What’s an Object Walk?

- Related Reading
- Setting Up
  - struct rev_cmdline_info
  - struct rev_info
- Basic Commit Walk
  - Setting Up
    - Default Setups
    - Configuring From .gitconfig
    - Setting Up rev_info
    - Tweaking rev_info For the Walk
    - Preparing rev_info For the Walk
    - Performing the Walk!
  - Adding a Filter
  - Changing the Order
- Basic Object Walk
  - Adding a Filter
  - Counting Omitted Objects
  - Changing the Order
- Wrapping Up

The object walk is a key concept in Git - this is the process that underpins operations like object transfer and fsck. Beginning from a given commit, the list of objects is found by walking parent relationships between commits (commit X based on commit W) and containment relationships between objects (tree Y is contained within commit X, and blob Z is located within tree Y, giving our working tree for commit X something like `y/z.txt`).

A related concept is the revision walk, which is focused on commit objects and their parent relationships and does not delve into other object types. The revision walk is used for operations like `git log`.

Related Reading

- `Documentation/user-manual.adoc` under "Hacking Git" contains some coverage of the revision walker in its various incarnations.
- `revision.h`
- Git for Computer Scientists gives a good overview of the types of objects in Git and what your object walk is really describing.

Setting Up

Create a new branch from `master`.

```
git checkout -b revwalk origin/master
```

We'll put our fiddling into a new command. For fun, let's name it `git` `walken`. Open up a new file `builtin/walken.c` and set up the command handler:

```
/*
 * "git walken"
 *
 * Part of the "My First Object Walk" tutorial.
 */

#include "builtin.h"
#include "trace.h"

int cmd_walken(int argc, const char **argv, const char *prefix, struct repository *repo)
{
    trace_printf(_("cmd_walken incoming...\n"));
    return 0;
}
```

> `trace_printf()`, defined in `trace.h`, differs from `printf()` in that it can be turned on or off at runtime. For the purposes of this tutorial, we will write `walken` as though it is intended for use as a "plumbing" command: that is, a command which is used primarily in scripts, rather than interactively by humans (a "porcelain" command). So we will send our debug output to `trace_printf()` instead. When running, enable trace output by setting the environment

> variable `GIT_TRACE`.

Add usage text and `-h` handling, like all subcommands should consistently do (our test suite will notice and complain if you fail to do so). We'll need to include the `parse-options.h` header.

```
#include "parse-options.h"

...

int cmd_walken(int argc, const char **argv, const char *prefix)
{
    const char * const walken_usage[] = {
        N_("git walken"),
        NULL,
    };
    struct option options[] = {
        OPT_END()
    };

    argc = parse_options(argc, argv, prefix, options, walken_usage, 0);

    ...
}
```

Also add the relevant line in `builtin.h` near `cmd_version()`:

```
int cmd_walken(int argc, const char **argv, const char *prefix, struct repository *repo);
```

Include the command in `git.c` in `commands[]` near the entry for `version`, maintaining alphabetical ordering:

```
{ "walken", cmd_walken, RUN_SETUP },
```

Add an entry for the new command in the both the Make and Meson build system, before the entry for `worktree`:

- In the `Makefile`: ---- ... BUILTIN_OBJS += builtin/walken.o ... ----
- In the `meson.build` file: ---- builtin_sources = [... 'builtin/walken.c', ...] ----

Build and test out your command, without forgetting to ensure the `DEVELOPER` flag is set, and with `GIT_TRACE` enabled so the debug output can be seen:

```
$ echo DEVELOPER=1 >>config.mak
$ make
$ GIT_TRACE=1 ./bin-wrappers/git walken
```

> For a more exhaustive overview of the new command process, take a look at `Documentation/MyFirstContribution.adoc`.
>
> A reference implementation can be found at https://github.com/nasamuffin/git/tree/revwalk.

`struct` `rev_cmdline_info`

The definition of `struct` `rev_cmdline_info` can be found in `revision.h`.

