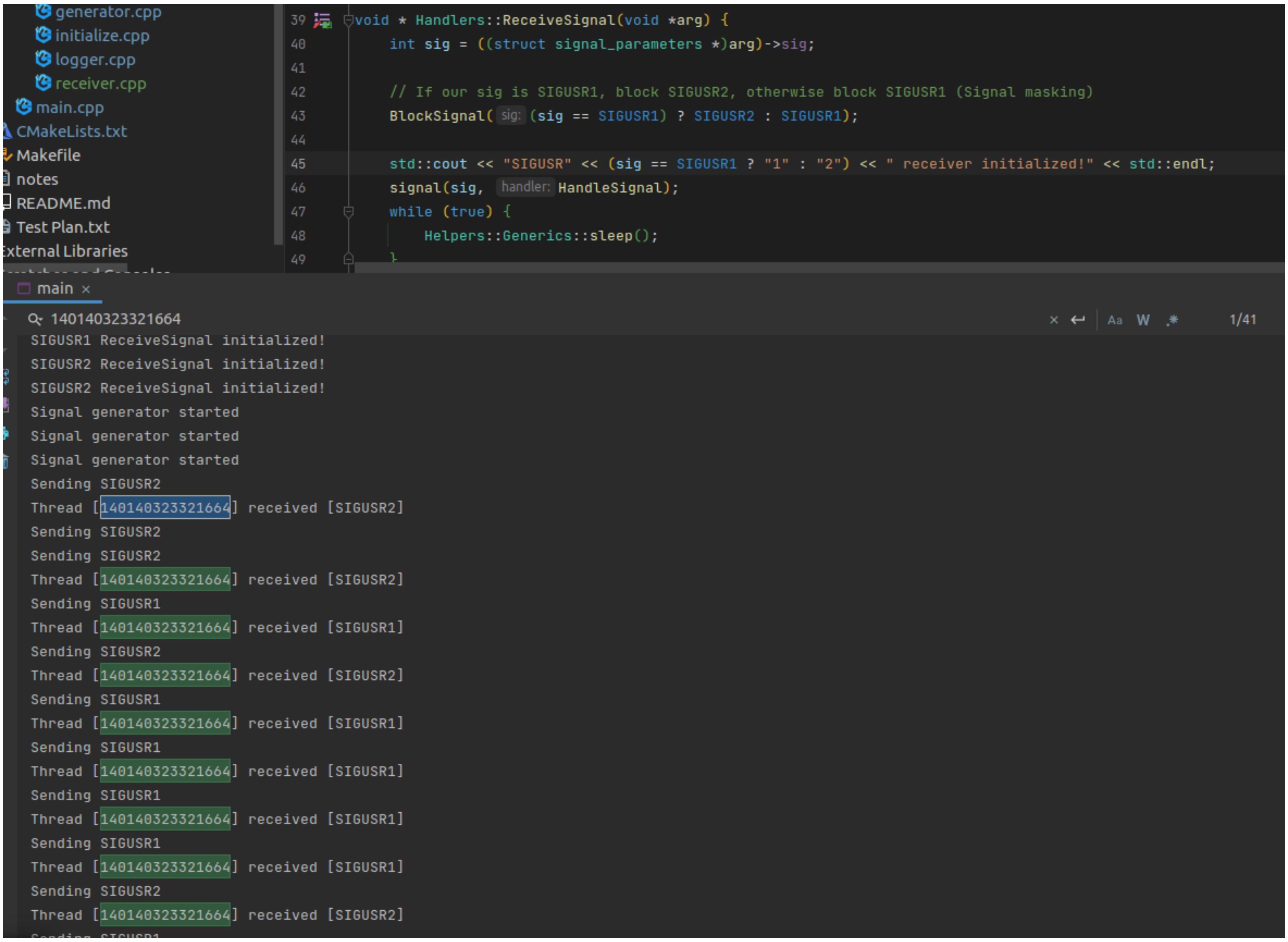
Ayser Jamshidi
Test Plan

## Receiver Thread Functionality

As we'll be requiring some statistics and logging it's imperative that all four receiver threads are being put to use and are fully functional. I initially had a problem where I had only one receiver thread being utilized for both SIGUSR1/2 even though I (supposedly?) blocked the signal a receiving thread wouldn't want to mess with (line 43).

