In the reading, we discussed pre-conditions and post-conditions which placed contracts upon the functions that we wrote. This allowed us to reason about the behavior of functions at the level of the function call. However, we do not have any hooks for reasoning about the internal behavior of the function. To do this, we introduce propositions, logical statements about our code that may or may not hold during its execution.

Propositions

We express propositions about the state of the variables in our program. For example, consider the following code snippet:

```c
// ...
int x = 1;
// Point A
if (b) {
    x = 5;
    // Point B
} else {
    x = 2;
    // Point C
}
// Point D
```

*b* is an arbitrary boolean expression. For example, consider the proposition $x < 3$ and whether it holds throughout the execution of this code snippet. Because $x$ is initialized to 1, we know that the proposition holds initially (Point A). If $b$ is true, then we go inside the if-branch of the conditional. Here we assign $x$ the value of 5 which breaks the proposition. Therefore, after the assignment (Point B), the proposition no longer holds. If $b$ is false, then we go inside the else-branch of the conditional. The assignment to $x$ here (to the value 2) does not break the proposition, so after this assignment (Point C), the proposition still holds.

Finally, we could have arrived outside of the conditional (Point D) either by entering the if-branch or the else-branch. In the former case, the proposition does not hold (because $x$ is now 5). In the latter case, the proposition does hold (because $x$ is now 2). Because of this, at Point D, we can only say that proposition may or may not hold.

In contrast, consider the proposition, $x > 0$. This proposition is always true at every point in the code snippet (verify this on your own!). Because of this, we say that the proposition is invariant with respect to the code snippet: our code ensures that it always true. Invariants are a powerful reasoning principle about code. By stating correctness in terms of invariants, we can determine if a program is correct by checking that its corresponding invariant holds at the end of the program. Furthermore, by setting up these invariants before we write the code, we can use the invariants to help guide us during program design.
Reasoning About Straight-line Code

Consider the following program:

```c
void mystery(int x) {
    int y = 0;
    // Point A
    if (x < 10) {
        y = x + 1;
        // Point B
    } else {
        x = x + 3;
        // Point C
    }
    // Point D
}
```

At each of the indicated points, determine if the following propositions hold never (×), sometimes (?), or always (✓).

<table>
<thead>
<tr>
<th></th>
<th>x &gt; 10</th>
<th>y = 0</th>
<th>y &gt; x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Point B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Point C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Point D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Point E</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Consider the following program:

```c
int mystery(int x, int y) {
    int c = 0;
    if (x < 0) {
        // Point A
        c = 0;
    } else if (y < 0) {
        // Point B
        c = x;
    } else {
        // Point C
        c = mystery(x, y - 1);
    }
    // Point D
    return c;
}
```

At each of the indicated points, determine if the following propositions hold never (×), sometimes (?), or always (✓).

<table>
<thead>
<tr>
<th></th>
<th>x &gt;= 0</th>
<th>y &gt;= 0</th>
<th>c &gt;= 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Point B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Point C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Point D</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reasoning About Looping Code

Consider the following program:

```c
void mystery(int n) {
    int sum = 0;
    int i = 0;
    // Point A
    if (n < 0) { return; }
    for (; i < n; i += 2) {
        if (i % 2 == 0) {
            // Point B
            sum += i;
        } else {
            // Point C
            sum -= i;
        }
    }
    // Point D
}
```

At each of the indicated points, determine if the following propositions hold never (×), sometimes (?), or always (✓).

<table>
<thead>
<tr>
<th></th>
<th>i &gt; n</th>
<th>i % 2 == 0</th>
<th>sum &gt;= 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Point B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Point C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Point D</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Recall that a `while` loop is used for code whose termination is not known or not a simple function of a loop variable. Consider the following function:

```c
int mystery(int n) {
    int ret = 0;
    // Point A
    if (n < 0) { n = -n; }
    while (n > 0) {
        if (n % 2 == 1) {
            // Point B
            ret += 1;
        }
        n = n / 10;
        // Point C
    }
    // Point D
    return ret;
}
```

At each of the indicated points, determine if the following propositions hold `never (×)`, `sometimes (?)`, or `always (√)`.

<table>
<thead>
<tr>
<th>Point</th>
<th>n &gt; 0</th>
<th>ret &gt;= 0</th>
<th>n % 2 == 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Point B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Point C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Point D</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In a sentence, describe what this function returns.
The `rand()` function of the `stdlib.h` header produces a random number in the range 0 to `RAND_MAX` (a constant also defined in `stdlib.h`). Consider the following program:

```c
#include <time.h>
#include <stdlib.h>

int mystery(int k, int n) {
    srand(time(NULL)); // Initializes rand()
    if (n <= 0 || k <= 0 || k < n) { return 0; }
    int q = rand() % k;
    // Point A
    while (q < n) {
        // Point B
        k = k - 1;
        if (k == 0) {
            q = -1; // Point C
        } else {
            q = rand() % k;
        }
    } // Point D
    return q;
}
```

At each of the indicated points, determine if the following propositions hold never (×), sometimes (?), or always (✓).

<table>
<thead>
<tr>
<th></th>
<th>q &gt; k</th>
<th>k &lt; n</th>
<th>q &gt; 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Point B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Point C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Point D</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>