How to C (as of 2016)

How to C in 2016

This is a draft I wrote in early 2015 and never got around to publishing. Here's the mostly unpolished version because it wasn't doing anybody any good sitting in my drafts folder. The simplest change was updating year 2015 to 2016 at publication time.

(Update: Many people have submitted revisions, notes, and improvements. All contributions have been incorporated throughout the page below.)

Feel free to submit fixes/improvements/complaints as necessary. -Matt

Adrián Arroyo Calle provides a Spanish translation at ¿Cómo programar en C (en 2016)?

Japanese POSTD provides a Japanese translation at 2016年、C言語はどう書くべきか (前編) and 2016年、C言語はどう書くべきか (後編).

Chinese InfoQ provides a Chinese translation at C语言的2016.

Programmer Magazine provides a Chinese translation at 2016年，C语言该怎样写 (as PDF too).

Keith Thompson provides a nice set of corrections and alternative opinions at howto-c-response.

Rob Graham provides a response covering other avenues out of scope here at Some notes C in 2016.

Nick Galbreath collected some relevant links about overall C programming as well.

Now on to the article...

The first rule of C is don't write C if you can avoid it.

If you must write in C, you should follow modern rules.

C has been around since the early 1970s. People have "learned C" at various points during its evolution, but knowledge usually get stuck after learning, so everybody has a different set of things they believe about C based on the year(s) they first started learning.

It's important to not remain stuck in your "things I learned in the 80s/90s" mindset of C development.

This page assumes you are on a modern platform conforming to modern standards and you have no excessive legacy compatibility requirements. We shouldn't be globally tied to ancient standards just because some companies refuse to upgrade 20 year old systems.

Preflight

Standard c99 (c99 means "C Standard from 1999"; c11 means "C Standard from 2011", so 11 > 99).

- clang, default
  - clang uses an extended version of C11 by default (`GNU C11 mode`), so no extra options are needed for modern features.
  - If you want standard C11, you need to specify `-std=c11`; if you want standard C99, use `-std=c99`.
  - clang compiles your source files faster than gcc
- gcc requires you specify `-std=c99` or `-std=c11`
  - gcc builds source files slower than clang, but *sometimes* generates faster code. Performance comparisons and regression testings are important.
  - gcc-5 defaults to `GNU C11 mode` (same as clang), but if you need exactly c11 or c99, you should still specify `-std=c11` or `-std=c99`.

Optimizations

- -O2, -O3
  - generally you want `-O2`, but sometimes you want `-O3`. Test under both levels (and across compilers) then keep the best performing binaries.
- -Os
  - `-Os` helps if your concern is cache efficiency (which it should be)

Warnings

- `-Wall -Wextra -pedantic`
  - newer compiler versions have `-Wpedantic`, but they still accept the ancient `-pedantic` as well for wider backwards compatibility.
- during testing you should add `-Werror` and `-Wshadow` on all your platforms
  - it can be tricky deploying production source using `-Werror` because different platforms and compilers and libraries can emit different warnings. You probably don't want to kill a user's entire build just because their version of GCC on a platform you've never seen complains in new and wonderous ways.
- extra fancy options include `-Wstrict-overflow -fno-strict-aliasing`

    - Either specify `-fno-strict-aliasing` or be sure to only access objects as the type they have at creation. Since so much existing C code aliases across types, using `-fno-strict-aliasing` is a much safer bet if you don't control the entire underlying source tree.
- as of now, Clang reports some valid syntax as a warning, so you should add `-Wno-missing-field-initializers`
    - GCC fixed this unnecessary warning after GCC 4.7.0

Building

- Compilation units
    - The most common way of building C projects is to decompose every source file into an object file then link all the objects together at the end. This procedure works great for incremental development, but it is suboptimal for performance and optimization. Your compiler can't detect potential optimizations across file boundaries this way.
- LTO — Link Time Optimization
    - LTO fixes the "source analysis and optimization across compilation units problem" by annotating object files with intermediate representation so source-aware optimizations can be carried out across compilation units at link time.
    - LTO can slow down the linking process noticeably, but `make -j` helps if your build includes multiple non-interdependent final targets (.a, .so, .dylib, testing executables, application executables, etc).
    - clang LTO (guide)
    - gcc LTO
    - As of 2016, clang and gcc releases support LTO by just adding `-flto` to your command line options during object compilation and final library/program linking.
    - `LTO` still needs some babysitting though. Sometimes, if your program has code not used directly but used by additional libraries, LTO can evict functions or code because it detects, globally when linking, some code is unused/unreachable and doesn't *need* to be included in the final linked result.

