Debugging

Lucas Frérot

Table of contents

Recognizing and debugging errors is a fundamental skill when programming in C++, as is writing safe, expressive, and modern C++. We will cover the basics in this note.

Prerequisites

Make sure you have a full GCC, GDB and CMake installation:

```
sudo apt install g++ gdb cmake
gdb --tui # <- this should not give an error message
```

We'll work on the following files, which you can now download:

- CMakeLists.txt
- nullpointer.cpp
- localaddress.cpp
- outofbounds.cpp
- signedintegers.cpp

Put these files in a folder **separate** from the class project.

Compile

Run the standard CMake workflow to get started:

```
mkdir builddir
cd builddir
cmake -DCMAKE_BUILD_TYPE=Debug ..
cmake --build .

# To compile after changes to sources:
cmake --build .

# To change build type
cmake -DCMAKE_BUILD_TYPE=Release ..
cmake --build .
```

More information on build types: `Debug`, `Release`, `RelWithDebInfo`, `MinSizeRel`.

Compiling other files

You'll notice that `localaddress.cpp` and `outofbounds.cpp` are not compiled. To create executables, add these lines at the end of `CMakeLists.txt` and compile again.

```
add_executable(localaddress localaddress.cpp)
target_link_libraries(localaddress PRIVATE Eigen3::Eigen)

add_executable(outofbounds outofbounds.cpp)
target_link_libraries(outofbounds PRIVATE Eigen3::Eigen)
```

Adding compilation flags

To add compilation flags for all targets, add them to your `CMakeLists.txt`:

```
# Enable all warnings
add_compile_options(-Wall -Wextra -Wpedantic)

# Or for a specific target:
target_compile_options(myapp PRIVATE -Wall -Wextra)
```

The AddressSanitizer (Asan) needs compilation **and** link flags. Add to your `CMakeLists.txt`:

```
add_compile_options(-fsanitize=address)
add_link_options(-fsanitize=address)
```

Debugging

Many bugs can be caught by enabling warnings and AddressSanitizer at compile time as shown above and running the program. However, this will most times only show the point of crash of your program, not where the bug actually is. To find where the bug is, it is very useful to use a debugger like the GNU Debugger (GDB). We'll now explain the basic workflow with GDB.

Compiling

Make sure you configured CMake with `Debug` mode:

```
cmake -DCMAKE_BUILD_TYPE=Debug ..
cmake --build .
```

This turns *off* code optimizations (`-O0` compiler flag) and adds debugging symbols to the executable (`-g` compiler flag), which are both necessary to properly debug. Additionally, the macro `NDEBUG` is not defined, which enables the `assert()` function from the standard header `<cassert>` and enables bounds check in Eigen arrays. All these features will make your life easier when writing code.

Running GDB

We will run GDB with a Terminal User Interface (TUI):

```
gdb --tui ./nullpointer
```

The top window should show you the source code (if not, make sure you compile with `-DCMAKE_BUILD_TYPE=Debug`). If the top windows is highlighted with a blue border, you can use the arrow keys to navigate the code. Use the shortcut `C-x o` (Ctrl-X then O), to alternate focus between the top and bottom window. If the focus is not on the top window, the arrow keys will cycle the command history (similar to arrows in bash). If you want to enable command history you can add `set history save on` to `~/.gdbinit`.

Manipulating program state

You can use the `run` command to run the program until the crash. The source window will then show where the crash occurs, and why, but that's often not where the bug is. You can still use `up` to see which call of `set_to_one` has the crash. Using `start` you can execute the program from the very beginning, line by line.

Use the `print` command to show the value of the `pointer` variable. Since it is tedious to `print` at every instruction, use `watch pointer`, then `next` to advance the program execution. The `watch` command will show when `pointer` changes value. When hitting enter without a command, it just repeats the last command. Advance the program until `pointer` has the value `0x0` and fix the bug.

Notable other GDB commands:

- `info locals`: show all variables in current stack
- `print <var>`: show value of variable

- `display <var>`: show value of variable at every step
- `watch <var>`: show when variable changes value
- `backtrace`: show the call stack
- `up`: move up the call stack
- `down`: move down the call stack
- `break <file>:<line>`: put a breakpoint at given line in a file
- `break <function>`: breakpoint when a function is called
- `continue`: resume execution until breakpoint, crash or end of program
- `finish`: resume execution until current function returns
- `until`: resume execution until a source line "greater" than the current location—essentially until the end of a scope (loop, if statement, etc.)
- `C-l` (Ctrl+L) refresh display: sometimes display of the source gets weird, `C-l` refreshes the source display so it's clear again
- `print *pointer@N` prints `N` values starting at address `pointer`

Memory leaks

After having fixed the null pointer bug, activate the AddressSanitizer (see above) and run the program. It should show a memory leak of 4 bytes. AddressSanitizer can also be used for the two other examples (localaddress.cpp and outofbounds.cpp), usually showing the call stack (backtrace) at the moment of the crash.

Fixed files

Below are versions of the files above where we used idiomatic C++, the standard library and Eigen instead of C++-flavoured C:

- CMakeLists.txt
- nullpointer.cpp
- localaddress.cpp
- outofbounds.cpp
- signedintegers.cpp

Printing Eigen objects in GDB

Let's work with the following file:

```cpp
#include <Eigen/Core>

int main() {
  Eigen::Array4d static_{0, 1, 2, 3};
  Eigen::ArrayXd dynamic_(4);

  dynamic_ << 0, 1, 2, 3;
  return 0;
}
```

When debugging, we'd like to know the contents of both arrays. If we use the `print` command on one of these objects we have a verbose output:

```
(gdb) print static_
$1 = {<Eigen::PlainObjectBase<Eigen::Array<double, 4, 1, 0, 4, 1> >> = {<Eigen::ArrayBase<Ei

(gdb) print dynamic_
$2 = {<Eigen::PlainObjectBase<Eigen::Array<double, -1, 1, 0, -1, 1> >> = {<Eigen::ArrayBase<
1, 1, 0, -1, 1> >> = {<Eigen::DenseBase<Eigen::Array<double, -1, 1, 0, -
1, 1> >> = {<Eigen::DenseCoeffsBase<Eigen::Array<double, -1, 1, 0, -1, 1>, 3>> = {<Eigen::De
1, 1, 0, -1, 1>, 1>> = {<Eigen::DenseCoeffsBase<Eigen::Array<double, -1, 1, 0, -
1, 1>, 0>> = {<Eigen::EigenBase<Eigen::Array<double, -1, 1, 0, -1, 1> >> = {<No data fields>
```

We can find values for `static_` at `array = {...}`, but this is inconvenient. A better way is to print the member variable `m_storage`:

```
(gdb) print static_.m_storage
$3 = {m_data = {array = {0, 1, 2, 3}}}
(gdb) print dynamic_.m_storage
$4 = {m_data = 0x55555556beb0, m_rows = 4}
```

We can see stored values for the static array because they are stored on the stack. The dynamic array values are on the heap so we only see the address of the first element and the number of elements. We can use that information to print the stored values with the `print *pointer@N` syntax, which shows `N` continguous values starting at address `pointer`:

```
(gdb) print *dynamic_.m_storage.m_data@4
$5 = {0, 1, 2, 3}
```

Note that this can only show values as a 1D sequence, so the order of the values for 2D matrices depends on if matrices are row- or column-major.

A less crude way to print values would be to use Eigen's pretty printer for GDB. Follow the instructions in the file to install. You might have to add `set auto-load safe-path /` to `~/.gdbinit` to enable loading of foreign python code.