OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

Architecting and Engineering
Main Memory Database Systems in Modern C
Database and Software Engineering Working Group

Magdeburg, October 20th 2017

Marcus Pinnecke, M.Sc.
Prof. Dr. Gunter Saake

Winter Term 2017/2018

Getting Started with Architecting & Engineering

submission deadline: Nov. 3rd 2017 11:59pm

This is a *per-student exercise* sheet, i.e., you are not allowed to submit a group solution.

You must submit your solution in time via a pull request to the lectures exercise Git repository[1] (cf. sheet No. 1). For this, branch from the official branch `master`, and commit your solutions in a directory `sheet_02/<your last name>`.

Prepare to present details of your solution during the tutorium.

Exercise sheet No. 2

After working on this sheet you will have learned the following

- ✓ apply a Git workflow by your own
- ✓ understand the purpose of C tools
- ✓ setup a toolchain for C development
- ✓ advance your Git skills by your own
- ✓ solve ever-occurring Git issues

This sheet consists of the following tasks:

Task 1	**Install & Setup your Tools for Development with C**	5 Point
Task 2	**More Hands on Version Control with Git**	11 Point

You successfully contributed first changes to the official repository of your project of interest. You came up with a novel fresh idea and decide to start a project by your own, called *My Main Memory Database System* (M3DB). It's time to setup the tools used to develop M3DB using the C programming language. Since you will not work on your project only by yourself, you expect some issues in the workflow, and take this as a change to improve your Git skills. Also, you want to try out the Markdown format for nice-looking readable documents. However, a friend of yours wants to join your project from the start up, and will contribute from the beginning. Unfortunately, he is unfamiliar with some more advanced Git concepts and ask you for a tutorial-like help. You agree.

Good Luck!

[1] https://github.com/Arcade-Lecture/exercises.git

Task 1 Install & Setup your Tools for Development with C 5 Point

Note: *In the following, you will install ten different kind of tools. Although all of these tools will be used during the lecture and, hence, must be installed on your system, it is enough for the submission if you consider the mandatory ones (marked with a red *) and* one *additional tool. Which tool you will install and submit additionally is up to you. However, feel to install, setup and explore all tools – you will need them later in time anyway.*

A toolchain is a sequence of programming tools out of a suite used to perform software development, and to produce executables or libraries. The basic toolchain for almost any programming language consists of tools for *compiling* source code, *linking* compiled units to an (executable) unit, *libraries* with interface to the operating system or third-party provided functionality, and a *debugger* to examine programs in order to find and eliminate bugs.

In addition to the basic toolchain, several other tools support the *development*, *inspection* and *fine-tuning*, *deployment*, *maintenance*, and testing of software products. Your task is to ensure a running basic *native* toolchain for compilation and code generating on x86/x64 and to install and setup software development tools for C.

In the following, `<repo>` is the absolute path to your local repository root directory, and `<your last name>` is the directory name of your personal submission in `<repo>/sheet_02/`.

***Compiler & Linker.** Those programs are responsible to transform source code from language to another. For C, this typically is the transformation of C source code to an output format that is acceptable by the assembler or linker program (i.e., *.o object files). Two popular tools from different vendors compete nowadays for native toolchains for C on x86/x64 architectures, *GCC* and *Clang*.

(1) Install at least one of these compilers on your machine (if not already installed) after you informed yourself which one to chose.

Tip: Your package manager or the official website will help you with that task.

(2) Run the following command in your bash

```
$ <compiler> --version >> <repo>/sheet_02/<your last name>/task_1/compiler-version
```

where `<compiler>` is the binary of the compiler of your choice (i.e., `gcc` or `clang`)

(3) Create a plain-text file `<repo>/sheet_02/<your last name>/task_1/compiler-choice` in which you give a short explanation on your reasons for the choice of a particular compiler.

Tip: Do not forget to include files added to `<repo>/sheet_02/<your last name>/task_1/` in your commit.