Arch

- `-march=native`
    - give the compiler permission to use your CPU's full feature set
    - again, performance testing and regression testing is important (then comparing the results across multiple compilers and/or compiler versions) is important to make sure any enabled optimizations don't have adverse side effects.
- `-msse2` and `-msse4.2` may be useful if you need to target not-your-build-machine features.

Writing code

Types

If you find yourself typing `char` or `int` or `short` or `long` or `unsigned` into new code, you're doing it wrong.

For modern programs, you should `#include <stdint.h>` then use *standard* types.

For more details, see the stdint.h specification.

The common standard types are:

- `int8_t`, `int16_t`, `int32_t`, `int64_t` — signed integers
- `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t` — unsigned integers
- `float` — standard 32-bit floating point
- `double` - standard 64-bit floating point

Notice we don't have `char` anymore. `char` is actually misnamed and misused in C.

Developers routinely abuse `char` to mean "byte" even when they are doing unsigned byte manipulations. It's much cleaner to use `uint8_t` to mean single a unsigned-byte/octet-value and `uint8_t *` to mean sequence-of-unsigned-byte/octet-values.

Special Standard Types

In addition to standard fixed-width like `uint16_t` and `int32_t`, we also have **fast** and **least** types defined in the stdint.h specification.

Fast types are:

- `int_fast8_t`, `int_fast16_t`, `int_fast32_t`, `int_fast64_t` — signed integers
- `uint_fast8_t`, `uint_fast16_t`, `uint_fast32_t`, `uint_fast64_t` — unsigned integers

Fast types provide a minimum of `X` bits, but there is no guarantee the underlying storage size is exactly what you request. If a larger type has better support on your target platform, a *fast* type will automatically use the better supported larger type.

The best example here is, on some 64-bit systems, when you request `uint_fast16_t` you actually get a `uint64_t` because operating on word-sized integers will be faster than operating on half of a 32-bit integer.

The *fast* guidelines aren't followed on every system though. One standout is OS X, where *fast* types are defined exactly as their corresponding fixed width counterparts.

Fast types can be useful for self-documenting code as well. If you know your counters only need 16 bits, but you prefer your math use 64 bit integers because they are faster on your platform, that's where `uint_fast16_t` would help. Under 64-bit Linux platforms, `uint_fast16_t` gives you a fast 64-bit counter while maintaining the code-level inline documentation of "we only need 16 bits here."

One thing to be aware of for fast types: it can impact certain test cases. If you need to test for storage width edge cases, having `uint_fast16_t` be 16 bits on some platforms (OS X) and 64 bits on other platforms (Linux) can increase the minimum number of platforms where your tests need to pass.

Fast types do introduce the same uncertainty as `int` not being a standard size across platforms, but with *fast* types, you can limit your uncertainty to known-safe locations in your code (counters, temporary values with checked bounds, etc).

Least types are:

- `int_least8_t`, `int_least16_t`, `int_least32_t`, `int_least64_t` — signed integers
- `uint_least8_t`, `uint_least16_t`, `uint_least32_t`, `uint_least64_t` — unsigned integers

Least types provide you with the most *compact* number of bits for the type you request.

The *least* guidelines, in practice, mean *least* types are just defined to standard fixed width types, since standard fixed width types already provide the exact minimum number of bits you request.

to `int` or not to `int`

Some readers have pointed out they truly love `int` and you'll have to pry it from their cold dead fingers. I'd like to point out is is technically *impossible* to program correctly if the sizes of your types change out from under you.

Also see RATIONALE included with inttypes.h for reasons why using non-fixed-width types is unsafe. If you are truly smart enough to conceptualize `int` being 16 bits on some platforms and 32 bits on other platforms throughout your development while also testing all 16 bit and 32 bit edge cases for every place you use `int`, please feel free to use `int`.

For the rest of us who can't hold entire multi-level decision tree platform specification hierarchies in our heads while writing fizzbuzz, we can use fixed width types and automatically have more correct code with much less conceptual hassle and much less required testing overhead.

Or, said more concisely in the specification: "the ISO C standard integer promotion rule can produce silent changes unexpectedly."

Good luck with that.

