Exercise: Basic extension

This exercise creates a minimal extension skeleton, with a single SQL function that does nothing.

Extensions are useful because they allow adding code without having to rebuild / reinitialize cluster. If you can, start experimenting with an extension instead of hacking on the core right away, it will make your life much easier.

Hint: Start by looking for extensions that alredy do something similar, and use them as a starting point.

Steps:

1. Go to the "extension" directory. The extension is called "hacking".
2. Explore the "hacking.control" file, with metadata. This is used by the server to know which version to install and various other features of the extension.
   For more details about all the available options see:
   https://www.postgresql.org/docs/current/extend-extensions.html#EXTEND-EXTENSIONS-FILES
3. Explore the "hacking.c" file. This is a minimal source, defining a function that is callable through SQL and that returns "nothing".
4. Explore the "sql/hacking--1.0.0.sql" script, which is what defines the SQL definitions for the C code - in this case a single function. The naming convention uses semantic versioning and upgrade scripts.
5. Explore the "Makefile" which defines how to build and install the extension, and various other things (testing, ...). This is a pretty regular makefile, except that it utilizes PGXS to interact with the Postgres installation (e.g. to determine where to install various files).
   For details about PGXN see: https://www.postgresql.org/docs/current/extend-pgxs.html
6. Run `make install` in the extension directory.
7. Connect to the database and execute `CREATE EXTENSION hacking`.
8. Run `SELECT hacking_function()`.

Exercise: SQL regression test

This exercise adds a trivial SQL regression test, to do simple testing of the extension code.

SQL tests simply define (a) file with SQL queries/commandss, executed in a single session/connection, and (b) another file with expected output. This is a great way to do basic testing of changes that affect results of SQL queries etc.

Steps:

1. Go to the "extension" directory. This is the same extension as in the previous exercise.
2. Try running `make check` and `make installcheck`.
   `make check` is not supported for out-of-tree extension, and fails
   `make installcheck` is supported, but requires a running instance
3. If you don't have an instance running, create one and try running `make installcheck` again. It should succeed (there are no tests).
4. The `Makefile` expects files with SQL tests in the `test/sql` directory, so create `test/sql/mytest.sql` and add some SQL commands to it. For example a trivial command:

   ```
   SELECT 1;
   ```

5. Now run `make installcheck` again. It should fail, because there is no file with expected output yet - add `test/expected/mytest.out` and run `make installcheck` again. This time it fails because the actual and expected outputs differ. See `regressions.diff` with a comparison of the files.
6. Modify the expected output file to have the correct output, as produced by the SQL script. Run `make installcheck` again, this time it should pass.
7. The test is not testing any actual extension code yet. Modify the SQL script to include this

   ```
   CREATE EXTENSION hacking;
   SELECT hacking_function();
   ```

   and run `make installcheck` again. If you haven't done `make install` earlier, it may fail because of that - so install it.
   It may also fail because the expected output is not updated yet, so update it and run the test, to make sure it passes.

Exercise: TAP regression test

This exercise adds a basic regression test written in Perl using TAP (Test Anything Protocol).

Each TAP test is a single Perl script, checking results of various actions (e.g. result of a query, exit code of a binary ...). Compared to SQL regression tests (in previous exercise), TAP allows imperative programming and thus more complex workflows, multiple instances, restarts, etc.

We'll create a minimum TAP test, that initializes an instance with a custom configuration, checks a result of a result, restarts the instance and checks the log. And then shuts the instance.

Steps:

1. Go to the "extension" directory, create an empty file `t/001_basic.sql` and run `make installcheck`. It should try to run the test and fail, complaining about the file not defining any tests.
2. Add this minimal skeleton of a TAP test to the file. It create an instance, starts it, stops it, and finishes testing.

