Deugging C++ with gdb

Efficiently overcoming bugs in C++ code in the Linux terminal environment

Hitham Hassan

IPPP, Dept. of Physics
Durham University

25/11/2021

Why Use a Debugger?

Bugs are varied in their nature - can cause the program to crash or introduce erroneous values when running.

When developing software *most* of the time is spent debugging.

Compilers will not often spot code that leads to segmentation faults (the most annoying type of error).

In-built debuggers are sophisticated and are designed to streamline this process.

Enter gdb - GNU DeBugger

You (should) have gdb. Open a terminal and run: `gdb`

Exit with: `q(uit)`

Can use to debug multiple languages - today we look at C++.

For help run: `gdb help [command]`

For more help the gdb manual is hosted online.

To make a C++ script debuggable compile with the `-g` flag e.g.
`g++ program.cc -o program -g`.

In the gdb console type `r(un)` to run your program

Basic gdb Commands

Set breakpoints : This stops your code in the debugger at a specific point that you provide e.g.
`(gdb) b(reak) file:line`
`(gdb) b(reak) function_name`

To get the file up in the terminal at the same time (useful on remote systems) use:
`gdb [executable] -tui` (text user interface)

to progress through after adding breakpoints, run the program and then use e.g.:
`(gdb) c(ontinue)` : continues to the next breakpoint.
`(gdb) n(ext)` : continues to the next subroutine.
`(gdb) s(tep)` : continues to the next line.

Can watch a variable - the running will stop whenever the variable is modified:
`(gdb) w(atch) variable_name`

Basic gdb Commands

Print a variable value - this is useful to do at different stages of large functions:
`(gdb) p(rint) variable`

Every time you print a new variable is assigned to the output e.g. `$1 $2 $3 ...`, similarly breakpoints are numbered. Delete these with:
`(gdb) d(elete) whatever you want to delete`

Define functions to use in the debugger:

```
(gdb) define func
Type commands for definition of "func".
End with a line saying just "end".
> type commands here
> separate new lines with ENTER/RETURN
> end
(gdb)
```

Exercise - Tracing Bugs

Clone the Git repository (see Git tutorial from a few weeks ago) and compile `HepCalculation.cc` with `make` which wraps around:

```
g++ HepCalculation.cc -o HepCalculation -lm -g
```

Open the executable in the `gdb` (use the TUI or have the source file open elsewhere).

In `main()` is implemented a complicated calculation on four momenta (using the bare bones `FourMomentum` class).

Run the program, you see `main()` produces a `nan` for our calculation.

Use `gdb` to spot from where the bug is introduced.

Resolving the Bug

The bug arose from momenta in the input that were not on-shell.

How is the issue resolved? Depends on your aims.

Imagine this formula came from HEP - we likely have conditions on the momenta going into the calculation (e.g. on-shell, time-like, come from momentum-conserving decays).

These could come from input files - one could add a check when generating input to make sure it is sensible **and/or** a check when reading the momenta.

The details of the calculation (as implemented) may be wrong.

One can filter out the problematic momenta by e.g. throwing *exceptions*.

Segmentation Faults and Backtrace

In general a segmentation fault occurs when **we try to access restricted memory**. Common culprits:

1. out of bounds arrays
2. for loops over too many iterations (`for(int j=0; j<=v.size(); ++j)`)
3. accessing uninitialised variables.
4. incorrect use of the reference `&` and dereference `*` operators (this could be a topic for a talk all on its own).
5. stack overflows (i.e. when memory on the stack is assigned above its limit. Example:

```
int func(int x){
  some code;
  return func(x);
}
```

Thank You