This struct is contained within the `rev_info` struct and is used to reflect parameters provided by the user over the CLI.

`nr` represents the number of `rev_cmdline_entry` present in the array.

`alloc` is used by the `ALLOC_GROW` macro. Check `alloc.h` - this variable is used to track the allocated size of the list.

Per entry, we find:

`item` is the object provided upon which to base the object walk. Items in Git can be blobs, trees, commits, or tags. (See `Documentation/gittutorial-2.adoc`.)

`name` is the object ID (OID) of the object - a hex string you may be familiar with from using Git to organize your source in the past. Check the tutorial mentioned above towards the top for a discussion of where the OID can come from.

`whence` indicates some information about what to do with the parents of the specified object. We'll explore this flag more later on; take a look at `Documentation/revisions.adoc` to get an idea of what could set the `whence` value.

`flags` are used to hint the beginning of the revision walk and are the first block under the `#include`s in `revision.h`. The most likely ones to be set in the `rev_cmdline_info` are `UNINTERESTING` and `BOTTOM`, but these same flags can be used during the walk, as well.

`struct rev_info`

This one is quite a bit longer, and many fields are only used during the walk by `revision.c` - not configuration options. Most of the configurable flags in `struct rev_info` have a mirror in `Documentation/rev-list-options.adoc`. It's a good idea to take some time and read through that document.

Basic Commit Walk

First, let's see if we can replicate the output of `git log --oneline`. We'll refer back to the implementation frequently to discover norms when performing an object walk of our own.

To do so, we'll first find all the commits, in order, which preceded the current commit. We'll extract the name and subject of the commit from each.

Ideally, we will also be able to find out which ones are currently at the tip of various branches.

Setting Up

Preparing for your object walk has some distinct stages.

1. Perform default setup for this mode, and others which may be invoked.
2. Check configuration files for relevant settings.
3. Set up the `rev_info` struct.
4. Tweak the initialized `rev_info` to suit the current walk.
5. Prepare the `rev_info` for the walk.
6. Iterate over the objects, processing each one.

Default Setups

Before examining configuration files which may modify command behavior, set up default state for switches or options your command may have. If your command utilizes other Git components, ask them to set up their default states as well. For instance, `git log` takes advantage of `grep` and `diff` functionality, so its `init_log_defaults()` sets its own state (`decoration_style`) and asks `grep` and `diff` to initialize themselves by calling each of their initialization functions.

Configuring From `.gitconfig`

Next, we should have a look at any relevant configuration settings (i.e., settings readable and settable from `git config`). This is done by providing a callback to `repo_config()`; within that callback, you can also invoke methods from other components you may need that need to intercept these options. Your callback will be invoked once per each configuration value which Git knows about (global, local, worktree, etc.).

Similarly to the default values, we don't have anything to do here yet ourselves; however, we should call `git_default_config()` if we aren't calling any other existing config callbacks.

Add a new function to `builtin/walken.c`. We'll also need to include the `config.h` header:

```
#include "config.h"

...

static int git_walken_config(const char *var, const char *value,
                 const struct config_context *ctx, void *cb)
{
    /*
     * For now, we don't have any custom configuration, so fall back to
     * the default config.
     */
    return git_default_config(var, value, ctx, cb);
}
```

Make sure to invoke `repo_config()` with it in your `cmd_walken()`:

```
int cmd_walken(int argc, const char **argv, const char *prefix, struct repository *repo)
{
    ...

    repo_config(repo, git_walken_config, NULL);

    ...
}
```

Setting Up `rev_info`

Now that we've gathered external configuration and options, it's time to initialize the `rev_info` object which we will use to perform the walk. This is typically done by calling `repo_init_revisions()` with the repository you intend to target, as well as the `prefix` argument of `cmd_walken` and your

`rev_info` struct.

