Homework 1: Getting Started

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.

1 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 duplicate code.
- Version control: write descriptive commit messages, and commit your changes frequently (but don't commit anything which 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.

You will gain more experience with pair programming during Project 1.

[Note: This course makes use of AWS and Git features which may not be available to all OCW users.]

4 C Primer

This section will be 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.

```
01 // All occurrences of ONE will be replaced by 1.
02 #define ONE 1
03
04 // Macros can also behave similar to inline functions.
05 // Note that parentheses around parameters are required to preserve order of
06 // operations. Otherwise, you can introduce bugs when substitution happens.
07 #define MIN(a,b) ((a) < (b) ? (a) : (b))
08
09 int c = ONE, d = ONE + 5;
10 int e = MIN(c, d);
11
12 #ifndef NDEBUG
13 // This code will be compiled only when
14 // the macro NDEBUG is not defined.
15 // Recall that if clang is passed -DNDEBUG on the command line,
16 // then NDEBUG will be defined.
17 if (something) {}
18 #endif
```

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

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

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

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

```
$ clang -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. However, the precisions of these types are weakly defined by the C standard, and may vary across compilers and machines. 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!

```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.

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).

```c
#include <stdint.h>

uint64_t unsigned_64_bit_int;
int16_t signed_16_bit_int;
```

Figure 3: Examples of explicit types in C.

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

[1]For further reading on this, try out the challenge at `https://blogs.oracle.com/ksplice/entry/the_ksplice_pointer_challenge` after class.

```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[] = "6.172";
  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.

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

Write-up 3: 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 writeup.

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?

```
80 void swap(int i, int j) {
81   int temp = i;
82   i = j;
83   j = temp;
84 }
85
86 int main() {
87   int k = 1;
88   int m = 2;
89   swap(k, m);
90   // What does this print?
91   printf("k = %d, m = %d\n", k, m);
92 }
```

Figure 6: An incorrect implementation of swap in C.

[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.

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.

Write-up 4: 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 writeup. Verify that the results of both `sizes.c` and `swap.c` are correct by using the python script `verifier.py`.

5 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 5: 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.

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: Start a debugging session in GDB:

```
$ gdb --args ./matrix_multiply
```

This command should give you a GDB prompt, at which you should type `run` or `r`:

```
$ (gdb) run
```

Your program will crash, giving you back a prompt, where you can type `backtrace` or `bt` to get a stack trace:

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400de4 in matrix_multiply_run ()
(gdb) bt
#0  0x0000000000400de4 in matrix_multiply_run ()
#1  0x0000000000400bbe in main ()
```

This stack trace says that the program crashes in `matrix_multiply_run`, but doesn't tell any other information about the error. In order to get more information, build a "debug" version of the code. First, quit GDB by typing `quit` or `q`:

```
(gdb) q
A debugging session is active.

        Inferior 1 [process 26817] will be killed.

Quit anyway? (y or n) y
```

Next, build a "debug" version of the code by typing `make DEBUG=1`:

```
$ make DEBUG=1
clang -g -DDEBUG -O0 -Wall -std=c99 -D_POSIX_C_SOURCE=200809L   -c -o testbed.o testbed.c
clang -g -DDEBUG -O0 -Wall -std=c99 -D_POSIX_C_SOURCE=200809L   -c -o matrix_multiply.o \
matrix_multiply.c
clang -o matrix_multiply testbed.o matrix_multiply.o -lrt -flto -fuse-ld=gold
```

The major differences from the optimized build are '`-g`' (add debug symbols to your program) and '`-O0`' (compile without any optimizations). Once you have created a debug build, you can start a debugging session again:

```
$ gdb --args ./matrix_multiply
(gdb) r
...
Program received signal SIGSEGV, Segmentation fault.
0x00000000004011cf in matrix_multiply_run (A=0x603270, B=0x6031d0, C=0x603130)
    at matrix_multiply.c:90
90              C->values[i][j] += A->values[i][k] * B->values[k][j];
```

Now, GDB can tell that a segmentation fault occurs at at `matrix_multiply.c` line 90. You can ask GDB to print values using `print` or `p`:

```
gdb) p A->values[i][k]
$1 = 7
(gdb) p B->values[k][j]
Cannot access memory at address 0x0
(gdb) p B->values[k]
$2 = (int *) 0x0
(gdb) p k
$3 = 4
```

This suggests that `B->values[4]` is `0x0`, which means B doesn't have row 5. There is something wrong with the matrix 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 a line to include `tbassert.h` at the top of the file. Then, build and run the program again using GDB. Make sure that you build using `make DEBUG=1`. You should see:

```
(gdb) r
...
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.
0x00007ffff7843c37 in raise () from /lib/x86_64-linux-gnu/libc.so.6
```

