# 3.5 — More debugging tactics

ALEX DECEMBER 31, 2023

In the previous lesson (3.4 -- Basic debugging tactics (https://www.learncpp.com/cpp-tutorial/basic-debugging-tactics/)), we started exploring how to manually debug problems. In that lesson, we offered some criticisms of using statements to print debug text:

1. Debug statements clutter your code.
2. Debug statements clutter the output of your program.
3. Debug statements require modification of your code to both add and to remove, which can introduce new bugs.
4. Debug statements must be removed after you're done with them, which makes them non-reusable.

We can mitigate some of these issues. In this lesson, we'll explore some basic techniques for doing so.

## Conditionalizing your debugging code

Consider the following program that contains some debug statements:

```
#include <iostream>

int getUserInput()
{
std::cerr << "getUserInput() called\n";
    std::cout << "Enter a number: ";
    int x{};
    std::cin >> x;
    return x;
}

int main()
{
std::cerr << "main() called\n";
    int x{ getUserInput() };
    std::cout << "You entered: " << x << '\n';

    return 0;
}
```

When you're done with the debugging statement, you'll either need to remove them, or comment them out. Then if you want them again later, you'll have to add them back, or uncomment them.

One way to make it easier to disable and enable debugging throughout your program is to make your debugging statements conditional using preprocessor directives:

```
#include <iostream>

#define ENABLE_DEBUG // comment out to disable debugging

int getUserInput()
{
#ifdef ENABLE_DEBUG
std::cerr << "getUserInput() called\n";
#endif
    std::cout << "Enter a number: ";
    int x{};
    std::cin >> x;
    return x;
}

int main()
{
#ifdef ENABLE_DEBUG
std::cerr << "main() called\n";
#endif
    int x{ getUserInput() };
    std::cout << "You entered: " << x << '\n';

    return 0;
}
```

Now we can enable debugging simply by commenting / uncommenting #define ENABLE_DEBUG. This allows us to reuse previously added debug statements and then just disable them when we're done with them, rather than having to actually remove them from the code. If this were a multi-file program, the #define ENABLE_DEBUG would go in a header file that's included into all code files so we can comment / uncomment the #define in a single location and have it propagate to all code files.

This addresses the issue with having to remove debug statements and the risk in doing so, but at the cost of even more code clutter. Another downside of this approach is that if you make a typo (e.g. misspell "DEBUG") or forget to include the header into a code file, some or all of the debugging for that file may not be enabled. So although this is better than the unconditionalized version, there's still room to improve.

---

## Using a logger

An alternative approach to conditionalized debugging via the preprocessor is to send your debugging information to a log. A **log** is a sequential record of events that have happened, usually time-stamped. The process of generating a log is called **logging**. Typically, logs are written to a file on disk (called a **log file**) so they can be reviewed later. Most applications and operating systems write log files that can be used to help diagnose issues that occur.

Log files have a few advantages. Because the information written to a log file is separated from your program's output, you can avoid the clutter caused by mingling your normal output and debug output. Log files can also be easily sent to other people for diagnosis -- so if someone using your software has an issue, you can ask them to send you the log file, and it might help give you a clue where the issue is.

C++ contains an output stream named `std::clog` that is intended to be used for writing logging information. However, by default, `std::clog` writes to the standard error stream (the same as `std::cerr`). And while you can redirect it to file instead, this is one area where you're generally better off using one of the many existing third-party logging tools available. Which one you use is up to you.



For illustrative purposes, we'll show what outputting to a logger looks like using the plog (https://github.com/SergiusTheBest/plog) logger. Plog is implemented as a set of header files, so it's easy to include anywhere you need it, and it's lightweight and easy to use.

```
#include <plog/Log.h> // Step 1: include the logger headers
#include <plog/Initializers/RollingFileInitializer.h>
#include <iostream>

int getUserInput()
{
    PLOGD << "getUserInput() called"; // PLOGD is defined by the plog library

    std::cout << "Enter a number: ";
    int x{};
    std::cin >> x;
    return x;
}

int main()
{
    plog::init(plog::debug, "Logfile.txt"); // Step 2: initialize the logger

    PLOGD << "main() called"; // Step 3: Output to the log as if you were writing to the console

    int x{ getUserInput() };
    std::cout << "You entered: " << x << '\n';

    return 0;
}
```

Here's output from the above logger (in the `Logfile.txt` file):

```
2018-12-26 20:03:33.295 DEBUG [4752] [main@19] main() called
2018-12-26 20:03:33.296 DEBUG [4752] [getUserInput@7] getUserInput() called
```

How you include, initialize, and use a logger will vary depending on the specific logger you select.

Note that conditional compilation directives are also not required using this method, as most loggers have a method to reduce/eliminate writing output to the log. This makes the code a lot easier to read, as the conditional compilation lines add a lot of clutter. With plog, logging can be temporarily disabled by changing the init statement to the following:

```
    plog::init(plog::none , "Logfile.txt"); // plog::none eliminates writing of most messages, essentially turning logging
off
```

We won't use plog in any future lessons, so you don't need to worry about learning it.

### As an aside...

If you want to compile the above example yourself, or use plog in your own projects, you can follow these instructions to install it:

First, get the latest plog release:

- Visit the plog repo.
- Click the green Code button in the top right corner, and choose "Download zip"

Next, unzip the entire archive to `somewhere` on your hard drive.

Finally, for each project, set the `somewhere\plog-master\include\` directory as an `include directory` inside your IDE. There are instructions on how to do this for Visual Studio here: A.2 -- Using libraries with Visual Studio (https://www.learncpp.com/cpp-tutorial/a2-using-libraries-with-visual-studio-2005-express/) and Code::Blocks here: A.3 -- Using libraries with Code::Blocks (https://www.learncpp.com/cpp-tutorial/a3-using-libraries-with-codeblocks/).

The log file will generally be created in the same directory as your executable.

**Tip**

In larger or performance-sensitive projects, faster and more feature-rich loggers may be preferred, such as spdlog (https://github.com/gabime/spdlog).


**Next lesson**
3.6 Using an integrated debugger: Stepping

**Back to table of contents**

**Previous lesson**
3.4 Basic debugging tactics