Before we begin, update your VM...

Ensure all local packages are up to date

- FIRST SET A CHECKPOINT FOR YOUR VM
- `sudo apt-get update`

Upgrade/Install some packages

- `sudo apt-get upgrade -y gdb`
- `sudo apt-get install -y python3-pip`
- `pip3 install --upgrade pip`
Tips and Tricks Update

- Edit your `~/.ssh/config` to contain the following (works on MacOS, Linux, VM, and WSL)

```bash
### CAEN
Host caen login.engin.umich.edu
    HostName login.engin.umich.edu
    User mmdarden  # Use your own uniqname
    ControlMaster auto
    ControlPath ~/.ssh/_%r%@h:%p
    ControlPersist 43200
```

- When connecting to CAEN (with `ssh caen`)
  - First login requires password and DUO
  - Subsequent logins connect instantly (for 12 hours, or until...)
  - When the multiplexing expires or is broken (rules unknown)
  - Works for everything that uses ssh (commands, sessions, 3rd party software, etc.)
  - 2 useful commands
    - `ssh -O check caen`
    - `ssh -O stop caen`
  - Also, look for the file `~/.ssh/_mmdarden@login.eecs.umich.edu:22`
TTU++

- Connect your local dev environment to CAEN
  - Use `rsync` and a "Post-build script"
- EECS 281 example: [https://gitlab.eecs.umich.edu/eecs281/makefile](https://gitlab.eecs.umich.edu/eecs281/makefile)
  - Look at `$(REMOTE_BASEDIR)`
  - Look at `$(REMOTE_PATH)`
  - Look at target `sync2caen`
- Xcode example:
  - Edit Scheme...
  - Add a "Build Post-action"
  - Name: "Sync to CAEN"
  - Shell: `/bin/bash`
  - Provide build settings from: `<current scheme>`
  - Add the following script

```bash
# Auto upload from Xcode to CAEN
make -C "${SRCROOT}" sync2caen > "${SRCROOT}/rsync.log"
open "${SRCROOT}/rsync.log"
```

- Check on CAEN in `~/$(REMOTE_PATH)`
- Sync happens after every successful build!
# Auto upload from Xcode to CAEN

1. make -C "${SRCROOT}" sync2caen > "${SRCROOT}/rsync.log"

2. open "${SRCROOT}/rsync.log"
Debuggers
What Does gdb Do?

Yes

- Start your program (with options and arguments)
- Stop your program
- Allow you to see into registers and memory
- Allow you to change values manually during execution
What Does gdb Do?

Yes

- Start your program (with options and arguments)
- Stop your program
- Allow you to see into registers and memory
- Allow you to change values manually during execution

No

- MAGIC
How Do I gdb?

To debug a program with gdb, simply put `gdb` in front of the program, i.e.:
How Do I gdb?

To debug a program with gdb, simply put `gdb` in front of the program, i.e.:

```
> ./prime  # running normally
> gdb ./prime  # debugging the program with gdb
```
How Do I gdb?

To debug a program with gdb, simply put `gdb` in front of the program, i.e.:

```bash
> ./prime # running normally
> gdb ./prime # debugging the program with gdb
```

One annoying gotcha shows up if the program to debug takes any options. The simple prime program does not, but if it did:

```bash
> ./prime --imaginary-option # running normally
> gdb ./prime --imaginary-option # will not work
  gdb: unrecognized option '--imaginary-option'
> gdb --args ./prime --imaginary-option # gdb will ignore everything after --args
```
GDB's Text User Interface

- It's a CLI program, get over it!
- Nope... Beast Mode... GDB TUI
  - At launch with --tui
  - After launch with `C-x 1`
GDB's Text User Interface

- It's a CLI program, get over it!
- Nope... Beast Mode... GDB TUI
  - At launch with --tui
  - After launch with C-x 1

GDB TUI Key Bindings (partial)

<table>
<thead>
<tr>
<th>Binding</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-x a</td>
<td>Enter/exit TUI</td>
</tr>
<tr>
<td>C-x 1</td>
<td>Change TUI layout?</td>
</tr>
<tr>
<td>C-x 2</td>
<td>Change TUI layout</td>
</tr>
<tr>
<td>C-x o</td>
<td>Switch window focus</td>
</tr>
<tr>
<td>C-x s</td>
<td>Single Key mode</td>
</tr>
<tr>
<td>C-l</td>
<td>Refresh screen</td>
</tr>
<tr>
<td>C-p, C-n, C-b, C-f</td>
<td>Readline navigation (Emacs FTW!)</td>
</tr>
</tbody>
</table>
**GDB TUI Single Key Mode**

- This is truly GDB Beast Mode... on steroids!