Add the `struct` `rev_info` and the `repo_init_revisions()` call. We'll also need to include the `revision.h` header:

```
#include "revision.h"

...

int cmd_walken(int argc, const char **argv, const char *prefix, struct repository *repo)
{
    /* This can go wherever you like in your declarations.*/
    struct rev_info rev;
    ...

    /* This should go after the repo_config() call. */
    repo_init_revisions(repo, &rev, prefix);

    ...
}
```

Tweaking `rev_info` For the Walk

We're getting close, but we're still not quite ready to go. Now that `rev` is initialized, we can modify it to fit our needs. This is usually done within a helper for clarity, so let's add one:

```
static void final_rev_info_setup(struct rev_info *rev)
{
    /*
     * We want to mimic the appearance of `git log --oneline`, so let's
     * force oneline format.
     */
    get_commit_format("oneline", rev);

    /* Start our object walk at HEAD. */
    add_head_to_pending(rev);
}
```

> Instead of using the shorthand `add_head_to_pending()`, you could do something like this:

```
    struct setup_revision_opt opt;

    memset(&opt, 0, sizeof(opt));
    opt.def = "HEAD";
    opt.revarg_opt = REVARG_COMMITTISH;
    setup_revisions(argc, argv, rev, &opt);
```

> Using a `setup_revision_opt` gives you finer control over your walk's starting point.

Then let's invoke `final_rev_info_setup()` after the call to `repo_init_revisions()`:

```
int cmd_walken(int argc, const char **argv, const char *prefix, struct repository *repo)
{
    ...

    final_rev_info_setup(&rev);

    ...
}
```

Later, we may wish to add more arguments to `final_rev_info_setup()`. But for now, this is all we need.

Preparing `rev_info` For the Walk

Now that `rev` is all initialized and configured, we've got one more setup step before we get rolling. We can do this in a helper, which will both prepare the `rev_info` for the walk, and perform the walk itself. Let's start the helper with the call to `prepare_revision_walk()`, which can return an error without dying on its own:

```
static void walken_commit_walk(struct rev_info *rev)
{
    if (prepare_revision_walk(rev))
        die(_("revision walk setup failed"));
}
```

> `die()` prints to `stderr` and exits the program. Since it will print to `stderr` it's likely to be seen by a human, so we will localize it.

Performing the Walk!

Finally! We are ready to begin the walk itself. Now we can see that `rev_info` can also be used as an iterator; we move to the next item in the walk by using `get_revision()` repeatedly. Add the listed variable declarations at the top and the walk loop below the `prepare_revision_walk()` call within your `walken_commit_walk()`:

```
#include "pretty.h"

...

static void walken_commit_walk(struct rev_info *rev)
{
    struct commit *commit;
    struct strbuf prettybuf = STRBUF_INIT;

    ...

    while ((commit = get_revision(rev))) {
        strbuf_reset(&prettybuf);
        pp_commit_easy(CMIT_FMT_ONELINE, commit, &prettybuf);
        puts(prettybuf.buf);
    }
    strbuf_release(&prettybuf);
}
```

> `puts()` prints a `char*` to `stdout`. Since this is the part of the command we expect to be machine-parsed, we're sending it directly to stdout.

Give it a shot.

```
$ make
$ ./bin-wrappers/git walken
```

You should see all of the subject lines of all the commits in your tree's history, in order, ending with the initial commit, "Initial revision of "git", the information manager from hell". Congratulations! You've written your first revision walk. You can play with printing some additional fields from each commit if you're curious; have a look at the functions available in `commit.h`.

Adding a Filter

Next, let's try to filter the commits we see based on their author. This is equivalent to running `git log --author=<pattern>`. We can add a filter by modifying `rev_info.grep_filter`, which is a `struct grep_opt`.

First some setup. Add `grep_config()` to `git_walken_config()`:

```
static int git_walken_config(const char *var, const char *value,
                  const struct config_context *ctx, void *cb)
{
    grep_config(var, value, ctx, cb);
    return git_default_config(var, value, ctx, cb);
}
```

Next, we can modify the `grep_filter`. This is done with convenience functions found in `grep.h`. For fun, we're filtering to only commits from folks using a `gmail.com` email address - a not-very-precise guess at who may be working on Git as a hobby. Since we're checking the author, which is a specific line in the header, we'll use the `append_header_grep_pattern()` helper. We can use the `enum grep_header_field` to indicate which part of the commit header we want to search.

