This README is just a fast *quick start* document. You can find more detailed documentation at redis.io.

What is Redis?

Redis is often referred to as a *data structures* server. What this means is that Redis provides access to mutable data structures via a set of commands, which are sent using a *server-client* model with TCP sockets and a simple protocol. So different processes can query and modify the same data structures in a shared way.

Data structures implemented into Redis have a few special properties:

- Redis cares to store them on disk, even if they are always served and modified into the server memory. This means that Redis is fast, but that it is also non-volatile.
- The implementation of data structures emphasizes memory efficiency, so data structures inside Redis will likely use less memory compared to the same data structure modelled using a high-level programming language.
- Redis offers a number of features that are natural to find in a database, like replication, tunable levels of durability, clustering, and high availability.

Another good example is to think of Redis as a more complex version of memcached, where the operations are not just SETs and GETs, but operations that work with complex data types like Lists, Sets, ordered data structures, and so forth.

If you want to know more, this is a list of selected starting points:

- Introduction to Redis data types. https://redis.io/topics/data-types-intro
- Try Redis directly inside your browser. https://try.redis.io
- The full list of Redis commands. https://redis.io/commands
- There is much more inside the official Redis documentation. https://redis.io/documentation

Building Redis

Redis can be compiled and used on Linux, OSX, OpenBSD, NetBSD, FreeBSD. We support big endian and little endian architectures, and both 32 bit and 64 bit systems.

It may compile on Solaris derived systems (for instance SmartOS) but our support for this platform is *best effort* and Redis is not guaranteed to work as well as in Linux, OSX, and *BSD.

It is as simple as:

```
% make
```

To build with TLS support, you'll need OpenSSL development libraries (e.g. libssl-dev on Debian/Ubuntu) and run:

```
% make BUILD_TLS=yes
```

To build with systemd support, you'll need systemd development libraries (such as libsystemd-dev on Debian/Ubuntu or systemd-devel on CentOS) and run:

```
% make USE_SYSTEMD=yes
```

To append a suffix to Redis program names, use:

```
% make PROG_SUFFIX="-alt"
```

You can build a 32 bit Redis binary using:

```
% make 32bit
```

After building Redis, it is a good idea to test it using:

```
% make test
```

If TLS is built, running the tests with TLS enabled (you will need `tcl-tls` installed):

```
% ./utils/gen-test-certs.sh
% ./runtest --tls
```

Fixing build problems with dependencies or cached build options

Redis has some dependencies which are included in the `deps` directory. `make` does not automatically rebuild dependencies even if something in the source code of dependencies changes.

When you update the source code with `git pull` or when code inside the dependencies tree is modified in any other way, make sure to use the following command in order to really clean everything and rebuild from scratch:

```
make distclean
```

This will clean: jemalloc, lua, hiredis, linenoise.

Also if you force certain build options like 32bit target, no C compiler optimizations (for debugging purposes), and other similar build time options, those options are cached indefinitely until you issue a `make distclean` command.

Fixing problems building 32 bit binaries

If after building Redis with a 32 bit target you need to rebuild it with a 64 bit target, or the other way around, you need to perform a `make distclean` in the root directory of the Redis distribution.

In case of build errors when trying to build a 32 bit binary of Redis, try the following steps:

- Install the package libc6-dev-i386 (also try g++-multilib).
- Try using the following command line instead of `make 32bit`: `make CFLAGS="-m32 -march=native" LDFLAGS="-m32"`

Allocator

Selecting a non-default memory allocator when building Redis is done by setting the `MALLOC` environment variable. Redis is compiled and linked against libc malloc by default, with the exception of jemalloc being the default on Linux systems. This default was picked because jemalloc has proven to have fewer fragmentation problems than libc malloc.

To force compiling against libc malloc, use:

```
% make MALLOC=libc
```

To compile against jemalloc on Mac OS X systems, use:

```
% make MALLOC=jemalloc
```

Monotonic clock

By default, Redis will build using the POSIX clock_gettime function as the monotonic clock source. On most modern systems, the internal processor clock can be used to improve performance. Cautions can be found here: http://oliveryang.net/2015/09/pitfalls-of-TSC-usage/

To build with support for the processor's internal instruction clock, use:

```
% make CFLAGS="-DUSE_PROCESSOR_CLOCK"
```

Verbose build

Redis will build with a user-friendly colorized output by default. If you want to see a more verbose output, use the following:

```
% make V=1
```

Running Redis

To run Redis with the default configuration, just type:

```
% cd src
% ./redis-server
```

If you want to provide your redis.conf, you have to run it using an additional parameter (the path of the configuration file):

```
% cd src
% ./redis-server /path/to/redis.conf
```

It is possible to alter the Redis configuration by passing parameters directly as options using the command line. Examples:

```
% ./redis-server --port 9999 --replicaof 127.0.0.1 6379
% ./redis-server /etc/redis/6379.conf --loglevel debug
```

All the options in redis.conf are also supported as options using the command line, with exactly the same name.