   ```
   use strict;
   use warnings FATAL => 'all';
   use PostgreSQL::Test::Cluster;
   use PostgreSQL::Test::Utils;
   use Test::More;
   my $node = PostgreSQL::Test::Cluster->new('main');
   $node->init;
   $node->start;
   ... some tests ...
   $node->stop;
   done_testing();

   Try running `make installcheck` again. It should fail, there are
   still no "checks" in the test.
   ```

3. To run a query, you'll need to use a function `safe_sql`, defined in the PostgreSQL Cluster module.

   ```
   $node->safe_psql($dbname, $sql) => stdout
   ```

   Use it to (a) create the extension in a database (you may use either "postgres" or create a new one), and (b) run the function defined in the extension.
   Searching for definitions in the PostgreSQL git repository can be easily done using `git grep`, for example like this:

   ```
   git grep safe_psql -- *.pm
   ```

   There's also a lot of existing TAP tests, which you can find using

   ```
   git grep safe_psql -- *.pl
   ```

   Use them as an inspiration, i.e. copy one of them.
4. You'll also need to add a "check" for the TAP test to actually test anything. One of the existing TAP tests certainly does that using Perl functions as `like(...)` or `is(...)`.
5. Once you add the missing commands, try running `make installcheck`.
6. There's a lot of TAP tests already, mostly in the `src/test/...` subdirectories. Go and explore e.g. `src/test/recovery` (which tests recovery and physical replication) or `src/test/subscription`, which tests logical replication, etc.

Exercise: Basic logging

Let's add some basic logging to the C code, writing messages either to server log or to the client. This is mostly about `elog` and `ereport` functions, defined in Postgres.

Steps:

1. Open the `hacking.c` file, and add this to the `hacking_function`:

   ```
   elog(WARNING, "hello world");
   ```

   Then rebuild the extension by doing `make install` (and also reopen connection to the database, to load the new `.so`).
2. Now run the function from `psql`:

   ```
   SELECT hacking_function();
   ```

   You should see the warning.
3. Search for the `elog` definition, e.g. using `git grep` again:

   ```
   git grep elog -- *.h
   ```

   which should point you to `src/include/utils/elog.h`.
4. Try changing the level to `ERROR`, and run the function again and check the server log.
5. Try setting `log_error_verbosity = verbose`, run the function and check the server log again.
6. The `elog` API is simple, but very limited - no translation support, no context information, ... It's meant to be used only for internal messages the user is not expected to see (e.g. can't happen errors).
   There's a more elaborate `ereport` interface, addressing all those limitations:

   ```
   ereport(ERROR,
           (errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
            errmsg("compression method lz4 not
   supported"),
            errdetail("This functionality requires the
   server to be built with lz4 support.")));
   ```

   It's defined in the same `elog.h` header, and it supports "context" in the form of `errcode`, `errmsg`, `errdetail`, `errhint` etc. It's porobably easier to look for places that emit similar message (e.g. check privileges, data types, ...).
   The errcodes are defined in the SQL standard and you may find a list in the `src/backend/utils/errcodes.txt` file.
7. Modify the extension to use the `ereport` interface. Add `errhint` and `errdetail` with additional details that should be logged.
8. Try runinng the tests using `make installcheck` again, fix them.

Exercise: Using asserts

Asserts are a convenient way to enforce conditions (invariants) during development. This should prevent "can't happen" cases and verify "must happen" cases.

This is an "extension" of the asserts provided by standad library, integrated into the Postgres configure / build system. In production builds asserts are disabled.

Steps:

1 Open the `hacking.c` file, and add this to the `hacking_function`:

```
Assert(false);
```

This will always fail and call "abort()". So rebuild/reinstall the extension, and run tests using `make installcheck`. Did it fail?

2 If it didn't fail, that's probably because you did not enable asserts when building Postgres. So use `--enable-casserts` configure flag and rebuild Postgres

```
./configure --enable-cassert  ... CPPFLAGS="-O0
-ggdb3"
./make -s -j4 install
```

And try running the tests again. It should fail. (The `CPPFLAGS` is optional - it disables optimizations, which makes it easiert to analyze core dumps.)

3 It's often useful to investigate why/where an ABORT happened, which you can approach in various ways. First, inspect the server log, which may have information about the command that failed.

