Homework 1: Getting Started

Due: 10:00 P.M. (ET) on Friday, January 21, 2025

Last Updated: January 14, 2025

This homework introduces the environment and tools you will be using to complete your future project assignments. It includes a quick C primer. You should use this assignment to familiarize yourself with the tools you will be using throughout the course.

Contents

1 Immediate action items

Make sure to complete homework 0 before starting this homework.

2 Software engineering

Best 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, and follow a consistent style.

- *Code organization*: Break up large functions into smaller subroutines, write reusable helper functions, and avoid duplicating code.
- *Version control*: Write descriptive commit messages, and commit your changes frequently (but don't commit anything that doesn't compile).
- *Assertions*: Frequently make assertions within your code so that you know quickly when something goes wrong.

Pair programming

Pair programming is a technique in which two programmers work on the same machine. According to Laurie Williams from North Carolina State University: "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 piece of software as they periodically switch roles.

3 Virtual machine setup

Follow the instructions in the following link to set up your course development environment: https://github.com/CSE491-spring25/student_software.

If you have trouble with any of these steps, please double-check to ensure you have followed the sequence of instructions precisely, and then ask a TA and/or post a Piazza note to the instructors.

4 Version control

We will use the Git distributed version control system for code release and submission. We will be using github.com for this class. The baseline code for this assignment is in the repository https://github.com/CSE491-spring25/homework1.

Cloning Homework 1

To make a clone—a local copy of the repository for your work—type:

```
$ git clone git@github.com:CSE491-spring25/homework1_<your-netid>.git homework1
```

Note that this clones your particular repository, and renames it to homework1.

You should frequently commit and push your changes back to the repository:

```
$ git commit -am 'Your commit message'
$ git push
```

For more advanced usage of Git, please refer to your favorite search engine.

5 Visual Studio Code

We recommend using Visual Studio Code as your primary editor for code in your VM, either through the Remote-SSH extension or through the shared filesystem via Orbstack. In section 3, you configured Visual Studio Code as the editor for your SPE Virtual Machine. The setup script included some recommended extensions that will be helpful for the class, such as syntax highlighting for C and a GUI for Git. You may install any extensions that you find useful through the VS Code Marketplace, such as syntax highlighting for C, a GUI for Git, or key bindings for Vim or Emacs.

Checkoff: Show the TA that you are able to view the Homework 1 code both in the browser (on Github) and in VS Code.

6 C primer

This section provides a short introduction to the C programming language. The code used in the exercises is located in `homework1/c-primer` in your repository.

Why use C?

- *Simple*: No complicated object-oriented abstractions like Java/C++.
- *Powerful*: Offers direct access to memory (but does not offer protection in accessing memory).
- *Fast*: No overhead of a runtime or JIT compiler, and no behind-the-scenes runtime features (like garbage collection) that use machine resources.
- *Ubiquitous*: C is the most popular language for low-level and performance-intensive software like device drivers, operating-system kernels, and microcontrollers.

Preprocessing

The C preprocessor modifies source code before it is passed to the compilation phase. Specifically, it performs string substitution of `#define` macros, conditionally omits sections of code, processes `#include` directives to import entire files' worth of code, and strips comments from code.

As an example, consider the code in `preprocess.c`, which is replicated in Figure 1.

Exercise: Direct `clang` to preprocess `preprocess.c`.

```
$ clang-spe -E preprocess.c
```

The preprocessed code will be output to the console. Now, rerun the C preprocessor with the following command:

```
$ clang-spe -E -DNDEBUG preprocess.c
```

You will notice that the `if` statement won't appear in the preprocessor output.

Data types and their sizes

C supports a variety of primitive types, including the types listed in Figure 2.