One Exception to never-`char`

The *only* acceptable use of `char` in 2016 is if a pre-existing API requires `char` (e.g. `strncat`, printf'ing "%s", ...) or if you're initializing a read-only string (e.g. `const char *hello = "hello";`) because the C type of string literals (`"hello"`) is `char []`.

ALSO: In C11 we have native unicode support, and the type of UTF-8 string literals is still `char []` even for multibyte sequences like `const char *abcgrr = u8"abc😬";`.

One Exception to never-{`int`,`long`,etc}

If you are using a function with native return types or native parameters, use types as described by the function prototype or API specification.

Signedness

At no point should you be typing the word `unsigned` into your code. We can now write code without the ugly C convention of multi-word types that impair readability as well as usage. Who wants to type `unsigned long long int` when you can type `uint64_t`? The `<stdint.h>` types are more *explicit*, more *exact* in meaning, convey *intentions* better, and are more *compact* for typographic *usage* and *readability*.

Pointers as Integers

But, you may say, "I need to cast pointers to `long` for dirty pointer math!"

You may say that. But you are wrong.

The correct type for pointer math is `uintptr_t` defined by `<stdint.h>`, while the also useful `ptrdiff_t` is defined by stddef.h.

Instead of:

```
long diff = (long)ptrOld - (long)ptrNew;
```

Use:

```
ptrdiff_t diff = (uintptr_t)ptrOld - (uintptr_t)ptrNew;
```

Also:

```
printf("%p is unaligned by %" PRIuPTR " bytes.\n", (void *)p, ((uintptr_t)somePtr & (sizeof(void *) - 1)));
```

System-Dependent Types

You continue arguing, "on a 32 bit platform I want 32 bit longs and on a 64 bit platform I want 64 bit longs!"

If we skip over the line of thinking where you are *deliberately* introducing difficult to reason about code by using two different sizes depending on platform, you still don't want to use `long` for system-dependent types.

In these situations, you should use `intptr_t` — the integer type capable of holding a pointer value for your platform.

On modern 32-bit platforms, `intptr_t` is `int32_t`.

On modern 64-bit platforms, `intptr_t` is `int64_t`.

`intptr_t` also comes in a `uintptr_t` flavor.

For holding pointer offsets, we have the aptly named `ptrdiff_t` which is the proper type for storing values of subtracted pointers.

Maximum Value Holders

Do you need an integer type capable of holding any integer usable on your system?

People tend to use the largest known type in this case, such as casting smaller unsigned types to `uint64_t`, but there's a more technically correct way to guarantee any value can hold any other value.

The safest container for any integer is `intmax_t` (also `uintmax_t`). You can assign or cast any signed integer to `intmax_t` with no loss of precision, and you can assign or cast any unsigned integer to `uintmax_t` with no loss of precision.

That Other Type

The most widely used system-dependent type is `size_t` and is provided by stddef.h.

`size_t` is basically as "an integer capable of holding the largest array index" which also means it's capable of holding the largest memory offset in your program.

In practical use, `size_t` is the return type of `sizeof` operator.

In either case: `size_t` is *practically* defined to be the same as `uintptr_t` on all modern platforms, so on a 32-bit platform `size_t` is `uint32_t` and on a 64-bit platform `size_t` is `uint64_t`.

There is also `ssize_t` which is a signed `size_t` used as the return value from library functions that return `-1` on error. (Note: `ssize_t` is POSIX and does not apply to Windows interfaces.)

So, should you use `size_t` for arbitrary system-dependent sizes in your own function parameters? Technically, `size_t` is the return type of `sizeof`, so any functions accepting a size value representing a number of bytes is allowed to be a `size_t`.

Other uses include: `size_t` is the type of the argument to malloc, and `ssize_t` is the return type of `read()` and `write()` (except on Windows where `ssize_t` doesn't exist and the return values are just `int`).

Printing Types

You should never cast types during printing.

Always use proper type specifiers as defined by inttypes.h.

