CPSC 490 Final Report

C-Bugger:
A Visual Integration of gdb for Remote Debugging

By: Addie Elwood
Advisor: James Glenn

May 2, 2019

1. Abstract

In computer science classes, learning how to debug a program can be just as difficult as learning how to write it in the first place. Due to a limited number of tools for C programs, the main debugging challenges for Yale students taking the core sequence are: the most popular debugger (gdb) has a steep learning curve because it is text-based, the results of the debugger are divorced from the source code, and programs should ideally be debugged on the Zoo where the final test scripts are run. Therefore, the purpose of this project is to provide a more intuitive and accessible interface for debugging C programs that displays the debugging results in relation to the code that produced it, and allows debugging to take place on a remote computer system. This was achieved by developing a plugin for Sublime Text 3, a popular text editor used by Yale students. It already has a plugin for remotely editing files, which was the inspiration for creating a plugin for remote debugging. The Sublime Text API provides front-end functionality, and the python library paramiko is used to open an SSH connection to the remote server, where gdb is launched to perform the actual debugging commands. This plugin is currently integrated with the SFTP plugin to retrieve information about the remote servers in order to login. The user selects what remote server to debug on, and then selects what executable file to run in the debugger. From there, the user can perform standard debugging operations by clicking in Sublime: setting and removing breakpoints with a right-click on a line, recompiling the executable, running the debugger, selecting what to do next after hitting a breakpoint. This project successfully functions as a simple remote debugging tool compatible with C programs. There are many possible enhancements such as increasing configurability of gdb commands, displaying results directly tied on a line of code, and adding valgrind capabilities.

2. Background

2.1 State of the Art of Debugging

Students struggle in introductory computer science classes for a lot of reasons. Learning how to code is hard, which is why debugging tools are so important for helping new programmers figure out what's going on. However, many classes don't leave space for teaching students how to debug, and instead they resort to clunky print statements or just staring at their code. As an undergraduate learning assistant, I know that the best way to help a student is to help them help themselves, by showing them the powerful capabilities of a real debugger. Such tools allow programmers to step through their program, displaying different kinds of information about what's happening, so they can understand their own code.

The core sequence in the Computer Science department at Yale is based on the C programming language, which offers many learning opportunities as a low-level language. However, unlike Java, Racket, and other languages, there is no IDE (Integrated Development Environment) for C. Instead, students use text editors to write their code, makefiles to help compile it, and bash commands typed into their terminals to run it. For new C programmers, this workflow, with details in each component, is a lot to learn on top of the complexities of the language itself. Adding a debugging tool into the mix can take significant effort, and these students often don't realize the ultimate benefits when faced with the difficulty of learning another unfamiliar tool.

Another unique challenge for computer science students at Yale is that class assignments are submitted and final test scripts are run on the Zoo, the CS department's network of computers. Because there can be differences between the student's operating system and the Zoo (ranging from minor environment settings to completely missing the right compiler) the best practice is to develop code directly on the Zoo. This means that students use a combination of SSHing, SCPing, and FTP/SFTP (transferring files) to edit their code and run it on the remote computer system. This makes it harder to use standard applications to debug code, since those tools run on the local computer. Learning the difference between the remote and local systems is yet another challenge for students in the introductory sequence.

Debugging is an extremely important skill for computer programmers. Learning good debugging strategies and the full capabilities of a debugger benefits students throughout their computer science careers in school and beyond. The challenge emerging is how to help students at Yale learn to love debugging, even during the difficulties of the introductory computer science sequence. However, there just aren't a lot of options for debuggers for C, and the existing tools present particular challenges for new programmers in introductory classes, and for remote debugging like on Yale's computer system.

2.2 Existing Tools

The primary debugging tool is gdb. Although it isn't a formal part of most computer science classes, it's often taught informally at office hours when students encounter issues in their code. While gdb is very powerful, and can be used on a remote system like the Zoo via the terminal, it can be challenging to learn, for two main reasons. First, gdb is text-based, so each command needs to be typed into the terminal. This means that students need to memorize gdb commands in addition to all the new C syntax. It also means that the extent of gdb's capabilities gets hidden. Students may only learn or remember a few simple commands, so they end up using gdb inefficiently. The second disadvantage of a terminal-based tool like gdb is that the results of the debugger are divorced from the code. Students writing in a text editor and debugging in the terminal have to switch back and forth, and have to remember both what line they're on and what the debugging information means as they do. Overall, gdb is an extremely useful tool, but its power gets limited by the inefficient workflow and lack of knowledge of its users.

Such challenges have been recognized before, which is why IDEs are so popular with other languages. These tools are full applications that allow users to edit the source code, and then build (compile) and debug their programs from the same window. Usually there's a huge array of buttons and menus to configure everything. However, because C is relatively unpopular, and most projects made with C are small (for example, to be uploaded onto a memory-limited embedded system), there is no IDE designed for C programs. In addition, trying to use an existing IDE for C is overly complicated, especially for the simple assignments in the core sequence of classes. The setup is difficult, the numbers of buttons and options is overwhelming, and the actual capabilities get lost in the mess. These also offer no remote support, since these applications run on the local computer.

Some attempts have been made to make gdb more accessible. One example is ddd, the Data Display Debugger. This tool is a visual front end, a graphical user interface for gdb that can be launched from the command line. It's powerful, and it does allow the user to see the source code alongside the debugging information. However, ddd is very old and has a significant learning curve. It's ugly, which doesn't detract from its capabilities, but is off-putting to new computer science students. It also doesn't allow the user to directly edit the code while viewing the debugging information. Another project that created a graphical user interface for gdb is gdbgui, found at gdbgui.com. This is a much more modern tool that combines the capabilities of gdb with the ease of a clickable user interface, making the debugging functionality intuitive and accessible. These tools are also challenging to use to debug remote programs, although gdbgui recently revamped and now appears to have this capability. However, it also does not allow users to edit the source code while debugging.