4 Alternatively, attach debugger (e.g. `gdb`) to the backend before running the command that fails. The PID may be easily determined using the `pg_backend_pid()` function, and `gdb` can attach to the process using the `-p` switch.

```
SELECT pg_backend_pid();
 pg_backend_pid
----------------
         802583

gdb -p 802583
```

Make the backend execute using the `c` command, and invoke the SQL function that triggers the assert. The `gdb` session should interrupt, allowing you to investigate.

```
SELECT hacking_function()
```

6 Alternatively, you may enable `core` files, so that the failing backend dumps memory into a file. To do that, you may need to (a) set the limit on core file size, and (b) specify where the core files should be written.
For (a), run `ulimit -c` and if it says `0`, then modify `/etc/security/limits.conf` to make it `unlimited`. You may need to start a new session and restart the instance from it.
For (b), run `sysctl kernel.core_pattern` and make sure it points to existing directory. There's a list of format specifiers for the field in kernel. For example this will ensure core files are written to `/mnt/data/cores` directory, and each `core` file will have PID suffix:
`sysctl -w kernel.core_pattern=/mnt/data/cores/core.%p`
With this in place (and instance restarted), run `installcheck` again. You should get a core file.

6 Now inspect the core file using `gdb` by speficying the paths to the postgres binary and the core file.

```
gdb ~/builds/master/bin/postgres
/mnt/data/cores/core.34432
```

7 Now you can inspect a complete state of the process at the moment of the crash. The process it not running, of course, but you can use gdb commands like `print`, `up`, `down` etc.

Exercise: Simple data types

Postgres has an extensible type system, but in principle you can divide the data types in two dimensions:

- passed by value vs. passed by reference
- fixed-length vs. variable length

There are no variable-length types passed by value, which means there are three combinations:

- passed by value / fixed length (e.g. integer)
- passed by reference / fixed length (e.g. macaddr)
- passed by reference / variable length (e.g. text)

This exercise is about the first category, which is the simplest to work with. The next exercise will be about the third category.

You can see all this in the `pg_type` catalog, which lists all the data types known to Postgres.

The types between SQL and C are quite different, and the C type system is not as extensible as in Postgres. These type systems are not easily compatible, e.g. the casting is much more restrictive in C, so there needs to be some "compatibility" layer.

Similarly for the function call interface - the call conventions are very different (SQL allows overloading, ...).

The compatibility layer between SQL and C types is represented by the "Datum" data type, which can represent any SQL data type. If you only have a `Datum` value, you don't know if it's for a by-value type (and how many bytes are valid) or for by-reference type (i.e. a pointer).

You need enough "context" - either implicit (when accessing a function argument, you know the type) or explicit (when processing tuples, you have tuple descriptor with info about types of attributes).

Steps:

1. Open the `hacking.c` file, and add copy the `hacking_function` into a new `hacking_function_2` one.
2. In the SQL script, copy the function definition, add a new `int` argument and make sure it references the new `hacking_function_2`.
3. Modify the C function to have this piece of code:

   ```
   int32  value = PG_GETARG_INT32(0);
   elog(WARNING, "value = %d", value);
   ```

4. Rebuild / reinstall the extension (you have to drop / create it, and reconnect to load the new `.so`).
5. Try calling the function with the `int` argument.

   ```
   SELECT hacking_function(1244);
   ```

   And you should get a warning message.
6. Explore what `PG_GETARG_INT32` actually does:

   ```
   git grep PG_GETARG_INT32 -- *.h
   ```

   The interesting parts are in `fmgr.h` (for int and many other built-in data types), where it says this:

   ```
   #define PG_GETARG_INT32(n)
   DatumGetInt32(PG_GETARG_DATUM(n))
   ```

7. `PG_GETARG_DATUM` extracts the argument `0` from function arguments, which are hidden by another macro `PG_FUNCTION_ARGS`.

   ```
   #define PG_GETARG_DATUM(n)  (fcinfo->args[n].value)
   ```

   Ultimately it's accessing `FunctionCallInfoBaseData`, i.e. an array of `Datum` values, passed to the function (and some more data). The "fmgr" means "function manager" and it's the layer at the boundary between SQL and C functions.

