Programming Tips for CS758/858

August 21, 2025

1 Introduction

The programming assignments for CS758/858 will be done in C. If you are not very familiar with C, we recommend the book *The C Programming Language* by Kernighan and Ritchie. Feel free to use C99 (or some later standard) if you happen to know it. This handout is intended to give you some tips on programming in C on `agate.cs.unh.edu` which is the machine that will be used to grade your assignments. Your submissions will need to run on this system without any changes, so you probably want do most of your work there. (If you really want to work elsewhere and like hacking, you might consider setting up `sshfs` so you can keep your files on agate. Few things are less fun than realizing that you submitted the wrong version of your code. Note that the remote ssh option of VisualStudio is buggy and leaves many zombie processes on `agate` which, unless you kill them, will eventually prevent you from logging in.)

2 Accessing `agate.cs.unh.edu`

The easiest way to access `agate.cs.unh.edu` is to use a secure shell (ssh) client.

2.1 UNIX

From a UNIX machine (this includes Macs) this can typically be done with the `ssh`. For example the command

```
ssh <username>@agate.cs.unh.edu
```

will prompt you for your password on `agate.cs.unh.edu` and then will give you a shell. Since most of the assignments allow you to see a graphical display of the performance of your algorithms you may want to enable X11 forwarding by using the `-X` option. By using the `-X` option, programs on `agate.cs.unh.edu` can open an X11 window on your local X server to display plots.

2.2 Windows

To access `agate.cs.unh.edu` from a Windows machine we recommend that you use `putty` as your ssh client (Google "putty tatham"). From the downloads page, you should download the `putty.exe` file (probably for 64-bit x86, but Windows Arm machines do exist). You can then run this program to log into `agate.cs.unh.edu`.

While putty is supposed to support X forwarding, you will need a local X server to use it. Windows itself does not typically come with an X server and, due to the complexity of X, we do not provide detailed instructions or support. However, we will mention that VcXsrv is a free X server for Windows or, if you have installed WSL2, it comes with a (Wayland-based) X server.

3 Programming Assignments

All of the programming assignments must be done in C. We will provide some skeleton code for you so that you don't have to write everything from scratch. You are not required to use the provided code, but we recommend it. (The provided code should at least handle I/O and can also give an example of the coding style that we expect.) If you are going to use the skeleton code, it is important that you read and understand it first, before designing and implementing your part of the solution. You will implement all of the remaining functions necessary for a complete working assignment. You should not have to change very much of the skeleton code in order to complete the assignments, but you certainly can if you want to. For most assignments, other libraries (even the standard math library) should not be needed. Please post a question on Piazza if it is unclear what you are supposed to implement or if you have questions about what you can use!

While the class is about algorithms and not programming aesthetics, it is in your best interests to write nice clean well-organized code that the TA will be able to understand quickly and easily. Please break large functions into small descriptively-named functional components.

In addition to skeleton code, you will be given a reference solution and a harness. Be warned that the reference solution may not be perfect and you should use it only as a guideline. The harness distributed with each assignment will run your solution and try to verify that its results are legal. It will not tell you if your program uses the wrong algorithm or is implemented inefficiently or with messy convoluted code! As mentioned before, the harness will sometimes allow you to see a graphical display of the performance of your algorithms (more information on using this will be given in each assignment handout). You are required to run your solution with the harness before submitting your assignments to ensure that: 1) the output from your solution is readable by the harness (which will be used for grading) and 2) to make sure that your solution gives correct output. If you have trouble getting your solution to work with the harness please setup a meeting with the TA immediately to get the issue resolved.

3.1 Submitting an assignment

In Fall 2025, source code and written work will be submitted separately. Source code will be submitted using a script on `agate` (described below) and written work will be submitted as a PDF via a SharePoint folder (distributed to you by the TA).

Electronically submit your source code using the `~cs758/scripts/sub758` script on agate. To submit your assignment, make sure that all of your source code and the makefile that compiles your code are in a directory. You will then submit this entire directory. This script takes two arguments: 1) the assignment name (in this case "1-undergrad") and 2) the name of the directory containing your code. You should run this command one directory up from your code directory. For example:

```
jtd7@agate:~$ ~cs758/sub758 <assignment name> <directory with your submission>
```

So, to submit the directory "sort_ints" for assignment 1, the command would look like:

```
jtd7@agate:~$ ~cs758/sub758 1-undergrad sort_ints
```

If you are enrolled in 858, please substitute `-grad` for `-undergrad`, as in `cs758/sub758 1-grad sort_ints`.

The submission script will keep all submitted versions and the last one submitted before the assignment deadline is the one that will be graded. The script preserves modification times, so if you realize after the deadline that you submitted the wrong files, you should be able to contact the TA and submit the correct directory with the `agate` modification times preserved (assuming that the correct files were on agate by the deadline).