2. 3 Project Goal

Clearly, there is a need in the Yale Computer Science Department, and hopefully more widely, for a debugging tool for C programs that provides a simple and intuitive interface for the debugger commands, that allows the program to be debugged on a remote computer system, and that displays the debugging information in the context of the source code, which can then be edited on the fly.

To create a debugger that meets these criteria, I decided to integrate gdb with Sublime Text, which is the most popular text editor used by students in the core sequence of computer science classes. Besides being a clean, simple text editor perfect for C code, its capabilities can be extended through plugins. Students at Yale use the SFTP plugin to open and edit their source files on the Zoo. Although not all students use this tool, it is popular enough that there would be an audience for a debugger that is compatible with Sublime Text, and the plugin extensibility means that I can rely on the complex text editor capabilities without having to build them up myself. Also, because students are already using SFTP, I can use some of that infrastructure to launch the remote access for my debugger. For example, when setting up SFTP, users create files with information such as host name, username, and even password. This also means I don't have to rewrite the process of opening and saving a remote file.

So, the goal of my project was to include basic debugging capabilities in the user interface of Sublime Text, while having the debugger itself run on a remote server. I had choices about what gdb commands to include, how the user should select what to do, and how to display the resulting information. I also faced the challenge of how to implement these capabilities.

3. Project

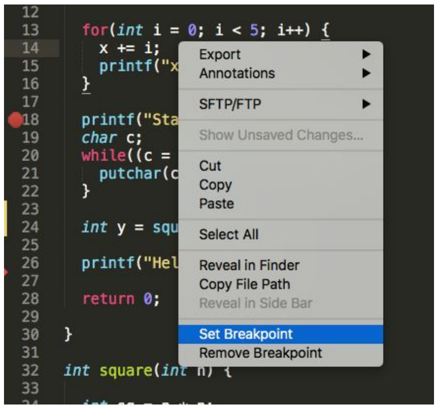
Ultimately, I was able to build a debugging tool that provides an interface for remote gdb through Sublime Text. Users can access most of the basic gdb commands, which will be discussed in more detail in the next section, and can continuously edit and recompile their code while using the debugger. There are still some limitations, but the project works as a debugger for simple C programs. The working title for the plugin is “C-bugger” as an homage to the language and purpose it was designed for.

3.1 Functionality

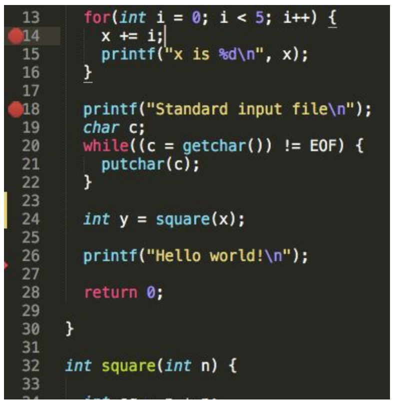
To demonstrate the capabilities of the tool, imagine that the user has successfully installed the Cbugger plugin — the specific steps to install and set up this plugin are covered in the next section. The user has opened a C source file using the SFTP plugin, and now they are ready to debug.

3.1.0 Breakpoints

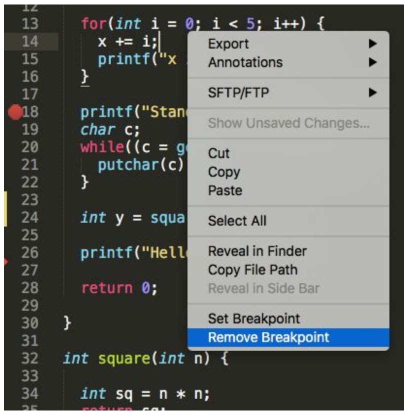
Breakpoints can be set at any time, before or after the remote gdb process is launched. Currently, breakpoints are set by line numbers. The user right-clicks on the line where they would like to modify a breakpoint. At the bottom of the pop-up menu are two options, to set or remove a breakpoint. A set breakpoint is indicated with a red stop-sign-shape on the left side of the editor; the stop sign is removed when the breakpoint is.



```
for(int i = 0; i < 5; i++) {
    x += i;
    printf("x
}

printf("Sta
char c;
while((c =
    putchar(c
}

int y = squ

printf("Hel

return 0;
}

int square(int n) {
```



```
for(int i = 0; i < 5; i++) {
    x += i;
    printf("x is %d\n", x);
}

printf("Standard input file\n");
char c;
while((c = getchar()) != EOF) {
    putchar(c);
}

int y = square(x);

printf("Hello world!\n");

return 0;
}

int square(int n) {
```



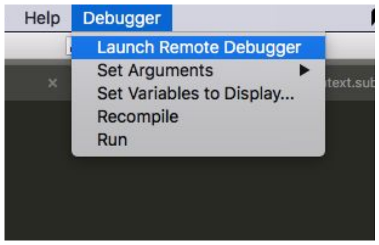
```
for(int i = 0; i < 5; i++) {
    x += i;
    printf("x
}

printf("Stan
char c;
while((c = g
    putchar(c)
}

int y = squa

printf("Hell

return 0;

}

int square(int n) {

    int sq = n * n;
```

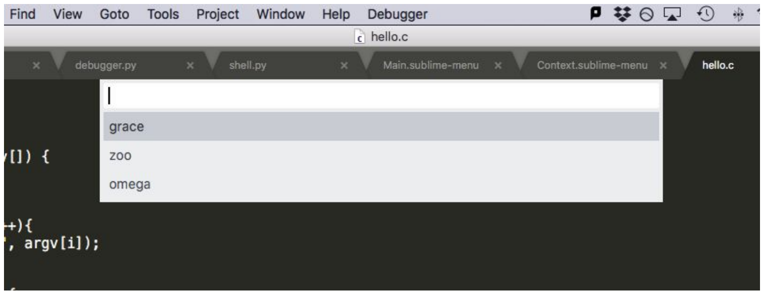
Limited support for multiple source files is available. Breakpoints can be set in multiple files, and the debugger will set breakpoints associated with the file they're in. The debugger keeps track of what files are valid source files for the executable.