Note: On most 64-bit machines and compilers, a standard-precision value (e.g. `int`, `float`) is 32 bits. A `short` is usually 16 bits, and a `long` or a `double` is usually 64. The precisions of these types are weakly defined by the C standard, however, and may vary across compilers and machines.

```c
// All occurrences of ONE will be replaced by 1.
#define ONE 1

// Macros can also behave similar to inline functions.
// Note that parentheses around parameters are required to preserve order of
// operations. Otherwise, you can introduce bugs when substitution happens.
#define MIN(a,b) ((a) < (b) ? (a) : (b))

int c = ONE, d = ONE + 5;
int e = MIN(c, d);

#ifndef NDEBUG
// This code will be compiled only when
// the macro NDEBUG is not defined.
// Recall that if clang is passed -DNDEBUG on the command line,
// then NDEBUG will be defined.
if (something) {}
#endif
```

Figure 1: A sample C program. If `-DNDEBUG` is not on, the preprocessor will include the `if` statement in line 17.

```c
short s;        // short signed integer
unsigned int i; // standard-length unsigned integer
long l;         // long signed integer
long long l;    // extra-long signed integer
char c;         // represents 1 ASCII character (1 byte)
float f;        // standard-precision floating point number
double d;       // double-precision floating point number
```

Figure 2: Some of the primitive types in C.

Confusingly, sometimes `int` and `long` are the same precision, and sometimes `long` and `long long` are the same, both longer than `int`. Sometimes, `int`, `long`, and `long long` all mean the exact same thing!

For throwaway variables or variables which will stay well under precision limits, use a regular `int`. The precisions of these values are set in order to maximize performance on machines with different word sizes. If you are working with bit-level manipulation, it is better to use unsigned data types such as `uint64_t` (unsigned 64-bit `int`). Otherwise, it is often better to use a non-explicit variable such as a regular `int`.

Furthermore, if you know the architecture you're working with, it is often better to write code with explicit data types instead (such as the ones in Figure 3).

You can define more complex data types by composing primitive types into a `struct`. For exam-

```c
#include <stdint.h>

uint64_t unsigned_64_bit_int;
int16_t signed_16_bit_int;
```

Figure 3: Examples of explicit types in C.

ple, one example of a `struct` definition in C is provided in Figure 4.

```c
typedef struct {
  int id;
  int year;
} student;

student you;
// access values on a struct with .
you.id = 12345;
you.year = 3;
```

Figure 4: Examples of a `struct` declaration in C.

Exercise: Edit `sizes.c` to print the sizes of each of the following types: `int`, `short`, `long`, `char`, `float`, `double`, `unsigned int`, `long long`, `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`, `uint_fast8_t`, `uint_fast16_t`, `uintmax_t`, `intmax_t`, `__int128`, `int[]` and `student`. Note that `__int128` is a clang C extension, and not part of standard C. To check the size of an `int` array, print the size of the array `x` declared in the provided code.

To compile and run this code, use the following command:

```
$ make sizes && ./sizes
```

To avoid creating repetitive code, you may find it useful to define a `macro` and call it for each of the types.

If you are interested in learning more about built-in types, check out
http://en.cppreference.com/w/c/types/integer .

Pointers

Pointers are first-class data types that store addresses in memory. A pointer can store the address of anything in memory, including another pointer. In other words, it is possible to have a pointer to a pointer.

Arrays behave very similarly to pointers: both hold information about the type and location of values in memory. There are a few gotchas involved with treating pointers and arrays equivalently, however.[1] Consider the following (buggy) snippet of code from `pointer.c` in Figure 5.

Exercise: Compile `pointer.c` using the following command:

```
$ make pointer
```

You will see compilation errors corresponding to the invalid statements mentioned in the above program. Why are these statements invalid? Comment out those invalid statements and recompile the program. (Do not worry if you see additional warnings about unused variables.)

Write-up 1: Answer the questions in the comments in `pointer.c`. For example, why are some of the statements valid and some are not?

Write-up 2: For each of the types in the `sizes.c` exercise above, print the size of a pointer to that type. Recall that obtaining the address of an array or `struct` requires the `&` operator. Provide the output of your program (which should include the sizes of both the actual type and a pointer to it) in the write-up.

