as the app cannot typically re-use memory across modules. – SmacL Jan 1, 2009 at 11:15

6

Some parsing operations can be performed on streams as the bytes arrive rather than copy into buffer and parse.

Some examples of this:

1. Parse a NMEA steam with a state machine, collecting only the required fields into a much more efficient struct.
2. Parse XML using SAX instead of DOM.

Share Improve this answer

Follow

edited Jan 1, 2009 at 19:05

strager

89.9k • 27 • 138 • 179

answered Jan 1, 2009 at 14:08

JeffV

54.4k • 33 • 105 • 124

+1 for avoid DOM. This is even true on desktop apps with large serialized data. – SmacL Jan 2, 2009 at 8:51

1 Avoid text, use binary. XML is even worse than text. There's a reason why our predecessors came up with ASN.1 and the likes, and it wasn't ease of use. – MSalters Jan 9, 2009 at 8:59

5

1) Before you start the project, build in a way of measuring how much memory you're using, preferably on a per-component basis. That way, each time you make a change you can see its effects on memory use. You can't optimise what you can't measure.

2) If the project is already mature and hits the memory limits, (or is ported to a device with less memory), find out what you're using the memory for already.

My experience has been that almost all the significant optimisation when fixing an over-size application comes from a small number of changes: reduce cache sizes, strip out some textures (of course this is a functional change requiring stakeholder agreement, i.e. meetings, so may not be efficient in terms of your time), resample audio, reduce the upfront size of custom-allocated heaps, find ways to free resources that are only used temporarily and reload them when required again. Occasionally you'll find some structure which is 64 bytes that could be reduced to 16, or whatever, but that's rarely the lowest-hanging fruit. If you know what the biggest lists and

arrays in the app are, though, then you know which structs to look at first.

Oh yes: find and fix memory leaks. Any memory you can regain without sacrificing performance is a great start.

I've spent a *lot* of time in the past worrying about code size. Main considerations (aside from: make sure you measure it at build time so that you can see it change), are:

1) Find out what code is referenced, and by what. If you discover that an entire XML library is being linked into your app just to parse a two-element config file, consider changing the config file format and/or writing your own trivial parser. If you can, use either source or binary analysis to draw a big dependency graph, and look for large components with only a small number of users: it may be possible to snip these out with only minor code rewrites. Be prepared to play diplomat: if two different components in your app use XML, and you want to cut it, then that's two people you have to convince of the benefits of hand-rolling something that's currently a trusted, off-the-shelf library.

2) Mess around with the compiler options. Consult your platform-specific documentation. For instance, you may want to reduce the default acceptable code size increase due to inlining, and on GCC at least you can tell the compiler only to apply optimisations which don't typically increase code size.

3) Take advantage of libraries already on the target platform(s) where possible, even if it means writing an adaptor layer. In the XML example above, you may find that on your target platform there's an XML library in memory at all times anyway, because the OS uses it, in which case link to it dynamically.

4) As someone else mentioned, thumb mode can help on ARM. If you only use it for the code which is not performance critical, and leave the critical routines in ARM, then you won't notice the difference.

Finally, there may be clever tricks you can play if you have sufficient control over the device. The UI only allows one application to run at a time? Unload all the drivers and services your app doesn't need. Screen is double buffered, but your app is synched to the refresh cycle? You may be able to reclaim an entire screen buffer.

Share Improve this answer Follow

edited Jan 1, 2009 at 14:35

answered Jan 1, 2009 at 14:18

Steve Jessop

279k ● 40 ● 469 ● 709

---

3

Recommend this book [Small Memory Software: Patterns for systems with limited memory]

Share Improve this answer

answered May 18, 2009 at 6:34

Follow

kcwu

**7,011** 4 29 32

2

- Reduce the length of and eliminate as many string constants as possible to reduce code space
- Carefully consider the tradeoffs of algorithms versus lookup tables where needed
- Be aware of how different kinds of variables are allocated.
    - Constants are probably in code space.
    - Static variables are probably at fixed memory locations. Avoid these if possible.
    - Parameters are probably stored on the stack or in registers.
    - Local variables may also be allocated from the stack. Don't declare large local arrays or strings if the code might run out of stack space under worst case conditions.
    - You may not have a heap - there may not be an OS to manage the heap for you. Is that acceptable? Do you need a malloc() function?

Share Improve this answer

Follow

answered Sep 7, 2009 at 16:31

Ben Gartner

**14.9k** ● 10 ● 38 ● 36

---

1

I have a presentation from the Embedded Systems Conference available on this topic. It is from 2001, but still it is very applicable. See paper .

Also, if you can choose the architecture of the target device, going with something like a modern ARM with Thumb V2, or a PowerPC with VLE, or MIPS with MIPS16, or selecting known-compact targets like Infineon TriCore or the SH family is a very good option. Not to mention the NEC V850E family that is nicely compact. Or move to an AVR which has excellent code compactness (but is an 8-bit machine). Anything but a fixed-length 32-bit RISC is a good choice!

Share Improve this answer

Follow

answered Jan 9, 2009 at 8:18

jakobengblom2

**5,843** ● 2 ● 28 ● 34

---

**1**

In addition to the suggestions other have given, remember that local variables declared in your functions will normally be allocated on the stack.

If stack memory is limited or you want to reduce the size of the stack to make room for more heap or global RAM, then consider the following:

1. *Flatten your call tree* to reduce the number of variables on the stack at any given time.
2. *Convert large local variables to globals* (decreases the amount of stack used, but increases the amount of global RAM used). Variables can be declared:
    - Global scope: Visible to entire program
    - Static at file scope: Visible in the same file

- Static at function scope: Visible within the function
- **NOTE:** Regardless, if theses changes are made, you must be wary of issues with `reentrant` code if you have a `preemptive` environment.

Many embedded systems do not have a stack monitor diagnostic to insure that a stack overflow is caught, so some analysis is required.

PS: Bonus for appropriate use of *Stack Overflow*?

Share Improve this answer

Follow

answered Sep 7, 2009 at 17:12

Tim Henigan

62.1k ● 11 ● 86 ● 79

---

1

Another useful trick can be the use of #define instead of variables whose values are not meant to change. For example: if something is defined as const, replace it with #define as these are decoded at preprocessor stage and do not occupy space. Any other datatype defined will need space in memory. This will help reduce the memory usage. This is applicable only if the type of variable is not of concern as #define will not take care of it.

Share Improve this answer

Follow

answered Apr 7, 2021 at 13:02

Apoorva R

39 ● 6

0

One trick that is useful in applications is to create a rainy day fund of memory. Allocate single block on startup that is big enough that it will suffice for cleanup tasks. If malloc/new fail, release the rainy day fund and post a message letting the user know that resources are tight and they should save and quite soon. This was a technique used in many Mac applications circa 1990.

Share Improve this answer

Follow

answered Jan 1, 2009 at 14:30

plinth

**49.1k** ● 11 ● 83 ● 123

0

A great way of constraining memory requirements is to rely as much as possible on libc or other standard libraries that can be linked dynamically. Every additional DLL or shared object you have to include in your project is a significant chunk of memory you may be able to avoid burning.

Also, make use of unions and bit fields, where applicable, only load the part of your data your program is working on in memory, and make sure you're compiling with the -Os (in gcc; or your compiler's equivalent) switch to optimize for program size.

Share Improve this answer

Follow

answered Jan 2, 2009 at 7:27



Max

**1,044** ● 10 ● 19