8 `DatumGetInt32` is what does the actual casting. You can see what it does in `postgres.h` - it mostly just casts to/from a `Datum` value.

9 Look at `DatumGetInt64`. Why does it need to be more complicated?

10 SQL functions can handle arguments in smarter ways, using `PG_NARGS` and `PG_ARGISNULL` to handle calls with NULLs and different numbers of arguments etc.

Exercise: Varlena data types

Remember that there are three combinations for data types in PostgreSQL:

- passed by value / fixed length (e.g. integer)
- passed by reference / fixed length (e.g. macaddr)
- passed by reference / variable length (e.g. text)

This exercise is about the third category.

Variable length data types, often called as varlena, refer to data whose length can vary. There are several types with varying lengths, one example which will be used in this exercise is `text`.

You can find other varlena type and more information about varlena struct by running the following command.

```
git grep 'struct varlena' -- '*\.h'
```

You'll find more varlena types specifically in `c.h`.

Steps:

1. Open the `hacking.c` file, and create a `hacking_function_3` function similar to the `hacking_function_2` from previous exercise.
2. In the SQL script, copy the function definition from previous exercise, change its argument type to `text` and make sure it references the new `hacking_function_3`.
3. Modify the C function to have this piece of code:

   ```
   struct varlena *msg = PG_GETARG_RAW_VARLENA_P(0);
   ```

   `PG_GETARG_RAW_VARLENA_P` is similar to `PG_GETARG_INT32` that we used before. Main difference is that it returns the argument as `varlena` instead of an integer.
   You'll find appropriate `PG_GETARG_*` macros in `fmgr.h` for most data types. Choose the one that would work with your argument type.
   Notice that there are already `PG_GETARG_TEXT_*` macros which can be used with `text` type. This will be covered in the next exercise.
4. Modify the C function to add the following code:

   ```
   char   *msg_str = VARDATA_ANY(msg);
   int   msg_size = VARSIZE_ANY_EXHDR(msg);
   elog(WARNING, "message = %s, size = %d", msg_str,
   msg_size);
   ```

   `VARDATA_ANY` and `VARSIZE_ANY_EXHDR` are utility macros to extract the data and size from a varlena. Check `varatt.h` for more such macros.
5. Do not only log the text, but also return the string. Change the `PG_RETURN_VOID` line with:

   ```
   PG_RETURN_CSTRING(msg_str);
   ```

   Similar to `PG_GETARG_*` macros, you can chech other variants of `PG_RETURN_*` macros.
6. Rebuild / reinstall the extension (you have to drop / create it, and reconnect to load the new `.so`).
7. Try calling the function with the `text` argument.

   ```
   SELECT hacking_function('hello world');
   ```

   And you should get a warning message along with the returned value.
8. However, it likely prints some garbage data after the string. Why? Try fixing that.

Exercise: TOASTed data types

This is a follow up to the previous exercise about `varlena` data types, that allow values with arbitrary length. Well, up to 1GB. Still, that's much more than can fit into a single data page (8kB by default).

So how does Postgres handle that? Let's explore that a bit!

Steps:

1. Create a table with a single `text` column, and insert a long random value into it:

    ```
    CREATE TABLE t (a text);
    INSERT INTO t SELECT string_agg(md5(i::text),'') FROM
    generate_series(1,1000) s(i);
    -- see how much disk space it uses
    \d+ t
    -- see how long it is
    SELECT length(a) FROM t;
    ```

2. Try calling the `hacking_function(text)` on the value:

    ```
    SELECT hacking_function(a) FROM t
    ```

    This should fail, printing some garbage. How come?
3. Postgres stores over-sized values (that can't fit onto a single 8kB data page) in a TOAST table. It can also compress them transparently, using either pglz or lz4. Each table with attributes that might need TOAST have an associated TOAST relation.