Argument passing

In C, arguments[2] to a function are passed **by value**. That means that if you pass an integer to function `foo(int f)`, a new variable `f` will be initialized inside `foo` with the same value as the integer you passed in.

For instance, consider the code in Figure 6 that swaps two integers. Why doesn't it work as expected?

There are two ways to fix this code. One way is to change `swap()` to be a macro, causing the operations to be evaluated in the scope of the macro invocation. Another way is to change `swap()` to use pointers. We will now ask you to fix the code by using pointers.

[1] For further reading, try the challenge at `https://blogs.oracle.com/linux/the-ksplice-pointer-challenge-v2` after class.

[2] In general, `parameters` are the variables that appear in a function definition, and `arguments` are the data that are actually passed in at runtime.

```c
int main(int argc, char* argv[]) {   // What is the type of argv?
  int i = 5;
  // The & operator here gets the address of i and stores it into pi
  int* pi = &i;
  // The * operator here dereferences pi and stores the value -- 5 --
  // into j.
  int j = *pi;

  char c[] = "6.106";
  char* pc = c;   // Valid assignment: c acts like a pointer to c[0] here.
  char d = *pc;
  printf("char d = %c\n", d);  // What does this print?

  // compound types are read right to left in C.
  // pcp is a pointer to a pointer to a char, meaning that
  // pcp stores the address of a char pointer.
  char** pcp;
  pcp = argv;  // Why is this assignment valid?

  const char* pcc = c;   // pcc is a pointer to char constant
  char const* pcc2 = c;   // What is the type of pcc2?

  // For each of the following, why is the assignment:
  *pcc = '7';  // invalid?
  pcc = *pcp;  // valid?
  pcc = argv[0];  // valid?

  char* const cp = c;   // cp is a const pointer to char
  // For each of the following, why is the assignment:
  cp = *pcp;  // invalid?
  cp = *argv;  // invalid?
  *cp = '!';  // valid?

  const char* const cpc = c;   // cpc is a const pointer to char const
  // For each of the following, why is the assignment:
  cpc = *pcp;  // invalid?
  cpc = argv[0];  // invalid?
  *cpc = '@';  // invalid?

  return 0;
}
```

Figure 5: An example of valid and invalid pointer usage in C.

```c
void swap(int i, int j) {
  int temp = i;
  i = j;
  j = temp;
}

int main() {
  int k = 1;
  int m = 2;
  swap(k, m);
  // What does this print?
  printf("k = %d, m = %d\n", k, m);
}
```

Figure 6: An incorrect implementation of `swap()` in C.

Write-up 3: File `swap.c` contains the code to swap two integers. Rewrite the `swap()` function using pointers and make appropriate changes in `main()` function so that the values are swapped with a call to `swap()`. Compile the code with `make swap` and run the program with `./swap`. Provide your edited code in the write-up. Verify that the results of both `sizes.c` and `swap.c` are correct by using the python script `verifier.py`.

7 Basic tools

The code that we will be using in this section is located in `homework1/matrix-multiply`.

Building and running your code

You can build the code by going to the `homework1/matrix-multiply` directory and typing `make`. The program will compile using Tapir, a cutting-edge derivative of Clang/LLVM. Notice that we are only compiling with optimization level 1 (i.e., `-O1`).

Exercise: Modify your Makefile so that the program is compiled using optimization level 3 (i.e., `-O3`).

Write-up 4: Now, what do you see when you type `make clean; make`?

You can then run the built binary by typing `./matrix_multiply`. The program should print out something and then crash with a segmentation fault.

If a different error occurs, you may be compiling the file for an architecture incompatible with the one you are executing it on. Recompile the program with `make clean; make LOCAL=1`.

8 Using telerun

Directly running and timing the execution on your own VMs may give inaccurate results; you may be running a small VM, there might be architectural differences, and there might also be measurement errors due to interference with your editor or other programs you are running. To get a standardized environment and an accurate timing measure on a dedicated machine, you can use the `telerun` utility.