<table>
<thead>
<tr>
<th>Key</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>continue</td>
</tr>
<tr>
<td>d</td>
<td>down</td>
</tr>
<tr>
<td>f</td>
<td>finish</td>
</tr>
<tr>
<td>n</td>
<td>next</td>
</tr>
<tr>
<td>q</td>
<td>exit the Single Key mode</td>
</tr>
<tr>
<td>r</td>
<td>run</td>
</tr>
<tr>
<td>s</td>
<td>step</td>
</tr>
<tr>
<td>u</td>
<td>up</td>
</tr>
<tr>
<td>v</td>
<td>info locals</td>
</tr>
<tr>
<td>w</td>
<td>where</td>
</tr>
<tr>
<td>___</td>
<td>________________</td>
</tr>
</tbody>
</table>
gdb Commands

run

- Starting gdb will not run your program by default. You must use the `run` command to begin execution.
- Using `run` will start your program with the options originally specified, or you can pass new options with `run`.

```
(gdb) run --different-option
```

- If your project is recompiled, each `run` will automatically reload the new version. Debugging is easier if you don't quit gdb, but leave it running in a separate terminal.
While your program is running, it has a function call stack that is built up with frames that hold parameters, locals, and register information for each invocation. Consider math.c:

```cpp
#include <iostream>
using namespace std;
int subtract (int a, int b) { return a - b; }
int divide (int a, int *b) { return a / *b; }
int do_math (int x, int y, int z) {
    int temp = subtract(x, y);
    temp = divide(z, &temp);
    return temp;
}
int main () {
    int temp = do_math(10, 10, 20);
    cout << "Result: " << temp << endl;
    return 0;
}
```

Function call stack (growing to the right)
gdb Commands

list, break, continue, step, next, finish, set

- Look at your source with list or list <function>
gdb Commands

- list, break, continue, step, next, finish, set
  
  - Look at your source with list or list <function>
  
  - Stop and start your program with break and continue
gdb Commands

list, break, continue, step, next, finish, set

- Look at your source with list or list <function>
- Stop and start your program with break and continue
- Take things at your own pace with step (into), next, and finish (out)
gdb Commands

**list, break, continue, step, next, finish, set**

- Look at your source with `list` or `list <function>`
- Stop and start your program with `break` and `continue`
- Take things at your own pace with `step` (into), `next`, and `finish` (out)
- Make a change to variables and registers with `set`
More on breakpoints

- Generally specified by filename:linenumber
- Will also work in context
- List all current breakpoints with `info breakpoints`
- Remove with `delete <number>` or `disable <number>` until later
- Skip over working code with breakpoints on either side and `continue`

Conditional breakpoints

- Unbelievably efficient for debugging
- Can create with `break myfile:11 if x == 5`
- Can extend with `condition <breakpoint number> x == 5`
GDB Does Python!!

- Access to GDB internals
- Variables, functions, etc.
- Inline, short entry, and script
- A pretty printer

```python
class ObjectPrinter:
    '''Pretty print an Object'''

    def __init__(self, val):
        self.val = val

    def to_string(self):
        '''Change this to reflect real properties from the object'''
        return self.val

    def display_hint(self):
        return 'Object'

    def lookup_type(self, val):
        return ObjectPrinter(val) if val.type == 'Object' else None

gdb.pretty_printers.append(lookup_type)
```
The New Hotness... gdbgui

- pip3 install gdbgui --upgrade
- Rerun the previous debug session
- Start a new debug session
Open Problems with Debugging

Look at `inf.c`

```c
#include <stdio.h>

int recurse(int add_me) {
    if (add_me == 1) {
        return add_me;
    }
    return recurse(add_me + add_me);
}

int main() {
    printf("%d\n", recurse(2));
}
```