    ```
    SELECT toastrelid::regclass FROM pg_class WHERE
    relname = 't';
    SELECT * FROM pg_toast.pg_toast_24801;
    ```

4. If a value gets TOASTed (longer than some threshold ...), it's replaced with a TOAST pointer which contains the key into the TOAST relation. Which is why the function printed nonsense.
5. Explore the possible ways an attribute may be stored. This depends on the data type, but also on the column options.
    See docs for details about PLAIN/EXTENDED/EXTERNAL/MAIN, compression etc.
6. All the details and macros for accessing `varlena` types are in `varatt.h` and you can see how it maps to the storage features we just discussed. There are variants for EXTERNAL, COMPRESSED, ...
7. It's time to fix the function, to print valid data. The important part is explained by this comment:

    ```
    /*
     * In consumers oblivious to data alignment, call
    PG_DETOAST_DATUM_PACKED(),
     * VARDATA_ANY(), VARSIZE_ANY() and
    VARSIZE_ANY_EXHDR().  Elsewhere, call
     * PG_DETOAST_DATUM(), VARDATA() and VARSIZE().
    Directly fetching an int16,
     * int32 or wider field in the struct representing
    the datum layout requires
     * aligned data.  memcpy() is alignment-oblivious, as
    are most operations on
     * datatypes, such as text, whose layout struct
    contains only char fields.
     *
     * Code assembling a new datum should call VARDATA()
    and SET_VARSIZE().
     * (Datums begin life untoasted.)
     *
     * Other macros here should usually be used only by
    tuple assembly/disassembly
     * code and code that specifically wants to work with
    ```

```
still-toasted Datums.
 */
```

8 Modify the function to use `PG_DETOAST_DATUM(), VARDATA() and VARSIZE()`. Try doing `git grep PG_DETOAST_DATUM` to see how to call it.
9 Have you noticed there's `DatumGetTextP`? How would you use this?
10 Try to find the code that actually does the detoasting.
You'll need to get through multiple layers of indirection (macro -> AM callback -> ...), but ultimately you should get to

- `detoast_attr` in `detoast.c`
- `heap_fetch_toast_slice` in `heaptoast.c`

11 There's also `PG_DETOAST_DATUM_SLICE` / `DatumGetTextPSlice`. Why would it be interesting?
12 It's possible to define functions with `internal` data type, and use it to pass "raw" pointers (and use `PG_GETARG_POINTER`). In what situations might that be useful?

Exercise: Memory contexts

Handling memory allocations in C is difficult and error-prone. You have to track exactly what you allocated and then free it at the appropriate time. It's easy to end up with either memory leaks, double-free or use-after-free bugs etc.

This is especially true true for systems where the different pieces of memory have vastly different life spans. Some memory needs to exist for as long as the backend is running, other pieces are per transaction, query, operation, row, ...

This is why Postgres implements the concept of "memory context", which is "collection" of allocations, grouped by life span. Memory context form a hierarchy, from long-lived to short-lived.

There is a small number of pre-defined memory contexts:

```
// memutils.h
extern PGDLLIMPORT MemoryContext TopMemoryContext;
extern PGDLLIMPORT MemoryContext ErrorContext;
extern PGDLLIMPORT MemoryContext PostmasterContext;
extern PGDLLIMPORT MemoryContext CacheMemoryContext;
extern PGDLLIMPORT MemoryContext MessageContext;
extern PGDLLIMPORT MemoryContext TopTransactionContext;
extern PGDLLIMPORT MemoryContext CurTransactionContext;
```

for contexts that are expected to exist. The names should hint what the purpose is. There is also `CurrentMemoryContext` which always points to the "current" context (e.g. what the caller sets before calling a function). To switch to a different memory context the idiom is:

```
MemoryContext oldcxt = MemoryContextSwitchTo(newcxt);
... allocate stuff in current context ...
MemoryContextSwitchTo(oldcxt);
```

You may create a new memory context (e.g. to wrap all allocations in a function, and free them at once). There are multiple implementations, the most widely used / general purpose is `AllocSet` (see `aset.c`).

Creating a context is simple:

```
MemoryContext ctx = AllocSetContextCreate(CurrentMemoryContext,
                                          "name of the context",

ALLOCSET_DEFAULT_SIZES);
```

After this, you can access the context using the API in `memutils.h`.