After compiling your code with `make`, you can run

```
$ telerun <your_program>
```

and your job will be queued. The command will not return until the job has been completed and you get results unless you control-C (canceling the process). The output of the program is then shown.

Using a debugger

While running your program, if you encounter a segmentation fault, bus error, or assertion failure, or if you just want to set a breakpoint, you can use the debugging tool GDB.

Exercise: Obtain a program stack trace using GDB.

We will run GDB using `telerun`, as the virtualization software can cause problems with GDB on certain base operating systems. First, rebuild the program for the remote machine with `make clean; make` (no local flag).

```
$ telerun gdb -ex run -ex bt -batch ./matrix_multiply
```

This command should run the program until it crashes, then output the program stack trace.

You should get an output like this:

```
Program received signal SIGSEGV, Segmentation fault.
0x000000000022a784 in matrix_multiply_run ()
#0  0x000000000022a784 in matrix_multiply_run ()
#1  0x000000000022a4ee in main ()
```

This stack trace says that the program crashes in `matrix_multiply_run`, but it doesn't give any other information about the error. In order to get more information, we will build a "debug" version of the code. Run:

```
$ make clean
$ make DEBUG=1
$ telerun gdb -ex run -ex bt -batch ./matrix_multiply
```

The major differences from the optimized build are '`-gdwarf-2`' (add debug symbols to your program) and '`-O0`' (compile without any optimizations). The stack trace should now include file line numbers. GDB tells us that a segmentation fault occurs at at `matrix_multiply.c` line 90. dimensions.

Using assertions

The `tbassert` package is a useful tool for catching bugs before your program goes off into the weeds. If you look at `matrix_multiply.c`, you should see some assertions in `matrix_multiply_run` routine that check that the matrices have compatible dimensions.

Exercise: Uncomment these lines and add a line to include `tbassert.h` at the top of the file. Then, build and generate a new GDB stack trace. Make sure that you build using `make DEBUG=1`. You should see the following:

```
Running matrix_multiply_run()...
matrix_multiply.c:80 (int matrix_multiply_run(const matrix *, const matrix *, matrix *))
Assertion A->cols == B->rows failed: A->cols = 5, B->rows = 4

Program received signal SIGABRT, Aborted.
__pthread_kill_implementation (no_tid=0, signo=6, threadid=140737351526208) at
./nptl/pthread_kill.c:44
44        ./nptl/pthread_kill.c: No such file or directory.
#0  __pthread_kill_implementation (no_tid=0, signo=6, threadid=140737351526208) at
./nptl/pthread_kill.c:44
#1  __pthread_kill_internal (signo=6, threadid=140737351526208) at
./nptl/pthread_kill.c:78
#2  __GI___pthread_kill (threadid=140737351526208, signo=signo@entry=6) at
./nptl/pthread_kill.c:89
#3  0x00007ffff7dc7476 in __GI_raise (sig=sig@entry=6) at ../sysdeps/posix/raise.c:26
#4  0x00007ffff7dad7f3 in __GI_abort () at ./stdlib/abort.c:79
#5  0x00005555555565fc in matrix_multiply_run (A=0x555555559360, B=0x5555555592c0,
C=0x555555559460) at matrix_multiply.c:79
#6  0x0000555555556227 in main (argc=1, argv=0x7fffffffe3f8) at testbed.c:134
```

GDB tells us that "`Assertion 'A->cols == B->rows' failed`", which is much better than the former segmentation fault. The assertion provides a `printf`-like API that allows you to print values in your own output, as above.

You will also see an assertion failure with a line number for the failing assertion without using GDB. Since the extra checks performed by assertions can be expensive, they are disabled for optimized builds, which are the default in the Makefile. As a result, if you make the program without `DEBUG=1`, you will not see an assertion failure.

You should sprinkle assertions liberally throughout your code to check important invariants in your program because they will make your life easier when debugging. In particular, most non-trivial loops and recursive functions should have an assertion of the loop or recursion invariant.

