Recitation 1: Basic Tools

This recitation introduces basic tools for building and debugging software.

1 Software engineering practices

A good software engineer strives to write programs that are fast, correct, and maintainable. Here are a few best practices which we feel are worth reminding you of:

- Maintainability: comment your code, use meaningful variable names, insert whitespaces, follow a consistent style.
- Code organization: break up large functions into smaller subroutines, write reusable helper functions, and avoid duplicate code.
- Version control: write descriptive commit messages, and commit your changes frequently (but don’t commit anything which doesn’t compile).

2 Pair programming

Pair programming is a technique in which two programmers work on the same machine. “One of the programmers, the driver, has control of the keyboard/mouse and actively implements the program. The other programmer, the observer, continuously observes the work of the driver to identify tactical (syntactic, spelling, etc.) defects, and also thinks strategically about the direction of the work.” The programmers work equally to develop a software as they periodically switch roles.

We recommend that you find someone to work with for this assignment; you will gain more experience with pair programming during Project 1.

3 Getting started

We recommend that you work on the course’s cloud machines. You can access them using ssh:

$ ssh username@cloudN.csail.mit.edu

First, you will have to introduce yourself to Git (give Git your name and email so that you will be identified when you commit changes).

$ git config --global user.name "Ima A. Student"

followed by
$ git config --global user.email imastudent@csail.mit.edu

You should, of course, use your real name and CSAIL email address.
Now use Git to clone the recitation1 repository to obtain the code for this recitation.

$ git clone /afs/csail.mit.edu/proj/courses/6.172/student-repos/fa14/recitations/
recitation1/username.git recitation1

If you have problems with your CSAIL setup, you can also get the repository by running:

$ git clone https://github.com/mit6172/recitation1 recitation1

4 Building your code

Enter the recitation1 directory. The file main.c therein currently contains a simple program which generates an array and then prints it. We can compile this program using gcc, as follows:

$ gcc -Wall main.c -o rollingsum

This command will compile main.c into an executable called rollingsum. The flag -Wall causes the compiler to print all warnings during compilation.
Next, run the program as follows:

$ ./rollingsum

Now open main.c in your favorite text editor. If you don’t have one, you can use vim:

$ vim main.c

We would like to modify main.c to measure the time it takes to generate and print the array. To do this add the following timing code:

```c
fasttime_t start_time = gettime();
gen_array(array, ARRAY_SIZE);
print_array(array, ARRAY_SIZE);
fasttime_t end_time = gettime();

double elapsed_seconds = tdiff(start_time, end_time);
printf("Elapsed execution time: %lf sec\n", elapsed_seconds);
```

Since timing functions are in fasttime.h, we have to include it at the top of main.c.

```c
#include "./fasttime.h"
```

Now try to compile with the command:

$ gcc -Wall main.c -o rollingsum

Depending on your version of gcc, you might or might not see an error related to the clock_gettime() function, which is called in fasttime.h. This function is in the realtime library, which can be included by using the compiler flag -lrt. The following command should compile successfully:

$ gcc -Wall main.c -o rollingsum -lrt
5 Using printf

The source file `sum.c` contains the function `sum_array()`, which computes the sum of all elements in an array and then prints the result using `printf`. Modify `main.c` to use this function to print the sum of the elements in the generated array, immediately after printing out the contents of the array.

```c
sum_array(array, ARRAY_SIZE, 0, ARRAY_SIZE);
```

*Don’t forget to include `sum.h`.*

Now try to compile with the command:

```bash
$ gcc -Wall main.c -o rollingsum
```

You will get “undefined reference” errors because the compiler cannot find the code for some functions. To fix this add `sum.c` as an argument:

```bash
$ gcc -Wall main.c sum.c -o rollingsum
```

When you compile, you will see a warning about a format string conversion. Take a look at the `printf` call in `sum.c`:

```c
printf("The sum from %d to %d is %f.\n", start, stop, sum);
```

Note that `printf` requires a type for each variable in order to format it correctly. In this case, `sum` is an `int`, so you need to use `%d` instead of `%f`. The compiler detects this mistake and warns you. However, the compiler still generates a binary, which prints an incorrect number. Fix this error, and then commit your changes to your local repository.


Compiling with

```bash
$ gcc -Wall main.c sum.c -o rollingsum -lrt
```

works, but is inefficient if you are working with a large codebase. To see the result of modifying the code in one file, you must recompile the code in all files. A better way is to first compile each file separately and then link them together.

```bash
$ gcc -Wall -c main.c
$ gcc -Wall -c sum.c
$ gcc main.o sum.o -o rollingsum -lrt
```

The flag `-c` tells the compiler to generate an object file (e.g. `main.o`) for the provided source. The final command links the object files together to produce the `rollingsum` executable. If you modify `main.c` without changing `sum.c`, then you only need to run the first and last commands to produce an updated `rollingsum` executable.
6 Using make

Typing these commands by hand is tedious and error prone. You can use `make` to automate this work. To use `make`, you must write a Makefile that specifies rules of the form:

```
target: dependencies
    command
```

When run, `make` checks the modification times of all dependency files to determine whether a given target needs to be recompiled.