Importantly, the breakpoints move when the code is edited. This was surprisingly challenging to implement, but when the file is edited, the program checks whether the breakpoint has moved lines; if it has, then the breakpoint is removed and re-set to be on the new line. This is crucial for the program to function as a proper debugger. When the user edits their code to fix it, they don't need to re-set breakpoints every time. The debugger keeps track for them.

3.1.1 Launch

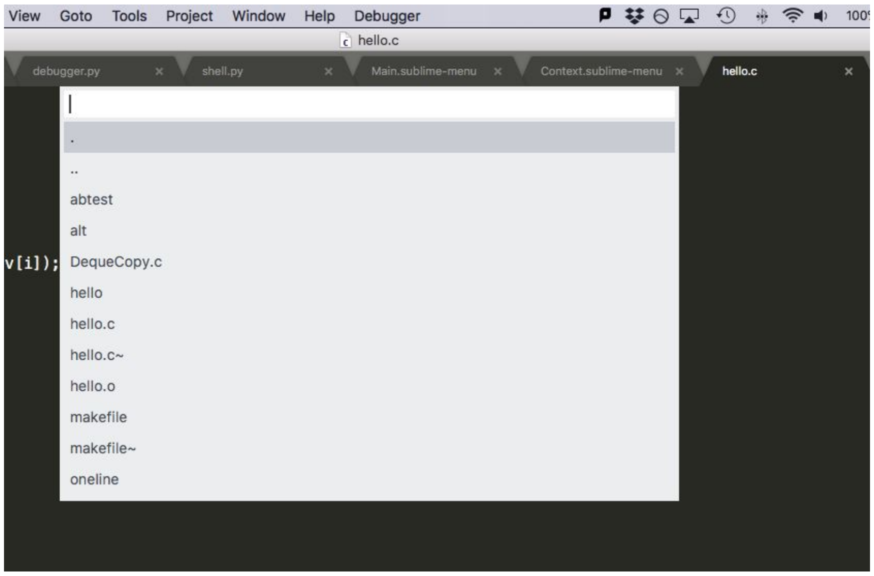
The first step to launching the debugger is to start the remote gdb process. In order to do this, the user selects the remote server and then which executable file should be debugged. The remote choice is made from the list of SFTP server files, which is why the plugin relies on the SFTP plugin. The connection to the remote is handled by the paramiko library. Currently, the information to start the SSH connection is extracted from the SFTP server files.





Names of server files created for SFTP

Because the file is being edited through the SFTP plugin, the program immediately navigates to the current directory of that file, then displays the contents for the user to select the proper executable. However, if the file can't be found, the debugger will open to the user's home directory. In either case, the user can navigate up and down in the directory system to find and select the correct executable. The program detects if the user selects a file that is not able to be debugged (for example, if it's not an executable) and issues an error message.



```
.
..
abtest
alt
DequeCopy.c
hello
hello.c
hello.c~
hello.o
makefile
makefile~
oneline
```

The list of files in the remote directory where the source file is located

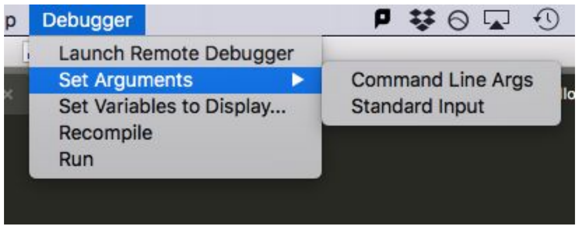
Establishing the connection to the remote server is where my plugin most heavily uses the SFTP plugin, and I debated how much to let my project rely on this other plugin. At first I wanted it to be completely independent. However, because the SFTP plugin handles remotely opening and editing files, I believe it's okay for these plugins to be a package deal. With that decision, one modification to make is that the user doesn't need to select which server file: that information can be extracted from the source file, which will make launching the debugger even more intuitive, since the remote connection can be handled completely automatically.

The remote connection was one of the most challenging parts of this project. The paramiko library was crucial, but because Sublime runs its own Python environment, importing libraries can be tricky. I even started to implement the SSH connection more crudely, using subprocess and the ssh bash command. However I was ultimately able to import it, and use the invoke_shell command to establish a continuous connection. This is a bare-bones communication channel that's usually used to implement terminal front-ends, so I needed to do a lot of string parsing to extract the relevant information beyond the terminal display information (such as the command prompt and regurgitated standard input).

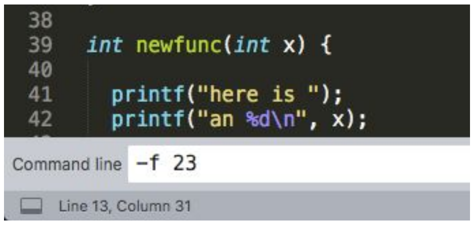
Once the remote connection has been established, there are three “modes” that the program could be communicating from: in the standard terminal, in gdb, or while gdb is running the executable. I wrote functions to communicate in each of the three modes, which allowed me to focus back on the debugging capabilities. For reading results from gdb, I used the machine interpreter mode, which outputs in a more easily processed format, and I then transformed those results, using the pygdbmi library and the python json library, into an neat dictionary of information. This allows access to different parts of the gdb output, making it easy to sort the types of gdb messages or find particular pieces of information like line number or file name. These libraries were very valuable for communicating easily with gdb.

3.1.2 Set Input

Once gdb has launched with the proper executable, the user may want to set up to actually run the program in the debugger, and to do so, the user may want to set inputs. C-bugger can handle two kinds of inputs. The user can enter command-line arguments as a line of input which is prompted. Or, the user can choose a file that standard input will read from. Currently, the user selects a file from the current directory of the source file.