These include, but are not limited to:

- `size_t` - `%zu`
- `ssize_t` - `%zd`
- `ptrdiff_t` - `%td`
- raw pointer value - `%p` (prints hex in modern compilers; cast your pointer to `(void *)` first)
- `int64_t` - `"%" PRId64`
- `uint64_t` - `"%" PRIu64`
  - 64-bit types should only be printed using `PRI[udixXo]64` style macros.
  - Why?
    - on some platforms a 64-bit value is a `long` and on others it's a `long long`. These macros provide the proper underlying format specification across platforms.
    - it is actually impossible to specify a correct cross-platform format string without these format macros because the types change out from under you (and remember, casting values before printing is not safe or logical).
- `intptr_t` — `"%" PRIdPTR`
- `uintptr_t` — `"%" PRIuPTR`

- `intmax_t` — `"%" PRIdMAX`
- `uintmax_t` — `"%" PRIuMAX`

One note about the `PRI*` formatting specifiers: they are *macros* and the macros expand to proper printf type specifiers on a platform-specific basis. This means you can't do:

```
printf("Local number: %PRIdPTR\n\n", someIntPtr);
```

but instead, because they are macros, you do:

```
printf("Local number: %" PRIdPTR "\n\n", someIntPtr);
```

Notice you put the `%` *inside* your format string literal, but the type specifier is *outside* your format string literal because all adjacent strings get concatentated by the preprocessor into one final combined string literal.

C99 allows variable declarations anywhere

So, do NOT do this:

```
void test(uint8_t input) {
    uint32_t b;

    if (input > 3) {
        return;
    }

    b = input;
}
```

do THIS instead:

```
void test(uint8_t input) {
    if (input > 3) {
        return;
    }

    uint32_t b = input;
}
```

Caveat: if you have tight loops, test the placement of your initializers. Sometimes scattered declarations can cause unexpected slowdowns. For regular non-fast-path code (which is most of everything in the world), it's best to be as clear as possible, and defining types next to your initializations is a big readability improvement.

C99 allows `for` loops to declare counters inline

So, do NOT do this:

```
    uint32_t i;

    for (i = 0; i < 10; i++)
```

Do THIS instead:

```
    for (uint32_t i = 0; i < 10; i++)
```

One exception: if you need to retain your counter value after the loop exits, obviously don't declare your counter scoped to the loop itself.

Modern compilers support `#pragma once`

So, do NOT do this:

```
#ifndef PROJECT_HEADERNAME
#define PROJECT_HEADERNAME
.
.
.
#endif /* PROJECT_HEADERNAME */
```

Do THIS instead:

```
#pragma once
```

`#pragma once` tells the compiler to only include your header once and you *do not* need three lines of header guards anymore. This pragma is widely supported across all compilers across all platforms and is recommended over manually naming header guards.

For more details, see list of supported compilers at [pragma once].

C allows static initialization of auto-allocated arrays

So, do NOT do this:

```
uint32_t numbers[64];
memset(numbers, 0, sizeof(numbers));
```

Do THIS instead:

```
uint32_t numbers[64] = {0};
```

C allows static initialization of auto-allocated structs

So, do NOT do this:

```
struct thing {
    uint64_t index;
    uint32_t counter;
};

struct thing localThing;

void initThing(void) {
    memset(&localThing, 0, sizeof(localThing));
}
```

Do THIS instead:

```
struct thing {
    uint64_t index;
    uint32_t counter;
};

struct thing localThing = {0};
```

IMPORTANT NOTE: If your struct has padding, the `{0}` method does not zero out extra padding bytes. For example, `struct thing` has 4 bytes of padding after `counter` (on a 64-bit platform) because structs are padded to word-sized increments. If you need to zero out an entire struct *including* unused padding, use `memset(&localThing, 0, sizeof(localThing))` because `sizeof(localThing) == 16 bytes` even though the addressable contents is only `8 + 4 = 12 bytes`.

If you need to re-initialize already allocated structs, declare a global zero-struct for later assignment:

```
struct thing {
    uint64_t index;
    uint32_t counter;
};

static const struct thing localThingNull = {0};
.
.
.
struct thing localThing = {.counter = 3};
.
.
.
localThing = localThingNull;
```

If you are lucky enough to be in a C99 (or newer) environment, you can use compound literals instead of keeping a global "zero struct" around (also see, from 2001, The New C: Compound Literals).

