Advanced Exercise – Week 6

Due: Before October 29, 10:00PM

Submission Instructions

We're going to try submitting via Gradescope for this advanced homework.

You must select the right page for each question on Gradescope.
If you do not select the right pages, we will not grade the question.

Advanced homeworks are all-or-nothing. When submitting via office hours, we could interact with you and give feedback for what you are missing. That's a little harder with Gradescope. We will try to keep up and grade submissions as they are submitted so that you know whether you have completed everything successfully before the deadline. However, there are many students and only a few course staff. *The earlier you submit, the more likely we will be able to look over your submission before the deadline.*

1 "printf debugging"

While debuggers and memory-checkers and all of the other tools can be very powerful, sometimes the easiest thing to do is simply print some things out as your program runs. Commenting and re-enabling lines all the time, however, gets pretty tedious pretty quickly. Consider something like this:

```
#include <stdio.h>

#ifdef NDEBUG
#define DBG(_s)
#else
#define DBG(_s) printf(_s)
#endif

int main() {
  DBG("Program starting\n");
  printf("Hello World\n");
}
```

One improvement on this is to distinguish debugging prints. Perhaps by prefixing them with `"Debug: "`, like so:

```
// Careful copying this in, the trailing \'s are important. Macros are by
// definition one line, but that can be hard to read. A trailing \ (with no
// whitespace after it!) means to continue as if the next line were part of
// this line, so this whole thing is "one line":
#define DBG(_s)\
  do {\
    printf("Debug: ");\
    printf(_s);\
  } while (0)
```

Notice that because we have multiple statements (multiple calls to `printf`), we wrap the macro in `do/while`. This "loop" is one statement long.

1.1 Why Block Macros?

Give an example of some code where using `BAD_DBG_MACRO` would give a different result than `DBG`

```
#include <stdio.h>
#define BAD_DBG_MACRO(_s)\
  printf("Debug: ");\
  printf(_s);

int main() {









    return 0;
}
```

Macros Tip: Confused by all of the macros? Try running `gcc -E main.c`, which will print the results after expanding all the macros (after the preprocessor has run) but before doing anything else. Note, this will include the expansion of all `#include`'s, so there's a lot of stuff at the beginning, but your expanded code will be at the bottom.

1.2 Adding some context information

Have you found yourself adding `printf("here\n");` and `printf("now here\n");` to code? Somewhere between `"okay here now"` and `"ooh, got here!"`, things can get a bit confusing... While this is probably a sign you should consider using a real debugger, we can make these debuggging statements a little better. Gcc has some built-in macros that can be very helpful, such as `__LINE__` which expands to the current line:

```
int main() {
  printf("Currently on line %d\n", __LINE__);
}
```

There are a few others as well that can be useful. Finish this `DBG` macro so that it prints the file, function, and line number. The output should look something like:

```
    Debug main.c::main:18 Program starting
```

```
#include <stdio.h>

#ifdef NDEBUG
#define DBG(_s)
#else
#define DBG(_s)\
```

```
#endif

int main() {
  DBG("Program starting\n");
  printf("Hello World\n");
}
```

1.3 Let's Make it Pretty

Recall that your terminal program a "terminal emulator". In early computers, a terminal simply took in a stream of characters to print and printed those characters. As terminals got fancier, however, it was sometimes interesting to send non-printing characters that would change how to terminal behaved. This was a bit of a mess in the 60s and 70s when every manufacturer had different "magic characters" that would change the terminal behavior. Fortunately, we standardized on a common set of escape codes that are still in use today.

For this question, you shouldn't need to look anything up, just try experimenting with things!

This version of the `DBG` macro uses some escape codes. Try it out and see what it does:

```
1 #define DBG(_s) do {\
2   printf("\033[1;31m");\
3   printf("Debug: ");\
4   printf("\033[0;31m");\
5   printf(_s);\
6   printf("\033[m");\
7   } while (0)
```

Between lines 2 and 4, the only change is a 1 turns to a 0. Looking at the resulting output, what property of the output does this control?

Check out `` `man ascii` ``, what does "033" mean? What does a leading 0 mean for a constant?

What can you change line 4 to so that it prints yellow instead?

1.4 And let's really make it useful

The big piece that has been missing so far is that `printf` is a *variadic function*, that is it's a function that takes a variable number of arguments. It would be nice to be able to print variables from our `DBG` macro, but to do that we need to set up `DBG` to take a variable number of arguments, we need a `variadic macro`.

