Reasoning About Code

Often functions make certain assumptions about their arguments, and it is the caller's responsibility to make sure those assumptions are valid. These are often called *preconditions*. A precondition for `f()` is an assertion (a logical proposition) that must hold at input to `f()`. The function `f()` is supposed to behave correctly and produce meaningful output as long as its preconditions are met. If any precondition is not met, all bets are off. Therefore, the caller must be sure to call `f()` in a way that will make these preconditions true. In short, a precondition imposes an obligation on the caller, and the callee may freely assume that the obligation has been met.

Here is a simple example of a function with a precondition:

```
/* Requires: p != NULL */
int deref(int *p) {
    return *p;
}
```

It is not safe to dereference a null pointer; therefore, we impose a precondition that must be met by the caller of `deref()`. The precondition is that $p \neq NULL$ must hold at the entrance to `deref()`. As long as all callers ensure this precondition, it will be safe to call `deref()`.

Assertions may be combined using logical connectives (and, or, implication). It is often also useful to allow existentially ($\exists$) and universally ($\forall$) quantified logical formulas. For instance:

```
/* Requires:
    a != NULL
    for all j in 0..n-1,  a[j] != NULL */
int sum(int *a[], size_t n) {
    int total = 0, i;
    for (i=0; i<n; i++)
        total += *(a[i]);
    return total;
}
```

The second precondition might be expressed in mathematical notation as something like

$$\forall \texttt{j} . (0 \leq \texttt{j} < \texttt{n}) \implies \texttt{a[j]} \neq \texttt{NULL}.$$

If you are comfortable with formal logic, you can write your assertions down in this way, and this will help you be precise. However, it is not necessary to be so formal. The primary purpose of preconditions is to help you think explicitly about precisely what assumptions you are making, and to communicate those requirements to other programmers and to yourself.

Postconditions are also useful. A postcondition for `f()` is an assertion that is claimed to hold when `f()` returns. The function `f()` has the obligation of ensuring that this condition is true when it returns. Meanwhile, the caller may freely assume that the postcondition has been established by `f()`. For example:

```
/* Ensures: retval != NULL */
void *mymalloc(size_t n) {
    void *p = malloc(n);
    if (!p) {
        perror("Out of memory");
        exit(1);
    }
    return p;
}
```

When you are writing code for a function, you should first write down its preconditions and postconditions. This specifies what obligations the caller has and what the caller is entitled to rely upon. Then, verify that, no matter how the function is called, as long as the precondition is met at entrance to the function, then the postcondition will be guaranteed to hold upon return from the function. You should prove that this is always true, for all inputs, no matter what the caller does. If you can find even one case where the caller provides some inputs that meet the precondition, but the postcondition is not met, then you have found a bug in either the specification (the preconditions or postconditions) or the implementation (the code of the function you just wrote), and you'd better fix whichever is wrong.

How do we prove that the precondition implies the postcondition? The basic idea is to try to write down a precondition and postcondition for every line of code, and then do the very same sort of reasoning. The requirement is that each statement's postcondition must match (or imply) the precondition of any statement that follows it. Thus, at every point between two statements, you write down an *invariant* that should be true any time execution reaches that point. The invariant is a postcondition for the preceeding statement, and a precondition for the next statement.

It is pretty straightforward to tell whether a statement in isolation fits its pre- and post-conditions. For instance, a valid postcondition for the statement "v=0;" would be $v = 0$ (no matter what the precondition is). Or, if the precondition for the statement "v=v+1;" is $v \geq 5$, then a valid postcondition would be $v \geq 6$. As another example, if the precondition for the statement "v=v+1;" is $w \leq 100$, then $w \leq 100$ is also a valid postcondition (assuming v and w do not alias).

This leads to a very useful concept, that of *loop invariants*. A loop invariant is an assertion that is true at the entrance to the loop, on any path through the code. The loop invariant has to be true before every iteration to the loop. To verify that a condition really is a valid loop invariant for the loop, you treat the condition as both a pre-condition and a post-condition for the loop body.

Let's try an example. Here is some code that computes the factorial function:

```
/* Requires: n >= 1 */
int fact(int n) {
    int i, t;
    i = 1;
    t = 1;
    while (i <= n) {
        t *= i;
        i++;
```

```
    }
    return t;
}
```

A prerequisite is that that input must be at least 1 for this implementation is not correct. Suppose we want to prove that the value of `fact(.)` is always positive. We'll annotate the code with invariants:

```
/* Requires: n >= 1
   Ensures: retval >= 0 */
int fact(int n) {
    int i, t;
            /* n>=1 */
    i = 1;
            /* n>=1 && i==1 */
    t = 1;
            /* n>=1 && i==1 && t==1 */
    while (i <= n) {
            /* 1<=i && i<=n && t>=1   <-- loop invariant */
        t *= i;
            /* 1<=i && i<=n && t>=1 */
        i++;
            /* 2<=i && i<=n+1 && t>=1 */
    }
            /* i>n && t>=1 */
    return t;
}
```