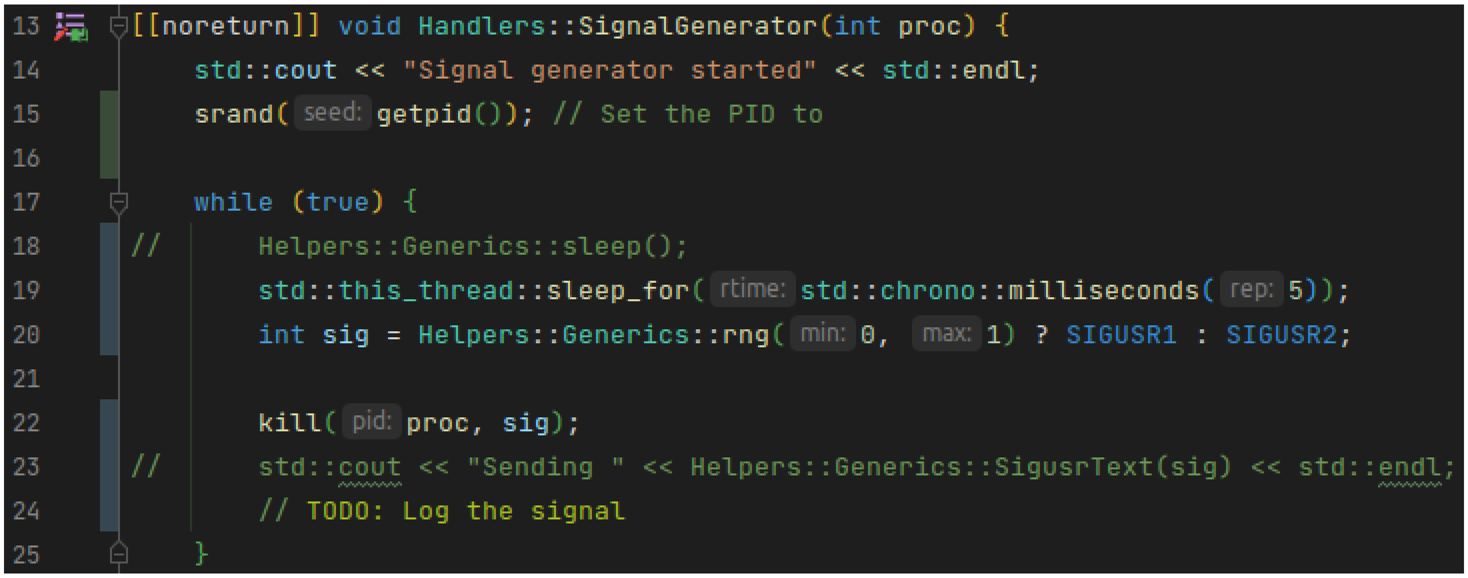


```
39 void * Handlers::ReceiveSignal(void *arg) {
40     int sig = ((struct signal_parameters *)arg)->sig;
41
42     // If our sig is SIGUSR1, block SIGUSR2, otherwise block SIGUSR1 (Signal masking)
43     BlockSignal((sig == SIGUSR1) ? SIGUSR2 : SIGUSR1);
44
45     std::cout << "SIGUSR" << (sig == SIGUSR1 ? "1" : "2") << " receiver initialized!" << std::endl;
46     signal(sig, HandleSignal);
47     while (true) {
48         Helpers::Generics::sleep();
49     }
```

```
SIGUSR1 ReceiveSignal initialized!
SIGUSR2 ReceiveSignal initialized!
SIGUSR2 ReceiveSignal initialized!
Signal generator started
Signal generator started
Signal generator started
Sending SIGUSR2
Thread [140140323321664] received [SIGUSR2]
Sending SIGUSR2
Sending SIGUSR2
Thread [140140323321664] received [SIGUSR2]
Sending SIGUSR1
Thread [140140323321664] received [SIGUSR1]
Sending SIGUSR2
Thread [140140323321664] received [SIGUSR2]
Sending SIGUSR1
Thread [140140323321664] received [SIGUSR1]
Sending SIGUSR1
Thread [140140323321664] received [SIGUSR1]
Sending SIGUSR1
Thread [140140323321664] received [SIGUSR1]
Sending SIGUSR1
Thread [140140323321664] received [SIGUSR1]
Sending SIGUSR2
Thread [140140323321664] received [SIGUSR2]
```