Menu

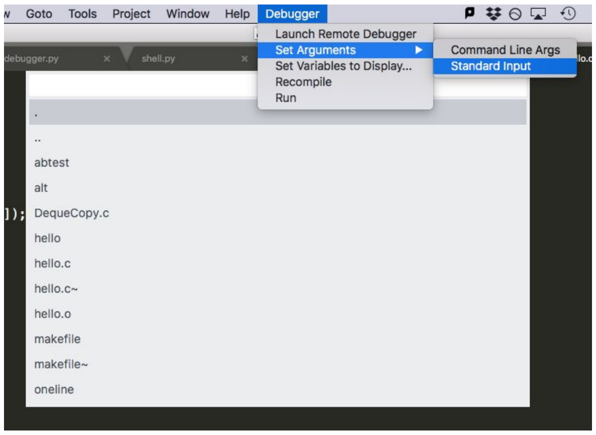


```
int newfunc(int x) {

  printf("here is ");
  printf("an %d\n", x);
```

Command line -f 23

Line 13, Column 31

Command line input



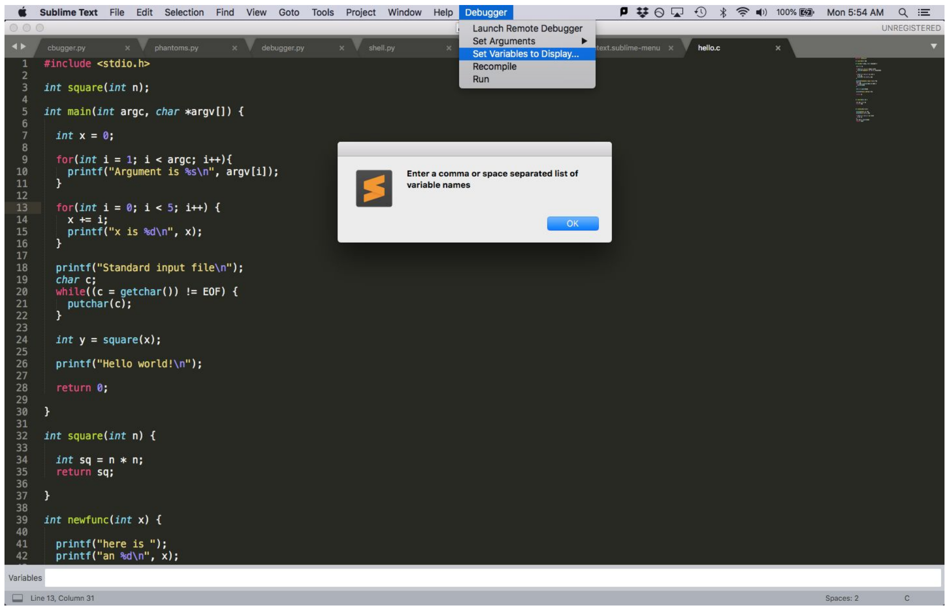
Choosing file to redirect to standard input

Displaying file options is implemented with another kind of remote connection in paramiko, in order to not interrupt the gdb process. This uses the exec_command function, which creates a new Channel on the same Transport (essentially a new “tunnel” using the same credentials from the existing SSH connection), executes a single command, and then closes the connection. Currently, because of this, the menu for selecting input files is not navigable.

Standard input presented a challenge, because the program being debugged hangs while it waits. To prevent this, if a file is not selected, standard input is redirected to /dev/null. Future enhancements could include the user directly entering input, or adding here docs.

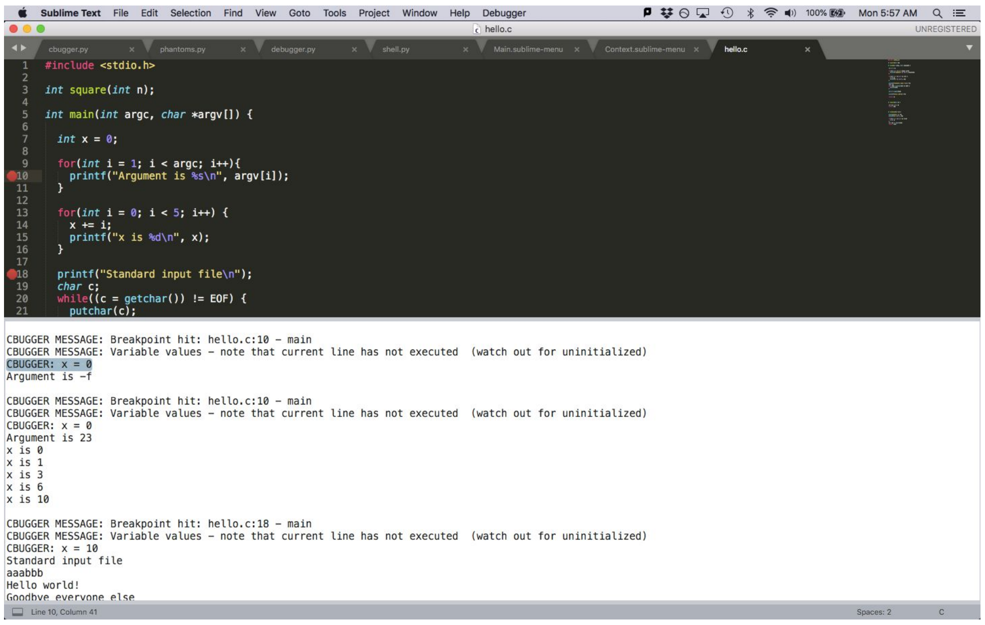
3.1.3 Display Variables

One method for displaying debugging information is to preselect what variables to inspect when reaching a breakpoint. This has mixed levels of usage, but it's helpful when the same variable needs to be displayed repeatedly. This menu option allows the user to enter a comma and/or space separated list of variable names. When a breakpoint is hit, or the debugger advances to a new line without exiting, these variables and their values are displayed if they are in scope. Because this option is a little confusing, I included a pop-up display message.



```c
#include <stdio.h>

int square(int n);

int main(int argc, char *argv[]) {

  int x = 0;

  for(int i = 1; i < argc; i++){
    printf("Argument is %s\n", argv[i]);
  }

  for(int i = 0; i < 5; i++) {
    x += i;
    printf("x is %d\n", x);
  }

  printf("Standard input file\n");
  char c;
  while((c = getchar()) != EOF) {
    putchar(c);
  }

  int y = square(x);

  printf("Hello world!\n");

  return 0;

}

int square(int n) {

  int sq = n * n;
  return sq;

}

int newfunc(int x) {

  printf("here is ");
  printf("an %d\n", x);
```