***Build Automation.** Those programs automate the building process of medium to large-scale software projects by automating (repeatable) tasks to resolve dependencies, execute code compilation and linking, testing, packaging into an executable or library and deployment. Classically, on UNIX-like systems that is the `make` tool which reads a `Makefile` that describes how a particular program is derived.

(1) Install `make` on your machine (if not already installed).

Tip: Your package manager or the official website will help you with that task.

(2) Run the following command in your bash

```
$ make --version >> <repo>/sheet_02/<your last name>/task_1/make-version
```

Tip: Do not forget to include files added to `<repo>/sheet_02/<your last name>/task_1/` in your commit.

Build Automation Generator. Make-based build automation is typically target of UNIX-like operating system, like Linux or macOS. However, build automation generator programs allow to specify how a program can be derived without depending to a particular build automation tool. The tool CMake is one popular build automation generator that created cross-plattform native build scripts.

(1) Install `cmake` on your machine (if not already installed).

Tip: Your package manager or the official website will help you with that task.

(2) Run the following command in your bash

```
$ cmake --version >> <repo>/sheet_02/<your last name>/task_1/cmake-version
```

Tip: Do not forget to include files added to `<repo>/sheet_02/<your last name>/task_1/` in your commit.

Libraries. A library is a pre-packaged unit that provides functionality (and sometimes other assets) to third-party developers. Typically, a software developer adds a library to a project in order to use these functionalities (or interfaces) without the need for writing code to implement that functionality. The C standard does not define a standard library having out-of-the-box advanced data structures or platform-independent interfaces. The Apache Portable Runtime (APR) aims for a library that closes this gab.

(1) Install `libapr1` and `libapr1-dev` on your machine (if not already installed).

Tip: Search for "apt-get install libapr"; your package manager with that task. Alternatively, build the library from its sources. The official website will help you with this task.

(3) Create manually a plain-text file `<repo>/sheet_02/<your last name>/task_1/libapr-version` in which you describe how you installed libapr and which version you use.

Tip: Do not forget to include files added to `<repo>/sheet_02/<your last name>/task_1/` in your commit.

***IDE**. Integrated Development Environments (IDE) are programs that aim to increase the productivity of a software developer by integrating tools (e.g., the debugger), functionality (e.g., quick navigation through the code base), language-specific editors (e.g., syntax-highlighting for C or XML), support (e.g., auto code completion), and more (e.g., automatic code generation, or context-sensitive documentation) under one hood. If and which IDE to use is (often) a choice that depends on the language, the project, third-party constraints, personal preferences, and more. We recommend to use the *CLion IDE* by JetBrains using a student-license. However, feel free to explore other IDEs and take you time to get familiar with your IDE of choice that supports the C language.

(1) Create manually a plain-text file `<repo>/sheet_02/<your last name>/task_1/ide-choice` in which you describe which IDE (or editor) you will use and three statements why you pick this particular tool.

Tip: Do not forget to include files added to `<repo>/sheet_02/<your last name>/task_1/` in your commit.

*__Debugger__. A bug is any error, failure or fault that causes incorrect or unexpected behavior or that lead to incorrect results. A debugger is a software tool to support the process of identifying and eliminating bugs by code correction or workarounds. Typically, debugging involves multiple steps including stepwise introspection of particular code blocks and evaluating values of variables during runtime. Prominent debugger for C programs are the GNU Debugger *GDB*, and *LLDB* by LLVM Developer Group.

(1) Install one of the debugger mentioned above (or an alternative of your choice) on your machine (if not already installed) after you informed yourself which one to chose.

Tip: Your package manager or the official website will help you with that task.

(2) Run the following command in your bash

```
$ <debugger> --version >> <repo>/sheet_02/<your last name>/task_1/debugger-version
```

where `<debugger>` is the binary of the debugger of your choice (i.e., `gdb` or `lldb`)

(3) Create a plain-text file `<repo>/sheet_02/<your last name>/task_1/debugger-choice` in which you give a short explanation on your reasons for the choice of a particular debugger.

Tip: Do not forget to include files added to `<repo>/sheet_02/<your last name>/task_1/` in your commit.