After reading the wonderful TA Rachel's slides she reminded us that if a signal was blocked for a process, it would also be blocked for any newly created threads. So my method was instead of dynamically blocking the undesired signal, I would <u>block</u> both SIGUSR1/2 **before** creating any new threads. After they were blocked, the process would continue to create 4 receiver threads which <u>unblock</u> their assigned signal to handle. Once that was set up, I double checked my logs but I was confused as now I had one receiver thread from each type showing in the logs instead of 2 of each. I looked through my code and didn't find anything odd so I went to my signal generator method and changed the sleep timer to be really low so it can annihilate the receiver threads to ensure that both threads of each type MUST pop up as beforehand, the receiver thread might've been done handling their signals before another assigned one popped up.

Once I made the modification

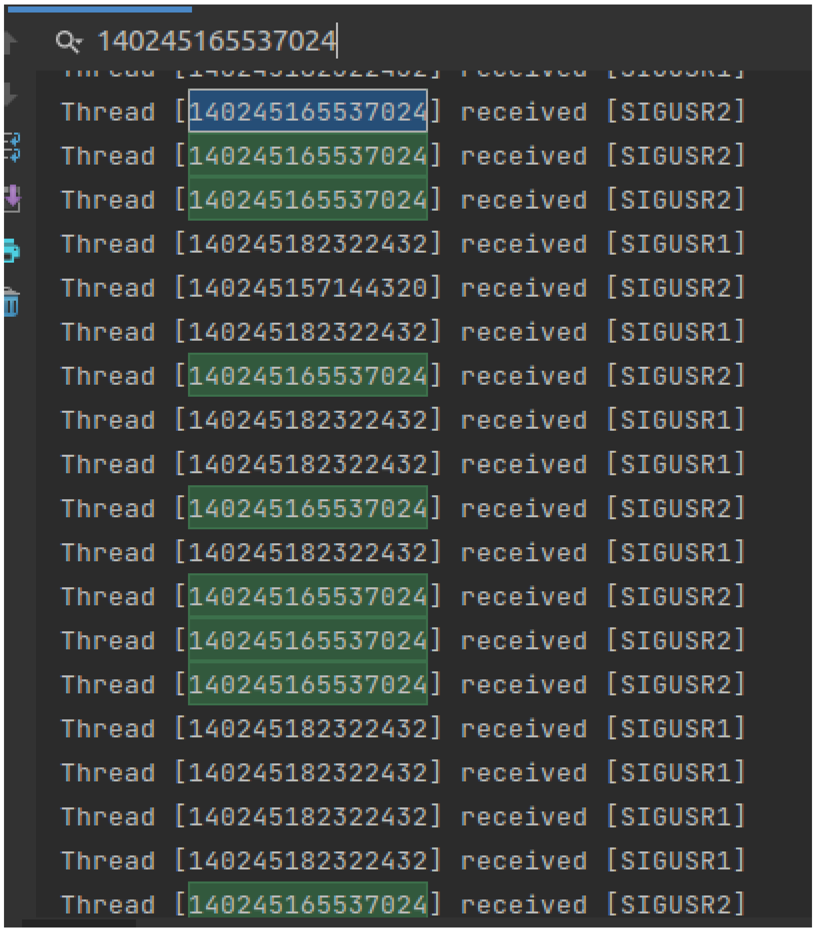


```cpp
[[noreturn]] void Handlers::SignalGenerator(int proc) {
    std::cout << "Signal generator started" << std::endl;
    srand( seed: getpid()); // Set the PID to

    while (true) {
//      Helpers::Generics::sleep();
        std::this_thread::sleep_for( rtime: std::chrono::milliseconds( rep: 5));
        int sig = Helpers::Generics::rng( min: 0,  max: 1) ? SIGUSR1 : SIGUSR2;

        kill( pid: proc, sig);
//      std::cout << "Sending " << Helpers::Generics::SigusrText(sig) << std::endl;
        // TODO: Log the signal
    }
```