Setting variables to display repeatedly (input box on bottom)



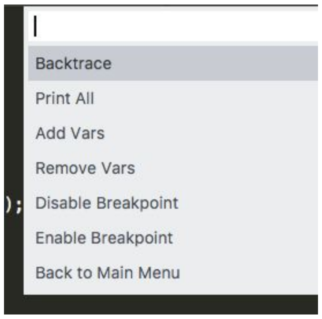
```c
#include <stdio.h>

int square(int n);

int main(int argc, char *argv[]) {

  int x = 0;

  for(int i = 1; i < argc; i++){
    printf("Argument is %s\n", argv[i]);
  }

  for(int i = 0; i < 5; i++) {
    x += i;
    printf("x is %d\n", x);
  }

  printf("Standard input file\n");
  char c;
  while((c = getchar()) != EOF) {
    putchar(c);
```

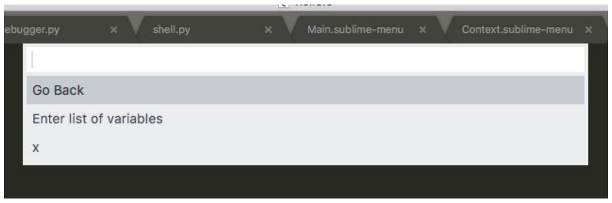
```
CBUGGER MESSAGE: Breakpoint hit: hello.c:10 - main
CBUGGER MESSAGE: Variable values - note that current line has not executed  (watch out for uninitialized)
CBUGGER: x = 0
Argument is -f

CBUGGER MESSAGE: Breakpoint hit: hello.c:10 - main
CBUGGER MESSAGE: Variable values - note that current line has not executed  (watch out for uninitialized)
CBUGGER: x = 0
Argument is 23
x is 0
x is 1
x is 3
x is 6
x is 10

CBUGGER MESSAGE: Breakpoint hit: hello.c:18 - main
CBUGGER MESSAGE: Variable values - note that current line has not executed  (watch out for uninitialized)
CBUGGER: x = 10
Standard input file
aaabbb
Hello world!
Goodbye everyone else
```

Variable value printed at each breakpoint (x was set)

3.1.3.a Add Vars, Remove Vars

Because the focus changes as the debugging process continues, the variables to display may need to be changed. When a breakpoint is hit, the user has the option (on the second menu, see 3.1.6) to add variables to the list to be displayed at every stopping point, or to choose variables currently on the list to remove so that they stop printing every time.