In `final_rev_info_setup()`, add your filter line:

```
static void final_rev_info_setup(int argc, const char **argv,
        const char *prefix, struct rev_info *rev)
{
    ...

    append_header_grep_pattern(&rev->grep_filter, GREP_HEADER_AUTHOR,
        "gmail");
    compile_grep_patterns(&rev->grep_filter);

    ...
}
```

`append_header_grep_pattern()` adds your new "gmail" pattern to `rev_info`, but it won't work unless we compile it with `compile_grep_patterns()`.

> If you are using `setup_revisions()` (for example, if you are passing a `setup_revision_opt` instead of using `add_head_to_pending()`), you don't need to call `compile_grep_patterns()`

> because `setup_revisions()` calls it for you.
>
> We could add the same filter via the `append_grep_pattern()` helper if we wanted to, but `append_header_grep_pattern()` adds the `enum` `grep_context` and `enum` `grep_pat_token` for us.

Changing the Order

There are a few ways that we can change the order of the commits during a revision walk. Firstly, we can use the `enum` `rev_sort_order` to choose from some typical orderings.

`topo_order` is the same as `git` `log` `--topo-order`: we avoid showing a parent before all of its children have been shown, and we avoid mixing commits which are in different lines of history. (`git` `help` `log`'s section on `--topo-order` has a very nice diagram to illustrate this.)

Let's see what happens when we run with `REV_SORT_BY_COMMIT_DATE` as opposed to `REV_SORT_BY_AUTHOR_DATE`. Add the following:

```
static void final_rev_info_setup(int argc, const char **argv,
        const char *prefix, struct rev_info *rev)
{
    ...

    rev->topo_order = 1;
    rev->sort_order = REV_SORT_BY_COMMIT_DATE;

    ...
}
```

Let's output this into a file so we can easily diff it with the walk sorted by author date.

```
$ make
$ ./bin-wrappers/git walken > commit-date.txt
```

Then, let's sort by author date and run it again.

```
static void final_rev_info_setup(int argc, const char **argv,
        const char *prefix, struct rev_info *rev)
{
```

```
    ...

    rev->topo_order = 1;
    rev->sort_order = REV_SORT_BY_AUTHOR_DATE;

    ...
}

$ make
$ ./bin-wrappers/git walken > author-date.txt
```

Finally, compare the two. This is a little less helpful without object names or dates, but hopefully we get the idea.

```
$ diff -u commit-date.txt author-date.txt
```

This display indicates that commits can be reordered after they're written, for example with `git rebase`.

Let's try one more reordering of commits. `rev_info` exposes a `reverse` flag. Set that flag somewhere inside of `final_rev_info_setup()`:

```
static void final_rev_info_setup(int argc, const char **argv, const char *prefix,
        struct rev_info *rev)
{
    ...

    rev->reverse = 1;

    ...
}
```

Run your walk again and note the difference in order. (If you remove the grep pattern, you should see the last commit this call gives you as your current HEAD.)

Basic Object Walk

So far we've been walking only commits. But Git has more types of objects than that! Let's see if we can walk *all* objects, and find out some information about each one.

We can base our work on an example. `git pack-objects` prepares all kinds of objects for packing into a bitmap or packfile. The work we are interested in resides in `builtin/pack-objects.c:get_object_list()`; examination of that function shows that the all-object walk is being performed by `traverse_commit_list()` or `traverse_commit_list_filtered()`. Those two functions reside in `list-objects.c`; examining the source shows that, despite the name, these functions traverse all kinds of objects. Let's have a look at the arguments to `traverse_commit_list()`.

