1 GIT tutorial

A version control system helps multiple developers working on the same code base. It keeps copies of previous code versions, which is helpful for debugging and software change analysis. It also allows to merge concurrent edits of text files. We use the version control system git for our tutorial.

If you don't already have one, create a GitHub account. Progress by creating a new public repository, naming it repotest.

In order to work on a copy locally, we need to clone the repository. We can do this by issuing the following command:

```
> git clone "https://"URL .
```

Note that you need to provide the URL of the GitHub Repository.

In order for files to be tracked by the version control system, they need to be explicitly added to the repository. Copy the file *file1.c* into the repository folder and use the following command to add it for tracking.

```
> git add file1.c
```

The file is now part of the git repository. Changes made to this file need to be committed in order to be visible to other users. We can verify that the repository has pending changes using the following command:

```
> git status
```

Before we can commit this initial change, however, we are required to set our user id. This is necessary so that git can keep track of which user introduced which changes.

```
> git config --global user.name <preferredusername>
> git config --global user.email some@email.com
```

We now commit the initial state of the file. In order to do so, we need to provide an obligatory commit message, summarising our changes for other users to see.

```
> git commit file1.c –m "We just created this"
```

As mentioned before, git keeps track of all changes applied to all files. We can verify this using the log command. This will list our previous change, as well as all following changes.

```
> git log
```

We now propagate this data back to the central repository. The benefit of git is that we have full version control locally without having to connect back to the central repository. Only when we want to share all of our edited versions with the rest of the team we issue the push command:

```
> git push origin master
```

In a real world environment, git repositories are shared between multiple users. Each of them has a separate clone of the repository on their machine. In our test example, we create another clone of our own demo repository by switching to another directory, and executing the following command:

```
> git clone "https://"URL second_demo
```

We can now see that the file we previously added is already part of the second clone. Edit the file and replace "John" by "Jane". Now commit the file using the following command:

```
> git commit file1.c –m "Changed name"
> git push origin master
```

Switch back to the first local copy of the repository. We may now see that the file has not been updated yet. We need to explicitly pull the new version from the central repo:

```
> git pull
```

This covers the basic functionalities of the git version control system. A more extensive demo can be found online:

`https://try.github.io/`

2 GDB tutorial

When a crash is encountered, it is always useful to run a debugger, such as GDB or LLDB to aid in identifying the root cause.

Go to the directory called *buggy*, and compile the program by calling *make*. Following by executing the binary, providing a 0 as an argument. This should lead to a crash.

In order to aid in identifying the root cause, we will be using GDB to print out the stack trace. Begin by loading the binary into the debugger as follows:

```
> gdb ex-03.exe
```

Next, run the debugger as follows:

```
> run 0
```

The debugger should catch the exception, and return control. Print the stack-trace by issuing the following command:

```
> bt
```

Follow up by inspecting the stack trace, and patching the program. You can exit the debugger by using the *quit* command.

It is important to note that we have access to program symbols such as function names because we compiled with the *-g* flag.

3 Dynamic Linked Lists

Linked List is a collection of elements stored linearly. Its size is dynamic, meaning that it may grow or reduce in size when an insertion or removal is performed respectively. In this exercise, we shall be implementing two versions.

Task 1 Implement Linked List

Open *linked-list.c* and implement the required functions, such that the tests pass.

Task 2 Implement a generic List

.

We will now be creating a generic list. The list can store different types of values, and is not limited to integers like the previous implementation.

Create a new file *gen-linked-list.c* and copy the code from your previous implementation. Modify it as well as well as the tests to achieve this task.

Hints:

- Rather than having a node store an int value, make it store a void *, which is a generic pointer.
- Have the remove function also take a function pointer to a comparison function. The prototype of this function should be `bool (*cmp)(void *, void *)`

4 Lookup tables

Lookup tables are an optimisation mechanism for functions with high computational complexity. They store a predefined amount of a functions input and output values in a data array. Instead of executing the computationally complex function at runtime, they then use the lookup table to retrieve previously calculated values.

This replaces a possibly very time-consuming algorithm executing by a simple array index access. Drawbacks of this approach are that a significant amount of memory needs to be reserved for this lookup table. Since most embedded systems have limited resources, the size of the lookup table needs to be restricted as well. This leads to imprecision in the lookup table results.

Take a look at the file *lookup-table.c*. It contains a stub method to fill the lookup table array and one stub method to actually perform the lookup. The algorithm to be mapped is the *model_function(...)* function. All we need to do is map the real range $[0, 2 \cdot \pi]$ to integer array indexes $[0, 1000]$.

The code already generates a gnuplot data file which can be examined using the provided .plt gnuplot script. GNUplot can be downloaded from here: `http://gnuplot.info/`.

Task 1 Implement *fill_model_data(...)*

This function initialises our lookup table with the precalculated model values.

Task 2 Implement *fast_model_function(...)*

This function retrieves a precalculated model value from our lookup table. Once both functions are implemented, examine the result using gnuplot. What do you notice if you zoom in?

Task 3 Use <time.h> to measure the performance improvement.

Task 4 Implement *scaled_fast_model_function(...)*

As you can see in gnuplot, our lookup table approach lacks precision. One way to increase precision are range-based resolutions. In our original data table, we use the same resolution for the whole domain of the function. However, the function shows very low gradients in the ranges $[0, \frac{\pi}{6}]$, $[\frac{2\cdot\pi}{3}, \frac{7\cdot\pi}{6}]$ and $[\frac{5\cdot\pi}{3}, 2\cdot\pi]$. On the other hand, the ranges $[\frac{\pi}{6}, \frac{2\cdot\pi}{3}]$ and $[\frac{7\cdot\pi}{6}, \frac{5\cdot\pi}{3}]$ show high gradients.

We can improve precision by increasing the resolution of our data table in areas with high gradients and decreasing it in areas with lower gradients. This allows us to use the same amount of data points more efficiently. However, this also means that the entries in our data table are no longer indexed linearly. Instead, we need to store both input and output values in our array and search for the correct index in the array.

A trivial search algorithm for the array would be to check every array entry and select the one which is closest to our value. Since we can make sure that the entries in our datatable are at least ordered, a better approach is to implement a binary search algorithm.

Task 5 Which C type would be suitable to store our new data table?

Task 6 Look up the binary search algorithm online and describe it in pseudo code. Wikipedia provides a good explanation.

Task 7 Implement the scaled lookup table model functions in our project