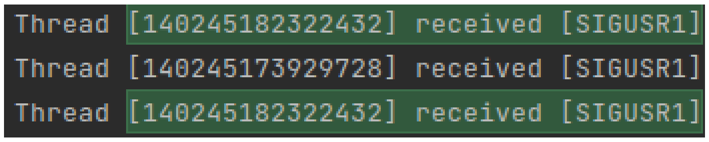
I reran it and struck gold



```
140245165537024
```

```
Thread [140245165537024] received [SIGUSR2]
Thread [140245165537024] received [SIGUSR2]
Thread [140245165537024] received [SIGUSR2]
Thread [140245182322432] received [SIGUSR1]
Thread [140245157144320] received [SIGUSR2]
Thread [140245182322432] received [SIGUSR1]
Thread [140245165537024] received [SIGUSR2]
Thread [140245182322432] received [SIGUSR1]
Thread [140245182322432] received [SIGUSR1]
Thread [140245165537024] received [SIGUSR2]
Thread [140245182322432] received [SIGUSR1]
Thread [140245165537024] received [SIGUSR2]
Thread [140245165537024] received [SIGUSR2]
Thread [140245165537024] received [SIGUSR2]
Thread [140245182322432] received [SIGUSR1]
Thread [140245182322432] received [SIGUSR1]
Thread [140245182322432] received [SIGUSR1]
Thread [140245182322432] received [SIGUSR1]
Thread [140245165537024] received [SIGUSR2]
```

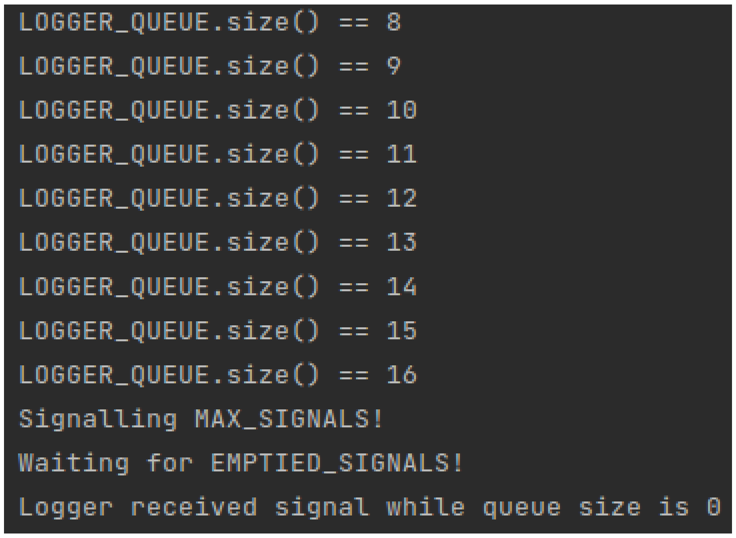
I was a little confused as I saw SIGUSR2 swapping logs between two different threads but SIGUSR1 was seemingly being handled by a single thread. I searched for specifically all SIGUSR1 requests and saw entries where it is in fact being handled by multiple threads.



```
Thread [140245182322432] received [SIGUSR1]
Thread [140245173929728] received [SIGUSR1]
Thread [140245182322432] received [SIGUSR1]
```