Memory Debugger. Basic memory management is implemented by explicit allocation and explicit deallocation of memory resources during runtime. When acquired memory is not released when it's no longer needed, a memory leak occurs. A memory leak is binding memory resources with no reason and no way to release it by the program in the worst case. Consequences range from waste of resources to system disaster. Tools to find memory leaks and other memory related bugs are called memory debugger. Such tools are fundamental for programming languages that rely on manual memory management, such as the *C* programming language. *Valgrind* is a popular Open Source profiler having a memory checker built-in.

(1) Install `valgrind` (or any other memory debugger of your choice) on your machine (if not already installed).

Tip: Your package manager or the official website will help you with that task.

(2) Run the following command in your bash

```
$ <mem-debugger> --version >> <repo>/sheet_02/<your last name>/task_1/mem-debugger-version
```

where `<mem-debugger>` is the memory debugger of your choice (e.g., `valgrind`)

Tip: Do not forget to include files added to `<repo>/sheet_02/<your last name>/task_1/` in your commit.

Profiler. Program analysis during runtime is achieved by instrumenting an executable with a profiler tool. Profiler tools are primarily used to inspect a program in order to optimize the performance of a program with respect to memory consumption, resource utilization, or function invocation. These tools allow to monitor counters, logs, file and network I/O, system calls,

threading, per-thread call trees with running times per function, and many more. For performance-critical systems (such as database systems), program optimization backed by profiling is one fundamental engineering task. *Perf* is a popular profiler on UNIX-like operating systems. On macOS, the built-in (visual) profiler *Instruments* is also worth a try.

(1) Install a profiler of your choice on your machine (if not already installed).

Tip: Your package manager or the official website will help you with that task.

(2) Create manually a plain-text file `<repo>/sheet_02/<your last name>/task_1/profiler-version` in which you describe which profile you installed and which version you use.

Tip: Do not forget to include files added to `<repo>/sheet_02/<your last name>/task_1/` in your commit.

Test Frameworks. Software testing is the art of evaluating that a program behaves like expected under particular circumstances. Testing is highly important to software quality since for non-trivial complex programs it's often impossible to guarantee correctness. However, testing is also crucial to design since by its nature testing only shows the presence of bugs rather than their absence (Dijkstra). In practice, two kinds of software testing are applied, unit testing and integration tests. While the first focuses on testing isolated pieces of code where the environment (e.g., a particular database) is mocked or stubbed out, the latter tests complex composition of system components in an environment that simulates the production environment. *Google C++ Test Framework* (aka Google Test or GTest) is a test framework for C/C++ that gained momentum due its rich feature set and wide-range platform support.

(1) Install `gtest` on your machine (if not already installed) by typing the following in your bash:

```
$ git clone https://github.com/google/googletest
$ cd googletest
$ mkdir build
$ cd build
$ cmake ..
$ make
$ make install
```

(2) Compile the first sample shipped with GTest, and add its output to your submission. You will find the sample code in `<googletest>/googletest/samples`. However, run `make` in `<googletest>/googletest/scripts/test`. This directory contains a Makefile that will be used to assemble an executable called `sample1_unittest`. Afterwards, type

```
$ ./sample1_unittest >> <repo>/sheet_02/<your last name>/task_1/gtest-output
```

where `<googletest>` is the absolute path to the gtest repository on your machine.

Tip: Do not forget to include files added to `<repo>/sheet_02/<your last name>/task_1/` in your commit.

Code Style Formatter. Coding style conventions are guidelines about code formatting style (e.g., text case usage) and code writing (e.g., maximum length of a single line of code) for a particular project. In practice, almost each project has its own code style to enable a consistent style across

a larger code base. However, applying and checking this style is automated by styling formatter programs and styling error checkers programs (cf. "linter" tools).

(1) Install `astyle` on your machine (if not already installed).

Tip: Your package manager or the official website hosted at http://astyle.sourceforge.net will help you.

