How to Debug with Contracts

Note: This guide is about how to *use* contracts, but does not explain what contracts are or how they work. This guide is also not about how to prove that contracts are valid. Please consult the lecture and recitation notes for more info on these topics.

1 When should I write my contracts?

During lecture and recitation, we often show you code that is informed by the contracts that have been written; thus, in this case, the contracts were written first. This is very helpful in writing reasonable code in the first place.

That being said, it is often helpful to also write contracts *after* you write your code as well! This will help you identify any lingering bugs in your code.

In this guide to success, we will talk about both methods and how they can be useful to you in your debugging experience.

2 Writing contracts before code

Contracts written before your code is written are useful in guiding the code you write. We call this process **deliberate programming**, and we generally recommend this approach as your main method of programming. Here are some examples:

- **Exclude inputs you don't want to deal with** – Writing a `//@requires` contract will allow you to narrow your focus on inputs you want your function to deal with; you can ignore any other inputs! Take this power function as an example:

  ```
  int POW(int a, int b)
  //@requires b >= 0;
  {
  // code
  }
  ```

  By requiring that `b` is not negative, we no longer need to worry about negative exponents! Only do this when the problem specification lets you do so (or requires you to!).

  In addition, if the function is called with a negative exponent somewhere else in your code, you can quickly identify the issue as a unsafe function call, and not an issue with the function itself.

- **Exclude unsafe inputs** – Make sure all your functions have appropriate requires statements to avoid clear `NULL` dereferences, array out-of-bounds errors, or plain weird answers. As an example:

  ```
  void incr(int* p)
  //@requires p != NULL
  {
      *p += 1;
  }
  ```

This is very similar to the previous bullet point. The main difference is that (with contracts disabled) calling a function with an unsafe input (e.g. `p(NULL)`) will cause the program to crash, while calling a function with an excluded input (e.g. `pow(2, -2)`) will return a meaningless result.

- **Create code that directly satisfies invariants** – If you have loop invariants written before your code, your code will likely be focused on making sure those loop invariants hold while still iterating through the loop.

  ```
  for (int i = 0; i < n; i++)
  //@loop_invariant 0 <= i && i <= n;
  //@loop_invariant b >= 2 * i;
  //@loop_invariant c == b + 1;
  {
    b = 2 * i;
    c = b + 1;
  }
  ```

  While this may seem like a very trivial example, the point here is that our goal is to actively fulfill the loop invariants, and we have achieved that goal in the simplest way we can.

- **Describe properties of your function's result** – Using careful and correct postconditions for all of your functions helps you write better code by making sure you do not rely on unspecified behaviour. Furthermore, writing these postconditions will assist you in your debugging journey by allowing you to determine exactly which function is incorrect, so you can catch bugs early on before they become a significant and highly complicated problem.

3 Writing contracts after code

If your code is failing at this point, make sure you've written all the types of contracts from the previous section; those will still be useful to you. Contracts written after your code are also useful, though, because a contract failure will give you a useful error message. If your concern is that you don't know where or why your code is crashing, a well-placed contract may be your solution!

- **Use `//@assert` contracts to find a specific failing line** – If you don't know what line your program is crashing on, assertions before every possible problematic statement can tell you what the specific problem is:

  ```
  //@assert p != NULL;
  int i = *p;
  //@assert i < \length(A);
  bool j = A[i];
  return j;
  ```

  Note the lack of assertion before the obviously safe statement of `return j`; we only need to assert for things that may crash our program. If `*p` is a `NULL` dereference, for example, our first contract will fail.

- **Temporary contracts** – Sometimes, you may want to create contracts to debug in the short-term, and delete them later. Oftentimes, these contracts will not be things that are generally true (such as asserting that a certain variable is 5), but that should be true for a very specific test case. Don't forget to delete these when done debugging!

- **Check for infinite loops** – Sometimes, you accidentally increment your loop in the wrong direction (it happens to the best of us). If you're getting an infinite loop, you can check for that by writing a loop invariant that bounds your loop invariant in both directions, like so:

  ```
  for (int i = 0; i < n; i--) //uh oh! we don't want to decrement here!
  //@loop_invariant 0 <= i && i <= n;
  {
  // code
  }
  ```

  As you can see, we will catch this sort of error immediately with this contract, since we will quickly decrement below 0. (Obviously, this won't catch something where `i` doesn't move at all; plan accordingly!)

- **Temporary contracts to stop infinite loops** – Sometimes you have a complicated loop that depends on a data structure being empty to exit. In this case, you can add a temporary counter and an assertion to ensure that if your code enters an infinite loop you'll detect that. As an example:

  ```
  int count = 0;
  while (!stack_empty(S))
  {
     // For this test case we shouldn't need more than 50 iterations!
     //@assert count < 50;
     count ++;

     // Some code to actually do the problem
  }
  ```