Unit 21: The `assert` Macro

Learning Objectives

After completing this unit, students should:

- be aware of the `assert` macro provided by C
- be able to use `assert` to defensively guard against bugs in code

`assert`

You have learned what an assertion is and how it can help you to reason about your programs. Previously we have only seen the use of an assertion as a commenting tool, to comment on certain properties that are guaranteed to be true at a certain point of a program.

Now, we will make the concept of assertion even more powerful -- we can cause our program to throw an error if an assertion is ever violated. This way, we can tell immediately if certain assumptions or properties that we made in our code are violated.

C provides a macro called `assert()` (in the header file `assert.h`) which takes in a logical expression. This logical expression must always evaluate to true when `assert` is used. Otherwise, assert will throw an error, giving the file and the line number where the error occurred.

Using assertions and the `assert` macro can help us pinpoint exactly where our code goes wrong, quickly.

Take the following code, for instance:

```
long nrows = 20;
long ncols = 10;
long matrix[nrows][ncols];

  :
    matrix[i][j] = -1;
  :
```

Remember our rule: we can only access memory that has been allocated to us. So we need to make sure that `i` and `j` must be within the correct range. We can add an assert statement to check that `i` and `j` are correct.

```
    assert(i >= 0 && i < nrows && j >= 0 && j < ncols);
    matrix[i][j] = -1;
```

If the assertion fails, the program will exit and print something like this:

```
assert: assert.c:12: int main(): Assertion `i >= 0 && i < nrows && j >= 0
&& j < ncols' failed.
Aborted
```

So that we know on which line in our code the assertion has failed -- in this case, which is the line in our code we have tried to access memory that does not belong to us.

As you reason about your code, sprinkle `assert` liberally in your code so that, if you are wrong about your reasoning or you make a careless mistake in your code, `assert` will spot that for you.

Problem Set 21

Problem 21.1

Consider the code:

```
1  void foo(long x) {
2      if (x % 2 == 0) {
3          // do something
4      } else {
5          assert(x % 2 == 1);
6      }
7  }
```

Would the assert in Line 5 above ever fail?

Problem 21.2

Consider the following code for selection sort:

```
1  long max(long length, const long list[])
2  {
3    long max_so_far = list[0];
4    long max_index = 0;
5    for (long i = 1; i <= length; i += 1) {
6      if (list[i] > max_so_far) {
7        max_so_far = list[i];
8        max_index = i;
9      }
10   }
11   return max_index;
```

```
}

void selection_sort(long length, long list[])
{
  long last = length;
  for (long i = 1; i <= length; i += 1) {
    long curr_pos = last - 1;
    long max_pos = max(last, list);
    if (max_pos != curr_pos) {
      long temp = list[max_pos];
      list[max_pos] = list[curr_pos];
      list[curr_pos] = temp;
    }
    last -= 1;
  }
}

int main()
{
  long n = cs1010_read_long();
  long *list = cs1010_read_long_array(n);
  selection_sort(n, list);
}
```

Add an `assert` statement whenever the array `list` is accessed to check if we ever violated the boundary of the array `list` (i.e, accessing beyond the memory allocated to `list`).