Now, GDB tells 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. However, even if you don't print values in your assertions, the debug build still has the symbols for GDB, as above. Unlike the above, however, if you try to print `A->cols`, you will fail. The reason is that GDB is not in the stack frame you want. You can get the stack trace to see which frame you want (#3 in this case), and type `frame 3` or `f 3` to move to frame #3. After that, you can print `A->cols` and `B->cols`.

```
(gdb) bt
#0  0x00007ffff7843c37 in raise () from /lib/x86_64-linux-gnu/libc.so.6
#1  0x00007ffff7847028 in abort () from /lib/x86_64-linux-gnu/libc.so.6
#2  0x000000000040121b in matrix_multiply_run (A=0x603270, B=0x6031d0, C=0x603130)
    at matrix_multiply.c:79
#3  0x0000000000400db2 in main (argc=1, argv=0x7fffffffdf58) at testbed.c:127
(gdb) f 3
#3  0x0000000000400db2 in main (argc=1, argv=0x7fffffffdf58) at testbed.c:127
127       matrix_multiply_run(A, B, C);
(gdb) p A->cols
$1 = 5
(gdb) p B->rows
$2 = 4
```

You should see the values 5 and 4, which indicates that we are multiplying matrices of incompatible dimensions.

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 our Makefile. As a result, if you make the program without `DEBUG=1`, you will not see an assertion failure.

You should consider sprinkling assertions throughout your code to check important invariants in your program, since they will make your life easier when debugging. In particular, most nontrivial 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 running

```
$ ./matrix_multiply
```

Commit and push your changes to the Git repository:

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

```
$ git push origin master
```

Next, check the result of the multiplication. Run

```
$ ./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

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

Using a memory checker

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

AddressSanitizer

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.

```
$ 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

```
$ ./matrix_multiply
```

Write-up 6: What output do you see from AddressSanitizer regarding the memory bug? Paste it into your writeup here.

Valgrind

Valgrind is another tool for checking memory leaks. If you want to check a program but are not able to instrument it, Valgrind is a good option for detecting memory bugs.

Exercise: First, do

```
$ make clean && make
```

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

```
$ 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

```
==43644== Use of uninitialised value of size 8
==43644==    at 0x508899B: _itoa_word (_itoa.c:179)
==43644==    by 0x508C636: vfprintf (vfprintf.c:1660)
==43644==    by 0x50933D8: printf (printf.c:33)
==43644==    by 0x401137: print_matrix (matrix_multiply.c:68)
==43644==    by 0x400E0E: main (testbed.c:133)
```

This output indicates that the program used a value before initializing it. The stack trace indicates that the bug occurs in `testbed.c:133`, 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 structs. Rebuild your program, and verify that it outputs a correct answer. Again, commit and push your changes to the Git repository.

Write-up 7: After you fix your program, run `./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 at the very end:

```
==2158== LEAK SUMMARY:
==2158==    definitely lost: 48 bytes in 3 blocks
==2158==    indirectly lost: 288 bytes in 15 blocks
==2158==      possibly lost: 0 bytes in 0 blocks
==2158==    still reachable: 0 bytes in 0 blocks
==2158==         suppressed: 0 bytes in 0 blocks
```

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`

```
$ 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 8: 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`. Try running your code normally with `./matrix_multiply -p`. Note that a number of new .gcda and .gcno files were created during your execution.

Now use the llvm-cov commandline 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:

```
164 1:       99:
165 if (use_zero_matrix) {
166 #####:  100:    for (int i = 0; i < A->rows; i++) {
167 #####:  101:        for (int j = 0; j < A->cols; j++) {
168 #####:  102:            A->values[i][j] = 0;
169 #####:  103:        }
170 #####:  104:    }
171 #####:  105:    for (int i = 0; i < B->rows; i++) {
172 #####:  106:      for (int j = 0; j < B->cols; j++) {
173 #####:  107:        B->values[i][j] = 0;
174 #####:  108:      }
175 #####:  109:    }
```

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 gcov matrix_multiply.c` and look at the output:

```
 10:   93:  for (int i = 0; i < A->rows; i++) {
 40:   94:    for (int j = 0; j < B->cols; j++) {
160:   95:      for (int k = 0; k < A->cols; k++) {
 64:   96:        C->values[i][j] += A->values[i][k] * B->values[k][j];
 64:   97:      }
 16:   98:    }
  4:   99:  }
```

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.

6 Using AWSRUN

You share the `athena.dialup.mit.edu` machines with all of MIT. Do not perform computationally intensive tasks on these machines. Directly running and timing the execution on your own Amazon VMs may give inaccurate results; you may be running a small VM, and there might also be measurement errors due to interference with your editor or other programs you are running. To get an accurate timing measure on a dedicated machine, you can use the AWSRUN utilities.

When you are ready to do a performance run you can use

```
$ awsrun ./matrix_multiply
```

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 twice (canceling the process). Once the job is complete, your output should look something like below:

```
$ awsrun ./matrix_multiply

Submitting Job: ./matrix_multiply
Waiting for job to finish...
==== Standard Error ====
Setup
Running matrix_multiply_run()...



==== Standard Output ====
Elapsed execution time: 0.000001 sec
```

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 first lecture. 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 9:** 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 AWSRUN.

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

> **Write-up 10:** Report the execution time of your programs compiled in debug mode with `-O0` and in non-debug mode with `-O3`.

7 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 your current directory use the command:

```
$ python clint.py *
```

The code the staff provides has 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 6.172 staff during your group projects to keep your code readable.