Compound literals allow your compiler to automatically create temporary anonymous structs then copy them onto a target value:

```
localThing = (struct thing){0};
```

C99 added variable length arrays (C11 made them optional)

So, do NOT do this (if you know your array is tiny or you are just testing something quickly):

```
uintmax_t arrayLength = strtoumax(argv[1], NULL, 10);
void *array[];

array = malloc(sizeof(*array) * arrayLength);

/* remember to free(array) when you're done using it */
```

Do THIS instead:

```
uintmax_t arrayLength = strtoumax(argv[1], NULL, 10);
void *array[arrayLength];

/* no need to free array */
```

IMPORTANT CAVEAT: variable length arrays are (usually) stack allocated just like regular arrays. If you wouldn't create a 3 million element regular array statically, don't attempt to create a 3 million element array at runtime using this syntax. These are not scalable python/ruby auto-growing lists. If you specify a runtime array length and the length is too big for your stack, your program will do awful things (crashes, security issues). Variable Length Arrays are convenient for small, single-purpose situations, but should not be relied on at scale in production software. If sometimes you need a 3 element array and other times a 3 million element array, definitely do not use the variable length array capability.

It's good to be aware of the VLA syntax in case you encounter it live (or want it for quick one-off testing), but it can almost be considered a dangerous anti-pattern since you can crash your programs fairly simple by forgetting element size bounds checks or by forgetting you are on a strange target platform with no free stack space.

NOTE: You must be certain `arrayLength` is a reasonable size in this situation. (i.e. less than a few KB, sometime your stack will max out at 4 KB on weird platforms). You can't stack allocate *huge* arrays (millions of entries), but if you know you have a limited count, it's much easier to use C99 VLA capabilities rather than manually requesting heap memory from malloc.

DOUBLE NOTE: there is no user input checking above, so the user can easily kill your program by allocating a giant VLA. Some people go as far to call VLAs an anti-pattern, but if you keep your bounds tight, it can be a tiny win in certain situations.

C99 allows annotating non-overlapping pointer parameters

See the restrict keyword (often `__restrict`)

Parameter Types

If a function accepts **arbitrary** input data and a length to process, don't restrict the type of the parameter.

So, do NOT do this:

```
void processAddBytesOverflow(uint8_t *bytes, uint32_t len) {
    for (uint32_t i = 0; i < len; i++) {
        bytes[0] += bytes[i];
    }
}
```

Do THIS instead:

```
void processAddBytesOverflow(void *input, uint32_t len) {
    uint8_t *bytes = input;

    for (uint32_t i = 0; i < len; i++) {
        bytes[0] += bytes[i];
    }
}
```

The input types to your functions describe the *interface* to your code, not what your code is doing with the parameters. The interface to the code above means "accept a byte array and a length", so you don't want to restrict your callers to only uint8_t byte streams. Maybe your users even want to pass in old-style `char *` values or something else unexpected.

By declaring your input type as `void *` then re-assigning or re-casting to the actual type you want inside your function, you save the users of your function from having to think about abstractions *inside* your own library.

Some readers have pointed out alignment problems with this example, but we are accessing single byte elements of the input, so everything is fine. If instead we were casting the input to wider types, we would need to watch out for alignment issues. For a different write up dealing with cross-platform alignment issues, see Unaligned Memory Access. (reminder: this page of generic overview details isn't about cross-architecture intricacies of C, so external knowledge and experience is expected to fully use any examples provided.)

Return Parameter Types

C99 gives us the power of `<stdbool.h>` which defines `true` to `1` and `false` to `0`.

For success/failure return values, functions should return `true` or `false`, not an `int32_t` return type with manually specifying `1` and `0` (or worse, `1` and `-1` (or is it `0` success and `1` failure? or is it `0` success and `-1` failure?)).

If a function mutates an input parameter to the extent the parameter is invalidated, instead of returning the altered pointer, your entire API should force double pointers as parameters anywhere an input can be invalidated. Coding with "for some calls, the return value invalidates the input" is too error prone for mass usage.

So, do NOT do this:

```
void *growthOptional(void *grow, size_t currentLen, size_t newLen) {
    if (newLen > currentLen) {
        void *newGrow = realloc(grow, newLen);
        if (newGrow) {
            /* resize success */
            grow = newGrow;
        } else {
            /* resize failed, free existing and signal failure through NULL */
            free(grow);
            grow = NULL;
        }
    }

    return grow;
}
```

Do THIS instead:

```
/* Return value:
 *  - 'true' if newLen > currentLen and attempted to grow
 *    - 'true' does not signify success here, the success is still in '*_grow'
 *  - 'false' if newLen <= currentLen */
bool growthOptional(void **_grow, size_t currentLen, size_t newLen) {
    void *grow = *_grow;
    if (newLen > currentLen) {
        void *newGrow = realloc(grow, newLen);
        if (newGrow) {
            /* resize success */
            *_grow = newGrow;
            return true;
        }

        /* resize failure */
        free(grow);
        *_grow = NULL;

        /* for this function,
         * 'true' doesn't mean success, it means 'attempted grow' */
        return true;
    }

    return false;
}
```