Exercise: Fix `testbed.c`, which creates the matrices, rebuild your program, and verify that it now works. You should see "Elapsed execution time..." after executing the following command:

```
$ telerun ./matrix_multiply
```

Commit and push your changes to the Git repository:

```
$ git commit -am 'Your commit message'
$ git push origin main
```

Next, check the result of the multiplication. Run the following command:

```
$ telerun ./matrix_multiply -p
```

The program will print out the result. The result seems to be wrong, however. You can check the multiplication of zero matrices by running

```
$ telerun ./matrix_multiply -pz
```

Using a memory checker

Some memory bugs do not crash the program, and consequently, GDB cannot tell you where the bug is. You can use the memory checking tools AddressSanitizer and Valgrind to track these bugs.

AddressSanitizer

Clang's built-in AddressSanitizer is a quick memory error checker that uses compiler instrumentation and a run-time library. It can detect a wide variety of bugs (including memory leaks). To use AddressSanitizer, we need to pass the appropriate flags. First, do

```
$ make clean
```

to get rid of the existing build. Next, do

```
$ make ASAN=1
```

to build with AddressSanitizer's instrumentation, yielding the following (you may also get a warning by the gold linker, `ld.gold`):

```
$ make ASAN=1
clang -O1 -g -fsanitize=address -Wall -std=c99 -D_POSIX_C_SOURCE=200809L -c \
testbed.c -o testbed.o
clang -O1 -g -fsanitize=address -Wall -std=c99 -D_POSIX_C_SOURCE=200809L -c \
 matrix_multiply.c -o matrix_multiply.o
clang -o matrix_multiply testbed.o matrix_multiply.o -lrt -flto -fuse-ld=gold \
-fsanitize=address
```

Finally, run the program with

```
$ telerun ./matrix_multiply
```

Write-up 5: What output do you see from AddressSanitizer regarding the memory bug? Paste it into your write-up here.

Valgrind

Valgrind is another tool for checking memory leaks. If you want to check a program without recompiling with instrumentation, Valgrind is a good option for detecting memory bugs.

Exercise: First, do

```
$ make clean && make DEBUG=1
```

to get rid of the existing build and get a fresh build. Run Valgrind using

```
$ telerun valgrind ./matrix_multiply -p
```

You need the `-p` switch, since Valgrind only detects memory bugs that affect outputs. You should also use a "debug" version to get a good result. This command should print out many lines. The important ones are

```
==1027219== Use of uninitialised value of size 8
==1027219==    at 0x48C22EB: _itoa_word (_itoa.c:177)
==1027219==    by 0x48DDABD: __vfprintf_internal (vfprintf-internal.c:1516)
==1027219==    by 0x48C879E: printf (printf.c:33)
==1027219==    by 0x10A3BB: print_matrix (matrix_multiply.c:68)
==1027219==    by 0x10A0E5: main (testbed.c:140)
```

This output indicates that the program used a value before initializing it. The stack trace indicates that the bug occurs in `testbed.c:140`, which is where the program prints out matrix `C`.

Exercise: Fix `matrix_multiply.c` to initialize values in matrices before using them. Keep in mind that the matrices are stored in `struct`'s. Rebuild your program, and verify that it outputs a correct answer. Again, commit and push your changes to the Git repository.

Write-up 6: After you fix your program, run `telerun ./matrix_multiply -p`. Paste the program output showing that the matrix multiplication is working correctly.

Memory management

The C programming language requires you to free memory after you are done using it, or else you will have a memory leak. Valgrind can track memory leaks in the program. Run the same Valgrind command, and you will see these lines towards the end of its output:

```
==1027765== LEAK SUMMARY:
==1027765==    definitely lost: 48 bytes in 3 blocks
==1027765==    indirectly lost: 288 bytes in 15 blocks
==1027765==      possibly lost: 0 bytes in 0 blocks
==1027765==    still reachable: 0 bytes in 0 blocks
==1027765==         suppressed: 0 bytes in 0 blocks
==1027765== Rerun with --leak-check=full to see details of leaked memory
```

This output suggests that there are indeed memory leaks in the program. To get more information, you can build your program in debug mode and again run Valgrind, using the flag `--leak-check=full`

```
$ telerun valgrind --leak-check=full ./matrix_multiply -p
```

The trace shows that all leaks are from the creations of matrices `A`, `B`, and `C`.