## Ensuring Variables Worked Properly

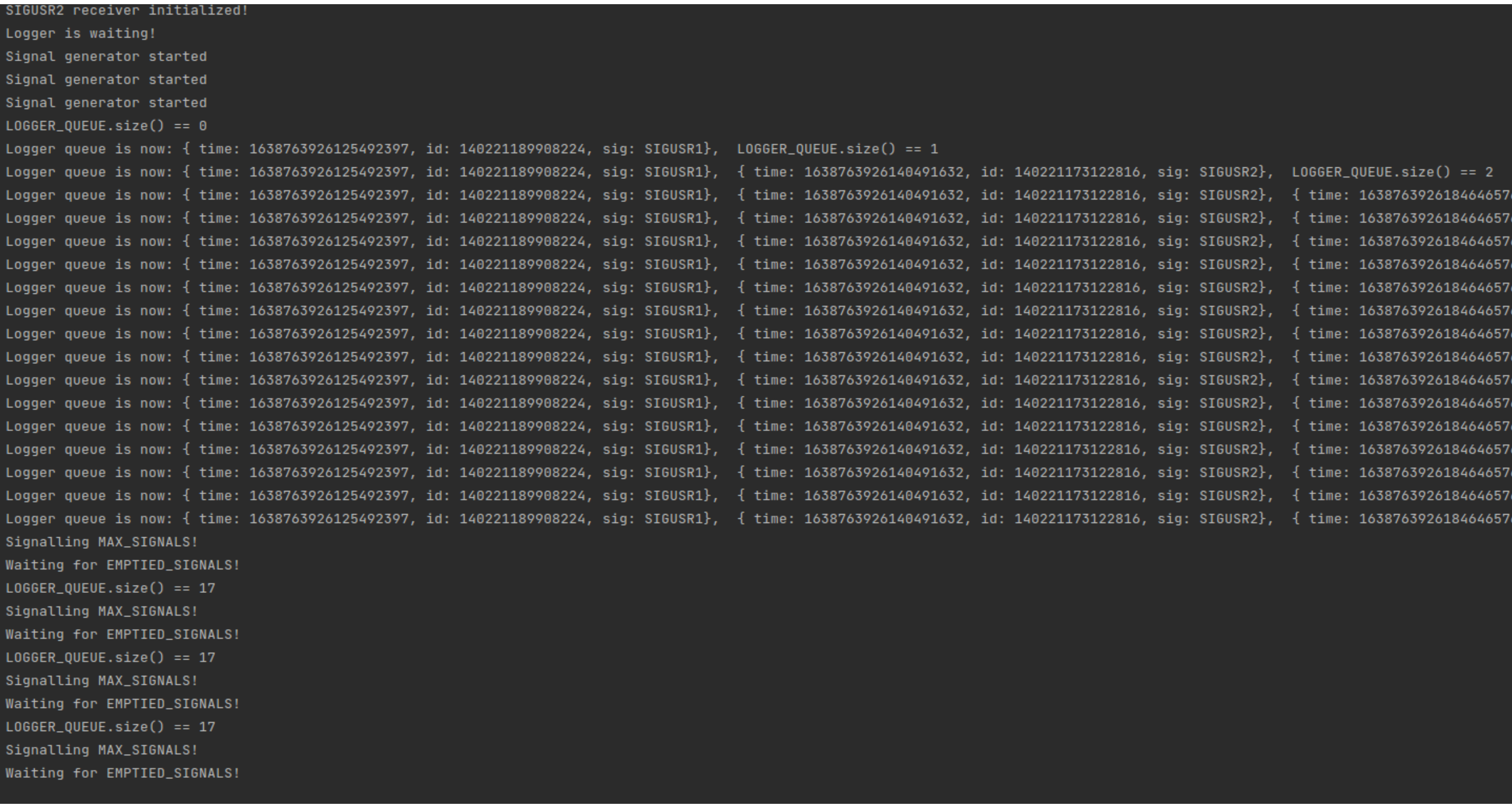
Throughout building this program I had many hiccups where variables were simply not maintaining values set to them. An example shown below is when I was testing the logger to start its job the moment 16 entries were in the queue. To the receiver threads the queue had 16 entries, but once the receiver threads signalled to the logger thread that it can continue its job, the logger thread would read the queue size and it would return 0!



```
LOGGER_QUEUE.size() == 8
LOGGER_QUEUE.size() == 9
LOGGER_QUEUE.size() == 10
LOGGER_QUEUE.size() == 11
LOGGER_QUEUE.size() == 12
LOGGER_QUEUE.size() == 13
LOGGER_QUEUE.size() == 14
LOGGER_QUEUE.size() == 15
LOGGER_QUEUE.size() == 16
Signalling MAX_SIGNALS!
Waiting for EMPTIED_SIGNALS!
Logger received signal while queue size is 0
```