Running Redis with TLS:

Please consult the TLS.md file for more information on how to use Redis with TLS.

Playing with Redis

You can use redis-cli to play with Redis. Start a redis-server instance, then in another terminal try the following:

```
% cd src
% ./redis-cli
redis> ping
PONG
redis> set foo bar
OK
redis> get foo
"bar"
redis> incr mycounter
(integer) 1
redis> incr mycounter
(integer) 2
redis>
```

You can find the list of all the available commands at https://redis.io/commands.

Installing Redis

In order to install Redis binaries into /usr/local/bin, just use:

```
% make install
```

You can use `make PREFIX=/some/other/directory install` if you wish to use a different destination.

Make install will just install binaries in your system, but will not configure init scripts and configuration files in the appropriate place. This is not needed if you just want to play a bit with Redis, but if you are installing it the proper way

for a production system, we have a script that does this for Ubuntu and Debian systems:

```
% cd utils
% ./install_server.sh
```

Note: `install_server.sh` will not work on Mac OSX; it is built for Linux only.

The script will ask you a few questions and will setup everything you need to run Redis properly as a background daemon that will start again on system reboots.

You'll be able to stop and start Redis using the script named `/etc/init.d/redis_<portnumber>`, for instance `/etc/init.d/redis_6379`.

Code contributions

Note: By contributing code to the Redis project in any form, including sending a pull request via Github, a code fragment or patch via private email or public discussion groups, you agree to release your code under the terms of the BSD license that you can find in the COPYING file included in the Redis source distribution.

Please see the CONTRIBUTING file in this source distribution for more information. For security bugs and vulnerabilities, please see SECURITY.md.

Redis internals

If you are reading this README you are likely in front of a Github page or you just untarred the Redis distribution tar ball. In both the cases you are basically one step away from the source code, so here we explain the Redis source code layout, what is in each file as a general idea, the most important functions and structures inside the Redis server and so forth. We keep all the discussion at a high level without digging into the details since this document would be huge otherwise and our code base changes continuously, but a general idea should be a good starting point to understand more. Moreover most of the code is heavily commented and easy to follow.

Source code layout

The Redis root directory just contains this README, the Makefile which calls the real Makefile inside the `src` directory and an example configuration for Redis and Sentinel. You can find a few shell scripts that are used in order to execute the Redis, Redis Cluster and Redis Sentinel unit tests, which are implemented inside the `tests` directory.

Inside the root are the following important directories:

- `src`: contains the Redis implementation, written in C.
- `tests`: contains the unit tests, implemented in Tcl.

- `deps`: contains libraries Redis uses. Everything needed to compile Redis is inside this directory; your system just needs to provide `libc`, a POSIX compatible interface and a C compiler. Notably `deps` contains a copy of `jemalloc`, which is the default allocator of Redis under Linux. Note that under `deps` there are also things which started with the Redis project, but for which the main repository is not `redis/redis`.

There are a few more directories but they are not very important for our goals here. We'll focus mostly on `src`, where the Redis implementation is contained, exploring what there is inside each file. The order in which files are exposed is the logical one to follow in order to disclose different layers of complexity incrementally.

Note: lately Redis was refactored quite a bit. Function names and file names have been changed, so you may find that this documentation reflects the `unstable` branch more closely. For instance, in Redis 3.0 the `server.c` and `server.h` files were named `redis.c` and `redis.h`. However the overall structure is the same. Keep in mind that all the new developments and pull requests should be performed against the `unstable` branch.

server.h

The simplest way to understand how a program works is to understand the data structures it uses. So we'll start from the main header file of Redis, which is `server.h`.

All the server configuration and in general all the shared state is defined in a global structure called `server`, of type `struct redisServer`. A few important fields in this structure are:

- `server.db` is an array of Redis databases, where data is stored.
- `server.commands` is the command table.
- `server.clients` is a linked list of clients connected to the server.
- `server.master` is a special client, the master, if the instance is a replica.