4 Tips on Programming in C

- Write very simple code. C will do just about anything that you tell it without regard for whether or not it is a good idea. If you make your code very complicated then there will be bugs and you will not be able to find them. It is a bad idea to be 'clever' or 'sophisticated' in C. Code hygiene is paramount.
- Check the return value of *all* non-`void` functions. Functions return a value for a reason, usually to report errors and you do not want to miss these. Think carefully before disregarding the return value of a function.
- Make functions very short. This really an extension of the first bullet point. If a function gets too long then you should probably break it into multiple functions. Since you are checking the return values of all functions that you call, your functions will get long quickly and this means that you will probably make a lot of small helper functions.
- Attempt to minimize the amount of memory that you allocate on the heap (using `malloc`, `calloc`, etc). Heap allocations must be `free`d to prevent memory leaks and we expect that you will free all of your heap allocated memory before your program exits.
- `valgrind` may be used to find and track down memory leaks and invalid memory access errors in your program (more below).
- `gdb` may be used to track down bugs in your program such as segfaults.

5 Tools

This section discusses a few tools may be helpful for tracking down bugs and memory leaks in your program. The `makefile` provided with the skeleton code for your assignments will build two binaries. The binary with the `_debug` extension is built without compiler optimizations and with debugging symbols. This means that the binary is laced with information that may be used by external applications to print information about the source code that generated portions of the binary.

5.1 `valgrind`

When you are initially developing your program, we recommend that you always use the `valgrind` application. `valgrind` is a simple tool that wraps around the execution of your program and can track down many types of bugs such as memory leaks and accesses to uninitialized and invalid memory locations. To run your program in `valgrind`, simply type `valgrind` before the command line to run your program. For example:

```
valgrind ./a.out_debug arg1 arg2 < input
```

will run a hypothetical binary called `a.out_debug` (compiled with debugging symbols) which takes two command-line arguments and an input file redirected to standard input. If there are errors in the execution of `a.out` then `valgrind` will report them to standard output along with a stack trace containing the source code file and line number of each function call leading up to the error.

If `valgrind` reports that there are memory leaks in the program than it may be re-run with the `--leak-check=full` option to perform finer grained memory leak detection. With this option `valgrind` will report more information such as where the leaked memory was allocated.

If `valgrind` is giving too much output then you may use the `--log-file=<filename>` option to have it output to a file instead of standard output.

It is harder to start using valgrind after your program is almost done, as there are likely to be many more error messages to deal with than if you had just used it from the beginning.

5.2 gdb

The GNU debugger (gdb) can allow you to track down harder-to-find bugs in your program. gdb is a fairly complicated program and we will only give some basic commands here:

5.2.1 The run command

To run gdb just type gdb <binary> at the command line where <binary> is the name of your binary that is compiled with debugging symbols. You will then be given a (gdb) prompt at which you can issue various commands. The first command that you will normally issue is the run command to run your program. The run command should be followed by any command-line arguments that you would like to be passed to your program. You can also use input redirection like at the shell. For example, the following is a transcript of gdb finding a segfault in a simple program:

```
$ gdb a.out_debug
GNU gdb (GDB) 7.1-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/aifs2/eaburns/tmp/gdb/a.out_debug...done.
(gdb) r world
Starting program: /home/aifs2/eaburns/tmp/gdb/a.out_debug world

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400538 in bad_function () at test.c:9
9 printf("accessing a null pointer: %d\n", *null);
```

5.2.2 The list command

The list command can be used to view some context when gdb reports an error in your program's execution. For example, if the list command is executed in the above program trace we will see:

```
(gdb) list
4
5 void bad_function(void)
6 {
7 int *null = NULL;
8
9 printf("accessing a null pointer: %d\n", *null);
10 }
11
12 void function_that_calls_bad_function(void)
13 {
```

5.2.3 The backtrace command

The backtrace command can be used to view a stack trace when gdb detects an error in program execution. So, in our running example, the backtrace command gives the following output.

```
(gdb) backtrace
#0  0x0000000000400538 in bad_function () at test.c:9
#1  0x0000000000400559 in function_that_calls_bad_function () at test.c:14
#2  0x000000000040058f in main (argc=2, argv=0x7fffffffe4d8) at test.c:20
```

Here, we see that `bad_function` (located in test.c on line 9) was called from a function named `function_that_calls_bad_funct` (located on line 14 in test.c) which was called by the `main` function (in test.c on line 20).

5.2.4 Other commands

`gdb` supports many more sophisticated commands for debugging programs such as setting break points, stepping through execution line-by-line or instruction-by-instruction and watching the values of variables as they change in your program. You can find these commands and more in the GDB documentation, which is located here: `http://www.gnu.org/software/gdb/documentation/`