Out of sheer confusion, I decided to create a function that outputs all of the log entries. Each new line shown below is an output of whenever my logger queue was added to. This confirms several things

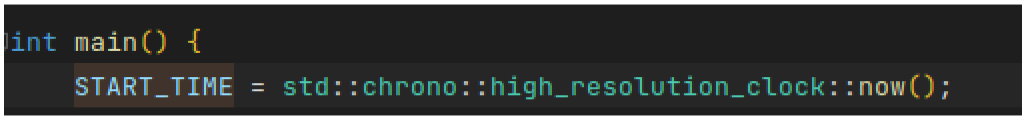
1) My queue is indeed being added to by the receivers
2) My log entry struct works very well for my needs!
3) The problem lies elsewhere, not the queue itself…



```
SIGUSR2 receiver initialized!
Logger is waiting!
Signal generator started
Signal generator started
Signal generator started
LOGGER_QUEUE.size() == 0
Logger queue is now: { time: 1638763926125492397, id: 140221189908224, sig: SIGUSR1},  LOGGER_QUEUE.size() == 1
Logger queue is now: { time: 1638763926125492397, id: 140221189908224, sig: SIGUSR1},  { time: 1638763926140491632, id: 140221173122816, sig: SIGUSR2},  LOGGER_QUEUE.size() == 2
Logger queue is now: { time: 1638763926125492397, id: 140221189908224, sig: SIGUSR1},  { time: 1638763926140491632, id: 140221173122816, sig: SIGUSR2},  { time: 163876392618464657
Logger queue is now: { time: 1638763926125492397, id: 140221189908224, sig: SIGUSR1},  { time: 1638763926140491632, id: 140221173122816, sig: SIGUSR2},  { time: 163876392618464657
Logger queue is now: { time: 1638763926125492397, id: 140221189908224, sig: SIGUSR1},  { time: 1638763926140491632, id: 140221173122816, sig: SIGUSR2},  { time: 163876392618464657
Logger queue is now: { time: 1638763926125492397, id: 140221189908224, sig: SIGUSR1},  { time: 1638763926140491632, id: 140221173122816, sig: SIGUSR2},  { time: 163876392618464657
Logger queue is now: { time: 1638763926125492397, id: 140221189908224, sig: SIGUSR1},  { time: 1638763926140491632, id: 140221173122816, sig: SIGUSR2},  { time: 163876392618464657
Logger queue is now: { time: 1638763926125492397, id: 140221189908224, sig: SIGUSR1},  { time: 1638763926140491632, id: 140221173122816, sig: SIGUSR2},  { time: 163876392618464657
Logger queue is now: { time: 1638763926125492397, id: 140221189908224, sig: SIGUSR1},  { time: 1638763926140491632, id: 140221173122816, sig: SIGUSR2},  { time: 163876392618464657
Logger queue is now: { time: 1638763926125492397, id: 140221189908224, sig: SIGUSR1},  { time: 1638763926140491632, id: 140221173122816, sig: SIGUSR2},  { time: 163876392618464657
Logger queue is now: { time: 1638763926125492397, id: 140221189908224, sig: SIGUSR1},  { time: 1638763926140491632, id: 140221173122816, sig: SIGUSR2},  { time: 163876392618464657
Logger queue is now: { time: 1638763926125492397, id: 140221189908224, sig: SIGUSR1},  { time: 1638763926140491632, id: 140221173122816, sig: SIGUSR2},  { time: 163876392618464657
Logger queue is now: { time: 1638763926125492397, id: 140221189908224, sig: SIGUSR1},  { time: 1638763926140491632, id: 140221173122816, sig: SIGUSR2},  { time: 163876392618464657
Logger queue is now: { time: 1638763926125492397, id: 140221189908224, sig: SIGUSR1},  { time: 1638763926140491632, id: 140221173122816, sig: SIGUSR2},  { time: 163876392618464657
Logger queue is now: { time: 1638763926125492397, id: 140221189908224, sig: SIGUSR1},  { time: 1638763926140491632, id: 140221173122816, sig: SIGUSR2},  { time: 163876392618464657
Logger queue is now: { time: 1638763926125492397, id: 140221189908224, sig: SIGUSR1},  { time: 1638763926140491632, id: 140221173122816, sig: SIGUSR2},  { time: 163876392618464657
Logger queue is now: { time: 1638763926125492397, id: 140221189908224, sig: SIGUSR1},  { time: 1638763926140491632, id: 140221173122816, sig: SIGUSR2},  { time: 163876392618464657
Signalling MAX_SIGNALS!
Waiting for EMPTIED_SIGNALS!
LOGGER_QUEUE.size() == 17
Signalling MAX_SIGNALS!
Waiting for EMPTIED_SIGNALS!
LOGGER_QUEUE.size() == 17
Signalling MAX_SIGNALS!
Waiting for EMPTIED_SIGNALS!
LOGGER_QUEUE.size() == 17
Signalling MAX_SIGNALS!
Waiting for EMPTIED_SIGNALS!
```