To manage memory in a context, you must not use `malloc` / `free` or any of that glibc API. Instead, use `palloc` / `pfree`, or the API defined in `palloc.h`.

Memory contexts also serve as "cache" on top of `malloc`. That is, when you do `palloc`, the context will allocate a larger block of memory using `malloc` and then use it to handle many `palloc` calls before the next `malloc` call. Also, `pfree` does not return memory to glibc, it just returns it to the context (and it can use it for a `palloc` with similar size).

To destroy a memory context, use either `MemoryContextDelete` or (if you want to keep using it) `MemoryContextReset`.

In the following exercise we'll modify the `hacking_function` to create a new memory context, do the allocations in it, and then discard the allocated memory.

Steps:

1. Open the `hacking_function` from previous exercise, and create a new memory context using `AllocSetContextCreate` above. Also add `MemoryContextDelete` at the end.
2. Switch to the memory context (and then back) using the `MemoryContextSwitchTo` idiom mentioned above.

3. Add some `palloc` calls between the switch-to calls, to allocate memory in the new memory context. It could be a couple bytes, kilobytes or MBs.
4. There's a handy function `MemoryContextStats` for inspecting memory context state - how much memory is allocated, etc. Try adding a call with either `TopMemoryContext` or the new context, and see what happens. The output goes to server log.
5. The function can also be called from gdb, which is useful while investigating issues. Attach GDB to running backend and try this:

   ```
   call MemoryContextStats(TopMemoryContext);
   ```

6. Memory contexts have built-in protections in development builds (with asserts enabled), e.g. randomization of memory. This is useful to identify use-after-free bugs etc. because the patterns are quite typical.
   Try defining a struct with int32/int64/char[] fields, allocate it using palloc, free it (or destroy the context), and try accessing it. Or inspect the contents using gdb.
7. Memory contexts also have optional valgrind support, if you build with `CPPFLAGS="-DUSE_VALGRIND"`. This is useful for investigating strange memory corruption issues - valgrind tells you not only where the crash happened, but when was the memory allocated/freed etc. It's much slower, though (roughly 100x slower).
8. Are memory contexts a solution to memory leaks? Can you think about cases when we'd still leak memory?

Exercise: Adding GUC parameters

GUC stands for Grand Unified Configuration, and it's also used as "config parameter" in the postgresql.conf file. Adding a new parameter can be handy during development, to allow "enabling/disabling" the new behavior in an easy way.

A new GUC can be added in two ways - in an extension (easy) and in the core server (harder). There are differences, e.g. GUC from an extension has to have a "prefix".

In the following exercise we'll add a new GUC - first to the core, then in an extension.

Steps:

1. Find and open the file `guc_tables.c`, which defines which GUCs are built into the core server itself. Skim through the file, to get some basic idea what it contains - some basic " definitions" first (e.g. labels, constants, ...) followed by tables of options with different types (bool, int, ...).
2. Find lines with `ConfigureNames` definitions. What types of options we support, what types? How do the tables end?
3. Let's add a new config option - a true/false flag to drive something. The easiest way is to find a similar entry, copy and modify it. Copy `enable_seqscan` option, rename it to `enable_my_feature` or whatever else.
4. You'll need to point the GUC to a new variable. `enable_seqscan` uses variable defined in `cost.h` and `cost.c`, so just copy those too. Now every place that includes `cost.h` can use the C variable backing the GUC option.
5. Rebuild the server, restart the instance, try setting the new `GUC`.
6. The `config_bool` GUC option type is defined in `guc_tables.h`, see what are the fields. What are the hooks for?
7. Now let's do a similar thing from an extension. There's an API for adding GUC options - try

   ```
   git grep DefineCustom
   ```

   and see how e.g. `passwordcheck` extension does that.
8. Try adding a new GUC option to the extension. You'll need to copy both the `DefineCustomIntVariable` and `MarkGUCPrefixReserved` calls, and define the C variable referenced by the define call.
9. Use the GUC in one of the functions (e.g. add it to `elog` call), set the option and see the function uses the new value.
10. Both approaches allow who/when can set the option (e.g. PGC_USERSET) and logical group (e.g. DEVELOPER_OPTIONS).

