GDB (using the Fraction example)

- A decent resource: tutorialspoint gdb
- A better resource:
  - https://sourceware.org/gdb/onlinedocs/gdb/Stopping.html#Stopping
  - https://sourceware.org/gdb/onlinedocs/gdb/Altering.html#Altering
- **To follow this tutorial with the Fraction example, type the bold instructions into your command line**
- Need -g flag for g++ when compiling/linking to produce debugging information for GDB
  - **$ g++ -g -Wall -std=c++11 -c Fraction.cpp FractionUser.cpp**
  - **$ g++ -g -Wall -std=c++11 Fraction.o FractionUser.o -o Fraction**
- Run GDB with the Fraction program
  - **$ gdb ./Fraction**
- Let's investigate the copy constructor in FractionUser.cpp at line 11. We can do that by setting a breakpoint at that line.
  - **(gdb) break FractionUser.cpp:11 OR** (gdb) b FractionUser.cpp:11
- We can also set breakpoints by giving a function name, for example, the command to create a breakpoint at Fraction::reduce is
  - **(gdb) b Fraction.cpp:Fraction::reduce**
- Check for the new breakpoints
  - **(gdb) info break OR** (gdb) i b
- Note the first column Num: the value there can be used with delete to remove a breakpoint
  - **(gdb) delete 1 OR** (gdb) d 1
  - Command will remove the breakpoint for line 11 in FractionUser.cpp
- Set the breakpoint again
  - **(gdb) b FractionUser.cpp:11**
- Note that this breakpoint has Num=3
- Since we don't care about the breakpoint we set for Fraction::reduce yet, we can disable the breakpoint
  - **(gdb) disable 2**
- We can enable it later with
  - (gdb) enable 2
- Let's run the program
  - **(gdb) run OR** (gdb) r
- We can pass in arguments to programs by
  - (gdb) r <arg0> <arg1> …
  - The program will run as if it was invoked as *./progName* <arg0> <arg1> …
- Note that the program stopped running at line 11. We can inspect our Fraction object f3 before the copy constructor is called by
  - **(gdb) print f3 OR** (gdb) p f3
- Let's go inside the copy constructor
  - **(gdb) step OR** (gdb) s

    - Step executes the next instruction, and since the next instruction is the copy constructor, we go to the first line of the copy constructor and pause.
- If you lose track of where you are in your code, you can use list to show the surrounding code
    - **(gdb) list OR** (gdb) l
- We don't really care about what's happening inside this print statement, so we can skip it with
    - **(gdb) next OR** (gdb) n
    - Next executes until the next line, so we skip the details of operator<<, and it's simply executed. As a result, we see "Called copy constructor" in the terminal.
- Let's inspect the parameter and the members
    - **(gbd) p f2**
    - **(gbd) p numerator**
    - **(gdb) p denominator**
- If we wanted to inspect numerator and denominator with every step, we can save some effort with
    - **(gdb) display numerator OR** (gdb) disp numerator
    - **(gdb) display denominator OR** (gdb) disp denominator
- You can see which variables are auto-displayed with
    - **(gbd) i disp**
- Note the Num column again. It can be used with undisplay or undisp to remove auto-displayed variables
    - **(gdb) undisplay 2 OR** (gdb) undisp 2
    - Command will remove auto-display for denominator
- You can enable/disable auto-dispove auto-display for denominator with
    - **(gdb) disable disp 1**
    - **(gdb) enable disp 1**
- **Step** again and notice how gdb displays the updated value of numerator
- **Step** again. GDB prints out the value of numerator again.
- Let's print the f3 Fraction object to see the effects of the copy constructor
    - **(gdb) p f3**
    - This will give an error message: No symbol "f3" in current context. Why? Recall that we are still executing within the copy constructor and not main. But there is a way to refer to the f3 object.
    - **(gdb) p *this**
- **Next**. Notice how you get an error message: <error: current stack frame does not contain a variable named 'this'> Why? We setup the auto-display inside the constructor with numerator, which is actually short for this->numerator. But now we are outside the constructor so the pointer this does not exist. There is a way to inspect the numerator.
    - **(gdb) p f3.numerator**
- **Disable** auto-display for numerator so we stop getting those error messages
- We can run until the next breakpoint by
    - **(gbd) continue**

- Exit the debugger with
    - **(gdb) quit OR** (gdb) q
- Other useful commands
    - **whatis *variable*** - shows the type of *variable*
    - **finish** - run until current function completes then pause
    - **watch *expression*** - set a watchpoint (let's you stop the program when the value of *expression* so this is extremely helpful)
        - (gdb) watch foo
            - Stops the program when foo changes value
        - (gdb) watch (myPointer != 0x0)
            - Stops the program if myPointer becomes nullptr (I hope you can see how useful this is for segmentation faults)
- Example of watch (crash.cpp)
    - Notice line 15: **ptr = nullptr;**
    - And line 18: **ptr->myMember;** (causes segmentation fault)
    - Compile crash.cpp: **$ g++ -g -std=c++11 crash.cpp -o crash**
    - Run gdb with crash: **$ gdb ./crash**
    - Set breakpoint: **(gdb) b main**
    - Run crash: **(gdb) r**
    - Set watchpoint: **(gdb) watch (ptr != 0x0)**
    - Continue running: **(gdb) c**
    - Notice how the program stops immediately after line 15 because it detected that **ptr** was set to nullptr.

Valgrind

- A good resource: http://valgrind.org/docs/manual/quick-start.html
- See also: http://es.gnu.org/~aleksander/valgrind/valgrind-memcheck.pdf
- Memory leak: memory that was allocated but not released back to the system after use.
- Run with
    - **$ valgrind *progName args...***
- To get more details
    - **$ valgrind --leak-check=full *progName args...***
- Leak Summary:
    - **Definitely lost**: program is leaking memory, must fix.
        - at the end of program execution, there is no pointer to the allocated memory
    - **Indirectly lost**: program is leaking memory, must fix.
        - at the end of program execution, there is a pointer to the allocated memory, but that pointer is in another directly lost block.
    - **Possibly lost**: valgrind is not sure if this memory is definitely lost or still reachable.
        - at the end of program execution, there is no pointer to the allocated memory, but there is at least one pointer to inside the allocated memory.

    - **Still reachable**: usually not harmful.
        - at the end of program execution, there is at least one pointer to the allocated memory.
- Also reports invalid reads/writes, meaning writing or reading outside of allocated memory
    - If we have **int *arr = new int[5];**
    - **int temp = arr[6];** this is reading out of bounds
    - **arr[-1] = 5;** this is writing out of bounds
- Can try valgrind with leak.cpp
    - Notice line 18: **myClass *ptr1 = new myClass();**
    - And line 19: **myClass *ptr2 = new myClass(100);**
    - Notice also that there are no **delete** statements in main().
    - Run valgrind: **$ valgrind ./leak**
        - Notice how it says we have leaked memory in the summary
    - To get more information on the leak run valgrind again
        - **$ valgrind --leak-check=full ./leak**
        - Notice how valgrind reports the where the leaks occurs
            - by … : main (leak.cpp:18)
            - by … : main (leak.cpp:19)