How do we verify that the invariants are correct? This might look pretty complicated, but don't get discouraged—it's actually pretty easy if you just take the time to look at each step. Notice that the function's precondition implies the invariant at the beginning of the function body. Also, the invariant at the end of the function body implies the function's postcondition. Thus, if each statement matches the invariant immediately before and after it, everything will be ok. The only non-trivial reasoning is in the loop invariant. First, we must prove that at the entrance to the first iteration of the loop, the loop iteration will be true, and this follows since the logical proposition $n \geq 1 \wedge i = 1 \wedge t = 1$ implies $1 \leq i \leq n \wedge t \geq 1$ (e.g., if $i = 1$, then certainly $i \geq 1$). Also, we must prove that if the loop invariant holds at the beginning of any iteration of the loop, then it will hold at the beginning of the next iteration, if there is another iteration. This is true, since the invariant at the end of the loop body ($2 \leq i \leq n + 1 \wedge t \geq 1$) together with the loop termination condition ($i \leq n$) implies the invariant at the beginning of the loop body ($1 \leq i \leq n \wedge t \geq 1$). It follows by induction on the number of iterations that the loop invariant is always true on entrance to loop body. The conclusion is that `fact()` will always make the postcondition true, so long as the precondition is established by its caller.

To give you some more practice, we'll show another example implementation of `fact()`, this time using recursion. Here goes:

```
/* Requires: n >= 1 */
int fact(int n) {
    int t;
    if (n == 1)
```

```
        return 1;
    t = fact(n-1);
    t *= n;
    return t;
}
```

Do you see how to prove that this code always outputs a positive integer? Let's do it:

```
/* Requires: n >= 1
   Ensures: retval >= 0 */
int fact(int n) {
    int t;
    if (n == 1)
        return 1;
            /* n>=2 */
    t = fact(n-1);
            /* t>=0 */
    t *= n;
            /* t>=0 */
    return t;
}
```

Before the recursive call to `fact()`, we know that $n \geq 1$ (by the precondition), that $n \neq 1$ (since the if statement didn't follow its then branch), and that `n` is an integer. It follows that $n \geq 2$, or that $n - 1 \geq 1$. That's very good, because it means the precondition is met when making the recursive call, and thus we're entitled to conclude that the return value from `fact(n-1)` is positive (by virtue of the postcondition for `fact()`). The rest is straightforward.

In general, any time we see a function call, we have to verify that its precondition will be met. Then we are entitled to conclude that its postcondition holds, and to use this fact in our reasoning.

If we annotate every function in the program with pre- and post-conditions, this allows *modular reasoning*. This means that I can verify function `f()` by looking only at the code of `f()` and the annotations on every function that `f()` calls—but I do *not* need to look at the code of any other functions, and I do not need to know everything that `f()` calls transitively. Reasoning about a function then becomes an almost purely local activity. We don't have to think hard about what the rest of the program is doing.

Preconditions and postconditions also serve as useful documentation. If Bob writes down pre- and post-conditions for the module he has built, and Alice wants to invoke Bob's code, she only has to look at the pre- and post-conditions—she does not need to look at or understand Bob's code. This is a useful way to coordinate activity between multiple programmers: each module is assigned to one programmer, and the pre- and post-conditions become a kind of contract between caller and callee. For instance, if Alice knows she is going to have to invoke Bob's code, then when the system is designed Alice and Bob might negotiate the interface between their code and the contract on who is responsible for what.

There is one more major use for this kind of reasoning. If we want to avoid security holes and program crashes, there are usually some implicit requirements the code must meet: for instance, it must not divide by zero, it must not make out-of-bounds accesses to memory, it must not dereference null pointers, and so on. We can then try to prove that our code meets these requirements using the same style of reasoning. For instance, any time a pointer is dereferenced, there is an implicit precondition that the pointer is non-null and in-bounds.

Here is an example of using this kind of reasoning to prove that array accesses are within bounds:

```
/* Requires: a != NULL and a[] holds n elements */
int sum(int a[], size_t n) {
    int total = 0, i;
    for (i=0; i<n; i++)
            /* Loop invariant: 0 <= i < n */
        total += a[i];
    return total;
}
```

In this example, the loop invariant is straightforward to establish. It is true at the entrance to the first iteration (since the first iteration ensures $i = 0$), and it is true at the entrance to every subsequent iteration (since the loop termination condition ensures $i < n$, and since i only increases), so the array access $a[i]$ is within bounds.

Of course, in general, proving the absence of buffer overruns might be much more difficult, depending on how the code is structured. However, if your code is structured in such a way that it is hard to provide a proof of no buffer overruns, perhaps you may wish to consider re-structuring the code to make the absence of buffer overruns more evident.

This might all look awfully tedious. The good news is that it does get a lot easier over time. With practice, you won't need to down detailed invariants before every statement; there is so much redundancy that you'll be able to derive them in your head easily. In practice, good programmers might write down the preconditions and postconditions and a loop invariant for every loop, and that will be enough to confirm that all is well. The bad news is that, even with practice, reasoning about your code still does take time and energy—however, it seems to be worth it for code that needs to be highly secure.

While we have presented this in a fairly formal way, in practice good programmers often do the same kind of reasoning without bothering with the formal notation. Also, good programmers sometimes omit the obvious parts of the invariants and write down only the parts that seem most important. Often, we think about data structures and code in terms of the invariants it ought to satisfy first, and only then write the code.

This kind of formal reasoning can be formalized very carefully using the tools of mathematical logic. In fact, there has been a lot of research into tools that use automated theorem provers to try to mathematically prove the validity of a set of alleged pre- and post-conditions (or even to help infer such invariants). You could take a whole course on the topic, but for reasons of time, we won't go any further in this course. Our goal was merely to show you enough to get started on your own, and maybe stir you to investigate further on your own.