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. A *precondition* for `f()` is an assertion (a logical proposition) that must hold for the inputs supplied to `f()`. The function `f()` is presumed 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 ensure that these preconditions hold. 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()`.[1]

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

The third part of the precondition might be expressed in mathematical notation as something

[1] Technically speaking, we also need to know that `p` is a valid pointer: i.e., it is safe to dereference. To be strictly correct, we ought to add that to the precondition. However, here we will follow the convention that every pointer-typed variable is implicitly assumed to have, as an invariant condition, that it is either `NULL` or valid. This convention simplifies and shortens preconditions, postconditions, and invariants.

like

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

If you are comfortable with formal logic, you can write your assertions 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 at the level of a single line of code. 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 preceding statement, and a precondition for the next statement.

It is pretty straightforward to tell whether a statement in isolation meets its pre- and post-conditions. For instance, a valid postcondition for the statement "`v=0;`" would be $\texttt{v} = 0$ (no matter what the precondition is). Or, if the precondition for the statement "`v=v+1;`" is $\texttt{v} \ge 5$ **and** $\texttt{v} \le 101$, then a valid postcondition would be $\texttt{v} \ge 6$ **and** $\texttt{v} \le 102$.[2] As another

[2] Note that if the precondition had simply been $\texttt{v} \ge 5$ then the postcondition would actually require some careful thinking to express due to the possibility of *integer overflow*. Here we take advantage of the fact that we know that incrementing a value ≤ 101 can't lead to an overflow on any reasonable system.

example, if the precondition for the statement "`v=v+1;`" is `w` ≤ 100, then `w` ≤ 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 and you use a proof by induction to prove it valid.

Let's try an example. Here is some code that prints out the decimal representation of an integer, but reversed (least significant digit first):

```
/* requires: n >= 0 */
void binpr(int n) {
    char digits[] = "0123456789";
    while (n != 0) {
        int d = n % 10;
        putchar(digits[d]);
        n = n / 10;
    }
    putchar('0');
}
```

A prerequisite is that the input `n` must be non-negative for this function to work correctly, hence the precondition. Suppose we want to prove that the array dereference (`digits[d]`) never goes outside the bounds of the array. We'll annotate the code with invariants (in blue):

```
/* requires: n >= 0 */
void binpr(int n) {
    char digits[] = "0123456789";     /* n >= 0 */
    while (n != 0) {                  /* n > 0 */
        int d = n % 10;               /* 0 <= d && d < 10 && n > 0*/
        putchar(digits[d]);           /* 0 <= d && d < 10 && n > 0*/
        n = n / 10;                   /* 0 <= d && d < 10 && n >= 0*/
    }
    putchar('0');
}
```

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. For instance, the function's precondition implies the invariant after the first line of the function body. Also, we can prove by induction that `n` > 0 is a loop invariant: we know that `n` ≥ 0 holds at the entry to the loop and at the end of the prior iteration; if we enter the loop, then we also know `n` $\neq 0$; and these two, taken together, imply `n` > 0. Tracing forward, we can see that if `n` > 0 holds at the beginning of the loop body, then `n` ≥ 0 holds at the end of the loop body. The validity of the loop invariant follows by induction on the number of iterations of the loop. The conclusion is that the array accesses in `binpr()` will always be in-bounds, as long as `binpr()`'s precondition is met.

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

```
/* requires: n >= 0 */
void binpr(int n) {
    char digits[] = "0123456789";
    if (n == 0) {
        putchar('0');
        return;
    }
    int d = n % 10;
    putchar(digits[d]);
    int m = n / 10;
    binpr(m);
}
```

Do you see how to prove that the array accesses are always valid? Let's do it:

```
/* requires: n >= 0 */
void binpr(int n) {
    char digits[] = "0123456789";
    if (n == 0) {
        putchar('0');
        return;
    }                       /* n > 0 */
    int d = n % 10;         /* n > 0 && 0 <= d && d < 10 */
    putchar(digits[d]);     /* n > 0 && 0 <= d && d < 10 */
    int m = n / 10;         /* n > 0 && 0 <= d && d < 10 && m >= 0 */
    binpr(m);
}
```

Before the recursive call to `binpr()`, we know that $\texttt{m} \geq 0$ (by the annotations). That's very good, because it means the precondition is met when making the recursive call. As a result, we're entitled to conclude that `binpr(m)` is safe (does not perform any out-of-bounds array access). Also, we can easily see that the expression `digits[d]` is safe, by virtue of the annotations we've filled in. It follows that, as long as the precondition to `binpr()` is respected, the function is memory-safe.

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 we can verify function `f()` by looking only at the code of `f()` and the annotations on every function that `f()` calls—but we do *not* need to look at the code of any other functions, and we 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 negotate 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 && size(a) >= n */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* Loop invariant: 0 <= i && i < n && n <= size(a) */
        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 during the first iteration, `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, and since `n` and `size(a)` never change), so the array access `a[i]` is always within bounds.

Of course, proving the absence of buffer overruns in general 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 should 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 write 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, you 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, you can do the same kind of reasoning

without bothering with the formal notation. Also, you can often omit the obvious parts of the invariants and write down only the parts that seem most important. Sometimes, it is helpful to 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 reasoning can be formalized more precisely 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 CS 161. Your basic intuition should be enough to handle most cases on your own.

By the way, you may have noticed how useful it is to be able to "speak mathematics" fluently. Now you know one reason why we make you take Math 55 or CS 70 as part of your computer science education.