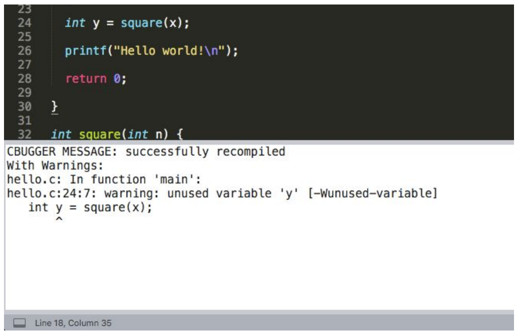




Add Vars, Remove Vars options *Example of Remove Vars menu*

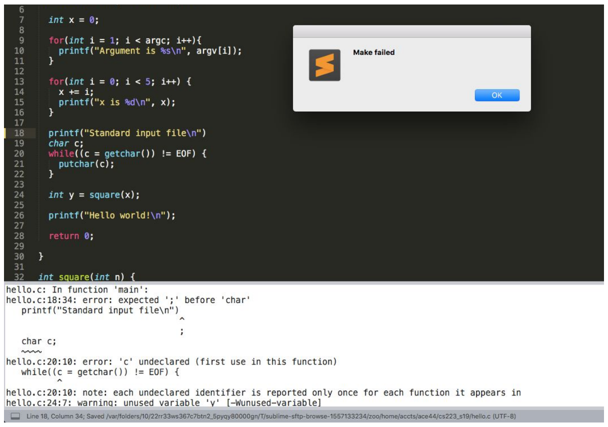
3.1.4 Recompile

An important part of the debugging process is fixing things incrementally, re-testing, and finding new errors. In order for this be a truly end-to-end tool, when the user edits their code, they must be able to run the new version. For C programs, this means they need to be able to recompile the executable. Luckily, gdb can handle when an executable is recompiled in the middle of running. This menu option allows the user to recompile their code with the click of a button, instead of having to navigate back to the terminal, type the command, and return. The debugger also displays the results of the recompilation in a panel that pops up at the bottom. It displays a success message that does include warnings. For errors it displays the content and has an additional warning pop up.



```
    int y = square(x);

    printf("Hello world!\n");

    return 0;

  }

  int square(int n) {
```

```
CBUGGER MESSAGE: successfully recompiled
With Warnings:
hello.c: In function 'main':
hello.c:24:7: warning: unused variable 'y' [-Wunused-variable]
   int y = square(x);
       ^
```



```
    int x = 0;

    for(int i = 1; i < argc; i++){
      printf("Argument is %s\n", argv[i]);
    }

    for(int i = 0; i < 5; i++) {
      x += i;
      printf("x is %d\n", x);
    }

    printf("Standard input file\n")
    char c;
    while((c = getchar()) != EOF) {
      putchar(c);
    }

    int y = square(x);

    printf("Hello world!\n");

    return 0;

  }

  int square(int n) {
```

```
hello.c: In function 'main':
hello.c:18:34: error: expected ';' before 'char'
   printf("Standard input file\n")
                                  ^
                                  ;
   char c;
   ~~~~
hello.c:20:10: error: 'c' undeclared (first use in this function)
   while((c = getchar()) != EOF) {
          ^
hello.c:20:10: note: each undeclared identifier is reported only once for each function it appears in
hello.c:24:7: warning: unused variable 'y' [-Wunused-variable]
```

Make failed

OK

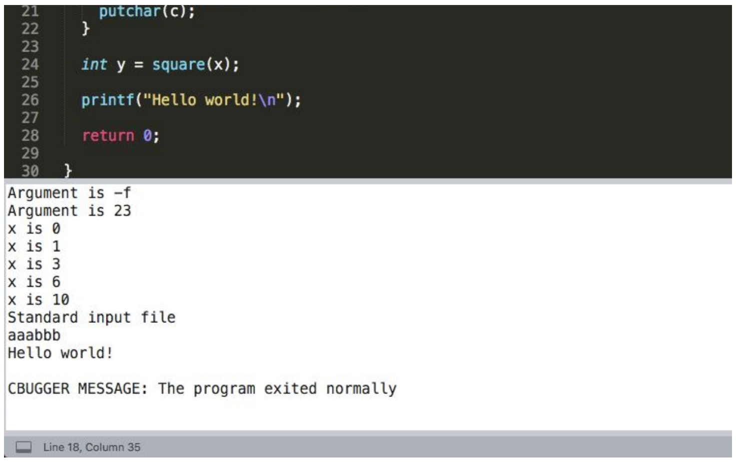
Line 18, Column 34; Saved /var/folders/10/22rr33ws367c7btn2_5pyqy80000gn/T/sublime-sftp-browse-1557133234/zoo/home/accts/ace44/cs223_s19/hello.c (UTF-8)

Successfully compiled, with warnings *Failed with compiler errors*

The main limitation of this command is that it currently only runs the 'make' command. Other ways to compile and more configurations for 'make' would improve the flexibility of the debugger. However, the makefile can be edited in Sublime, and ‘make’ is the way students are taught to compile their C programs in the core sequence classes that this debugger is designed for.

3.1.5 Run

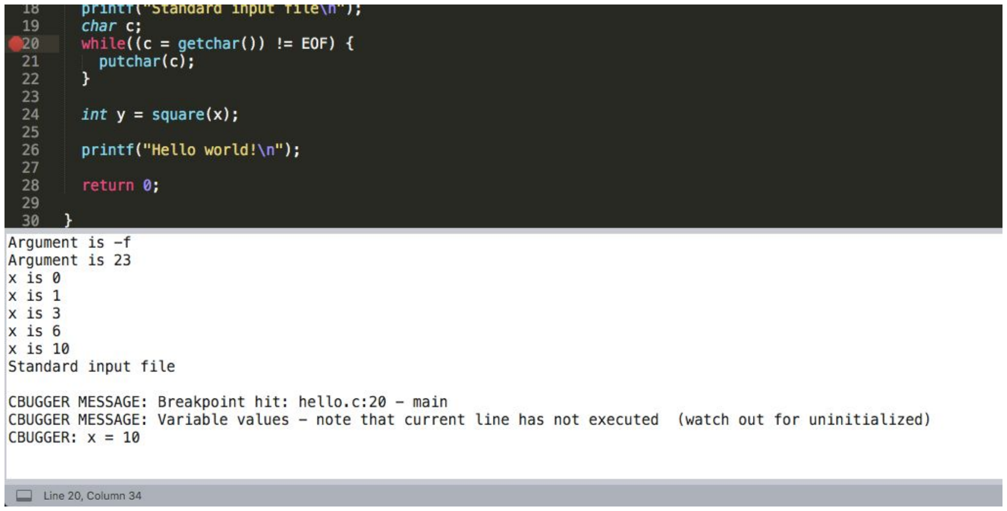
After all of that setup, the user is finally ready to run their program in the debugger. All arguments and breakpoints are set beforehand, there won’t be a prompt. This is a simple command since it really just runs the debugger. The output from the program is displayed in a panel at the bottom. Debugger information is printed with each line starting with “CBUGGER”, so standard output from the program can be differentiated.



```
    putchar(c);
  }

  int y = square(x);

  printf("Hello world!\n");

  return 0;

}
```

```
Argument is -f
Argument is 23
x is 0
x is 1
x is 3
x is 6
x is 10
Standard input file
aaabbb
Hello world!

CBUGGER MESSAGE: The program exited normally
```

Standard output from a single run with no breakpoints



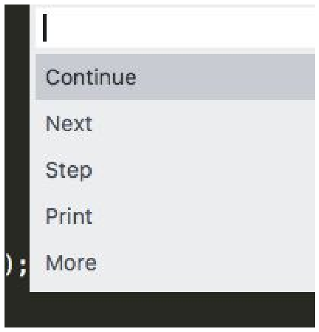
```
  printf("Standard input file\n");
  char c;
  while((c = getchar()) != EOF) {
    putchar(c);
  }

  int y = square(x);

  printf("Hello world!\n");

  return 0;

}
```

```
Argument is -f
Argument is 23
x is 0
x is 1
x is 3
x is 6
x is 10
Standard input file