- `struct rev_info *revs`: This is the `rev_info` used for the walk. If its `filter` member is not `NULL`, then `filter` contains information for how to filter the object list.
- `show_commit_fn show_commit`: A callback which will be used to handle each individual commit object.
- `show_object_fn show_object`: A callback which will be used to handle each non-commit object (so each blob, tree, or tag).
- `void *show_data`: A context buffer which is passed in turn to `show_commit` and `show_object`.

In addition, `traverse_commit_list_filtered()` has an additional parameter:

- `struct oidset *omitted`: A linked-list of object IDs which the provided filter caused to be omitted.

It looks like these methods use callbacks we provide instead of needing us to call it repeatedly ourselves. Cool! Let's add the callbacks first.

For the sake of this tutorial, we'll simply keep track of how many of each kind of object we find. At file scope in `builtin/walken.c` add the following tracking variables:

```
static int commit_count;
static int tag_count;
static int blob_count;
static int tree_count;
```

Commits are handled by a different callback than other objects; let's do that one first:

```
static void walken_show_commit(struct commit *cmt, void *buf)
{
    commit_count++;
}
```

The `cmt` argument is fairly self-explanatory. But it's worth mentioning that the `buf` argument is actually the context buffer that we can provide to the traversal calls - `show_data`, which we mentioned a moment ago.

Since we have the `struct commit` object, we can look at all the same parts that we looked at in our earlier commit-only walk. For the sake of this tutorial, though, we'll just increment the commit

counter and move on.

The callback for non-commits is a little different, as we'll need to check which kind of object we're dealing with:

```
static void walken_show_object(struct object *obj, const char *str, void *buf)
{
    switch (obj->type) {
    case OBJ_TREE:
        tree_count++;
        break;
    case OBJ_BLOB:
        blob_count++;
        break;
    case OBJ_TAG:
        tag_count++;
        break;
    case OBJ_COMMIT:
        BUG("unexpected commit object in walken_show_object\n");
    default:
        BUG("unexpected object type %s in walken_show_object\n",
            type_name(obj->type));
    }
}
```

Again, `obj` is fairly self-explanatory, and we can guess that `buf` is the same context pointer that `walken_show_commit()` receives: the `show_data` argument to `traverse_commit_list()` and `traverse_commit_list_filtered()`. Finally, `str` contains the name of the object, which ends up being something like `foo.txt` (blob), `bar/baz` (tree), or `v1.2.3` (tag).

To help assure us that we aren't double-counting commits, we'll include some complaining if a commit object is routed through our non-commit callback; we'll also complain if we see an invalid object type. Since those two cases should be unreachable, and would only change in the event of a semantic change to the Git codebase, we complain by using `BUG()` - which is a signal to a developer that the change they made caused unintended consequences, and the rest of the codebase needs to be updated to understand that change. `BUG()` is not intended to be seen by the public, so it is not localized.

Our main object walk implementation is substantially different from our commit walk implementation, so let's make a new function to perform the object walk. We can perform setup which is applicable to all objects here, too, to keep separate from setup which is applicable to commit-only walks.

We'll start by enabling all types of objects in the `struct rev_info`. We'll also turn on `tree_blobs_in_commit_order`, which means that we will walk a commit's tree and everything it points to immediately after we find each commit, as opposed to waiting for the end and walking through all trees after the commit history has been discovered. With the appropriate settings configured, we are ready to call `prepare_revision_walk()`.

```
static void walken_object_walk(struct rev_info *rev)
{
    rev->tree_objects = 1;
    rev->blob_objects = 1;
    rev->tag_objects = 1;
    rev->tree_blobs_in_commit_order = 1;

    if (prepare_revision_walk(rev))
        die(_("revision walk setup failed"));

    commit_count = 0;
    tag_count = 0;
    blob_count = 0;
    tree_count = 0;
```

Let's start by calling just the unfiltered walk and reporting our counts. Complete your implementation of `walken_object_walk()`. We'll also need to include the `list-objects.h` header.

```
#include "list-objects.h"

...

    traverse_commit_list(rev, walken_show_commit, walken_show_object, NULL);

    printf("commits %d\nblobs %d\ntags %d\ntrees %d\n", commit_count,
        blob_count, tag_count, tree_count);
}
```

> This output is intended to be machine-parsed. Therefore, we are not sending it to `trace_printf()`, and we are not localizing it - we need scripts to be able to count on the formatting to be exactly the way it is shown here. If we were intending this output to be read by humans, we would need to localize it with `_()`.

