How to Debug with Print Statements

1 How do I print?

First, it's important to make sure we know how to actually print data, as the behavior of the print statements in C0 can seem unintuitive. Before we talk about the actual functions, let us establish a golden rule:

Important: Whenever you want to see output from your program, make sure you print a newline (`\n`) at the end of anything you want to see.

You can print a newline either using the `println` function (which will print a newline at the end of its string argument), or by printing the newline character as part of the string.

You need to do this because printing is "buffered" in C0: C0 accumulates data in a buffer until it sees a newline, at which points it "flushes" the buffer and actually displays it to the terminal. This can also be manually done with the `flush` function.

There are two ways to print something in C0: using `printf` and using type-specific print functions. Let's look at both.

1.1 Printing using `printf`

The most convenient way to print in C0 is to use the `printf` function. The first argument of `printf` is the string you wish to print. This string may contain *format specifiers*, which are used to print the value of expressions supplied as additional arguments.

For example, if you wanted to print a pair of integers `a` and `b` followed by a new line, you would use the following command:

```
printf("(%d,%d)\n", a, b);
```

When this command is executed, the first `%d` will be replaced by the value of `a`, and the second `%d` by the value of `b`. So, if `a` is 4 and `b` is 5, then this will print `(4,5)`.

In C0, `printf` supports three format specifiers: `%d` for values of type **`int`**, `%c` for characters, and `%s` for string. You can find more information about `printf` in the C0 Library Reference.

1.2 Printing using Type-Specific Functions

If you want to print the value of a variable, you can also to use appropriate print function for the type of that variable. These are as follows:

- `print` – Strings
- `println` – Strings, but also prints a newline at the end (see the above important note)
- `printbool` – Booleans
- `printchar` – Characters
- `printint` – Integers

You can also find more information on these functions in the C0 Library Reference.

For example, to print the pair of integers `a` and `b` in this way, we would do the following:

```
print("(");
printint(a);
print(",");
printint(b);
println(")");
```

Notice that we use `println` to print the closing parenthesis. If `a` is 4 and `b` is 5, then this will print `(4,5)`.

Clearly, that's more cumbersome than using `printf`, but that's occasionally useful, in particular since `printf` does not provide a format specifier for booleans.

2 Where do I put my print statements?

There are a lot of interesting uses for print statements in the debugging world, some of which you may not even think of at first glance. Here are some useful applications:

- **What is the value of a particular variable at this point in the program?** – The classic use of a print statement is to check the value of a variable at a given point. If your variable is a complicated data structure, it is helpful to write a function to print it nicely. See Section How do I print a data structure?.
- **Did I enter this conditional or loop?** – Sometimes your concern is whether your code is actually being executed when it's supposed to, such as when it's inside an if statement or loop. In that case, you may want to put a print statement inside the branch to see whether it prints or not.

  ```
  int a = 0;
  if (a == 0 && d > 213)
  {
    a = 1;
    printf("Inside the first if statement\n");
  }
  ```

  If the sentence is printed, we know that we entered the if statement.
- **Did I enter this function?** – Same as the above; you can place your print statement at the beginning of your function to see if you called it. It is often also helpful to print out the values of at least some of the arguments:

  ```
  void merge(int[] A, int lo, int mid, int hi)
  // Some contracts (omitted)
  {
    printf("Entered Merge!\n");
    printf("lo is %d, mid is %d, hi is %d.\n", lo, mid, hi);

    // Some more code (omitted)
  }
  ```

- **When did I crash?** – Possibly the most underrated use of the print statement is to figure out exactly when your program terminated if it crashed unexpectedly. Place a print statement before and after every statement you think could crash, and base further action off of what doesn't print.

  ```
  printf("A\n");
  int a = *r;
  printf("B\n");
  int b = *q;
  printf("C\n");
  int c = *p;
  printf("D\n");
  return 0;
  ```

  If this program prints A and B, but doesn't print C or D, for example, then I know that `q` is a `NULL` pointer.

Important: If you have a lot of print statements, it is often a good idea to format them so that it is easy for you to scan through them to find information. One useful technique is to indent the data you print based on the indentation of the code they are in, and to print out the name of the function you are in with no indentation applied. That way, print statements for a function are clearly distinguished from print statements for a different function, and print statements within the same function but at different points can have some distinguishing factors as well.

3 How do I print a data structure?

Often, you may need to write your own printing function for a more complex data structure for more careful debugging. We will demonstrate how to do so by writing an `array_print` function for integer arrays. First, we'll need a header:

```
void array_print(int[] A, int n)
//@requires 0 <= n && n <= \length(A);
```

Make sure you include any arguments you need to make your function work; in this case, we need the length of the array.

Note: We'll be writing this function to align with the other print functions, which means it will **not** print a newline at the end of the data structure.

Next, we want to make sure we print all the useful information in our data structure:

```
void array_print(int[] A, int n)
//@requires 0 <= n && n <= \length(A);
{
  for (int i = 0; i < n; i++) {
    printf("%d", A[i]);
  }
}
```

This technically prints all the information we need, but we will likely run into a problem if we try to use the function as-is. Consider the array `[1,2,3,4]`. That will print as `1234`; but does that mean we have an array with the numbers `1`, `2`, `3`, and `4`? Or does it mean we have an array with just the number `1234`? This function currently prints something ambiguous and therefore not very useful.

In order to rectify this, we want to format our data structure nicely; you may see this called "pretty printing" in some other languages.

```
void array_print(int[] A, int n)
//@requires 0 < n && n <= \length(A);
{
  printf("[");
  for (int i = 0; i < n; i++) {
    printf("%d", A[i]);
    if (i != n-1) printf(",")
  }
  printf("]");
}
```

Now, an array containing `1`, `2`, `3`, and `4` will print `[1,2,3,4]` as expected, while an array containing `1234` will print `[1234]`. Now, our data structure is clear and readable, and we can print it whenever we want!

Note: Writing a good print function for a data structure seems like a lot of work. However, this is an investment that pays off handsomely while debugging complex code!

Now, we can use `array_print` any time we want to print the contents of an array of integers. Taking the merge example from before, we might add a call to `array_print` in order to print out all the arguments:

```
void merge(int[] A, int lo, int mid, int hi)
  // Some contracts (omitted)
  {
    printf("Entered Merge!\n");
    printf("lo is %d, mid is %d, hi is %d.\n", lo, mid, hi);
    printf("A is ");     // New!
    array_print(A, hi);   // New!
    printf("\n");

    // Some more code (omitted)
  }
```

Notice that we don't have any guarantee that `hi` is actually equal to the length of the array — but that's fine: all we care about is the array segment from `lo` to `hi`, so it is ok if we miss some things outside of that segment.