CBUGGER MESSAGE: Breakpoint hit: hello.c:20 - main
CBUGGER MESSAGE: Variable values - note that current line has not executed  (watch out for uninitialized)
CBUGGER: x = 10
```

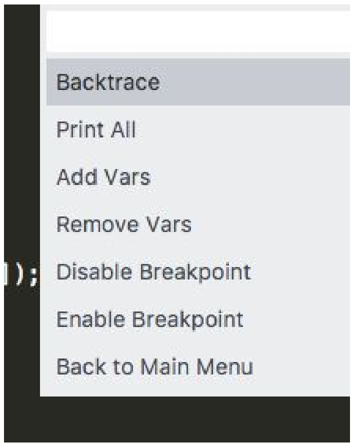
Output from a run that stopped at a breakpoint: note the debugger messages

3.1.6 At A Breakpoint

Once the debugger hits a breakpoint in the program, the fun can really begin. This is where the user can display information about what’s going on in the program, and decide where to follow it next. I chose a few gdb commands that I've used most often to offer to the user. Because there are so many options, there are actually two menus: a shorter main menu and a slightly longer secondary menu.



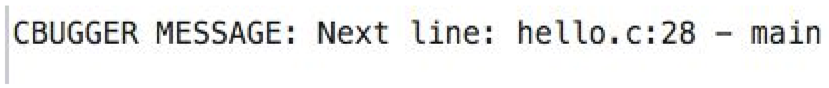
Main menu



Secondary menu

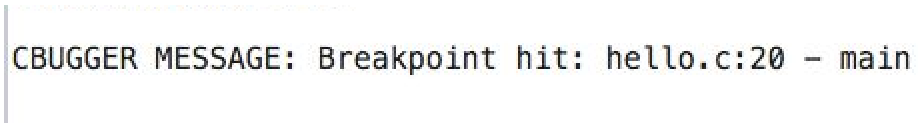
3.1.6.a Advance In Program

The main menu is mostly focused on controlling the flow, where the debugger will go next. There are three options: the first is 'continue', which advances to the next breakpoint or until the program ends. The others are 'next', which advances to the next line in the current function, and 'step', which advances to the next line of code that executes (it will step inside a function call). In any case, the debugger displays any output that the program produced, and prints the line and function that the debugger is now stopped at.



```
CBUGGER MESSAGE: Next line: hello.c:28 - main
```

Next or Step message example



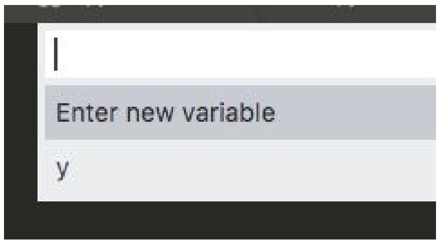
```
CBUGGER MESSAGE: Breakpoint hit: hello.c:20 - main
```

Continue message example

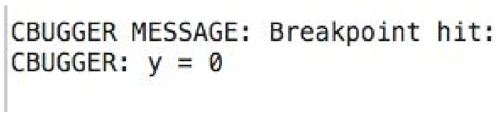
There are three handled ways the program could end: exits normally, exits with a non-zero status code, or gets a signal. Each of these is displayed with a simple message in the output panel. Currently the debugger handles segmentation faults (a particular kind of signal) with a simple message as well.

3.1.6.b Display Debugging Information

There are two main kinds of debugging information that this debugger can currently display: variable values and the function call stack. There are multiple options for variable value printing. The first, displayed in the main menu, is a simple 'print'. This option prompts the user for a variable name, and then prints the value to the output panel if it exists and is in scope. It also displays a list of the 6 most recent variable names, so the user can click on a menu item instead of retyping every time.



Print menu



```
CBUGGER MESSAGE: Breakpoint hit:
CBUGGER: y = 0
```

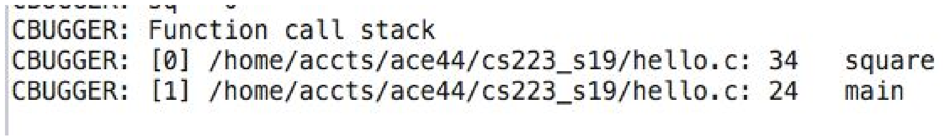
Variable print message

The second option for printing, as discussed earlier, is to pre-set variables to print at every stopping point. A third option is also on the secondary menu, called 'Print All'. This prints all the variables and their values that are in scope at the current stopped point.

A fourth method of printing is a precursor to a more complicated feature. When the debugger stops at a line, if there is an in-scope variable referenced in that line, the value is printed. Right now this is limited to whitespace-separated variables.

Currently, simple variable values are displayed best. More complicated variables like structs, strings, arrays, and pointers in general need additional support for the user to fully examine.

The secondary menu includes an option for printing the function call stack, called 'Backtrace'. This is a useful gdb command that displays the functions that were called to reach whatever stopping point the debugger is currently at. The debugger displays this call stack from the top down.

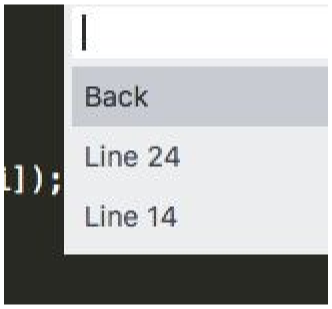


```
CBUGGER: Function call stack
CBUGGER: [0] /home/accts/ace44/cs223_s19/hello.c: 34   square
CBUGGER: [1] /home/accts/ace44/cs223_s19/hello.c: 24   main
```

Example of output from Backtrace command

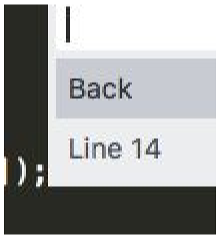
3.1.6.c More Options

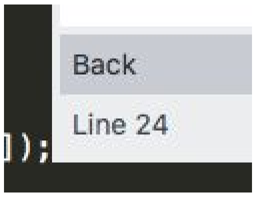
The remaining two options on the secondary menu are for enabling and disabling breakpoints. These useful gdb commands allow the user to make the debugger not stop at a breakpoint without permanently deleting the breakpoint, thus allowing it to be added back (enabled) later. This manifests in the visual debugger as the breakpoint icon (red stop sign) doesn't disappear. The enabled breakpoints appear in the list of breakpoints that can be disabled; the disabled breakpoints appear in the list that can be enabled.



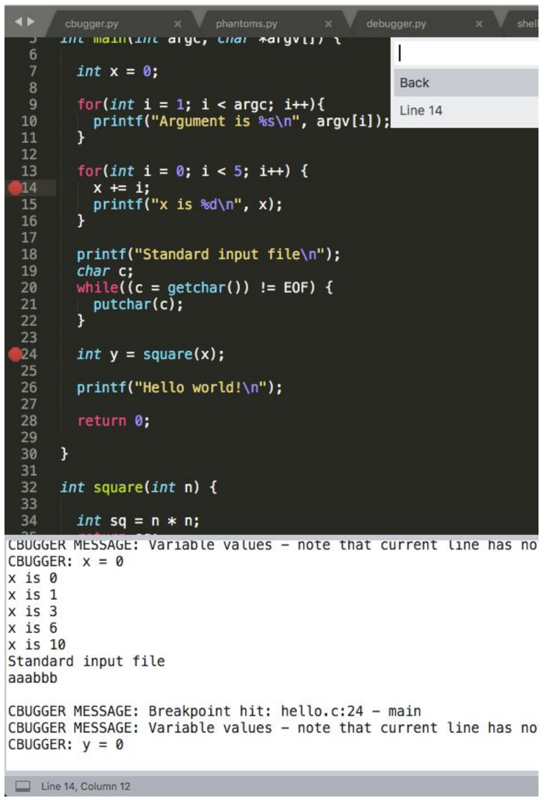
Disable menu

After selecting 14:





Enable menu *Disable menu*



