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.

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

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 compatability requirements. We shouldn't be globally tied to ancient standards just because some companies refuse to upgrade 20 year old systems.

Preflight

Standard c99.

- clang, default
  - C99 is the default C implementation for clang, no extra options needed.
  - If you want C11, you need to specify `-std=c11`
  - clang compiles your source files faster than gcc
- gcc requires you sepecify `-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 `-std=gnu11`, but you should still specify a non-GNU `c99` or `c11` for practical usage.

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 compatability.
- 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.
- you can be extra fancy and add `-Wstrict-aliasing -Wstrict-overflow` too.
- 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 optimization 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 (this slows down the linking process noticeably, but `make -j` helps).
  - 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` stil 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.

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.

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□";`.

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*.

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 in `<stddef.h>`.

Instead of:

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

Use:

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

System-Dependent Types

You continue arguing, "on a 32 bit patform 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 *deliberatly* 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 defined to be the word size of your current platform.

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

On 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`.

`size_t` is defined 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.

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()`.

Printing Types

You should never cast types during printing. You should use proper type specifiers.

These include, but are not limited to:

- `size_t` - `%zu`
- `ssize_t` - `%zd`
- `ptrdiff_t` - `%td`
- raw pointer value - `%p` (prints hex value; cast your pointer to `(void *)` first)
- 64-bit types should be printed using `PRIu64` (unsigned) and `PRId64` (signed)
  - on some platforms a 64-bit value is a `long` and on others it's a `long long`
  - it is actualy 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, but the type specifier is *outside* your format string.

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 defininig 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 exists, obviously don't declare your counter scoped to the loop itself.

C allows static initialization of stack-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 stack-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};
```

If you need to re-initialize already allocated structs, delcare 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;
```

C99 allows variable length array initializers

So, do NOT do this:

```
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 */
```

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 *` and re-casting inside your function, you save the users of your function from having to think about abstractions *inside* your own library.

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 simultaenously 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 clang-format script:

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

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

Then call it as:

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

The `-i` option to `clang-format` means overwrite existing files 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 have "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 readabiltiy, 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 two 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, ...)

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.

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 platformas, 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.

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 a global zero'd out structure).

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

If you really want to write everything perfectly, memorize the thousand individual pages at Secure Coding.

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.

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 posess the ability to spell "initilization" or "initializers."

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 writting 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.

redis
what is redis?intro to data typesthinking in redisredis architecture
redis ready-to-use
short cluster intro
dynamic redis
geo commands
quicklist
crcspeed
redis experiments
pub/sub scripts
command loading
json storage
compare compilers
redis presentations
Intro to Redis (2013)
books
swift: power of types
personal
my codehowto c (2016)koshaboutiOS apps
future
searching (past)errors at scaleemployees
audio
(may 30, 2014)