Or, even better, Do THIS instead:

```
typedef enum growthResult {
    GROWTH_RESULT_SUCCESS = 1,
    GROWTH_RESULT_FAILURE_GROW_NOT_NECESSARY,
    GROWTH_RESULT_FAILURE_ALLOCATION_FAILED
} growthResult;

growthResult growthOptional(void **_grow, size_t currentLen, size_t newLen) {
    void *grow = *_grow;
    if (newLen > currentLen) {
        void *newGrow = realloc(grow, newLen);
        if (newGrow) {
            /* resize success */
            *_grow = newGrow;
            return GROWTH_RESULT_SUCCESS;
        }

        /* resize failure, don't remove data because we can signal error */
        return GROWTH_RESULT_FAILURE_ALLOCATION_FAILED;
    }

    return GROWTH_RESULT_FAILURE_GROW_NOT_NECESSARY;
}
```

Formatting

Coding style is simultaneously very important and utterly worthless.

If your project has a 50 page coding style guideline, nobody will help you. But, if your code isn't readable, nobody will *want* to help you.

The solution here is to **always** use an automated code formatter.

The only usable C formatter as of 2016 is clang-format. clang-format has the best defaults of any automatic C formatter and is still actively developed.

Here's my preferred script to run clang-format with good parameters:

```
#!/usr/bin/env bash

clang-format -style="{BasedOnStyle: llvm, IndentWidth: 4, AllowShortFunctionsOnASingleLine: None, KeepEmptyLinesAtTheStartOfBlocks: false}" "$@"
```

Then call it as (assuming you named the script `cleanup-format`):

```
matt@foo:~/repos/badcode% cleanup-format -i *.{c,h,cc,cpp,hpp,cxx}
```

The `-i` option overwrites existing files in place with formatting changes instead of writing to new files or creating backup files.

If you have many files, you can recursively process an entire source tree in parallel:

```
#!/usr/bin/env bash

# note: clang-tidy only accepts one file at a time, but we can run it
#       parallel against disjoint collections at once.
find . \( -name \*.c -or -name \*.cpp -or -name \*.cc \) |xargs -n1 -P4 cleanup-tidy

# clang-format accepts multiple files during one run, but let's limit it to 12
# here so we (hopefully) avoid excessive memory usage.
find . \( -name \*.c -or -name \*.cpp -or -name \*.cc -or -name \*.h \) |xargs -n12 -P4 cleanup-format -i
```

Now, there's a new cleanup-tidy script there. The contents of `cleanup-tidy` is:

```
#!/usr/bin/env bash

clang-tidy \
    -fix \
    -fix-errors \
    -header-filter=.* \
    --checks=readability-braces-around-statements,misc-macro-parentheses \
    $1 \
    -- -I.
```

clang-tidy is policy driven code refactoring tool. The options above enable two fixups:

- `readability-braces-around-statements` — force all `if`/`while`/`for` statement bodies to be enclosed in braces
  - It's an accident of history for C to allow "brace optional" single statements after loop constructs and conditionals. It is *inexcusable* to write modern code without braces enforced on every loop and every conditional. Trying to argue "but, the compiler accepts it!" has *nothing* to do with the readability, maintainability, understandability, or skimability of code. You aren't programming to please your compiler, you are programming to please future people who have to maintain your current brain state years after everybody has forgotten why anything exists in the first place.
- `misc-macro-parentheses` — automatically add parens around all parameters used in macro bodies

`clang-tidy` is great when it works, but for some complex code bases it can get stuck. Also, `clang-tidy` doesn't *format*, so you need to run `clang-format` after you tidy to align new braces and reflow macros.

Readability

the writing seems to start slowing down here...

Comments

logical self-contained portions of code file

File Structure

Try to limit files to a max of 1,000 lines (1,500 lines in really bad cases). If your tests are in-line with your source file (for testing static functions, etc), adjust as necessary.

misc thoughts

Never use `malloc`

You should always use `calloc`. There is no performance penalty for getting zero'd memory. If you don't like the function protype of `calloc(object count, size per object)` you can wrap it with `#define mycalloc(N) calloc(1, N)`.