For example, a Makefile for our program in its current state is:

```bash
CC := gcc
CFLAGS := -Wall
LDFLAGS := -lrt

all: rollingsum

sum.o: sum.c
    $(CC) $(CFLAGS) -c $<

main.o: main.c
    $(CC) $(CFLAGS) -c $<

rollingsum: main.o sum.o
    $(CC) -o $@ $(LDFLAGS)

clean:
    rm -f rollingsum *.o
```

Note: the command lines must be indented with TAB’s, not spaces. You can now compile the program by typing:

```bash
$ make
```

Don’t forget to add your new Makefile to your recitation repository using git:

```bash
$ git add Makefile
```

7 Using GDB

GDB is a debugging tool that allows you to view your program’s state as it is executing. For example, it can help you debug when you get a segmentation fault.

Change the stop argument for `sum_array()` (the final argument for `sum_array()` in `main.c` to `ARRAY_SIZE * 9999`. Recompile and run the program. You will get a segmentation fault.
debug your program using GDB, you should first build a “debug” version of your program. Modify your Makefile to add the compiler flags `-g` (adds debug symbols) and `-O0` (disables optimizations) as follows:

```
CFLAGS := -Wall -g -O0
```

Now recompile your program. If `make` reports “Nothing to be done for all.”, then it has not actually recompiled your program. You can force `make` to recompile the program by running either

```
$ make clean; make
```

or

```
$ make -B
```

You can now start a debugging session in GDB.

```
$ gdb ./rollingsum
...
(gdb) r
...
Program received signal SIGSEGV, Segmentation fault.
0x0000000000040095c in sum_array (array=0x7fffffffe1f0, size=10,
start=0, stop=99990) at sum.c:31
9  sum += array[i];
(gdb) bt
#0 0x0000000000040095c in sum_array (array=0x7fffffffe1f0, size=10,
start=0, stop=99990) at sum.c:31
#1 0x000000000004007be in main () at main.c:56
(gdb) p i
$1 = 900
(gdb) p array
$2 = (int *) 0x7fffffffe1f0
```

You will learn more about GDB in Homework 1 and while working on Project 1.

8 Using assertions

Assertions are useful tools that simplify debugging. Include `tbassert.h` in `sum.c` and add the following lines to `sum_array()`.

```
tbassert(start >= 0, "start = %d\n", start);
tbassert(start < stop, "start = %d, stop = %d\n", start, stop);
tbassert(stop <= size, "stop = %d, size = %d\n", stop, size);
```
Build and run the program. You will see output about an assertion error.

Although assertions are useful when debugging, it is inefficient to check their conditions. For this reason, we want to disable assertions when we compile an optimize build. To do this, you can add a compiler flag `-DNDEBUG` to define the `NDEBUG` macro; which disables assertion checks.

```
$ CFLAGS := -Wall -O3 -DNDEBUG
```

After adding the flag, recompile the program and verify that assertions are not checked in your optimized build. You should get a segmentation fault, which means that the assertions were not checked.

*Before you go on to the next section, remember to fix the bug we introduced into the program by reverting the stop argument to `sum_array()` to its original value.*

## 9 Code coverage using gcov

Sometimes we are interested in how many lines of code are executed when we run a program. For example, if we run a set of tests, we would like every line of the code we are testing to be executed at least once. Code coverage tools like `gcov` are useful for checking the comprehensiveness of your test suites. We will practice using `gcov` to measure the code coverage of `rollingsum`.

First, edit `main.c` to set the array size to 1. Then add `-fprofile-arcs -ftest-coverage` to both the `CFLAGS` and `LDFLAGS` of your Makefile, preserving the `-DNDEBUG` flag already present from the previous step. Now recompile and run the following commands:

```
$ ./rollingsum > /dev/null
$ gcov main
$ gcov sum
```

This will produce the files `main.c.gcov` and `sum.c.gcov` which have been annotated by `gcov` to display the number of times each line of code has been executed. Open these files in a text editor. Lines which have not been executed will be marked by `#####`.

Using `gcov`, identify a section of code which is no longer executed due to our modification to the array size. Now set the array size back to 10 and verify that you get more code coverage.

## 10 Exercise

Modify your `Makefile` such that

- `make` builds an “optimized” version.
- `make DEBUG=1` builds a “debug” version with `gcov` enabled.

*Hint*: You can check for `DEBUG=1` by
ifeq ($(DEBUG),1)
...
else
...
endif

11 Using lanka

Now that we have spent some time engineering our code, let us try running it on the lanka cluster, which we shall use this term to test our programs' performance. Running jobs on the lanka cluster will require you to have your CSAIL account setup with the lanka cluster. You must also run the student setup script to gain access to the lexec command:

$ /afs/csail.mit.edu/proj/courses/6.172/scripts/student-setup

First, build an optimized version of your program, without gcov enabled. To run rollingsum on the lanka cluster, use the command lexec before the command you would normally run, as follows:

$ lexec ./rollingsum
Connection to lanka.csail.mit.edu closed.
srun: job 483 queued and waiting for resources
srun: job 483 has been allocated resources
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
The sum from 0 to 10 is 45.
Elapsed execution time: 0.000043 sec
Connection to lanka.csail.mit.edu closed.

You might be prompted for your CSAIL password when you run lexec.

The lexec command enqueues your job, ./rollingsum, for execution on the lanka cluster. Do not worry if it takes a moment for your job to begin executing. Because other students and researchers also use the lanka cluster, it can take some time for computation resources to become available.

12 Finished!

Commit your changes to the local repository. Then, check your work by running the verification script in your repository:

$ python verifier.py
and check your code quality by running clint.py:

$ python clint.py *.c *.h

If these scripts pass, show your work to a TA or UTA to complete the checkoff for this recitation. If you worked with someone else, you should grant permission to access the repository to your peer.