Exercise: Accessing relations

Let's do something more complicated - let's access a relation (=table). In this exercise we'll learn to lock a relation and access the tuple descriptor, in the next exercise we'll learn to read rows from it.

To access a relation, we need to lock the relation and get a descriptor (metadata describing the structure). Then we need to do whatever we want to do (e.g. print the descriptor info), and finally close the relation (which unlocks it).

Steps:

1. Define a new SQL function that gets a `regclass` data type. This is a special data type that accepts a relation name, and translates it into an `oid`. Point it at a new C function that can access the parameter using `PG_GETARG_OID`.
2. Get the `OID` and open it using `table_open`. which returns a `Relation` struct with all kinds of metadata of the table. Look at the definition of `RelationData` in `src/include/utils/rel.h`. Also look at `src/include/access/table.h`.
3. The necessary lock level depends on what you intend to do with the relation. For our purpose `AccessShareLock` is good enough.
4. Make sure to also close the relation using `table_close`.
5. We want to print information about structure of the table, i.e. attributes and types. That is represented by `TupleDesc`, and you can obtain it from `Relation` using `RelationGetDescr`.

   ```
   TupleDesc tdesc = RelationGetDescr(rel);
   ```

6. Then you need to iterate over the attributes. There are `tdesc->natts` attributes in the descriptor, 0-indexed. To get i-th attribute, use `TupleDescAttr` which returns `FormData_pg_attribute` struct. See `src/include/catalog/pg_attribute.h` for details of the fields.
7. For each attribute, print `attnum`, `attname`, `atttypid`, `attlen` and anything else you deem interesting.
8. Build the extension and try running it on arbitrary table.

Exercise: Accessing tuples

In the previous exercise we opened a relation and printed some basic info from the tuple descriptor. In this example we'll actually access tuples - we'll scan the relation (using sequential scan), count the tuples, and possibly also parse the tuples.

Steps:

1. To access tuples, we need to do the same basic stuff as in the earlier exercise - open/lock the relation, etc. But then we also need to start a "scan" using `table_beginscan`.
   This is part of the `table AM` API defined in `access/tableam.h`, so maybe take a look at that. Then search for places already doing this (`pgrowlocks.c`). We're however going to mix this with "heap" API, because we know that's the storage format. But it's not quite right.
2. Once we have a scan, you can call `heap_next()` to get `HeapTuple` from the scan, until you get NULL.
   Simply increment a counter, and return it from the function (so the function should now count rows, not attributes). Overall it should look like this:

   ```
   Datum
   hacking_function_4(PG_FUNCTION_ARGS)
   {
    // get OID of relation for argument 0
    // open the relation (table_open)
    // begin a scan on the relation (table_beginscan)
    // iterate tuples (heap_getnext)
    // return close the relation (table_close)
    // return the count (PG_RETURN_INT32)
   }
   ```
3. Rebuild/reinstall the extension, try calling the function. Did it work, did it print some warnings?
4. Add the missing `table_endscan` call.
5. Instead of just counting the rows, you can try doing something else with the tuples. For example, you can use `HeapTupleHeaderGetNatts` to show the number of attributes in a tuple.
   Question: When could this be different from the tuple descriptor?
   Similarly, you can access attributes using the `heap_getattr` function, which returns `Datum` value and sets `isnull` flag. Search for places doing that, and try doing that.
6. Another thing you might try is defining scan keys (i.e. conditions). For that you need to allocate `ScanKeyData` and pass it to the `beginscan` function.
   See for example how `aclchk.c` does this. It's on catalogs, but the general approach is exactly the same.
7. These examples mix `heap` and `table AM` code, because we know the relation is `heap` and thus the rows are `HeapTuple`. But in general that would not work, and we should use the `table AM` consistently. But that uses a concept of "slot" as an abstract tuple format. The executor however does exactly that.