Finally, we'll ask `cmd_walken()` to use the object walk instead. Discussing command line options is out of scope for this tutorial, so we'll just hardcode a branch we can change at compile time. Where

you call `final_rev_info_setup()` and `walken_commit_walk()`, instead branch like so:

```
    if (1) {
        add_head_to_pending(&rev);
        walken_object_walk(&rev);
    } else {
        final_rev_info_setup(argc, argv, prefix, &rev);
        walken_commit_walk(&rev);
    }
```

> For simplicity, we've avoided all the filters and sorts we applied in `final_rev_info_setup()` and simply added `HEAD` to our pending queue. If you want, you can certainly use the filters we added before by moving `final_rev_info_setup()` out of the conditional and removing the call to `add_head_to_pending()`.

Now we can try to run our command! It should take noticeably longer than the commit walk, but an examination of the output will give you an idea why. Your output should look similar to this example, but with different counts:

```
Object walk completed. Found 55733 commits, 100274 blobs, 0 tags, and 104210 trees.
```

This makes sense. We have more trees than commits because the Git project has lots of subdirectories which can change, plus at least one tree per commit. We have no tags because we started on a commit (`HEAD`) and while tags can point to commits, commits can't point to tags.

> You will have different counts when you run this yourself! The number of objects grows along with the Git project.

Adding a Filter

There are a handful of filters that we can apply to the object walk laid out in `Documentation/rev-list-options.adoc`. These filters are typically useful for operations such as creating packfiles or performing a partial clone. They are defined in `list-objects-filter-options.h`. For the purposes of this tutorial we will use the "tree:1" filter, which causes the walk to omit all trees and blobs which are not directly referenced by commits reachable from the commit in `pending` when the walk begins. (`pending` is the list of objects which need to be traversed during a walk; you can imagine a breadth-first tree traversal to help understand. In our case, that means we omit trees and blobs not directly referenced by `HEAD` or `HEAD`'s history, because we begin the walk with only `HEAD` in the `pending` list.)

For now, we are not going to track the omitted objects, so we'll replace those parameters with `NULL`. For the sake of simplicity, we'll add a simple build-time branch to use our filter or not. Preface the line calling `traverse_commit_list()` with the following, which will remind us which kind of walk we've just performed:

```
    if (0) {
        /* Unfiltered: */
        trace_printf(_("Unfiltered object walk.\n"));
    } else {
        trace_printf(
            _("Filtered object walk with filterspec 'tree:1'.\n"));

        parse_list_objects_filter(&rev->filter, "tree:1");
    }
    traverse_commit_list(rev, walken_show_commit,
                 walken_show_object, NULL);
```

The `rev→filter` member is usually built directly from a command line argument, so the module provides an easy way to build one from a string. Even though we aren't taking user input right now, we can still build one with a hardcoded string using `parse_list_objects_filter()`.

With the filter spec "tree:1", we are expecting to see *only* the root tree for each commit; therefore, the tree object count should be less than or equal to the number of commits. (For an example of why that's true: `git commit --revert` points to the same tree object as its grandparent.)

Counting Omitted Objects

We also have the capability to enumerate all objects which were omitted by a filter, like with `git log --filter=<spec> --filter-print-omitted`. To do this, change `traverse_commit_list()` to `traverse_commit_list_filtered()`, which is able to populate an `omitted` list. Asking for this list of filtered objects may cause performance degradations, however, because in this case, despite filtering objects, the possibly much larger set of all reachable objects must be processed in order to populate that list.