Write a `DBG` macro so that this program gives the expected output
(Note, the line numbers do not need to match)

```
> make -B main && ./main
cc main.c -o main
Debug main.c::main:21 Program name is ./main and was passed 1 argument(s)
Hello World
Debug main.c::main:24 a is 2, a++ is 1, ++a is 2
Modifying variables in debug macros is a bad idea. a is 2
> make -B CFLAGS='-DNDEBUG' main && ./main
cc -DNDEBUG main.c -o main
Hello World
Modifying variables in debug macros is a bad idea. a is 0
> cat main.c
  #include <stdio.h>
```

```
int main(int argc, char **argv) {
  int a = 0;
  int b = 10;

  DBG("Program name is %s and was passed %d argument(s)\n", argv[0], argc);
  printf("Hello World\n");

  DBG("a is %d, a++ is %d, ++a is %d\n", a, a++, ++a);
  printf("Modifying variables in debug macros is a bad idea. a is %d\n", a);
}
```

2 Extending GDB

While GDB can be very useful, it can sometimes benefit from some assistance from custom "pretty printer"s. Consider a simple class such as:

```
class Point {
public:
  int x, y;

  Point(int x_coord, int y_coord) {
    x=x_coord; y=y_coord;
  }
};

int main() {
  std::vector<Point> points;
  points.push_back(Point( 1,-2));
  points.push_back(Point( 5, 4));
  points.push_back(Point(-8, 2));
  points.push_back(Point( 9,12));
  points.push_back(Point(-1, 2));

  // Break
```

While gdb can print *primitive data types*, int's, char's, or other types built into the language, older versions struggled when asked to print complex data types such as a vector:

```
(gdb) print points
points = {<std::_Vector_base<Point, std::allocator<Point> >> = {
  _M_impl = {<std::allocator<Point>> = {<__gnu_cxx::new_allocator<Point>> = {
    <No data fields>}, <No data fields>}, _M_start = 0x615c90, _M_finish = 0x615cb8,
      _M_end_of_storage = 0x615cd0}}, <No data fields>}
```

As of gdb 7.0, however, gdb added a pretty printing mechanism that understands standard library containers, so now we get something more like:

```
(gdb) print points
points = std::vector of length 5, capacity 8 = {{x = 1, y = -2}, {x = 5, y = 4},
  {x = -8, y = 2}, {x = 9, y = 12}, {x = -1, y = 2}}
```

Notice how this both printed information about the vector and then recursively printed information about its contents. Gdb doesn't really understand `Points`, but it can make an educated guess, which is to print the public members.

We can help gdb out, however, but teaching it how to print a `Point` and what's interesting for us to see:

```
(gdb) print points
points = std::vector of length 5, capacity 8 = {(1, -2), (5, 4), (-8, 2), (9, 12), (-1, 2)}
```

Here's the Python written to make that happen:

```
import gdb

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

   def to_string(self):
      x = self.val['x'] # This is a dictionary lookup in Python
      y = self.val['y'] # You can read any class member variable this way

      # This line creates the string that gdb will use. {}'s are like %d or %s in printf
      return '({}, {})'.format(x,y)
```

```
def lookup_type(val):
   if str(val.type) == 'Point':
      return PointPrinter(val)

gdb.pretty_printers.append(lookup_type)
```

In gdb, loading the custom pretty printer is one command:

```
(gdb) source my_pretty_printer.py
```

After this line, any time gdb tries to print a `Point`, it will call the `to_string` function defined here.

The Assignment:

Pick any current or old project with an interesting data type and write a custom pretty printer for it.

Explain why you chose the elements you did and how they are the most helpful for debugging. Do not simply print the whole structure (like my example did here). You must explain why you chose what you did and why it's the most useful for debugging.

Include the declaration of your class (not the whole definition, aka the stuff in `.h`, not `.c`), your python pretty printer, and a transcript of a gdb session using your new pretty printer.

There is a lot of information on writing pretty printers available online. I found Red Hat's guide the easiest to understand, **but** the last line `python execfile` no longer seems to work, use `source` instead. You may find other examples more helpful.

Why display these values?

Your Class, Pretty Printer, and GDB Session:

Extra Space If Needed