Readers have commented on a few things here:

- `calloc` *does* have a performance impact for **huge** allocations
- `calloc` *does* have a performance impact on weird platforms (minimal embedded systems, game consoles, 30 year old hardware, ...)
- wrapping `calloc(element count, size of each element)` is not always a good idea.
- a good reason to avoid `malloc()` is it can't check for integer overflow and is a potential security risk
- `calloc` allocations remove valgrind's ability to warn you about unintentional reads or copies of uninitialized memory since allocations get initialized to `0` automatically

Those are good points, and that's why we always must do performance testing and regression testing for speed across compilers, platforms, operating systems, and hardware devices.

One advantage of using `calloc()` directly without a wrapper is, unlike `malloc()`, `calloc()` can check for integer overflow because it multiplies its arguments together to obtain your final allocation size. If you are only allocating tiny things, wrapping `calloc()` is fine. If you are allocating potentially unbounded streams of data, you may want to retain the regular `calloc(element count, size of each element)` calling convention.

No advice can be universal, but trying to give *exactly perfect* generic recommendations would end up reading like a book of language specifications.

For references on how `calloc()` gives you clean memory for free, see these nice writeups:

- Benchmarking fun with calloc() and zero pages (2007)
- Copy-on-write in virtual memory management

I still stand by my recommendation of always using `calloc()` for most common scenarios of 2016 (assumption: x64 target platforms, human-sized data, not including human genome-sized data). Any deviations from "expected" drag us into the pit of despair of "domain knowledge," which are words we shan't speak this day.

Subnote: The pre-zero'd memory delivered to you by `calloc()` is a one-shot deal. If you `realloc()` your `calloc()` allocation, the grown memory extended by realloc is *not* new zero'd out memory. Your grown allocation is filled with whatever regular uninitialized contents your kernel provides. If you need zero'd memory after a realloc, you must manually `memset()` the extent of your grown allocation.

Never memset (if you can avoid it)

Never `memset(ptr, 0, len)` when you can statically initialize a structure (or array) to zero (or reset it back to zero by assigning from an in-line compound literal or by assigning from a global zero'd out structure).

Though, `memset()` is your only choice if you need to zero out a struct including its padding bytes (because `{0}` only sets defined fields, not undefined offsets filled by padding).

Learn More

Also see Fixed width integer types (since C99)

Also see Apple's Making Code 64-Bit Clean

Also see the sizes of C types across architectures — unless you keep that entire table in your head for every line of code you write, you should use explicitly defined integer widths and never use char/short/int/long built-in storage types.

Also see size_t and ptrdiff_t

Also see Secure Coding. If you really want to write everything perfectly, simply memorize their thousand simple examples.

Also see Modern C by Jens Gustedt at Inria.

Also see Understanding Character/String Literals in C/C++ for details about Unicode support in C11.

Closing

Writing correct code at scale is essentially impossible. We have multiple operating systems, runtimes, libraries, and hardware platforms to worry about without even considering things like random bit flips in RAM or our block devices lying to us with unknown probability.

The best we can do is write simple, understandable code with as few indirections and as little undocumented magic as possible.

-Matt — @mattsta — ♣mattsta

Attributions

This made the twitter and HN rounds, so many people helpfully pointed out flaws or biased thoughts I'm promulgating here.

First up, Jeremy Faller and Sos Sosowski and Martin Heistermann and a few other people were kind enough to point out my `memset()` example was broken and provided the proper fix.

Martin Heistermann also pointed out the `localThing = localThingNull` example was broken.

The opening quote about not writing C if you can avoid it is from the wise internet sage @badboy_.

Remi Gacogne pointed out I forgot `-Wextra`.

Levi Pearson pointed out gcc-5 defaults to gnu11 instead of c89 as well as clarifying the default clang mode.

Christopher pointed out the `-O2` vs `-O3` section could use a little more clarification.

Chad Miller pointed out I was being lazy in the clang-format script params.

Many people also pointed out the `calloc()` advice isn't *always* a good idea if you have extreme circumstances or non-standard hardware (examples of bad ideas: huge allocations, allocations on embedded jiggers, allocations on 30 year old hardware, etc).

Charles Randolph pointed out I misspelled the world "Building."

Sven Neuhaus pointed out kindly I also do not possess the ability to spell "initialization" or "initializers." (and also pointed out I misspelled "initialization" wrong the first time here as well)

Colm MacCárthaigh pointed out I forgot to mention `#pragma once`.