After an hour or two of hassle, I found out that this is caused by making my global variables static. This is improper and was fixed by me making all of my global variables to be “extern” instead and initializing them outside of globals.h.

Another annoying issue I had was, when attempting to set the start time of the program via the following line:

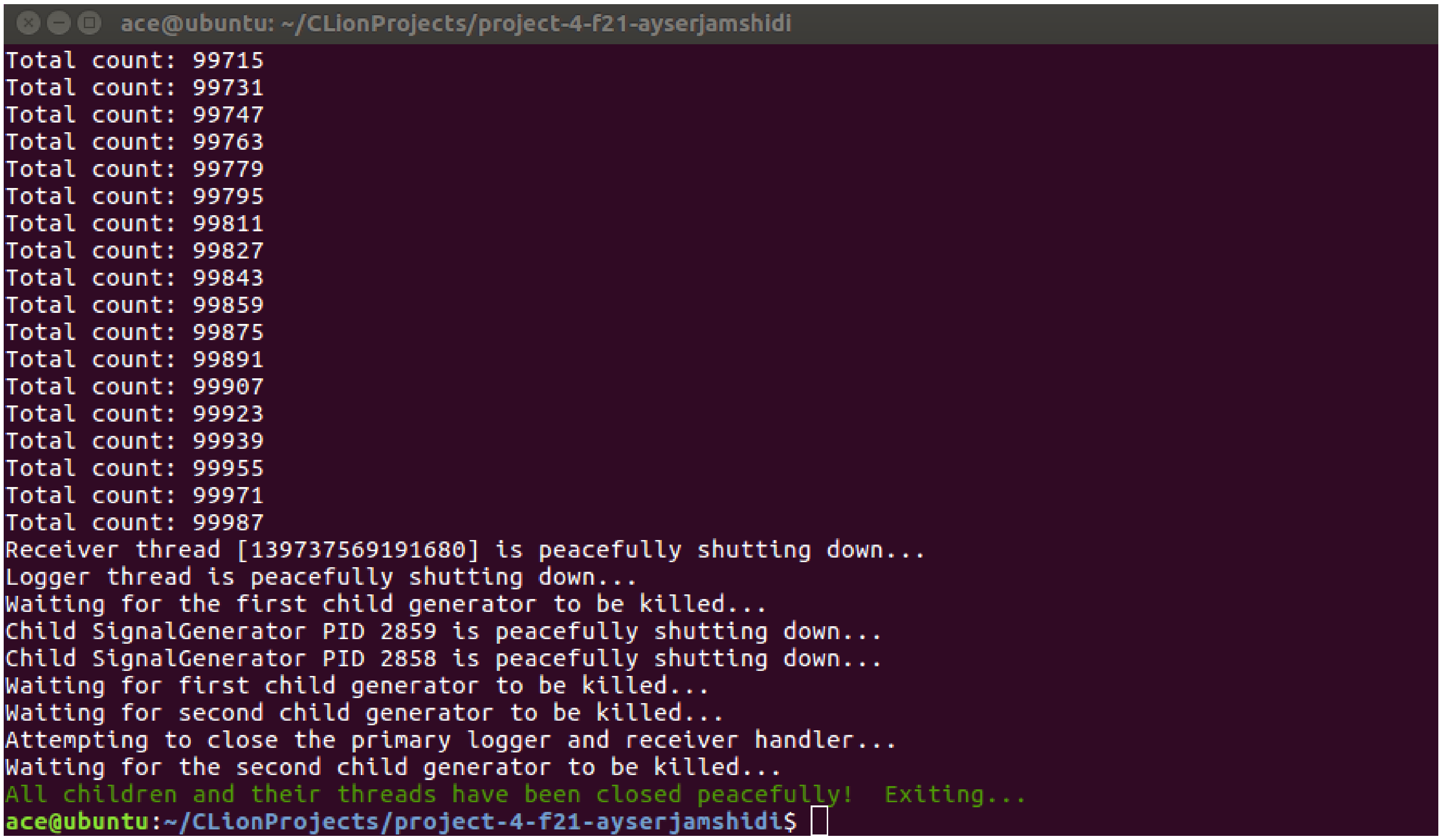


```
int main() {
    START_TIME = std::chrono::high_resolution_clock::now();
```