First, add the `struct oidset` and related items we will use to iterate it:

```
#include "oidset.h"

...

static void walken_object_walk(
```

```
    ...

    struct oidset omitted;
    struct oidset_iter oit;
    struct object_id *oid = NULL;
    int omitted_count = 0;
    oidset_init(&omitted, 0);

    ...
```

Replace the call to `traverse_commit_list()` with `traverse_commit_list_filtered()` and pass a pointer to the `omitted` oidset defined and initialized above:

```
    ...

        traverse_commit_list_filtered(rev,
            walken_show_commit, walken_show_object, NULL, &omitted);

    ...
```

Then, after your traversal, the `oidset` traversal is pretty straightforward. Count all the objects within and modify the print statement:

```
    /* Count the omitted objects. */
    oidset_iter_init(&omitted, &oit);

    while ((oid = oidset_iter_next(&oit)))
        omitted_count++;

    printf("commits %d\nblobs %d\ntags %d\ntrees %d\nomitted %d\n",
        commit_count, blob_count, tag_count, tree_count, omitted_count);
```

By running your walk with and without the filter, you should find that the total object count in each case is identical. You can also time each invocation of the `walken` subcommand, with and without `omitted` being passed in, to confirm to yourself the runtime impact of tracking all omitted objects.

Changing the Order

Finally, let’s demonstrate that you can also reorder walks of all objects, not just walks of commits. First, we’ll make our handlers chattier - modify `walken_show_commit()` and `walken_show_object()` to

print the object as they go:

```
#include "hex.h"

...

static void walken_show_commit(struct commit *cmt, void *buf)
{
    trace_printf("commit: %s\n", oid_to_hex(&cmt->object.oid));
    commit_count++;
}

static void walken_show_object(struct object *obj, const char *str, void *buf)
{
    trace_printf("%s: %s\n", type_name(obj->type), oid_to_hex(&obj->oid));

    ...
}
```

> Since we will be examining this output directly as humans, we'll use `trace_printf()` here. Additionally, since this change introduces a significant number of printed lines, using `trace_printf()` will allow us to easily silence those lines without having to recompile.

(Leave the counter increment logic in place.)

With only that change, run again (but save yourself some scrollback):

```
$ GIT_TRACE=1 ./bin-wrappers/git walken 2>&1 | head -n 10
```

Take a look at the top commit with `git` `show` and the object ID you printed; it should be the same as the output of `git` `show` `HEAD`.

Next, let's change a setting on our `struct` `rev_info` within `walken_object_walk()`. Find where you're changing the other settings on `rev`, such as `rev→tree_objects` and `rev→tree_blobs_in_commit_order`, and add the `reverse` setting at the bottom:

```
    ...

    rev->tree_objects = 1;
    rev->blob_objects = 1;
    rev->tag_objects = 1;
```

```
    rev->tree_blobs_in_commit_order = 1;
    rev->reverse = 1;


    ...
```

Now, run again, but this time, let’s grab the last handful of objects instead of the first handful:

```
$ make
$ GIT_TRACE=1 ./bin-wrappers/git walken 2>&1 | tail -n 10
```

The last commit object given should have the same OID as the one we saw at the top before, and running `git` `show` `<oid>` with that OID should give you again the same results as `git` `show` `HEAD`. Furthermore, if you run and examine the first ten lines again (with `head` instead of `tail` like we did before applying the `reverse` setting), you should see that now the first commit printed is the initial commit, `e83c5163`.

Wrapping Up

Let’s review. In this tutorial, we:

- Built a commit walk from the ground up
- Enabled a grep filter for that commit walk
- Changed the sort order of that filtered commit walk
- Built an object walk (tags, commits, trees, and blobs) from the ground up
- Learned how to add a filter-spec to an object walk
- Changed the display order of the filtered object walk

□□□ #2
□ □□□ □□□ 2026-10-31 19:44:59 UTC
□ □□□ □□□ 2026-10-31 19:45:00 UTC