Jeffrey Yasskin pointed out we should kill strict aliasing too (mainly a gcc optimization).

Jeffery Yasskin also provided better wording around the `-fno-strict-aliasing` section.

Chris Palmer and a few others pointed out calloc-vs-malloc parameter advantages and the overall drawback of writing a wrapper for `calloc()` because `calloc()` provides a more secure interface than `malloc()` in the first place.

Damien Sorresso pointed out we should remind people `realloc()` doesn't zero out grown memory after an initial zero'd `calloc()` request.

Pat Pogson pointed out I was unable to spell the word "declare" correctly as well.

@TopShibe pointed out the stack-allocated initialization example was wrong because the examples I gave were global variables. Updated wording to just mean "auto-allocated" things, be it stack or data sections.

Jonathan Grynspan suggested harsher wording around the VLA example because they **are** dangerous when used incorrectly.

David O'Mahony kindly pointed out I can't spell "specify" either.

Dr. David Alan Gilbert pointed out `ssize_t` is a POSIXism and Windows doesn't have it.

Chris Ridd suggested we explicitly mention C99 is C from 1999 and C11 is C from 2011 because otherwise it looks strange having 11 be newer than 99.

Chris Ridd also noticed the `clang-format` example used unclear naming conventions and suggested better consistency across examples.

Anthony Le Goff pointed us to a book-length treatment of many modern C ideas called Modern C.

Stuart Popejoy pointed out my inaccurate spelling of deliberately was truly inaccurate.

jack rosen pointed out my usage of the word 'exists' does not mean 'exits' as I intended.

Jo Booth pointed out I like to spell compatibility as compatibility, which seems more logical, but English commonality disagrees.

Stephen Anderson decoded my aberrant spelling of 'stil' back into 'still.'

Richard Weinberger pointed out struct initialization with `{0}` doesn't zero out padding bytes, so sending a `{0}` struct over the wire can leak unintended bytes on under-specified structs.

@JayBhukhanwala pointed out the function comment in Return Parameter Types was inaccurate because I didn't update the comment when the code changed (story of our lives, right?).

Lorenzo pointed out we should explicitly provide a warning concerning potential cross-platform alignment issues in section Parameter Types.

Paolo G. Giarrusso re-clarified the alignment warning I previously added to be more correct regarding the examples given.

Fabian Klötzl provided the valid struct compound literal assignment example since it's perfectly valid syntax I just hadn't run across before.

Omkar Ekbote provided a very thorough walkthrough of typos and consistency problems here including that I couldn't spell "platform," "actually," "defining," "experience," "simultaneously," "readability," as well as noted some other unclear wordings.

Carlo Bellettini fixed my aberrant spelling of the word aberrant.

Keith S Thompson provided many technical corrections in his great article how-to-c-response.

Marc Bevand pointed out we should say the `fprintf` type specifiers come from inttypes.h.

Brian Cain pointed out we should mention the *fast* and *least* types too.

Michal Marzec kindly pointed out I still can't spell anonymous.

Nick Galbreath used misspell to discover four more sneaky misspellings.

Many people on reddit went apeshit because this article originally had `#import` somewhere by mistake. Sorry, crazy people, but this started out as an unedited and unreviewed year old draft when originally pushed live. The error has since been remedied.

Some people also pointed out the static initialization example uses globals which are always initialized to zero by default anyway (and that they aren't even initialized, they are statically allocated). This is a poor choice of example on my part, but the concepts still stand for typical usage within function scopes. The examples were meant to be any generic "code snippet" and not necessarily top level globals.

A few people seem to have read this as an "I hate C" page, but it isn't. C is dangerous in the wrong hands (not enough testing, not enough experience when widely deployed), so paradoxically the two kinds of C developers should only be novice hobbyists (code failure causes no problems, it's just a toy) or people who are willing to test their asses off (code failure causes life or financial loss, it's not just a toy) should be writing C code for production usage. There's not much room for "casual observer C development." For the rest of the world, that's why we have Erlang.

Many people have also mentioned their own pet issues as well or issues beyond the scope of this article (including new C11 only features like George Makrydakis reminding us about C11 generic abilities).

Perhaps another article about "Practical C" will show up to cover testing, profiling, performance tracing, optional-but-useful warning levels, etc.

series