The program wouldn't update START_TIME properly, due to it being static. I would get data such as **Thread [139855729477376] received [SIGUSR1] at runtime [23455.834794445ms]** but I knew this wasn't right as that was the very first signal received and it was just a second after the program starting. This also solved itself after turning the variable to "extern" instead of static and initializing it elsewhere.

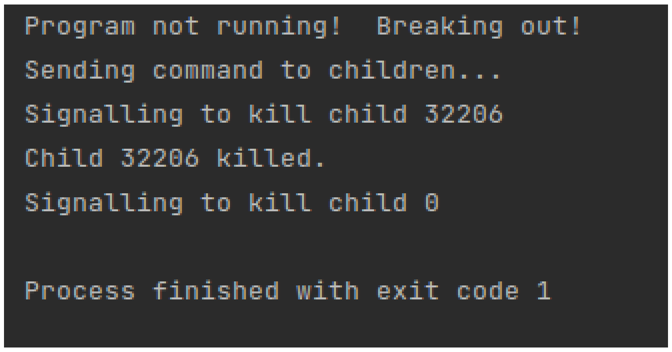
## Peaceful Exiting

I wanted my program to peacefully exit any and all threads and children before exiting the main process. My program has a very detailed output cycle whenever the program is set to terminate, as seen below:



```
ace@ubuntu: ~/CLionProjects/project-4-f21-ayserjamshidi
Total count: 99715
Total count: 99731
Total count: 99747
Total count: 99763
Total count: 99779
Total count: 99795
Total count: 99811
Total count: 99827
Total count: 99843
Total count: 99859
Total count: 99875
Total count: 99891
Total count: 99907
Total count: 99923
Total count: 99939
Total count: 99955
Total count: 99971
Total count: 99987
Receiver thread [139737569191680] is peacefully shutting down...
Logger thread is peacefully shutting down...
Waiting for the first child generator to be killed...
Child SignalGenerator PID 2859 is peacefully shutting down...
Child SignalGenerator PID 2858 is peacefully shutting down...
Waiting for first child generator to be killed...
Waiting for second child generator to be killed...
Attempting to close the primary logger and receiver handler...
Waiting for the second child generator to be killed...
All children and their threads have been closed peacefully!  Exiting...
ace@ubuntu:~/CLionProjects/project-4-f21-ayserjamshidi$
```

The program announces what is being killed in order, every time. I ran into a multitude of issues making this work the way I wanted to as, during testing, several threads – specifically the generators – would maintain their loop regardless of if the program was attempting to terminate or not. I also came across an interesting issue where, when storing the pid_t of the generator children, sending a kill signal to the second child would actually send a signal to the parent (for whatever reason?) and close the program not-so-peacefully.



```
Program not running!  Breaking out!
Sending command to children...
Signalling to kill child 32206
Child 32206 killed.
Signalling to kill child 0

Process finished with exit code 1
```

Oddly enough, this fixed itself after some time.  I’m unsure if it was a small unnoticeable change I’ve done or if it was me rebooting my computer.