There are tons of other fields. Most fields are commented directly inside the structure definition.

Another important Redis data structure is the one defining a client. In the past it was called `redisClient`, now just `client`. The structure has many fields, here we'll just show the main ones:

```
struct client {
    int fd;
    sds querybuf;
    int argc;
    robj **argv;
    redisDb *db;
    int flags;
```

```
    list *reply;
    // ... many other fields ...
    char buf[PROTO_REPLY_CHUNK_BYTES];
}
```

The client structure defines a *connected client*:

- The `fd` field is the client socket file descriptor.
- `argc` and `argv` are populated with the command the client is executing, so that functions implementing a given Redis command can read the arguments.
- `querybuf` accumulates the requests from the client, which are parsed by the Redis server according to the Redis protocol and executed by calling the implementations of the commands the client is executing.
- `reply` and `buf` are dynamic and static buffers that accumulate the replies the server sends to the client. These buffers are incrementally written to the socket as soon as the file descriptor is writeable.

As you can see in the client structure above, arguments in a command are described as `robj` structures. The following is the full `robj` structure, which defines a *Redis object*:

```
typedef struct redisObject {
    unsigned type:4;
    unsigned encoding:4;
    unsigned lru:LRU_BITS; /* lru time (relative to server.lruclock) */
    int refcount;
    void *ptr;
} robj;
```

Basically this structure can represent all the basic Redis data types like strings, lists, sets, sorted sets and so forth. The interesting thing is that it has a `type` field, so that it is possible to know what type a given object has, and a `refcount`, so that the same object can be referenced in multiple places without allocating it multiple times. Finally the `ptr` field points to the actual representation of the object, which might vary even for the same type, depending on the `encoding` used.

Redis objects are used extensively in the Redis internals, however in order to avoid the overhead of indirect accesses, recently in many places we just use plain dynamic strings not wrapped inside a Redis object.

server.c

This is the entry point of the Redis server, where the `main()` function is defined. The following are the most important steps in order to startup the Redis server.

- `initServerConfig()` sets up the default values of the `server` structure.

- `initServer()` allocates the data structures needed to operate, setup the listening socket, and so forth.
- `aeMain()` starts the event loop which listens for new connections.

There are two special functions called periodically by the event loop:

1. `serverCron()` is called periodically (according to `server.hz` frequency), and performs tasks that must be performed from time to time, like checking for timed out clients.
2. `beforeSleep()` is called every time the event loop fired, Redis served a few requests, and is returning back into the event loop.

Inside server.c you can find code that handles other vital things of the Redis server:

- `call()` is used in order to call a given command in the context of a given client.
- `activeExpireCycle()` handles eviction of keys with a time to live set via the `EXPIRE` command.
- `performEvictions()` is called when a new write command should be performed but Redis is out of memory according to the `maxmemory` directive.
- The global variable `redisCommandTable` defines all the Redis commands, specifying the name of the command, the function implementing the command, the number of arguments required, and other properties of each command.

commands.c

This file is auto generated by utils/generate-command-code.py, the content is based on the JSON files in the src/commands folder. These are meant to be the single source of truth about the Redis commands, and all the metadata about them. These JSON files are not meant to be used directly by anyone directly, instead that metadata can be obtained via the COMMAND command.

networking.c

This file defines all the I/O functions with clients, masters and replicas (which in Redis are just special clients):

- `createClient()` allocates and initializes a new client.
- the `addReply*()` family of functions are used by command implementations in order to append data to the client structure, that will be transmitted to the client as a reply for a given command executed.
- `writeToClient()` transmits the data pending in the output buffers to the client and is called by the *writable event handler* `sendReplyToClient()`.
- `readQueryFromClient()` is the *readable event handler* and accumulates data read from the client into the query buffer.

- `processInputBuffer()` is the entry point in order to parse the client query buffer according to the Redis protocol. Once commands are ready to be processed, it calls `processCommand()` which is defined inside `server.c` in order to actually execute the command.
- `freeClient()` deallocates, disconnects and removes a client.

aof.c and rdb.c

As you can guess from the names, these files implement the RDB and AOF persistence for Redis. Redis uses a persistence model based on the `fork()` system call in order to create a process with the same (shared) memory content of the main Redis process. This secondary process dumps the content of the memory on disk. This is used by `rdb.c` to create the snapshots on disk and by `aof.c` in order to perform the AOF rewrite when the append only file gets too big.