(2) Run the following command in your bash

```
$ astyle --version >> <repo>/sheet_02/<your last name>/task_1/astyle-version
```

Tip: Do not forget to include files added to `<repo>/sheet_02/<your last name>/task_1/` in your commit.

Task 2 More Hands on Version Control with Git 11 Point

In the last exercise sheet, you get familiar with the basic workflow and usage of Git in a fork-based branching model. The following task requires you to reason more about solutions to typical tasks that typically occur while working on a project managed with Git. For this tasks, add a text file `<repo>/sheet_02/<your last name>/task_2/submission.md` to your submission. The file `submission.md` must contain your answers to the questions and sub-tasks below, and must be formatted using the Markdown[2] syntax. Each of the following sub task must be a single section in your `submission.md` file.

Tip: Take a look into the official built-in tutorial (`run $man gittutorial` in your bash), or visit https://git-scm.com
Tip: See http://markdown.de and https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet
Tip: Do not forget to include files added to `<repo>/sheet_02/<your last name>/task_2/` in your commit.

(1) **Initializing your own local repository**. You want to create a repository for a project that you will do with a friend of yours. Your friend is unfamiliar with repository initialization. Describe the proceeding of initializing a local repository on your machine with Git! Which commands are reasonable, what is changed in your file system? Your description should be helpful and complete, giving your friend a good “cheat sheet” at hand.

(2) **Remote repositories & branching workflows**. You and your friend agree to use one (or more) remote repository while each of you will clone the remote repository for local development. Your friend is unfamiliar with branching workflows. Provide him a tabular comparison of the centralized, the feature branch, the GitFlow, and the forking workflow! Your comparison must contain per-workflow properties regarding the following: number of remote repositories, number of remote branches and their lifetime, and who is responsible to merge changes into the official code base. In addition, state one benefit and one drawback of each workflow! Your friend is curious about if the choice of a particular workflow affects his work on his local repository. Give a statement (along with a reason) about the effect of a particular workflow choice to the workflow your friend applies in his local repository! Finally, you and your friend want to decide for a particular workflow. Provide an argumentation for each workflow that gives a “rule of thumb” when to use which workflow!

[2] Markdown is an industry-standard markup language to achieve text-to-html conversation without losing the capability of human-readability.

(3) **Committing one out of two**. During your local development, you create two new files A and B in your repository that you add to the Git index by running `git add A B`. However, as it turns out, you want to split your commit in such a way that exactly one commit will contain exactly one out of both files rather than one commit containing both files, i.e., you want to commit first A and then file B each in a dedicated commit. How can you achieve that?

(4) **Ignoring files**. You and your friend work with different IDEs (e.g., Eclipse or CLion) each populating its own workspace-configuration files into your repositories. Neither you nor your friend want to add IDE-specific files to your remote code base. Describe how both of you can configure your shared repository in such a way that none of these files are added to the remote code base! Is there alternatively a way that each of you can specify this procedure without affecting the other (i.e., local ignorance of files rather than remote global ignorance of files)? If so, give a short description on how you will achieve that and is different compared to the global approach!

(5) **Merging vs rebasing**. You already applied merging to apply the changes made in one branch to the history of another branch. An alternative to the merging functionality is rebasing. State the difference between both approaches and give one statement when one is more useful that the other!

(6) **Rewriting History**. You read the message of a change that you just committed into your local repository. As it turns out, you find a typo in your message: you intended to write “fix” but you wrote “fox”. Before you push your changes to the remote repository, you want to remove that typo from the message of your latest commit. Describe how you can achieve that! Afterwards, you realize that the same typo is on a commit that was done days before. Hence, you decide to rewrite that old typo too. Describe roughly what option you have to rewrite old commits in the history!

(7) **Stash your work**. You are working on a particular feature in a certain local branch of your repository. Out of nowhere and totally unexpected, your friend appears right behind you and he is super excided about his latest changes. He wants you to check out his remote branch and have a closer look on it. After a funny chat on heart attacks, you agree. However, having a look at the sources in web UI of your remote repository is not enough since you must build and run your project he said. How can you switch to another branch without committing your current changes? Give a short description!