Exercise: Fix `testbed.c` by freeing these matrices after use with the function `free_matrix`. Rebuild your program, and verify that Valgrind doesn't complain about anything. Commit and push your changes to the Git repository.

Write-up 7: Paste the output from Valgrind showing that there is no error in your program.

Checking code coverage

Bugs may exist in code that doesn't get executed in your tests. You may find it surprising when someone testing your code (like a professor or a TA) uncovers a crash on a line that you never exercised. Additionally, lines that are frequently executed are good candidates for optimization. The Gcov tool provides a line-by-line execution count for your program.

Exercise: To use Gcov, modify your Makefile and add the flags `-fprofile-arcs` and `-ftest-coverage` to the `CFLAGS` and `LDFLAGS` variables. You will have to rebuild from scratch using `make clean` followed by `make DEBUG=1 LOCAL=1`. Try running your code normally with `./matrix_multiply -p`. Observe that several new `.gcda` and `.gcno` files were created during your execution.

Now use the `llvm-cov` command-line utility on `testbed.c`:

```
$ llvm-cov gcov testbed.c
```

A new file, `testbed.c.gcov` was created that is identical to the original `testbed.c`, except that it has the number of times each line was executed in the code. In that file, you will see:

```
        1:   99:  if (use_zero_matrix) {
    #####:  100:    for (int i = 0; i < A->rows; i++) {
    #####:  101:      for (int j = 0; j < A->cols; j++) {
    #####:  102:        A->values[i][j] = 0;
    #####:  103:      }
    #####:  104:    }
    #####:  105:    for (int i = 0; i < B->rows; i++) {
    #####:  106:      for (int j = 0; j < B->cols; j++) {
    #####:  107:        B->values[i][j] = 0;
    #####:  108:      }
    #####:  109:    }
    #####:  110:  } else {
```

The hash marks indicate lines that were never executed. In general, it is unusual to run a code-coverage utility on a testbed, but a set of untested lines in your core code could lead to unexpected results when executed by someone else.

Another handy use of Gcov is identifying which lines got executed the *most* frequently. Code that gets run the most is often the most costly in terms of performance. Run `llvm-cov` on `matrix_multiply.c` and look at the output:

```
 5:   91:  for (int i = 0; i < A->rows; i++) {
20:   92:    for (int j = 0; j < B->cols; j++) {
80:   93:      for (int k = 0; k < A->cols; k++) {
64:   94:        C->values[i][j] += A->values[i][k] * B->values[k][j];
64:   95:      }
16:   96:    }
 4:   97:  }
```

These are the loops in `matrix_multiply_run`. Clearly, this function is a good candidate for optimization.

When you are done using Gcov, remove the flags you added to the Makefile because they add costly overhead to the execution, and will negatively impact your actual performance numbers. You should never run benchmarks on code that is instrumented with Gcov. Don't forget to `make clean` to remove the instrumented object files.

Performance enhancements

To get an idea of the extent to which performance optimizations can affect the performance of your program, we will first increase the size of the input to demonstrate the effects of changes in the code.

Exercise: Increase the size of all matrices to 1000×1000.

Now let's try one of the techniques from the Lecture 1. Right now, the inner loop produces a sequential access pattern on `A` and skips through memory on `B`. Let's rearrange the loops to produce a better access pattern.

Exercise: First, you should run the program as is to get a performance measurement. Next, swap the `j` and `k` loops, so that the inner loop strides sequentially through the rows of the `C` and `B` matrices. Rerun the program, and verify that you have produced a speedup. Commit and push your changes to the Git repository.

