Erik Poole
00719917

1.
```
res = open(path, "r");
assert(res >= 0);
```

This is a bad use of an assertion. Assertions should be used to catch errors in our program, but here we're checking if a system call (something out of our control) was successful. This is something that probably should be handed with error checking.

2.
```
assert(p = malloc(size));
```

This is also a bad assertion. This line is performing a (probably) critical function for the program and asserting it all at once. If this line were to be optimized out by a compiler it could cause major issues and be difficult to track down.

3.
```
p = malloc(size);
assert(p);
```

This is more appropriate than the last one but is still probably not great. Malloc is another process outside of our control and it failing is not likely to be indicative of a bug in our software. It's possible that it could show a developer that they're pushing it as far as memory goes, but that will vary based on machine and environment so I think it would be best to leave out.

4.
```
int array[length];
...
int index = hash_function(input);
assert(index >= 0 && index < length);
array[index] = -1;
```

This is the first assertion that seems great! Double checking that the index provided by a hash function is in the appropriate range for it's data structure is a great sanity check and if you ever have a failure it's highly indicative of either a bad input or a bad hash_function, both of which are relevant to the developer.

5.
```
res = fclose(fd);
assert(res == 0);
```

I think this one is probably OK. It is asserting a system call, which is usually not a good practice (as I've gone over in previous questions) but fclose only fails if the file descriptor isn't open which probably indicates a problem somewhere else in your program.

6.

```
assert(tree_is_balanced(t));
```

This is also a good assertion. It ensures that your data structure is functioning as expected and might be a good catch-all for a wide array of bugs.

7.

```
if (a == 42)
  doSomething();
else
  assert(false && "a should be 42");
otherCode();
```

This one feels bad; at best it's somewhat redundant. I would think if you wanted to make sure a == 42 you should simply state "assert(a == 42)" rather than messing around with an if/else statement.

8.

```
res = printf("%d\n", i);
assert(res > 0);
```

Also a bad assertion, similar to ones we've seen before. If the write systemcall that printf covers is failing that isn't really a problem with the developer's code.

9.

```
if (func())
  x = 3;
else
  x = 17;
assert (x==3 || x==17);
```

This assertion is bad and has no meaning. Unless your computer was failing in dramatic ways (and if so I bet this code doesn't function, either) this assertion will never fail. It should be removed.

10.

```
switch(x) {
case ...:
default:
  assert("unexpected value");
}
```

This is a tough one. If there were only a specific handful of x values that the program can handle it might be valuable to assert a warning if the default is reached and I do think this assertion has some merit depending on how the value of x was assigned. However, care must be taken that if the default is reached the program handles it in some other way since the compiler will optimize this statement out. I think I'm leaning towards this being bad practice, but I could see some times where it might be valuable.

11.

```
assert(!"unreachable");
```

This is dependent on context but I think it's a good assertion. If there is a section of your code that shouldn't ever be reached (given certain specifications) then the assert will be able to warn you if the code isn't running as you expected. Putting a line like this after a return statement doesn't make sense, but as a branch of a complicated algorithm it has merit.

12.

```
assert(sizeof(some_struct) == 72);
```

Probably not a good assertion. A struct's size should be set when it's initialized. This is another assert that is verifying something out of your control and if it's failing you have bigger problems.

13.

```
... calculate lengths of sides of a triangle ...
assert(short1 + short2 <= long + epsilon);
```

This is a good assertion. If you're calculating the sides of a known triangle this could be a valuable check to verify that a number of calculations haven't gone too far wrong. This assertion wouldn't catch all errors, but it would catch egregious ones.

14.

```
std::map<int, int> map;
... code ...
assert(map[42] == 0);
... code ...
map.at(42) = 32;
```

The assert probably isn't valuable here, though it depends on what occurs in "... code ..." if the value of the map at 42 is used for calculation it's possible that you want the assert statement before being reassigned. If it isn't used before map[42] is reassigned though the assertion statement isn't proving anything before the value is just changed again anyway.

15.
... calculate angles of a triangle ...
assert(angle1 + angle2 + angle3 - 180.0 < epsilon);

Also seems like a reasonable assertion. This is another example of checking the culmination of (potentially) a lot of math and it's a good sanity check that your angle calculations are correct.

16.
assert(Index++ < Length);

This is bad and dangerous. Changing a value during an assertion will cause problems when the assertion is optimized out and can be another example of a bug that's hard to catch in testing but could be nasty when released to customers.