The implementation inside `aof.c` has additional functions in order to implement an API that allows commands to append new commands into the AOF file as clients execute them.

The `call()` function defined inside `server.c` is responsible for calling the functions that in turn will write the commands into the AOF.

db.c

Certain Redis commands operate on specific data types; others are general. Examples of generic commands are `DEL` and `EXPIRE`. They operate on keys and not on their values specifically. All those generic commands are defined inside `db.c`.

Moreover `db.c` implements an API in order to perform certain operations on the Redis dataset without directly accessing the internal data structures.

The most important functions inside `db.c` which are used in many command implementations are the following:

- `lookupKeyRead()` and `lookupKeyWrite()` are used in order to get a pointer to the value associated to a given key, or `NULL` if the key does not exist.
- `dbAdd()` and its higher level counterpart `setKey()` create a new key in a Redis database.
- `dbDelete()` removes a key and its associated value.
- `emptyDb()` removes an entire single database or all the databases defined.

The rest of the file implements the generic commands exposed to the client.

object.c

The `robj` structure defining Redis objects was already described. Inside `object.c` there are all the functions that operate with Redis objects at a basic level, like

functions to allocate new objects, handle the reference counting and so forth. Notable functions inside this file:

- `incrRefCount()` and `decrRefCount()` are used in order to increment or decrement an object reference count. When it drops to 0 the object is finally freed.
- `createObject()` allocates a new object. There are also specialized functions to allocate string objects having a specific content, like `createStringObjectFromLongLong()` and similar functions.

This file also implements the `OBJECT` command.

replication.c

This is one of the most complex files inside Redis, it is recommended to approach it only after getting a bit familiar with the rest of the code base. In this file there is the implementation of both the master and replica role of Redis.

One of the most important functions inside this file is `replicationFeedSlaves()` that writes commands to the clients representing replica instances connected to our master, so that the replicas can get the writes performed by the clients: this way their data set will remain synchronized with the one in the master.

This file also implements both the `SYNC` and `PSYNC` commands that are used in order to perform the first synchronization between masters and replicas, or to continue the replication after a disconnection.

Script

The script unit is compose of 3 units * `script.c` - integration of scripts with Redis (commands execution, set replication/resp, ..) * `script_lua.c` - responsible to execute Lua code, uses script.c to interact with Redis from within the Lua code. * `function_lua.c` - contains the Lua engine implementation, uses script_lua.c to execute the Lua code. * `functions.c` - Contains Redis Functions implementation (FUNCTION command), uses functions_lua.c if the function it wants to invoke needs the Lua engine. * `eval.c` - Contains the `eval` implementation using `script_lua.c` to invoke the Lua code.

Other C files

- `t_hash.c`, `t_list.c`, `t_set.c`, `t_string.c`, `t_zset.c` and `t_stream.c` contains the implementation of the Redis data types. They implement both an API to access a given data type, and the client command implementations for these data types.
- `ae.c` implements the Redis event loop, it's a self contained library which is simple to read and understand.
- `sds.c` is the Redis string library, check https://github.com/antirez/sds for more information.

- `anet.c` is a library to use POSIX networking in a simpler way compared to the raw interface exposed by the kernel.
- `dict.c` is an implementation of a non-blocking hash table which rehashes incrementally.
- `cluster.c` implements the Redis Cluster. Probably a good read only after being very familiar with the rest of the Redis code base. If you want to read `cluster.c` make sure to read the Redis Cluster specification.

Anatomy of a Redis command

All the Redis commands are defined in the following way:

```
void foobarCommand(client *c) {
    printf("%s",c->argv[1]->ptr); /* Do something with the argument. */
    addReply(c,shared.ok); /* Reply something to the client. */
}
```

The command is then referenced inside `server.c` in the command table:

```
{"foobar",foobarCommand,2,"rtF",0,NULL,0,0,0,0,0},
```

In the above example `2` is the number of arguments the command takes, while `"rtF"` are the command flags, as documented in the command table top comment inside `server.c`.

After the command operates in some way, it returns a reply to the client, usually using `addReply()` or a similar function defined inside `networking.c`.

There are tons of command implementations inside the Redis source code that can serve as examples of actual commands implementations. Writing a few toy commands can be a good exercise to get familiar with the code base.

There are also many other files not described here, but it is useless to cover everything. We just want to help you with the first steps. Eventually you'll find your way inside the Redis code base :-)

Enjoy!