Project Structure Guide

This guide describes the idiomatic and scalable structure for organizing C projects that require multiple architectures, clean separation of libraries, tests, and applications, and support both static and shared builds. This structure is optimized for:

- Professional C-based systems projects
- Projects that scale from one app to many
- Multi-architecture builds (e.g., x86_64, aarch64)
- Clean test, deploy, and install paths

Directory Layout Overview

```
Debug/
├── deploy/
│   ├── x86_64/
│   │   ├── libs/
│   │   │   ├── static/           # Optional: if static libs are output
separately
│   │   │   └── shared/           # .so shared objects for deployment
│   │   ├── tests/                # CUnit or other test binaries
│   │   └── project_1/
│   │       ├── remote/bin/apps/  # Deployed binaries
│   │       └── local/bin/apps/   # Tools, debug-only binaries (optional)
│   └── aarch64/
│       └── ... same layout as above
```

Terminology

- **Domain**: Top-level purpose grouping of a build (e.g., `deploy/` for customer-bound builds).
- **Scope**: Install locality of a component (e.g., `remote/` for deliverables, `local/` for developer tools).
- **Architecture**: Target CPU architecture (e.g., `x86_64`, `aarch64`).

Build Output Policy

- **Static libraries (.a)** are used only during linking and do not need to be deployed.

- **Shared libraries (.so)** must be deployed and live in:

  - `deploy/<arch>/libs/shared/`
  - Optionally per-project under `project_n/libs/` if isolation is required

- **Test binaries** go to:

  - `deploy/<arch>/tests/`
  - Use RPATH so they can find shared libs from `../libs/shared/`

- **App binaries** go to:

  - `deploy/<arch>/project_n/remote/bin/apps/` (deployable targets)
  - `deploy/<arch>/project_n/local/bin/apps/` (optional internal tools)

CMake Integration Guidelines

- Use `cmake_parse_arguments()` for flexible `add_localized_app`, `add_localized_lib`, and `add_cunit_test` macros.

- Centralize paths in `paths-config.cmake` using `set_project_output_paths(<project>)`.

- Use RPATH in executable properties:

  ```
  set_target_properties(my_app PROPERTIES
      BUILD_RPATH "${CMAKE_LIBRARY_OUTPUT_DIRECTORY}/../libs"
      INSTALL_RPATH "$ORIGIN/../../../../libs/shared"
      INSTALL_RPATH_USE_LINK_PATH TRUE)
  ```

- Define install scopes (`remote`, `local`) and install domains (`deploy`, `internal`, etc.) with clear purpose.

Optional Enhancements

- Add `tools/`, `utils/`, or `libexec/` under each project for internal binaries
- Use `CTest` and `CTestCustom.cmake` for test dashboards
- Implement `install()` logic if packaging is required (`.deb`, `.rpm`, etc.)
- Add export targets for libraries to be consumed downstream

Example Use Case

A single repo with two projects:

- **project_1** depends on `Compare`, `IO`, `Core`, `Strings`, `Signals`
- **project_2** depends on `Threading`, `Networking`, `Core`, `Strings`, `Signals`

Deploy directory contains:

```
debug/deploy/x86_64/
├── libs/shared/
│   ├── libCore.so
│   ├── libStrings.so
│   └── libSignals.so
├── project_1/remote/bin/apps/project_1_server
└── project_2/remote/bin/apps/project_2_server
```

Both apps use shared versions of common libraries without duplication.

Summary

This structure is designed to:

- Be idiomatic for C and CMake
- Avoid manual artifact movement
- Scale across projects, libraries, and targets
- Provide clean separation between local/internal and deployable deliverables

Use this guide as a reference to keep your repository organized, extensible, and ready for multi-target deployment and testing.