Write-up 8: Report the execution time of your programs before and after the optimization.

Compiler optimizations

To get an idea of the extent to which compiler optimizations can affect the performance of your program, rebuild your program in "debug" mode and run it with `telerun`.

Exercise: Rebuild it again with optimizations (just type `make`), and run it with `telerun`. Both versions should print timing information, and you should verify that the optimized version is faster.

> **Write-up 9:** Report the execution time of your programs compiled in debug mode with `-O0` and normally with `-O3`.

9 C style guidelines

Code that adheres to a consistent style is easier to read and debug. Google provides a style guide for C++ which you may find useful: https://google.github.io/styleguide/cppguide.html

We have provided a Python script `clint.py`, which is designed to check a subset of Google's style guidelines for C code. To run this script on all source files in a given directory, use the following command:

```
$ python clint.py path-to-directory/*
```

The code the staff provides contains no style errors. We suggest, but do not require, that you use this tool to clean up your source code. Part of your code-quality grade on projects is based on the readability of your code. It can be difficult to maintain a consistent style when multiple people are working on the same codebase. For this reason, you may find it useful to use the style checkers provided by the course staff during your group projects to keep your code readable.

10 Practice with Instrumentation (Useful for Project 1)

We have earlier used the AddressSanitizer in section 7 to help us debug problems related to memory. We did this by passing the instrumentation option `-fsanitize=address` to `clang-spe`, so that it would add extra instructions to detect for these memory bugs.

Exercise: Investigate the `Makefile` in `matrix-multiply` to see how the instrumentation option is used.

There are other Sanitization instrumentation options that are useful. One such example is the Undefined Behavior Sanitizer. It can be used by passing the option `-fsanitize=undefined` to

`clang-spe`. It checks for some undefined operations like division by zero, incorrect shift operations, integer overflow errors, ...

To get some practice using it, navigate to the `homework1/is-power-of-two` directory.

This directory contains a small program that check if a number is a power of two in `is-power-of-two/is_power_of_two.c` and a number of test cases to check if for correctness in `is-power-of-two/testbed.c`

Compile and run the test cases using:

```
$ make LOCAL=1
$ ./is_power_of_two
```

You will notice that the program never finishes executing.

Now, this is a good time to try to use the Undefined Behavior Sanitizer. We have added a rule to the Makefile that should run this sanitizer (`make UBSAN=1`), but we left out the instrumentation for you to get some practice.

Exercise: Investigate the `Makefile` in `is-power-of-2` to see how the rule `ASAN` uses the Address Sanitizer `-fsanitize=address`. Then, modify the `Makefile` to get the rule `UBSAN` to pass the option `-fsanitize=undefined` where appropriate.

Write-up 10: What do you see when you run `make clean; make LOCAL=1 UBSAN=1`? Paste the compilation output.

Write-up 11: What do you see when you run `./is_power_of_two`? Paste the error that the sanitizer throws. Explain where and why this error happened. Then, fix the bug you found and explain your fix. Finally, paste the fixed code.

Your program should not get stuck again once you fix the bug, recompile, and run again.

You will notice that we calculate the multiplication of all the power of two numbers in the tests and print. However, if you investigate the output, you will find that the multiplication is incorrect.

Write-up 12: Compile again with `make clean; make LOCAL=1 UBSAN=1`. What do you see when you run `./is_power_of_two`? Paste the error that the sanitizer throws. In addition, explain where and why this error happened. Fix the error you found and explain your fix.

Finally, compile and run the program again and verify that the output is correct. Paste your program output.

11 Submission

Commit your changes to your repository and check your code style by running `clint.py`. Submit the write-up as described in Write-ups 1–12 to Gradescope. You do not need to prepare a separate document for your write-ups: most of them will be submitted as inline text on Gradescope. Finally, if you haven't already, commit and push your final submission to the repository:

```
$ git commit -am 'Done!'
$ git push origin main
```