```
 6
 7     int x = 0;
 8
 9     for(int i = 1; i < argc; i++){
10       printf("Argument is %s\n", argv[i]);
11     }
12
13     for(int i = 0; i < 5; i++) {
14       x += i;
15       printf("x is %d\n", x);
16     }
17
18     printf("Standard input file\n");
19     char c;
20     while((c = getchar()) != EOF) {
21       putchar(c);
22     }
23
24     int y = square(x);
25
26     printf("Hello world!\n");
27
28     return 0;
29
30   }
31
32   int square(int n) {
33
34     int sq = n * n;
```

```
CBUGGER MESSAGE: Variable values - note that current line has no
CBUGGER: x = 0
x is 0
x is 1
x is 3
x is 6
x is 10
Standard input file
aaabbb

CBUGGER MESSAGE: Breakpoint hit: hello.c:24 - main
CBUGGER MESSAGE: Variable values - note that current line has no
CBUGGER: y = 0
```

Debugger does not stop at Line 14;
Enable Breakpoint menu pictured in the top right

3.2 How to Run

To install this program on your own computer:

1. Install Sublime Text 3
2. Install the Sublime SFTP plugin
3. Download the code as a zip file from https://github.com/acelwood/cbugger.git
4. Unzip the file and rename the folder 'CBugger'
5. Move the file to '/User/<NAME>/Library/Application Support/Sublime Text 3/Packages/'
6. Reload Sublime Text 3

Currently only known to be compatible with Mac OSX due to some file hard-coding. If you change the path in the instructions to be the Linux location for Sublime Text 3 packages, and change the path in the code to be the Linux location for the SFTP file servers, it should work on Linux as well. Unfortunately, the Windows compatibility is unknown and unlikely.

To test or use the program:

1. Setup Sublime SFTP plugin if it's never been done before: https://docs.google.com/document/d/15kyAx28zdL9LvnCjwPzKDiFIzcNlsFVolT0F9bux4qE/edit?usp=sharing
2. Open a file to edit using SFTP
3. Select 'Launch Remote Debugger' under the 'Debugger' menu along the top bar
    a. If the 'Debugger' menu does not appear, reload Sublime Text and make sure the package was put in the right folder
    b. On Mac, selecting CTRL-` will display the Python console and show any error messages that might be occurring.
4. Set inputs like command line arguments or standard input files, set variables to display.
5. Set or remove breakpoints by right-clicking
6. Select 'Run' under 'Debugger' to run the debugger!

To develop or extend the work

1. Open the cbugger.py file
2. Open any files in the /src/ directory
3. Edit away!
4. Save the cbugger.py to reload changes to the plugin

4. Future Work

There are many opportunities for further development of this project, some of which I'm excited about, others of which I'm *very* excited about.

The first feature I'd want to finish, and one that I worked on but ultimately couldn't flesh out fully was displaying more complete debugging information at breakpoints. As a ULA, I've noticed that often students assume the issue with their code is in one place, when it's actually in another. So, they end up printing variables that don't demonstrate what's really wrong. This feature would, for each line up the stopping point, display the current value for any variables referenced on that line. There are two options that I explored for implementing this feature: Sublime Phantoms, and directly inserting text. Each of these has challenges, advantages, and drawbacks. An initial version of this feature is implemented by printing variables in the current line at breakpoints. The extension would be to do this for multiple lines, and display the results in-line instead of in the output panel. This would help tie the results of the debugger even more closely to the source code.

In terms of other enhancements to the debugger, I understand why many debuggers and IDEs are so complicated and busy, because there are so many options for configuring the debugging commands. Once you understand all the possibilities, you want to provide all of that to the user. There are many more gdb commands that could be included, including additional options for commands that already exist. The next ones I'd want to include would be setting breakpoints on function names and other points instead of just line numbers, and being able to continue a certain number of times (this can be helpful for breakpoints inside loops). However, there are many, many more, and although I do want to include more, I also think it's important to be restrained for this particular project. This debugger is designed for new users, and too many options will just turn this debugger into another overly complicated thing for them to learn. I think the secondary menu works well as a decluttering mechanism, so maybe there are more ways to hide the additional options so that more advanced users can find them without overwhelming the beginners.

To figure that out, I could ask users themselves: both experienced computer science majors, even the undergraduate learning assistants, and novice students who would truly be learning the debugger for the first time. In general, user feedback is an aspect of this project that is missing. When embarking on the next stages of development, I would want to incorporate user input on what current features are most useful, and what features they feel are missing or want. This includes everything from debugger commands and configurations to the design of how debugging information is displayed. Ultimately, I want the debugger to be as intuitive and simple as possible while providing powerful debugging capabilities.

One final feature idea that I think would be incredible, a key to making this tool as useful as possible to the computer science classes here and C programmers in general, is to integrate valgrind. This was never a feasible option to finish in a semester, but if this project continues, this is where it could end up. I’m not sure of the best way to incorporate valgrind (maybe it needs to be a completely different plugin) but being able to debug memory issues using valgrind through Sublime Text would complete the integration of C programming tools.

5. Acknowledgements

Thank you to the Yale Computer Science Department for four years of all kinds of lessons.
Thank you to my professors, especially my advisor James Glenn.
Thank you to my friends, who carried me through.